RICE UNIVERSITY

Enabling Distributed Reconfiguration in an Actor Model

by

Arghya Chatterjee

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

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

Zoran Budimlić
Senior Research Scientist
Department of Computer Science

Swarat Chaudhuri
Associate Professor of Computer Science

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

Houston, Texas

April, 2017

ABSTRACT

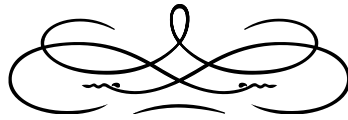
Enabling Distributed Reconfiguration in an Actor Model

by

Arghya Chatterjee

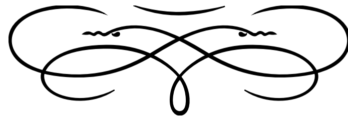
The demand for portable mainstream programming models supporting scalable, reactive and versatile distributed computing is growing dramatically with the proliferation of manycore/heterogeneous processors on portable devices and cloud computing clusters that can be elastically and dynamically allocated. With such changes, it's becoming extremely difficult for distributed software systems and applications to achieve scalability and programmability without complex coordination and synchronization patterns.

In this dissertation, we address the dynamic reconfiguration challenges that arise in distributed implementations of the Selector Model. We focus on the Selector Model in this work because of its support for multiple guarded mailboxes, which enables the programmer to easily specify coordination patterns that are more general than those supported by the actor model. The contributions of this dissertation are demonstrated in two implementations of distributed selectors, one for distributed servers and another for distributed Android devices. Both implementations run on distributed JVMs and feature the automated bootstrap and global termination capabilities introduced in this dissertation. In addition, the distributed Android implementation supports dynamic joining and leaving of devices, which is also part of the dynamic reconfiguration capabilities introduced in this dissertation.



To mom and dad,

Debi and Jagannath Chatterjee.



Acknowledgments

Firstly, I would like to thank my thesis advisor, Prof. Vivek Sarkar, for his unwavering support and mentorship throughout this work. I owe him the greatest degree of appreciation for providing me the opportunity to be a part of the Habanero Extreme Scale Research group.

I would also like to thank my thesis committee members; Vivek Sarkar, Zoran Budimlić, Swarat Chaudhuri and Lin Zhong, for their time, feedback and suggestions for improvements in my thesis.

Zoran Budimlić, thank you for your continuous guidance throughout the last two years. It was a real pleasure working with you.

I express my sincere gratitude to Prof. John Mellor-Crummey, Prof. Keith Cooper, and Prof. Linda Torczon. John's Parallel Computing course (COMP 422), and, Keith and Linda's Introduction to Compiler course (COMP 412), introduced me to the world of Parallel Programming and Compilers.

I would like to thank all the members of the Habanero Extreme Scale Research Group for their feedback and support during the length of my time at Rice. I would especially like to thank my co-authors in the works presented in this thesis, Branko Gvoka, Shams Imam, Srđan Milaković and Bing Xue. I also thank the undergraduate researchers at Rice that I have been fortunate enough to mentor and work with, Ashok, Tom, Timothy, and Hunter.

I would like to thank my friends for all the love and encouragement. Special thanks to Alexander, Alina, Ankush, Anwesha, Arvind, Belia, Deepak, Dragos, Jayvee, Kartik, Keliang, Kuldeep, Kumail, Lechen, Leo, Madhuparna, Melissa, Prasanth, Rabimba, Rishi, Rohan, Ryan Luna, Ryan Spring, Sarah, Simba, Srayan, Sriraj, Sugu-

man and Terry.

I would also like to thank the administrative staff at the Computer Science Department at Rice University, who promptly helped me in various ways throughout the last three years. I am also thankful to the executive board members of the CS Graduate Student Association, and my role as the President of the CSGSA during 2015-2106 has been an incredible experience.

Finally, I am indebted to my parents, Debi and Jagannath Chatterjee for their continuous support, unconditional love and encouragement. They stood by my side through all the difficult times. Love you both, and I hope I have made you proud.

Thank you all !!!

Contents

Abstract ii

Acknowledgments iv

List of Illustrations viii

1 Introduction 1

1.1 Motivation 1

1.2 Thesis Statement 5

1.3 Contributions 5

1.4 Organization 5

2 Background 7

2.1 Habanero-Java Library (HJlib) 7

2.2 Selector Model (SM) 7

2.2.1 Coordination and Synchronization patterns with the SM . . . 11

2.3 Distributed configuration in Akka and SALSA 18

2.4 Distributed Communication in Clusters and Mobile Platforms 23

3 Cluster Platform 27

3.1 HJDS Design 27

3.2 Automatic Bootstrap 32

3.3 Global Termination 35

4 Android Platform 41

4.1 DAMMP Design 41

4.2 Dynamic Joining 45
4.3 Dynamic Leaving 48

5 Experimental Evaluation **53**
5.1 Cluster based implementation (HJDS) 53
5.1.1 Hardware Setup 53
5.1.2 Benchmarks 53
5.1.2.1 NQueens First K Solutions 54
5.1.2.2 Pi Precision 56
5.2 Android based implementation (DAMMP) 58
5.2.1 Hardware Setup 59
5.2.2 Benchmarks 59
5.2.2.1 Trapezoidal Approximation 59
5.2.2.2 Pi Precision 61

6 Related Work **64**
6.1 Akka 64
6.2 SALSA: Simple Actor Language, System and Architecture 65
6.3 AmbientTalk 66
6.4 ActorDroid 67
6.5 ActorNet 68

7 Conclusion & Future Work **70**
7.1 Conclusion 70
7.2 Future Work 72

Bibliography **73**

Illustrations

2.1 Sample HJlib program 8

2.2 Selector Model with multiple guarded mailboxes, local state and a message processing unit 9

2.3 Life cycle of a Selector instance 11

2.4 Computation of a synchronous request-reply pattern using an Actor Model and a Selector Model 13

2.5 Left: Complex implementation of the join pattern using the Actor Model. The Adder actor aggregates the data items from each of the three sources and sums them up. Right: Similar join pattern, using the Selector Model for the Adder. Each mailbox in the Adder corresponds to a sequence number. Sources send messages to the mailbox which matches the sequence number. 14

2.6 Using Selectors to solve the Join Pattern problem of Figure 2.5. The aggregator selector version: `Adder Any Order` maintains one mailbox for each source. For simplicity, we assume sources are identified by consecutive integers starting at 0. 16

2.7 Using Selectors to solve the Join Pattern problem of Figure 2.5. The aggregator selector version: `Adder Round Robin` maintains one mailbox for each source. For simplicity, we assume sources are identified by consecutive integers starting at 0. 17

3.1 The Distributed Selector System 28

3.2 Proxy Actor forwarding messages to local selectors and remote selectors at each *place*. The underlying message forwarding protocol is transparent to the application programmer. 30
3.3 Sample configuration file, nodes are on Rice University's DAVinCI cluster. 33
3.4 Bootstrap process of the Distributed Selector program. 34
3.5 Stage 1: Termination process initiated when each of the remote nodes and the master node is in idle state. 36
3.6 Decomposition of the runtime and user-level view during the termination process. a) Before initiating the termination process(Stage 1) b) Places 1 and 2 are considered as 'idle' in this phase 37
3.7 Stage 2: Termination signal passed around the network in a conceptual place-ring fashion with the Master Place at the end of the sequence. 38
3.8 Stage 3: Terminate all nodes. Send termination signal around the network in a conceptual place-ring fashion. 40

4.1 Overview of the distributed selector system. On each hand-held device, we have a single selector system with the Mobile Communication Manager of each device exposed to the entire distributed network. All communications to external handheld devices are performed by the Mobile Communication Manager in the runtime. 43
4.2 Dynamic join protocol for new handheld devices. 47
4.3 Showing the dynamic leaving of devices while calculating a trapezoidal approximation of the function. We start with one device, and join upto three devices. Each of the colors depict which device contributed to the computation of that trapezoid in the function. . . 51

5.1 NQueens : Shows strong scaling of computing the NQueens problem on a 17×17 board using the described algorithm. Workers compute solutions sequentially when given a partial solution with six placed queens. The number of workers per node is constant (12). The solution limit is set to 1_477_251, which is a tenth of the size of a complete solution set. Mean Execution time in seconds from 20 iterations. The error bar on mean execution time plot shows the standard deviation of the 20 iterations. 55

5.2 Precise Pi : Shows strong scaling for the calculation of Pi with a precision of 80_000. Number of workers per node is constant (12). Work is evenly distributed among all selectors. Mean Execution time in seconds from 20 iterations. The error bar on mean execution time plot shows the standard deviation of the 20 iterations. 57

5.3 Trapezoidal Approximation: Shows the best and average time (over 20 executions) with 4 Nexus 5 devices to compute an approximation with 100, 000, 000 intervals for the integration. The x axis (in log scale) shows the number of work messages sent by the master to workers, and the y axis shows the best and average times. The total work remains the same for all experiments. For each work message, a reply message is sent back to the master with the result. 60

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 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 of the devices 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. 63

Chapter 1

Introduction

1.1 Motivation

Distributed applications for today's cloud and mobile platforms need more than mere computing capacity. Without improvements in programmability, the ever-growing complexity of interaction patterns in distributed applications can limit us from efficiently exploiting a diversity of available computational resources. For modern cloud services, the need for exploiting both multicore and multi-node parallelism is widely acknowledged, but there remains a conceptual gap between programming models for shared-memory parallelism and those for distributed concurrency. Mobile platforms, such as Android devices, have seen an increasing trend in available hardware parallelism but they remain resource constrained on the power consumption and thermal dissipation fronts. Aggregating the computing capabilities of multiple such mobile platforms in a distributed and dynamic setting opens the possibilites for both performance improvements and novel dynamic and distributed applications.

The *Actor Model*(AM) [1, 2], represented by isolated processes (*actors*) that interact solely via asynchronous message passing, is a natural fit for a unified concurrency model for both cluster-level and multi-core parallelism. However, we believe that the traditional Actor model poses certain limitations with respect to actor *coordination* and *synchronization* patterns, which is why we aim to develop a distributed runtime system for the more general Selector Model (SM) [3].

To address dynamic reconfiguration challenges, in this dissertation we introduce two implementations of the distributed selectors, one for distributed servers (HJDS: Habanero Java Distributed Selectors) and another for distributed Android devices (DAMMP: Distributed Actor Model on Mobile Platforms). HJDS features the automated bootstrap and global termination capabilities. In addition, the distributed Android implementation allows programmers to react to the reconfiguration of mobile devices and adapt to dynamic changes in device availability. DAMMP is a cross-platform runtime system that can span *distributed servers* and *mobile devices*. We built the HJDS runtime as an extension to the single-node shared-memory Habanero Java Runtime Library (HJlib) [4].

The Selector Model (SM) [3] is an extension of the Actor programming model [1] that supports multiple guarded mailboxes (which can be enabled or disabled independently) and combined with priority-based processing of messages. It overcomes known difficulties in implementing synchronization and coordination patterns using the pure actor model. SM offers a promising approach for building distributed concurrent applications with both productivity and scalability.

Although the Actor Model (AM) has been successfully used for many concurrent computations, not all concurrent problems are most effectively solved using this model. In some concurrent programming patterns, preserving the integrity of the objects requires synchronization protocols to control the order in which messages are processed in the mailbox [5]. Since AM forces all communication to be asynchronous, concurrent coordination involving multiple actors might be harder than using non-actor concurrency constructs such as locks [6]. The Habanero Java Distributed Selector (HJDS) model acts as an abstraction to support synchronization and coordination mechanisms among multiple selectors across multiple devices / cluster-nodes. The

multiple guarded mailboxes in the Selector Model allows efficient coordination and synchronization patterns such as:

- synchronous request-response,
- join patterns in streaming applications, and
- producer-consumer with bounded buffer

The mobile platform extension, Distributed Actor Model for Mobile Platforms (DAMMP) can be instrumental in improving the efficiency of parallel and distributed applications on modern mobile devices through aiding the on-device computation by harnessing the processing power of the available mobile devices in the ad-hoc network. While power and performance constrained resources such as mobile devices are not usually associated with distributed and high-performance computing, projects including distributed peer-to-peer file sharing in mobile applications [7] and off-the-grid agricultural distributed mobile applications [8] motivate real-world impact of distributed mobile applications. However, while most such projects are based on specialized hardware or specific models, we aim to provide a *portable* and more *extensible* framework with DAMMP. Some popular scenarios in image recognition [9, 10] and GPS triangulation [11, 12, 13, 14] illustrate the usability of the model in a dynamic and heterogeneous settings:

- Law enforcement agencies in the field may not have access to a secure and high-performance computing resource to run a facial-recognition application, but may instead have to rely on the combined computing power of their hand-held mobile devices. These compute intensive jobs may not be possible to run within the energy and computation limitations of a single device, but with the

use of DAMMP they can create a distributed network of multiple devices to collaboratively execute the face-recognition algorithm [9].

- Hikers are at a remote location trying to locate each other without Internet connection and with a set of devices with different computing power, battery states and temperatures. A DAMMP application can allow them to share the GPS data among their devices and triangulate their position using the collective computing power of their devices, while at the same time managing the power consumption and heat dissipation to maximize the collective usability and longevity of their ad-hoc distributed system of heterogeneous devices [11].

Scenarios such as these illustrate a need for a programming model and system that would allow an easy distribution and redistribution of work that needs to be done, a convenient mechanism for the application to adapt to dynamic changes in network topology (such as devices leaving or joining the group), and an effortless model for offloading computation to another device. DAMMP combines the flexibility of Actor runtimes (e.g., weak message ordering support for migration), the expressiveness of the Selector model (with multiple guarded mailboxes), distributed execution on ad-hoc networks using Wi-Fi Direct, support for Actor-friendly interface between the runtime and adaptation mechanisms that reach to the dynamic changes in the system (devices joining/leaving). In the remainder of this dissertation, we will use the term Selector, for user-defined entities, and the term, Actor, for entities internal to the DAMMP runtime (which happen not to require the multiple-mailbox functionality available in selectors).

1.2 Thesis Statement

The actor/selector model can be used to enable a cross-platform runtime system that can span both distributed servers (with support for automatically bootstrapping the system and global termination) and mobile devices (with support for dynamic joining and leaving of devices).

1.3 Contributions

This thesis makes the following contributions:

- Two implementations of a distributed runtime that supports the Selector Model, one for distributed servers (clusters) and another for distributed Android devices (developed jointly with co-authors [15, 16]).
- Support for Automated Bootstrap and Global Termination of the runtime on cluster platforms.
- Support for Dynamic Joining and Dynamic Leaving of devices on Android platforms.
- Introduces a standalone and seamless offload model for computation offloading on to nearby mobile devices.
- Experimental evaluation of performance benefits of using the Selector Model on clusters and heterogeneous Android devices (obtained jointly with co-authors [15, 16]).

1.4 Organization

This thesis is organized as follows:

- Chapter 2 contains background on the Selector Model, and on the Akka [17] and SALSA [18] distributed configuration modules for cluster-based applications.
- Chapter 3 discusses how we address the challenges of reconfiguration in distributed applications on the cluster platforms (HJDS). This chapter focuses on the Automated Bootstrapping and Global Termination services provided by our runtime.
- Chapter 4 discusses how we address the challenges of reconfiguration in distributed applications on the Android platforms (DAMMP), and focuses on the Dynamic Joining and Dynamic Leaving features of our runtime.
- Chapter 5 evaluates our runtime on cluster and Android devices.
- Chapter 6 discusses related work for both cluster and Android platforms.
- Chapter 7 wraps up by summarizing the thesis and potential areas for future research.

Chapter 2

Background

2.1 Habanero-Java Library (HJlib)

Habanero-Java (HJ) is a parallel programming model developed at Rice University. HJ is based around four orthogonal core concepts [19] :

- Lightweight dynamic task creation and termination using `async, finish, future, forall, forasync` constructs
- Collective and point-to-point synchronization using phasers
- Mutual exclusion and isolation
- Locality control using the "place" abstraction.

The Habanero-Java library (HJlib) is a library implementation of the HJ model in Java 8 [20]. HJlib is a parallel programming library that combines several parallel programming concepts. HJlib runtime uses Java threads as *workers*. The runtime creates small number of worker threads in a *thread pool*, typically one per core. Figure 2.1 shows a sample HJlib program that splits up a large input data into **nchunks** asynchronous tasks.

2.2 Selector Model (SM)

Selectors are an extension of the Actor model [3]. A Selector is an execution unit that has the capability to process incoming messages. Similar to actors, selectors encapsu-

```
// launches the code to be run by Habanero Java runtime
launchHabaneroApp ( () —> {
    // waits for all nested asynchronous tasks before exiting
    finish ( () —> {
      // chunks data to allocate to cores
      for (int chunk = 0; chunk < nchunks; chunk ++) {
          final int mychunk = chunk;
        // creates an asynchronous task to process my chunk
          async ( () —> {
            ...
          }); // end async block
      } // end for block
    }); // end finish block
}); // end application
```

Figure 2.1 : Sample HJlib program

late their local state and process incoming messages, one message at a time. Figure 2.2 shows the key components of the Selector model. The modularity and data locality of the Actor model are still preserved when using selectors.

Selectors differ from the conventional actor design in two ways:

a) Selectors have multiple mailboxes to receive messages, which allows messages with priority or purpose to be concurrently and asynchronously added to different mailboxes, eliminating the need for blocking coordination and reduce contention,

b) Each mailbox has a guard that can disable or enable the mailbox while pro-

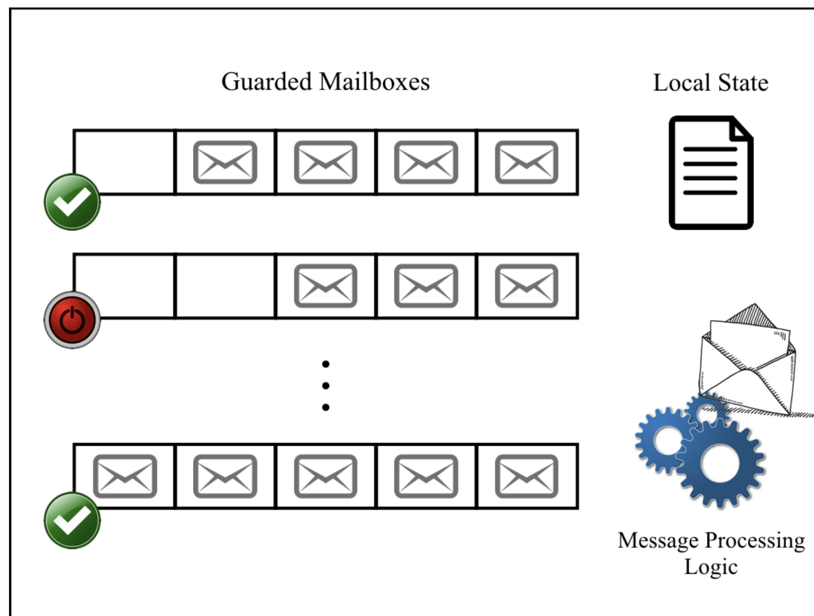


Figure 2.2 : Selector Model with multiple guarded mailboxes, local state and a message processing unit

cessing a message. This *guard* does not affect the mailbox's ability to receive messages; it only controls whether the messages in the mailbox can be processed or not.

An *actor* can be viewed as a *selector* with an always **enabled**, single mailbox.

The Distributed Selector life cycle (Figure 2.3) is very similar to its shared-memory counterpart [3], and displays the complete encapsulation and state isolation found in most Actor model interpretations. The Selector life cycle consists of the following three stages:

- **created**: A new *selector* is asked to be created, its location in the distributed runtime is hidden unless specified by the user. At this point, the selector object is not guaranteed to have been instantiated, but an access *handle* is immediately

created and passed to the caller and any entity holding this handle can start sending messages to the selector. Initially, all mailboxes are enabled, and the runtime will buffer all incoming messages for the selector.

- **started**: A *selector* has started processing messages. It processes messages one at a time from any enabled mailbox. During the processing of a message, the selector can choose to *enable* or *disable* some of its mailboxes, thus changing its own behavior. Since the mailboxes have priorities, the selector will try to process messages in mailboxes of higher priority first, however such priorities are not strict. To guarantee fairness among all mailboxes, the Selector rotates between mailboxes when processing messages (we use this feature to address join patterns in Streaming Applications when using the "round-robin" approach. See Section 2.2.1 for details).

- **terminated**: A *selector* terminates when it calls exit(). A Selector in such a state will not process any messages in its mailbox and ignores all incoming messages, aside from some special cases. The distributed runtime does not terminate a selector until all `new` operations requested by that selector are observed to have completed, and no outgoing message remains in the local buffer. A Selector in such a state cannot be restarted, and a system-wide termination of all selectors will signal the global termination of the application.

The multiple guarded mailboxes in a selector allows the programmer to optionally implement the following actions (each of these selector features are explored in our applications, used for evaluating our HJDS model [Section 5.1]):

- *Mailbox determined by sender*: The message sender can directly specify the mailbox to send a message to.

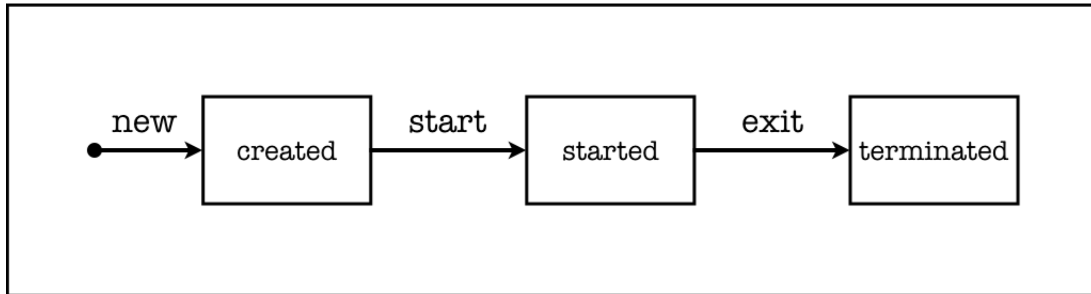


Figure 2.3 : Life cycle of a Selector instance

- *Mailbox determined by receiver*: A message can be sent without a target mailbox and the receiver can either choose to put the message in a default mailbox, or introduce processing logic to inspect the message and determine the mailbox it should be put in. Such approach can be extremely useful in dynamic load balancing and dynamic updates in many interactive or reactive systems.

The idea of guarded mailboxes in selectors is inspired by classical condition variables, in which a thread checks whether a condition is true before continuing execution in a critical section. The results in [3] 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 [3] focused on a single-node (shared-memory) implementation of the Selector Model.

2.2.1 Coordination and Synchronization patterns with the SM

In this section, we briefly explain the motivating examples discussed in Chapter 1 to show how the multiple guarded mailboxes in the Selector Model allow efficient coordination and synchronization patterns, and how the patterns are transparently

applied to our HJDS model. We demonstrate the Distributed Selector(HJDS) model's programmability by contrasting our pattern with a traditional actor-based solution.

- **Synchronous Request-Reply Pattern.** We can observe a synchronous request-reply pattern [21, 22] when a requestor sends a message to the replier, which receives and processes the message, and eventually sends a reply in response to the requestor. Using the Actor Model one may implement this pattern with either a non-blocking or a blocking approach. With a non-blocking mechanism approach, this pattern requires separate messages for request and reply. The incoming messages before the response message must be stashed and unstashed to the mailbox after processing the reply message. While it can be cumbersome for a user to manually code the non-blocking approach, Akka provides the `become` and `unbecome` constructs and the `Stash` trait to enable this pattern [23]. Using the blocking mechanism, implementation can be less complicated but it limits scalability since the worker thread executing the actor is blocked between the request and reply messages.

  Figure 2.4 shows how the computation differs in the Selector Model from the Actor Model. Using the SM, we can define two separate mailboxes, one to receive *regular* messages including all the *request* messages (label: REQUEST) and another mailbox to receive only *synchronous response* messages (label: RESPONSE). Whenever a selector is expected to process a synchronous response message, it disables the REQUEST mailbox, which ensures that the next message to be processed will be from the RESPONSE mailbox. We assume that all response messages from the responder are sent to the RESPONSE mailbox of the selector. The requestor selector stays in the reply-blocked state until a response is received, and after processing the message from the RESPONSE

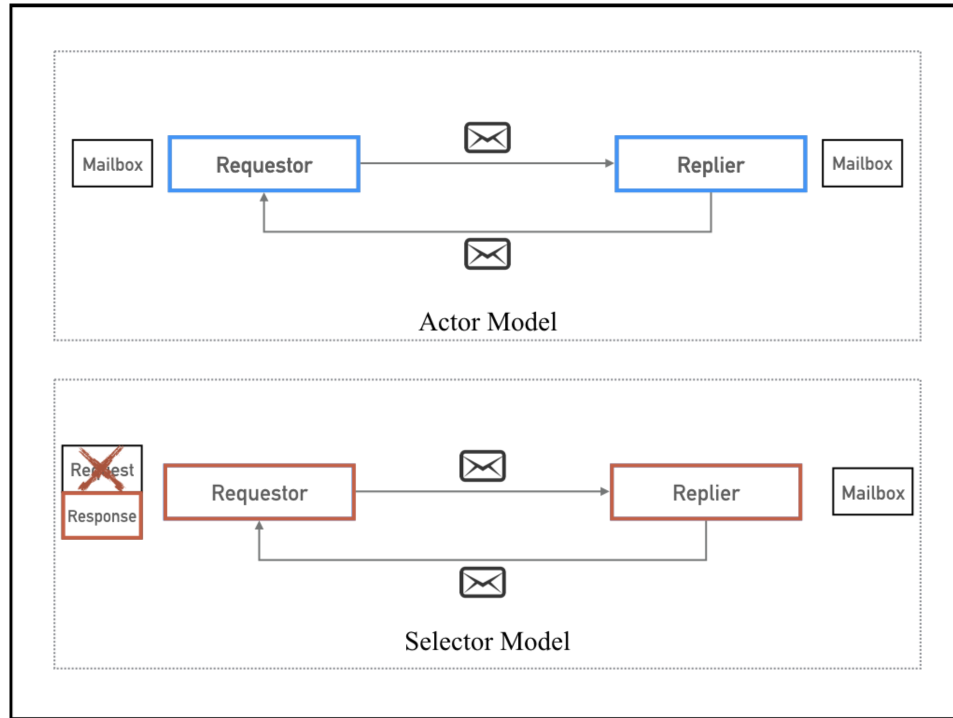


Figure 2.4 : Computation of a synchronous request-reply pattern using an Actor Model and a Selector Model

mailbox, it enables the REQUEST mailbox and starts processing other messages.

Such a pattern translates seamlessly to distributed applications, hides long message passing latencies, and can be a large contributor to improving efficiency in interactive applications and service-oriented architectures where a request-response pattern is commonly used.

- **Join Patterns in Streaming Applications.** Join patterns in any streaming application are usually blocking as they need to match the data from all sources and wait for all the data to arrive before processing the messages. The left im-

age in Figure 2.5 shows an *aggregator*, where the *Adder actor* is consuming data from the *Source actors* and adding streams of corresponding values. In general,

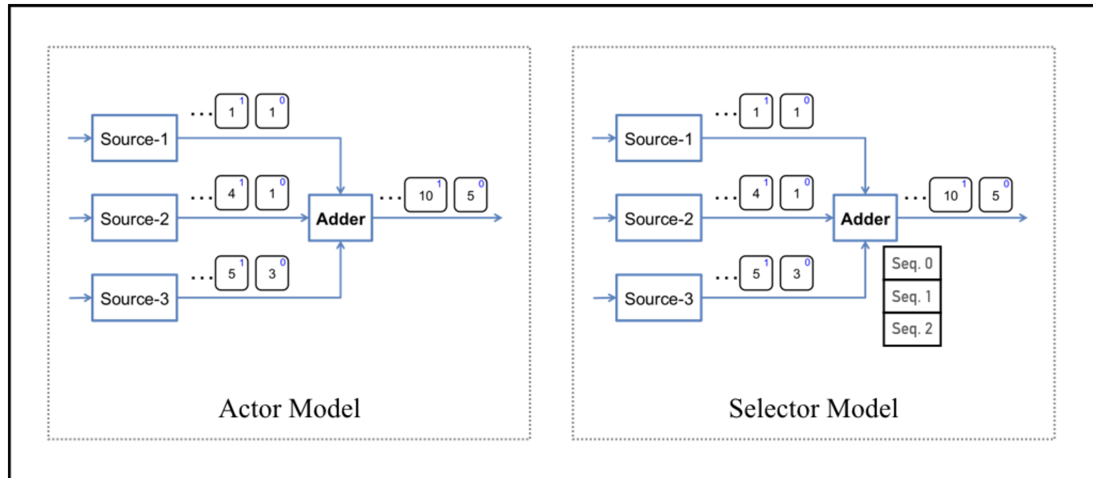


Figure 2.5 : Left: Complex implementation of the join pattern using the Actor Model. The Adder actor aggregates the data items from each of the three sources and sums them up. Right: Similar join pattern, using the Selector Model for the Adder. Each mailbox in the Adder corresponds to a sequence number. Sources send messages to the mailbox which matches the sequence number.

one would expect the Actor model to be an excellent choice for streaming applications since actors can be connected in a data flow chain of producer-consumer pairs with pipeline parallelism, and ensure that messages are processed in FIFO order. However, when using actors, it becomes difficult to mimic a join pattern where messages from two or more data streams are combined into a single message since the order of processing messages is not guaranteed with respect to the sender actors. Further complexity is added when we need to keep track of all the *in-flight* messages, since the actor model is push-based rather than pull-

based. To aggregate the results from the sources we also need to tag messages with source and sequence number. Akka provides support for the aggregator pattern that allows `match` patterns to be dynamically added to and removed from an actor from inside of the message processing logic. However, this implementation does not allow matching the sender (*Source*) of the message during aggregation which is a key part of the join pattern.

Figure 2.5 (Right) shows how the computation differs in the Selector Model, using this approach we need to make sure that the senders send their messages to the correct mailbox of the aggregator(*Adder selector*). We can achieve this in two ways: *a*) **Any order:** wrapping the send logic in the selector to forward messages from sources to a specific mailbox (label: sequence number) in the aggregator or *b*) **Round robin order:** configuring (initialization) the sources with different mailbox names so that the sources send only to specific mailboxes.

– For the first approach ("any order", see Figure 2.6, lines 1-19), ordering is not preserved when sending data from the sources to the aggregator's corresponding mailbox. As items are received the corresponding mailbox is disabled, and the pool of active mailboxes decrease [line 10]. When items from all sources have been received for the current sequence number, the result is computed and pushed to the consumer network, and all the mailboxes are enabled for the next sequence number [lines 12 -18].

```
1  // process items in any order
2  public class AdderAnyOrder(...) extends DistributedSelector {
3    int[] items = new int[numSrcs];
4    int srcMatched = 0
5    public void process(MessageType theMsg) {
6      if(theMsg instanceof ItemMesssage){
7        ItemMessage im = new ItemMessage(theMsg);
8          items[im.sourceId) = im.intValue();
9          // disable the current mailbox
10         this.mailbox.disable(im.sourceId)
11         srcMatched += 1;
12         if (srcMatched == numSrcs) {
13           SomeValue joinResult = computeJoin(items);
14           nextInChain.send(joinResult);
15           // reset locals
16           items = new int[numSrcs]; srcMatched = 0;
17           // enable all mailboxes for next seq
18            this.mailbox.enableAll();
19       } }}}
```

Figure 2.6 : Using Selectors to solve the Join Pattern problem of Figure 2.5. The aggregator selector version: `Adder Any Order` maintains one mailbox for each source. For simplicity, we assume sources are identified by consecutive integers starting at 0.

– For the second approach ("round robin", see Figure 2.7, lines 1-22), the aggregator selector disables all mailboxes except the first one, which corresponds to messages from the first source [line 7]. As each message is received by a mailbox, that mailbox is disabled, and next mailbox is enabled in a round-robin fashion [line 13]. When one message from each of the sources has reached the aggregator, the join operation is commenced and forwarded to the next consumer in the network [lines 15-19]. The first

mailbox, corresponding to the first source is then enabled to process items for the next sequence number.

```
1  // process items in round-robin order
2  public class AdderRoundRobinOrder(...)
3  extends DistributedSelector{
4    int[] items = new int[numSrcs];
5    int srcMatched = 0;
6    // expect item from first source
7    this.mailbox.disableAllExcept(0);
8    public void process(theMsg: AnyRef) {
9      if(theMsg instanceof ItemMessage){
10       ItemMessage im = new ItemMessage(theMsg);
11         items(im.sourceId) = im.intValue();
12         // disable the current mailbox
13          this.mailbox.disable(im.sourceId);
14         srcMatched += 1;
15         if (srcMatched == numSrcs) {
16             SomeValue joinResult = computeJoin(items);
17             nextInChain.send(joinResult);
18             // reset locals
19             items = new int[numSrcs]; srcMatched = 0;
20         }
21         //enable round-robin mailbox for next seq
22                 this.mailbox.get(srcMatched).enable();
23     } } }
```

Figure 2.7 : Using Selectors to solve the Join Pattern problem of Figure 2.5. The aggregator selector version: `Adder Round Robin` maintains one mailbox for each source. For simplicity, we assume sources are identified by consecutive integers starting at 0.

2.3 Distributed configuration in Akka and SALSA

Akka is an open-source toolkit and runtime for building highly concurrent, fault-tolerant and distributed systems on the JVM, based on the *Actor Model* [17]. Akka implements a unique hybrid model of the aforementioned charecteristics [24]:

- Use of actors provide a simple and high-level abstractions for distribution, concurrency, and parallelism. Actors are very lightweight event-driven processes, asynchronous, non-blocking and highly performant message-driven programming model.
- Akka achieves fault-tolerance by using supervisor hierarchies with "let-it-crash" semantics. Akka actors can span over multiple JVMs to provide truly fault-tolerant systems, effective for writing highly fault-tolerant systems that self-heal and never stop.
- Location transparency is achieved since everything in Akka is designed to work in a distributed environment and all interactions of actors use pure asynchronous message passing.
- Akka allows actors to recover their state even after a JVM crashes, or when being migrated to another node.

Setting up the Akka system. `Akka.cluster` is a module dedicated to aiding actor-based distributed application programming and provides a fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure/ bottleneck [17]. Akka achieves this using the gossip protocol [25] and their failure detector [26]. A node in the Akka system is a logical member of a cluster, there could

be multiple nodes on a physical machine with unique `hostname:port:uid` tuple. A *cluster* is a set of nodes joined through their membership service.

Akka applications can be distributed over a cluster with each node hosting some part of the application. For a node to join a cluster, one must initiate the process by issuing a `join` command to one of the nodes in the cluster to join. Akka uses the UID to be able to reliably trigger remote death watch. Death watch is a feature provided by Akka to register an actor for reception of the Terminated message dispatched by the other actor upon termination [27]. The registered actor will only notify the parent actor when it terminates (i.e. stops permanently, not a temporary failure and restart). Since Akka uses the UIDs the same actor can never join a cluster again once it's been removed, the system must be stopped and started again with a new UID for that actor to join the cluster.

Managing the Akka system. Akka based its cluster membership management on Amazon's *Dynamo* system [28], in which membership is communicated to other nodes in the cluster using the well know Gossip Protocol [29]. The information about the cluster converges locally at a node at certain points in time, using the Gossip convergence protocol where a node can prove that the cluster state it is observing has been observed by all other nodes in the cluster. The convergence cannot be reached when a node is unreachable, this will delay the convergence protocol, but Akka claims that the computation running on the cluster will not be affected [17]. After the convergence, a leader for the cluster is determined, based on the first node in sorted order of the UID. The leader is used to to shift members in and out of the cluster by changing `joining` members to the `up` state and `exiting` members to the `removed` state. The leader may "auto-down" a node if the failure detector considers a node to be *unreachable* (Note: For a leader to be selected, convergence from the

gossip protocol is needed).

Akka uses a failure detector model to detect failure when a node is unreachable. Specifically, they use an Accrual Failure Detector [30] that keeps a history of failure statistics for heartbeats from different nodes and generate a φ value representing the likelihood that the node is down. Heartbeats are sent out every second in a request-reply handshake with the replies which is used as an input to the failure detector. When all nodes, monitoring the unreachable node detects it as reachable, the detector will declare the node as *reachable*.

Termination of the Akka system. All actors in the Akka system are created in a hierarchical fashion (parent-child relation). A System guardian actor in the Akka system (created during initialization of the Akka System), is responsible for an orderly shut-down sequence where logging remains active while all normal actors terminate. Upon receiving the `ActorSystem.terminate` message the System guardian actor initiates its own shut-down, but before the guardian can initiate shutdown all actors in the system must be terminated. Actor termination is done in two steps: first, the actor suspends its mailbox processing and sends a `stop` command to all its children. Using the Death-watch protocol [27], the actor keeps processing termination messages from all its children. If any of the child actors don't respond to the `stop` message, the entire termination process halts. This procedure ensures that actor system sub-trees terminate in an orderly fashion, and hierarchically move up to the supervisor. Actors in a system can also be terminated using either the *Poison Pill* message, which is processed like a regular message [31] and stops the actor after the Pill message is processed or use the *graceful termination* method [32], if one needs to wait for the termination based on some computation or termination of another set of actors.

SALSA (Simple Actor Language, System and Architecture) is a Java-based actor programming language developed at RPI [18]. The language targets open, dynamically re-configurable Internet and mobile applications. SALSA provides active objects, asynchronous message passing, universal naming, migration, and advanced coordination constructs for concurrency as a part of their abstraction module. A SALSA program consists of *universal actors* that can be migrated around distributed nodes at runtime. Distributed SALSA programming involves universal naming, theaters, service actors, migration, and concurrency control [33]. The distributed language supports the World-Wide Computing architecture (harnesses underutilized resources in the Internet by providing various Internet users a unified interface that allows them to distribute their computation with location transparency and its also platform agnostic) [34].

The World-Wide Computing(WWC) architecture implementation of SALSA consists of the following key components:

- **Universal naming service.** which allows actors to become *universal actors*, and can migrate to other nodes during runtime. Every universal actor has a Universal Actor Name (UAN) and Locator (UAL). UAN is unique throughout the lifetime of the system, and the UAL provides the location of the node on which the actor currently resides.

- **Run-time environment.** defines and executes the theaters (programming construct of SALSA) and the naming service (protocol that defines how to interact with the WWC services)

- **Migration support.** migrates universal actors to desired location (UAL) set

by the user. SALSA migration also supports, multiple migration of a single actor if a set of UAL's are passed to the `migrate<ual>`.

Setting up the SALSA system. Distributed computing in SALSA is achieved using the concept of *theaters*, which serves as a separate process in which multiple actors perform on a set of stages [35]. *Stages* are heavyweight actors that simulate the execution of many concurrent lightweight actors in parallel. Each stage consists its own mailbox and thread control to handle all the lightweight actors running on that stage. SALSA actors are implemented as Java objects containing their state as an object field and a reference to the stage that the actor is performing on. One must note, that since multiple lightweight actors are hosted on the same stage, if any of the messages has an unbounded processing time, all actors on that stage will starve. The authors of SALSA claim that one of the solutions to avoid this problem is to host each actor on its own stage [35]. The distributed runtime system of SALSA consists of background daemon servers that host each stage, and these must be started and terminated by the user before and after the computation is completed.

Managing the SALSA system. The distributed implementation of SALSA-Lite introduces a new type of actors, mobile actors [33]. Mobile actors are defined as two objects (reference and state). Defining mobile actors as a single object would generate significant challenges since references to the migrating actor must be modified at each actor holding a reference to the migrating actor. The state of a mobile actor is stored in a `MobileActorStateRegistry` which is the only object with a reference to the actorâĂŹs state. Any invocation on the reference will trigger a lookup in the `MobileActorStateRegistry` which invokes the message on the state if the actor is present. On migration of any mobile actor the state object is put in a message to the `OutgoingTheaterConnection` it is being sent over,

and the `MobileActorStateRegistry` for that actor is replaced with a reference to the `OutgoingTheaterConnection` actor that sends messages to the theater the actor migrated to.

Termination of the SALSA system. The distributed implementation of SALSA Lite provides no support for the proper termination of the runtime. The user is responsible for terminating all the actor instances at a stage, and once the stage is no longer hosting any active actors the user may terminate the stages and the theaters they are running on.

2.4 Distributed Communication in Clusters and Mobile Platforms

Communication in Cluster Platform. All communications between the user-defined selectors are done using the Proxy Actor. To communicate between any two selectors, we must use a Selector-Handle as the access point. These Selector-Handles are lightweight and can easily be sent across the network. It contains a globally unique identifier or the selector object and method handle 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 needs an identifier that can encode both scenarios. The identifier is constructed upon a request for selector creation and is unique across the entire distributed system, and stays alive throughout the execution time of the application.

A selector object's, globally unique identifier is currently a 32-bit integer that encodes three pieces of information:

- an 8-bit value encoding the *place* p on which the selector is created;

- an 8-bit value encoding the *place* q on which the selector instance resides; and
- a 16-bit integer value representing a unique identifier for the selector on p.

For serialization we use the Kryo serialization framework [36]. As a Java–oriented framework, it is better suited for the Distributed Selector model than other high-performance serialization tools such as Google's Protocol Buffers [37], Apache Avro [38], or Apache Thrift [39], which works across multiple languages and platforms and have more restrictions on the data that can be sent [40].

The Proxy Actor acts as a routing point between local and remote selectors. When a message is sent to a non-local selector, the SelectorHandle's destination ID is decoded to extract the destination of the message:

- If the destination *place* matches the local *place* then the Proxy Actor looks up its local registry to find the selector and forwards the message.
- If local registry doesn't have an entry yet, then the Proxy Actor buffers any message to the non-existing selector.
- If the destination *place* is remote then the Proxy Actor forwards the message to that specified *place*.

When a user-defined selector sends a special message, there are two ways a new selector is created depending on its location:

- If the new selector to be created is at the same *place* (local) then we use *java reflections* to create a new selector at the current *place*; the Proxy Actor is not involved in the process.

- If the new selector to be created is at a different *place*, the Proxy Actor sends over the selector parameters to the destination *place* to create it locally at that *place*.

Communication in Android Platform. The user-defined selectors can only communicate with each other and the runtime system of the Android device(*place*) that hosts the selectors. Communications to other handheld devices (*places*) must be managed by the Mobile Communication manager.

A fully decoupled interface, `IMobileCommunicationManager` uses a runtime - userspace callback system based on the actor model to communicate when any change in network status is observed.

When a new selector system instance is created in the network, the selector system will call `start()` to initiate the communication manager, and the communication manager will call `ISystemCallback.onConnectionReady` once the device is ready to join a network. When a connected device leaves the network, the communication manager invokes `ISystemCallback.onPlaceLeft()` to notify the selector system, which in turn will alert the application of a network change via a system message.

New devices joining the network does not affect the selector system until an Actor message is exchanged. The `onPlaceJoined` callback only gets invoked if the application desires to be notified of such topological changes. On the other hand, a device leaving the network can cause a known remote selector reference to be invalid and is immediately reported to the application through the callback system. *non-blocking* data transmission in a selector system instance is guaranteed by forcing the communication manager to run on separate threads.

For any given message to a selector B from selector A, where A and B both reside on the same physical device, the message is passed by reference as a regular object

without performing a communication operation. For any given message to a selector D from selector C, where C and D reside on different physical devices (places), the message must be passed through the Mobile Communication Manager. The selector system of D's residence will decode the Global Unique Identifier of selector C to obtain its residence place ID, and invoke the `send` method on its communication manager to deliver the message. Upon receiving a message at the communication manager on C's residence, the communication manager will deliver the message to selector D.

Chapter 3

Cluster Platform

3.1 HJDS Design

Figure 3.1 shows the decomposition of the Habanero Java Distributed Selector (HJDS) system. We refer to each process in our runtime as a *place*. For our Java-based runtime, a place corresponds to a single instance of Java Virtual Machine (JVM). It is possible (an in some cases may be desirable) to deploy multiple places on a node. Unless otherwise specified, our default assumption will be the common case of deploying exactly one place per node, and we will use "place" and "node" interchangeably in those cases. A selector runtime system instance on one place consists of multiple user- defined selectors and two service actors: the System Actor and the Proxy Actor.

Section 3.2 explains how these *actors* are initialized and used for bootstrapping the system. The user can choose to denote a specific *place* in the configuration file as the *Master Node*. In a situation where a *Master Node* is not defined in the configuration file, our runtime selects the first node in the file as the Master.

In this diagram, the *distributed selector system* refers to the system on which an HJDS program is executed. The system is composed of several components:

- **Place.** A *place* refers to an individual Habanero Java(HJ) runtime instance. This concept is inherited from the *place* abstraction in HJlib to describe affinity among selector objects [4]. The *place* constructs in HJ provides a way for the programmer to specify affinity among the asynchronous tasks. Data locality

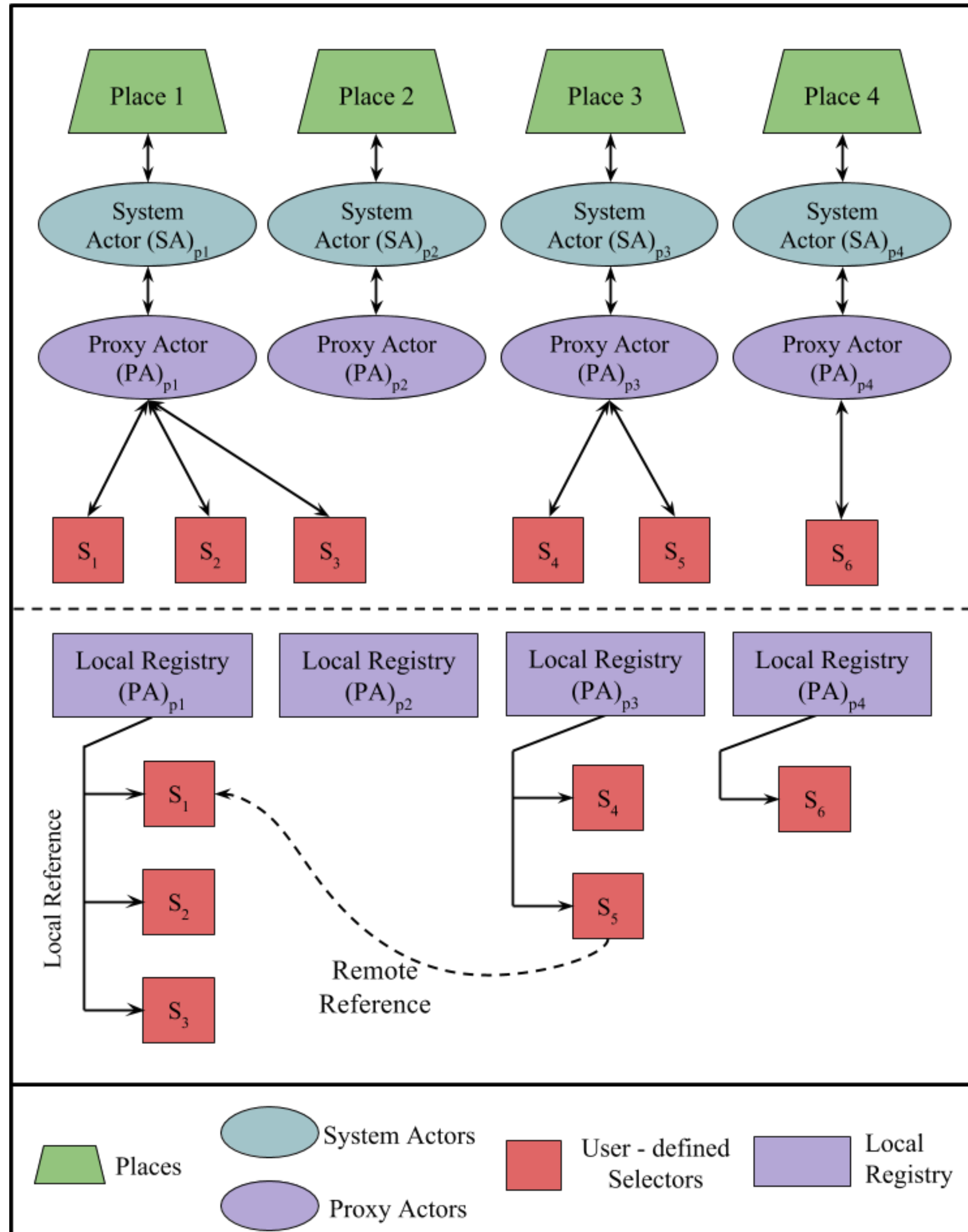


Figure 3.1 : The Distributed Selector System

can be controlled by assigning selectors with the same data affinity to execute in the same place. The management of individual worker threads within a *place* is not visible to the user. When a physical computing node contains multiple

places, each *place* needs a distinct port for message passing (See, Config File in Figure 3.3). The term *selector system* refers to the HJ runtime instance on a single *place*.

- **System Actor.** The System Actor is a service actor that maintains the internal state of its Selector System, also communicates such information to the rest of the distributed system. The System Actor on the *Master Node* maintains the internal state of the entire HJDS system. During the automatic bootstrap process (See Section 3.2 for details) the System Actor plays a critical role in setting up the system and updating all the other System Actors in the network with the relevant information about the network settings (location of each Proxy Actor with respect to their *places* and which *places* are live) The System Actor is also responsible for managing the system termination process for the *Global Finish Scope*(See, Section 3.3 for details). The System Actor on the *Master Node* initiates the termination process for the entire Selector System and is the final actor to be terminated before the application gracefully exits.

- **Proxy Actor.** The Proxy Actor is responsible for coordinating the messages between local and remote selectors, including remote selector creation and termination requests, as well as remote message passing. The end-user view of the Selector System includes instances of local user-defined selectors throughout the program execution as in [3]. Each Proxy Actor maintains a *local registry* of all the selectors that are located in the same *place* for easy communication between local selectors. As shown in Figure 3.1 the *local registry* can also contain remote references (S5 has a remote reference to S1). Remote references can only be obtained by a selector if, *a*) the selector holding the reference (S5 in

this case), created the selector (S1) at a remote location (*place 1*), or, *b*) any other selector in the system provided the selector(S5) with a remote handle of the selector (S1). Any message to a non-local selector will be sent to the Proxy Actor of the remote selector (whose location is encoded in the selector handle), and the Proxy Actor forwards the message directly to local selector instances(see Figure 3.2). Migration of selectors and dynamically updating the selector handle is a potential area for future research.

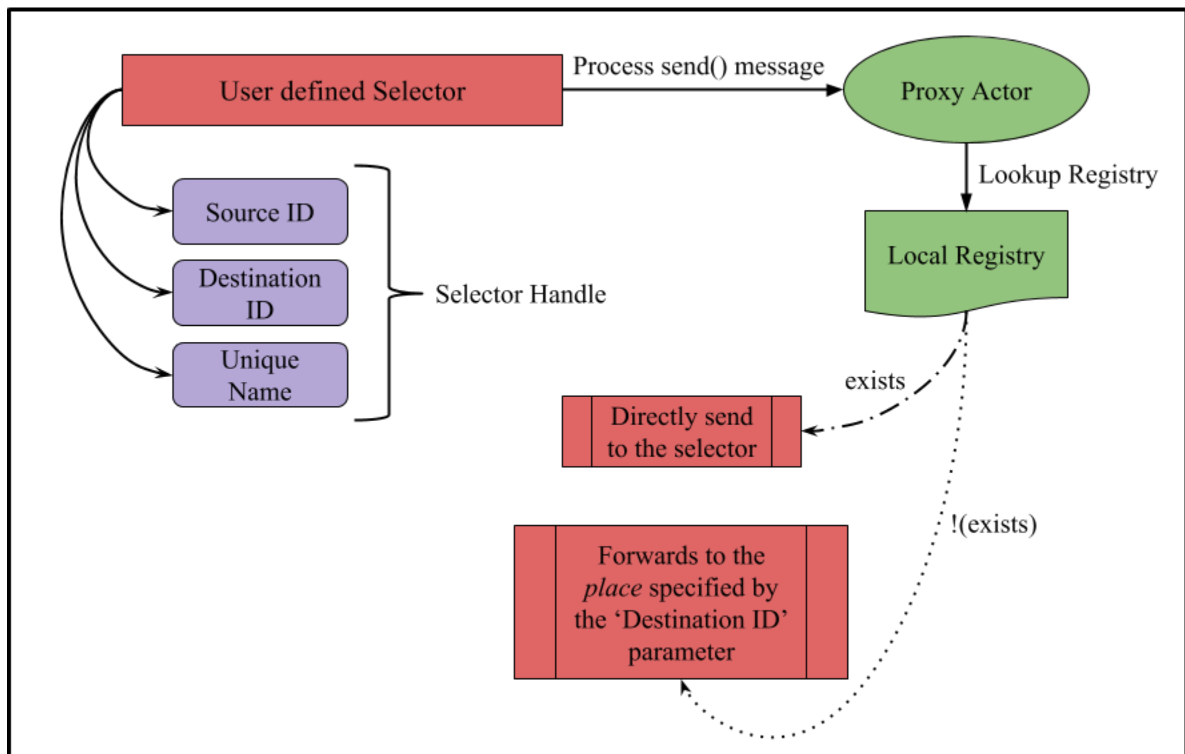


Figure 3.2 : Proxy Actor forwarding messages to local selectors and remote selectors at each *place*. The underlying message forwarding protocol is transparent to the application programmer.

- **User Defined Selectors.** In the end-user view of the HJDS system, a *user-defined selector* refers to any Distributed Selector object created in user code.

These selectors are responsible for any computation in the user's code and are not a part of our runtime. The user has full control over the initialization and termination of the selectors using the `selectorName.start()` and `selectorName.stop()` constructs respectively. The `start()` and `stop()` constructs can only be called by a selector on itself. These primitive constructs are provided by the shared memory selector model [3]. The user-defined selectors created at each *place* are recorded in the *local registry* of the **Proxy Actor**, and upon creation of new selectors the *registry* gets updated with the selector *handle* of the new selector. In our current implementation, the unique *handle* for a selector object consists of: *a*) an 8-bit value encoding the *place* p on which the selector is created (Source ID); *b*) an 8-bit value encoding the *place* q on which the selector instance resides (Destination ID); and *c*) a 16-bit integer value representing a unique identifier for the selector on p (USID).

- **Master Node.** In our HJDS system, *Master Node* refers to the *place* that controls system bootstrap and the global termination sequence. The user designates one of the available computing nodes to be the Master Node. It is responsible for managing the state of the entire Distributed Selector program, including the initiation and termination of the distributed system.

- **Global Finish Scope.** The system defines its *Global Finish Scope* as the enclosing join operation around the user program. Derived from the `finish` construct in AFM [4, 41], the single *Global Finish Scope* for the entire DS program waits for all tasks created by user code to finish and then automatically terminates the distributed system. The process for system termination is explained in detail in Section 3.3.

The HJ runtime fully encapsulates the bootstrap and termination of the entire distributed system for the user program, thereby requiring minimal user involvement to enable distributed execution of a selectors program. In fact, the same application code can be executed in local mode or distributed mode simply by modifying the configuration file.

3.2 Automatic Bootstrap

The design of our HJDS model is based on the Habanero Java Runtime Library (HJlib) [4]. We extended the shared-memory implementation of the Selector Model to support remote message passing, remote selector creation and bounded global termination in a transparent manner.

To set up the HJDS system, users provide a configuration file (see Figure 3.3) in which the IP addresses (or hostnames) and ports for all computing nodes are specified. If two *places* are assigned the same node, the system will run on multiple JVM instances (using different ports) on the same node (see port entries for place p1 and p2, lines 11 and 16). The `init` keyword specifies the bootstrap master node, while the `remote` keyword indicates other predefined *places* in the bootstrap. The bootstrap master node will be responsible for initiating the bootstrap process and will stay live until the entire application has successfully completed. Our runtime reads information from the configuration file and boots up the system. The current implementation requires the user to set the username and password for remote hosts as environment variables. The runtime can be extended to support more configuration parameters, like memory usage limit and the thread count, in the bootstrap config files.

Each application using our HJDS framework is considered as a single distributed

```
1  selectorSystem {
2  init {
3      place : p0,
4      hostname:cn16.davinci.rice.edu,
5      port: 5000,
6   }
7   remote : [
8     {
9        place : p1,
10       hostname:cn20.davinci.rice.edu,
11       port: 5000,
12    },
13    {
14       place : p2,
15       hostname:cn20.davinci.rice.edu,
16       port: 5002,
17    }
18   ]
19 }
```

Figure 3.3 : Sample configuration file, nodes are on Rice University's DAVinCI cluster.

system by the Habanero Java runtime. The usual practice of launching distributed application is to run server daemons on each computing node and host user program. Each HJDS application may be modeled as a single service and seamlessly integrated into a larger system. Using our approach the application developer doesn't need to worry about whether the application will be running as a distributed system or a shared-memory model.

In a single HJDS program, each computing node is set up with a configuration file (Figure 3.3) and SSH access for the initial master node. The programmer can adjust the number of places used in the program by modifying the configuration

files, the dynamic addition of a place is currently supported in our Android extension (DAMMP) of this model. For a detailed explanation on dynamic joining of places (Android devices) see Section 4.2, in Chapter 4. On the Master Node, as specified in the configuration file, the HJ runtime will stage the program executable on all *remote* places and start up a process on each to initiate the application.

Figure 3.4 demonstrates the bootstrap process in a distributed selector program. The Master Node (Place 1 in this example), waits for all other places in the configuration file to be ready before any user application code can be processed. Upon

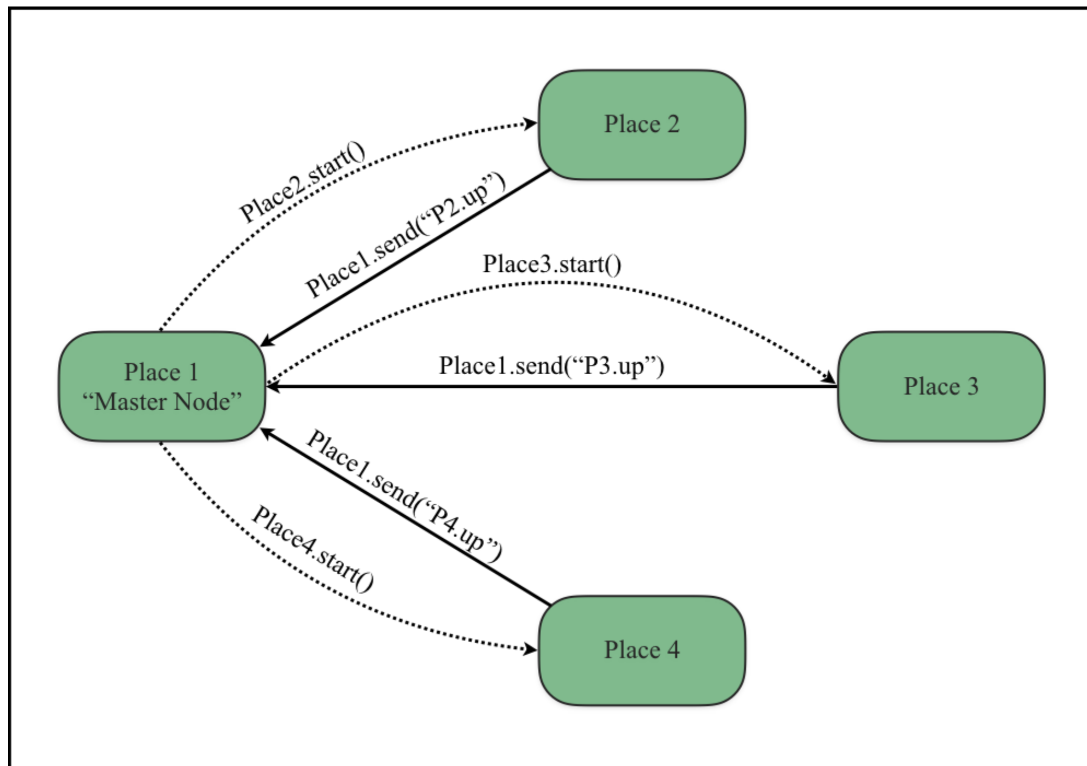


Figure 3.4 : Bootstrap process of the Distributed Selector program.

receiving messages labeled `'PlaceNumber.up'` from all places in the network, the System Actor logs into each remote location (Place 2, 3 and 4 in our case) through SSH, and starts up an HJ selector system on that *place*. The execution of any appli-

cation code will remain in a stalled state until all remote *places* are up and running a HJ selector system.

Upon successful bootstrap, the System Actor on the Master Node obtains information for all other *places* in the system from the configuration file. The system actors on each *place* will identify the Master Node from the configuration file. When initialized successfully, these system actors send periodic heartbeat messages to the Master Node to indicate their state. The Master Node collects ready messages for all known remote *places* and informs each *place's* proxy actor to start program execution. Optionally, users can choose to manually start up each *place* when it is correctly set up in accordance with the HJDS system requirement. This choice does not affect the automated global termination of the system.

3.3 Global Termination

An HJDS program waits for all the tasks and the selectors that the user code created to finish and then terminates the distributed system. The Master Node initiates the global termination process and is performed in three stages. The System Actor at each *place* is responsible for the graceful termination of that *place* and the System Actor at the master *place* guarantees the successful termination of the entire HJDS system. Each stage is discussed below in detail:

- **Stage 1: Initiate Termination.** A System Actor detects its *place* to be *idle* if its local user-defined selectors have all been terminated by the user, and no pending selector creation is in the system. The System Actor communicates such information to the Master Node as a periodic heartbeat, which will be reset by any new incoming request. When Master Node collects idle state from all *places* in the system, it attempts to **initiate the termination process** and

moves to *Stage 2*. During this Stage, every node is either *idle* or *active*. Figure 3.5 shows the stage 1 of the termination process where the master node is collecting state information from the remote nodes.

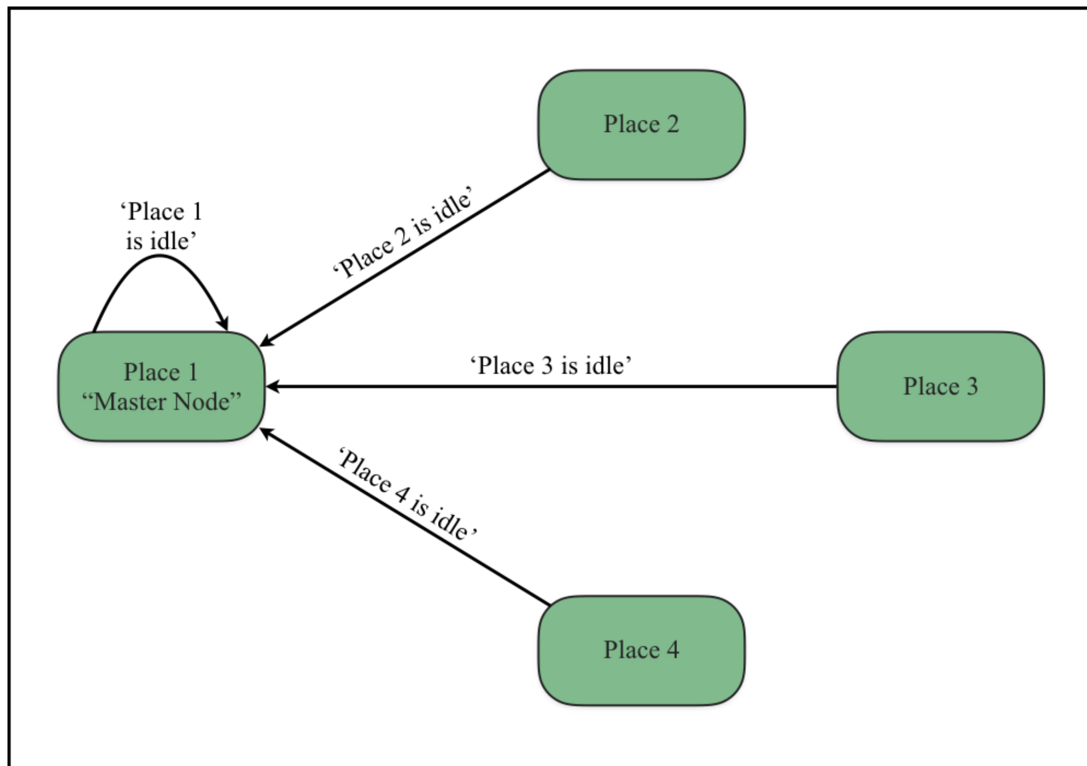


Figure 3.5 : Stage 1: Termination process initiated when each of the remote nodes and the master node is in idle state.

For a *place* to be considered idle (See Figure 3.6):

– all user-defined selectors on a place are terminated by the application programmer,

– there are no more buffered messages * for selectors that have not been created yet, and,

*During remote creation of selectors, if messages are sent to the remote selector before the selector

– there are no more messages to be processed in any of the user-defined selectors i.e., empty mailboxes.

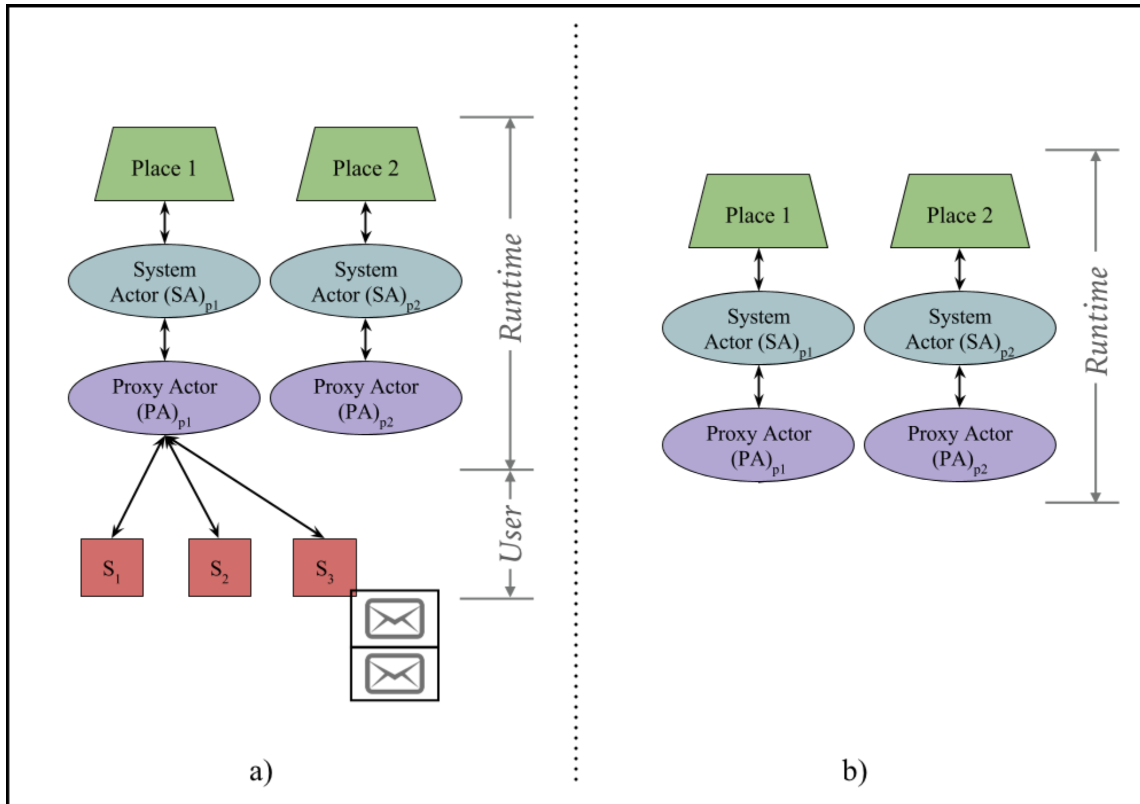


Figure 3.6 : Decomposition of the runtime and user-level view during the termination process. a) Before initiating the termination process(Stage 1) b) Places 1 and 2 are considered as 'idle' in this phase

Figure 3.6 shows the runtime and user-level view of our distributed system when the termination process is initiated. The image on the right depicts the

---

has been initialized on the remote *place* or ready to process any incoming message, the proxy actor at that *place* will buffer those messages. Note: There cannot be any selector creation messages in flight when a *place* is idle

scenario where both places (Place 1 and Place 2 in our case) can be considered as idle.

- **Stage 2: Ready To Terminate.** The System Actor at the Master Node passes a `signal` to prepare termination to any *place*, along with an order in which the `signal` should be passed around. Figure 3.7 shows how the signal sequence is implemented as a conceptual place-ring with the Master Node at the end of the sequence, but other arrangements can be used. The signaled *place* confirms its idle state by passing the signal down the ring. A signal with confirmation from all *places* to the Master Node will trigger *Stage 3*.

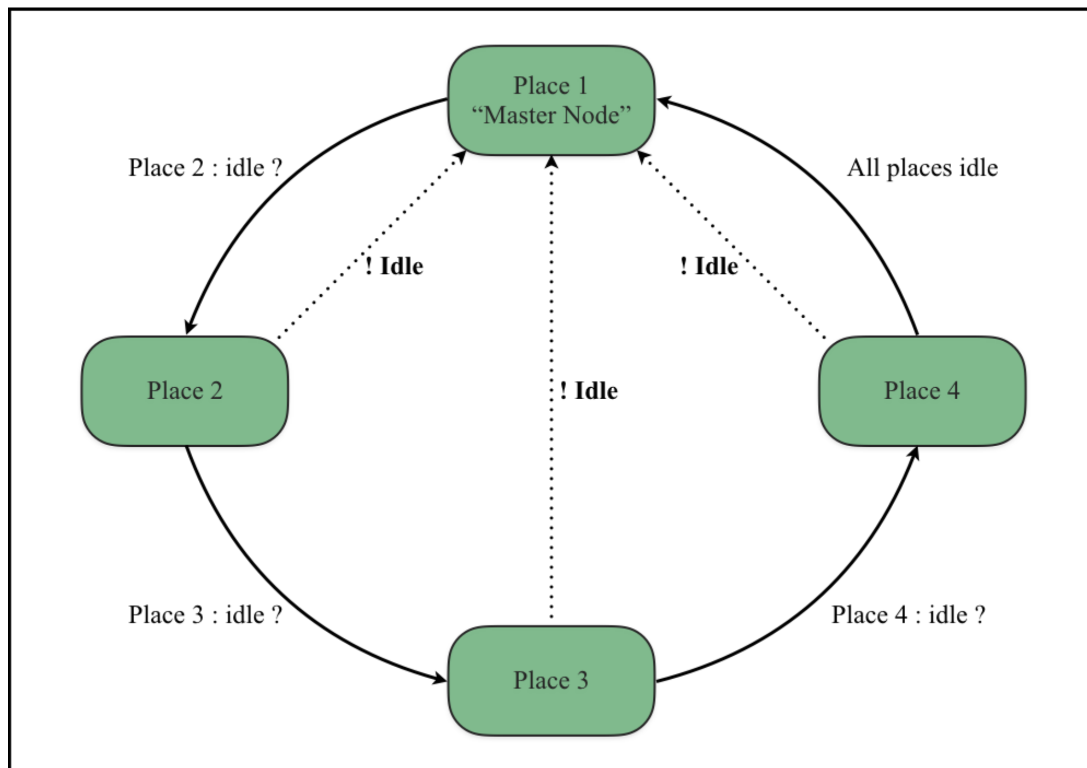


Figure 3.7 : Stage 2: Termination signal passed around the network in a conceptual place-ring fashion with the Master Place at the end of the sequence.

During this stage, a signaled *place* can short-circuit the ring and declare its active state directly to the Master Node to return the distributed selector system to the active state thus canceling the global termination process. Only if none of the *places* short-circuit the ring, and the Stage 3 signal completes the round trip around the ring, the global termination process moves on to the next stage. During this stage, every node is either *idle*, *active*, or in *Stage 2*. Stage 2 will capture any in-flight messages in the network as well since all Proxy Actors sending messages to remote *places* for new selector creation must wait for an acknowledgment from the remote selector.

- **Stage 3. Terminate All Nodes.** In the final stage of the global termination process, the Master Node knows all places are ready to exit and sends a termination signal down the place-ring (see Figure 3.8). Each *place* shuts down after forwarding a final *shutdown* message to the next *place* in the ring. The system gracefully exits by terminating each *place* and when the termination signal finally the reaches the master node, the System Actor at the master node terminates itself (graceful termination of the HJDS system). During this stage, every *place* is either in *Stage 2*, or it has been shut down.

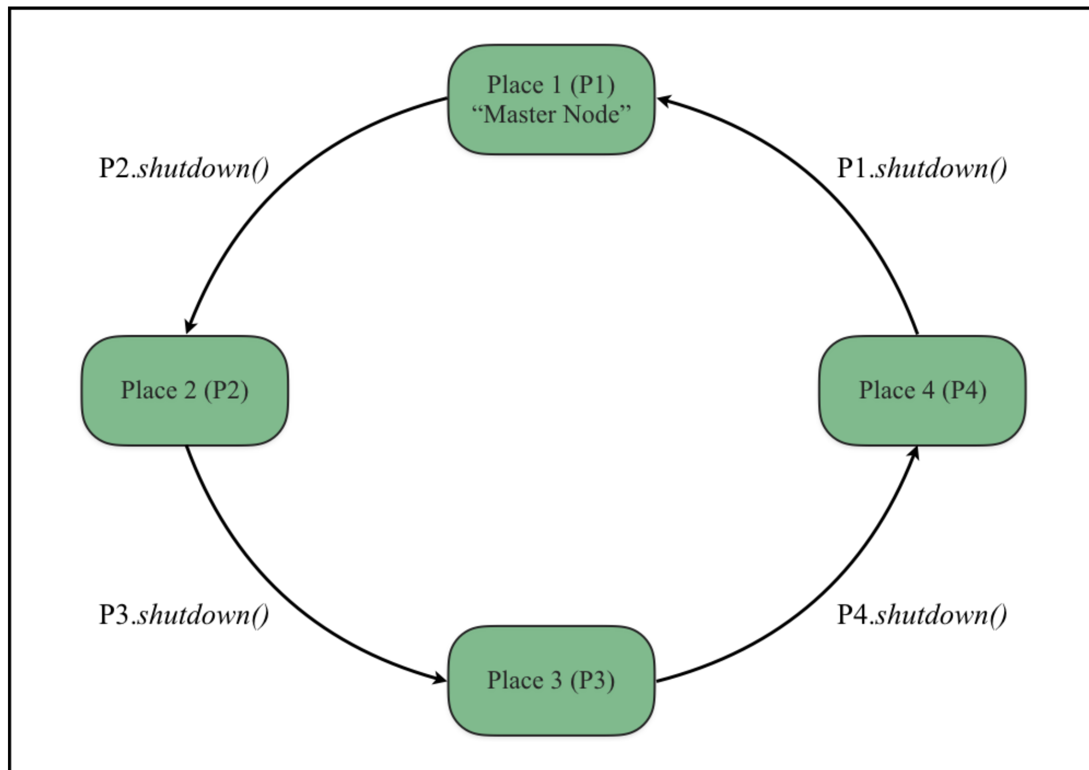


Figure 3.8 : Stage 3: Terminate all nodes. Send termination signal around the network in a conceptual place-ring fashion.

Chapter 4

Android Platform

4.1 DAMMP Design

The Distributed Actor Model for Mobile Platforms (DAMMP) extends the HJDS execution model and cluster-based implementation (as described in Chapter 3) to mobile platforms. There are several challenges that had to be addressed in this extension including dealing with unpredictable periods of unavailability for different devices, and ensuring that a reliable runtime system can be implemented across mobile devices. Our mobile implementation supports the lightweight location-transparent actor creation and actor-based runtime control mechanisms described in Chapter 3 and [15], with the extensions to mobile devices we limit a single physical device to host exactly one *place* [*]. Our approach extends HJDS's non-resilient high-performance runtime to obtain more decentralized mobile platforms, through the following contributions:

- To support the distributed actor runtime on *volatile* mobile networks, we have replaced the automated bootstrap and global termination feature in HJDS that assume stable network connectivity by a new communication manager that is more suitable for mobile computing,
- We introduce dynamic joining (Section 4.2) and leaving (Section 4.3) of these mobile devices, and, finally,

[*] As defined in Chapter 3, a place is the logical unit for a shared-memory runtime instance that can host multiple actors.

- We can optionally expose and delegate some of the runtime control (e.g.: offloading computation based on memory usage of an application, handling computation when a device joins / leaves the network of mobile devices) to the application level.

The cluster-based implementation of HJDS assumes network stability and low communication costs. However, those assumptions usually do not hold on volatile mobile platforms such as the one we considered in this work. Over-the-air data transmission is a far more expensive than that of wired connections, and the volatile nature of mobile networks due to devices leaving, joining, dropping out, going out of range or running out of power makes the automated bootstrap and global termination features in HJDS to be of limited use for the mobile platforms that we are considering. Also, automated bootstrap and global termination require a monolithic application model that limits the expressibility of the Actor model in many peer-to-peer applications. Instead, our mobile-oriented approach encourages a decentralized programming pattern, that we further enhance by the delegation of limited runtime control to applications through a publish-subscribe model on single devices.

An overview of a single *place* (device) in the selector system is shown in Figure 4.1. One may consider each device to be a local selector system. A local selector system in a single *place* (or device) consists of:

- a *runtime system* with a System Actor, a Proxy Actor, and Mobile Communication manager, and,
- a *developer view* with one or more user-defined selectors.

As shown in Figure 4.1, the user-defined selectors can only communicate with each other and the runtime system of the *place* that hosts the selectors. All communica-

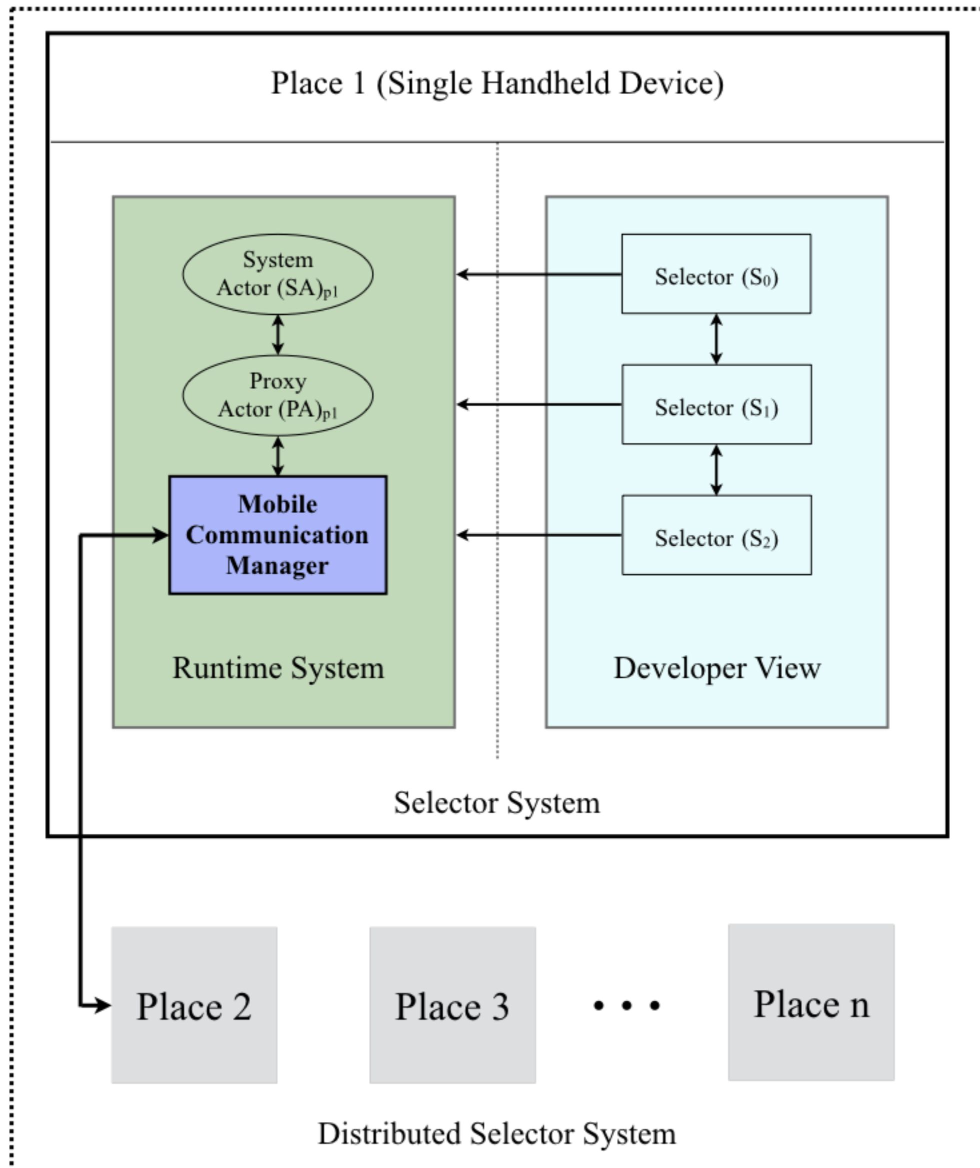


Figure 4.1 : Overview of the distributed selector system. On each hand-held device, we have a single selector system with the Mobile Communication Manager of each device exposed to the entire distributed network. All communications to external handheld devices are performed by the Mobile Communication Manager in the runtime.

tions to other handheld devices (*places*) must be managed by the Mobile Communication manager.

The Mobile Communication Manager is modeled as a runtime-userspace callback system based on the actor model as shown in Listing 4.1. The communication manager includes a system callback handle that is invoked after any change in the network status. Upon creation of any new selector in the DAMMP system, the selector system will call `start()` to initiate the communication manager. Once the device is ready to join the network, the communication manager calls the `ISystemCallback.onConnectionReady()` method.

```
1      public interface IMobileCommunicationManager {
2          interface ISystemCallback {
3              void onConnectionReady(Place localNode);

5              void onMessage(Message message);

7              void onPlaceJoin(Place place);

9              void onPlaceLeft(Place place);
10         }
11         void start();
12         void stop();
13         boolean send(Place place, Message message);
14         void setSystemCallback(ISystemCallback callback);
15 }
```

Listing 4.1: The communication API for mobile platform.

For any given message to a selector B from selector A, where A and B both reside on the same physical device, the message is passed by reference as a regular object without performing a communication operation. For any given message to a selector D from selector C, where C and D reside on different physical devices (*places*), the

message must be passed through the communication manager[†]. The selector system of C's residence will decode the Global Unique Identifier (GUID) of selector D, to obtain its residence *place* ID, and invoke the `send` method on its communication manager to deliver the message. Upon receiving a message at the communication manager on D's residence, the communication manager invokes a callback upon message receive and the selector system will deliver the message to selector D.

4.2 Dynamic Joining

Past implementations of distributed actors and distributed selectors [15, 17, 18] targeted cloud-like distributed servers, and did not account for the fact that devices can frequently join and leave a mobile network. Due to this volatile nature of mobile platforms, we have decided to extend our distributed implementation to (optionally) allow user-level control of adaptation to network changes to application-level actors.

As shown in Listing 4.2, applications can subscribe to specific message types (including runtime-generated alerts) through designated mailboxes in a selector. Using the publish-subscribe model one can be notified of any android runtime supported events (e.g.: Battery Usage, Network bandwidth), and based on the notification, each event can be handled differently. Since the Selector model is a asynchronous message passing model, these events are processed as messages, when a selector subscribes to a specific topic mailbox. On subscribing to these topics, the mobile devices can be notified about android system events about itself and all the mobile devices that are connected in the ad-hoc network with Wi-Fi Direct. Using other methods such as broadcasting all system messages to all mobile devices connected in the network may cause an added overhead, or redundant information on devices that need not

[†]See Section 2.4 for details on the Android communication manager

handle such messages. But with our puplish-subscription model, and with annotating classes, only devices that want to be notified of certian topics will receive such runtime information.

```
1 @Subscription(topics = {
2     @Topic(messageClass = NodeJoined.class , mailbox = 0),
3     @Topic(messageClass = NodeLeft.class , mailbox = 0)
4     })
```

Listing 4.2: A selector class can subscribe to different alerts from runtime.

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.

As an example, Figure 4.2 shows a dynamic join protocol implemented in on of our DAMMP applications. When a new device wishes to connect and join the network of devices, it first connects to the Group Owner (GO). Upon successful connection with the GO, the GO sends information about all the other devices that are already connected to the network. The new device can then connect to the other devices in the network as needed. The Wi-Fi direct layer is used to establish any new connections before any computation is processed using our selector model.

As an example of adaptation, the application can designate different priority values to different system alerts, choose to give higher priorities to topological changes for handling partial failures, and thereby avoid waiting for work-related messages to complete before topological changes are handled. Messages with higher priority values are then sent to higher priority mailbox, and using the selector models processing logic, messages in the higher priority mailbox get executed first before it looks for messages in the lower prioirty mailboxes. The subscription mechanism also al-

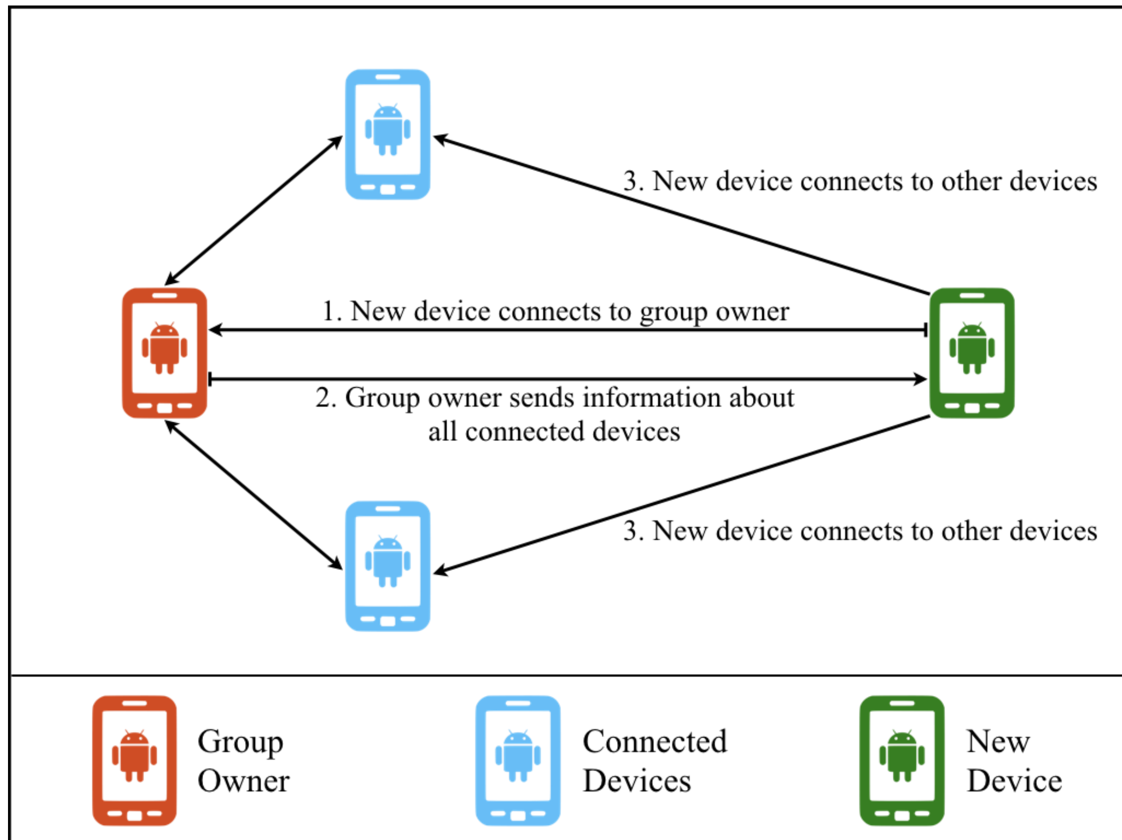


Figure 4.2 : Dynamic join protocol for new handheld devices.

lows the communication manager to communicate directly with the application if a specific implementation calls for user input/interaction. Finally, the communication manager can send custom messages through the selector's system callback and have the messages delivered to any selector subscribed to the custom message class.

Resilient offloading across mobile devices. In this work, we also use the dynamic join protocol to introduce a seamless *offload model* for computation offloading to other heterogeneous devices. The master-worker pattern is a common building block for parallel and distributed applications, and multiple workers in a single master-worker pattern can be composed hierarchically when there's a danger

of a single master actor becoming a bottleneck.

Typically, the main computation in the master will generate multiple sub-problems that can be delegated to multiple workers for parallel processing, for example in divide-and-conquer parallel algorithms (one master selector divides the work among multiple worker selectors which in turn can create new worker selectors and dividing up the computation among those newly created selectors). However, an application written using the DAMMP framework can easily offload sub-problems to other devices. A master-worker based paradigm should allow easy computation offloading through creating workers on more powerful devices (relative to smartphones), such as tablets. Each of these devices can further offload partial computations to clusters, and aggregating such layers of devices, DAMMP can handle memory or power intensive in a seamless fashion.

4.3 Dynamic Leaving

Due to the volatile nature of mobile networks, an application may experience devices leaving, dropping out, going out of range or running out of power. At the application level, the user program should be able to handle resilience and, in some cases might have to perform redundant computation on these handheld devices to address the resilience issue. At the runtime level, we support a resilient runtime system when any handheld device voluntarily leaves the ad-hoc network or drops out of the network due to running out of battery or going out of range. We need to address the following four scenarios when a device leaves the Wi-Fi Direct ad-hoc network. The scenarios are based on whether the leaving device is a group member (GM) or a group owner (GO), and whether its corresponding place is playing the role of a master place or a worker place.

- Group Member leaves the network and its a worker *place*
- Group Member leaves the network and its a master *place*
- Group Owner leaves the network and its a worker *place*
- Group Owner leaves the network and its a master *place*

In the first scenario, lets say a device leaves the ad-hoc group of phones connected with Wi-Fi Direct due to unrealiable network and the device is a group member (GM) in the Wi-Fi Direct layer:

- If the device is a worker *place* the master *place* will resend the work, either to the same worker (if the device is up and running), or our runtime will forward the computation to any other devices connected in the ad-hoc network.
- All master *places* are periodically backed up, and snapshots of the master place are sent to other worker *places*. In the event where the device, dropped out is a master, the runtime selects a worker *place* to become the new master device. The new master *place*, resumes computation. Some work performed by the new master *place* can be redundant work.

In the second scenario, where the device leaving the ad-hoc network is a group owner (GO) in the Wi-Fi Direct layer:

- Althought the Wi-Fi Direct specification allow recovery for this scenario, but currently Wi-Fi Direct implementation does not support re-establishment of the network when the GO leaves the network. To handle this scenario we restart our devices and use our dynamic join protocol to re-establish the network, and resume computation.

Our runtime supports a resilient master-worker pattern by periodically backing up program snapshots of the master selector (a selector instantiated locally without a parent, and have only outward dependencies to worker selectors on the same device/remote devices) on other nodes in the network. Upon a network change, selectors can continue execution as normal as the runtime will buffer all outgoing messages and restore a master copy once the network is re-established. The runtime will halt selector message processing by disabling any non-subscription mailbox once the outgoing buffer is full. This is to prevent any further outgoing application-level message being generated. The runtime will alert the application through subscription mailboxes that a device has left the network, and all selectors that are subscribed to the topic may take any additional action needed by that application. Upon network re-establishment, if a copy of the master selector is present in the network, the runtime will restore master selector processing, if not, the runtime notifies the application-level and restarts with a fresh copy of the master selector.

Since there are no direct dependencies between application-level selectors and the network layer, the application can maintain its resilience control mechanism, if needed, so as to further reduce redundancy of storing the selector messages. Worker selectors can buffer most recent processed messages in the event of network re-establishment with an older copy of the master, and can directly return processed results to the master. Similarly, the master can record delegated work and may resend work based on network change alerts if subscribed to a topic supported by the publish- subscription model.

We performed some additional evaluation on the Android devices using our DAMMP runtime, to demonstrate how the computation is not affected upon a device leaving the ad-hoc network and seamlessly distribute the work to the remaining devices in

the network. We use a benchmark that approximates the integral function over an interval $[a, b]$ by using the trapezoidal approximation [42, 18]. We approximate the integral of the function:

$$f(x) = \frac{1}{x+1} \times \sqrt{1 + e^{\sqrt{2x}}} \times \sin\left(x^3 - 1\right)$$

See, Section 5.2.2.1 for detailed explanation on the benchmark and experimental evaluation. In Figure 4.3 we are computing the area enclosed by the aforementioned

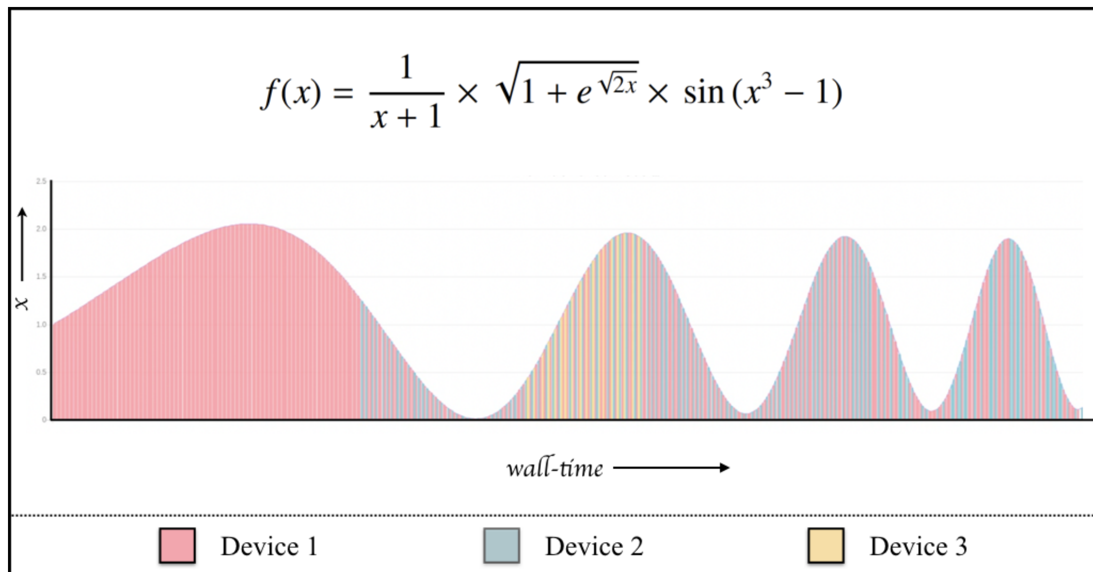


Figure 4.3 : Showing the dynamic leaving of devices while calculating a trapezoidal approximation of the function. We start with one device, and join upto three devices. Each of the colors depict which device contributed to the computation of that trapezoid in the function.

function using the trapezoidal approximation method. The interval is divided up into approximately $100,000,000$ trapezoids, and each color represents the device that contributed to the computation of that area in the function. We start the computation

with one Nexus 5 (device 1), and join 2 more Nexus 5's (devices 2 and 3) to divide the computation. As shown in the figure when the devices leave the network, remaining devices in the network seamlessly pick up the computation. The trapezoidal benchmark is implemented as a request-reply model, where work is sent to devices piece by piece, and upon getting a result, the master will generate another piece of work and send to the replying device when work is available.

Chapter 5

Experimental Evaluation

5.1 Cluster based implementation (HJDS)

Selectors can act naturally as a programming primitive in distributed setting, and more efficiently support the coordination patterns (as discussed in Section 2.2.1) than with an actor model explicitly implementing multiple guarded mailboxes [3]. The Selector Model, as a more generalized form of actors, also supports any readily available actor-based programming patterns. To demonstrate the scalability and programmability of selectors, we show results of some actor-based micro-benchmarks chosen from the SAVINA benchmark suite [43].

5.1.1 Hardware Setup

The benchmarks were run on a 12 core (two hex cores), 2.8GHz Westmere nodes with 48GB of RAM per node (4 GB per core), running Red Hat (RHEL 6.5). We used up to 16 nodes in our studies.

5.1.2 Benchmarks

For benchmarking, we use the number of workers equal to twelve *times* the number of nodes. On each node an 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. The selected benchmarks use the master-worker

parallelism to achieve both intra-node and inter-node parallelism. Each computing node is designated as a single *place*, with 12 workers on each *place* to minimize the effect from task scheduling on computation time. All implementations feature multiple mailboxes to differentiate control messages, and actual computational tasks, with control messages of higher priority. The master selector in computation is located on the Master Node of the HJDS system in all benchmarks.

5.1.2.1 NQueens First K Solutions

The *NQueens K Solutions* benchmark finds the first K solutions to placing N queens on the chessboard of size $N \times N$ in a way that no queen can threaten each other. This benchmark uses the classic master-worker programming model with a depth-first search to exhaustively enumerate through all solutions and prematurely terminates at finding the first K solutions. The master selector initiates computation by passing an empty $N \times N$ board (as a partial solution) to each worker. Each time a worker successfully place a non-attacking queen on the partial solution, the worker reports the partial board back to master as a new work item. Each time a worker reports a board configuration to the Master Node selector, the master either assigns the partial solution to a worker in a round-robin fashion or records that a valid solution is found.

This algorithm exploits the priority feature in our Distributed Selector implementation for more than the purpose of progress control, and places a higher priority on work items that contain complete partial solutions (i.e. with more safely placed queens). The priority restriction on partial solutions reduces the process on duplicate work items and supports early termination of the program by putting complete solution at highest priority to process aside from control messages. When K solutions have been found, the master will send out a termination message (of highest priority)

to all workers.

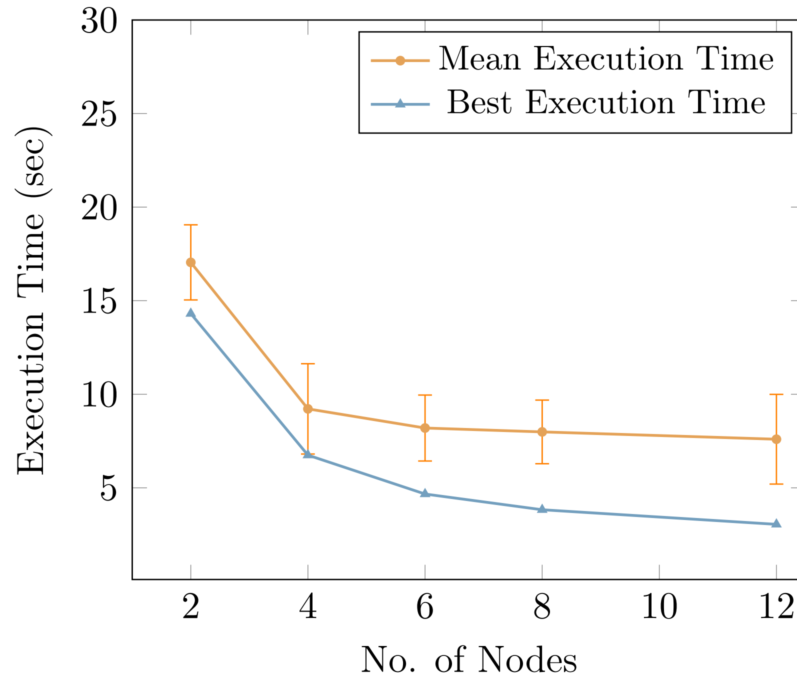


Figure 5.1 : NQueens : Shows strong scaling of computing the NQueens problem on a 17×17 board using the described algorithm. Workers compute solutions sequentially when given a partial solution with six placed queens. The number of workers per node is constant (12). The solution limit is set to 1_477_251, which is a tenth of the size of a complete solution set. Mean Execution time in seconds from 20 iterations. The error bar on mean execution time plot shows the standard deviation of the 20 iterations.

With a naive actor-based implementation, a solution limit to allow termination has no effect on the program, and the program has to exhaustively compute the solution space. Without using priority, depth-first search with a divide-and-conquer style, as shown above, is hard to implement, since messages are processed in their received order, and no guarantee is made on processing partial solutions deeper in the

search tree. While the priority feature can be emulated in an actor model through explicit pattern matching on its mailbox, the overhead generated for each message can be collectively large for a significant amount of message exchange, like in the NQueens K case.

This benchmark exhibits complicated interaction pattern between the master and its workers. In this algorithm, the master node becomes a bottleneck when the number of workers increases, as all communication about partial results routes through the master selector. The abundance of worker selectors when we use more nodes also poses the problem of extra duplicate partial solutions, we approach this by having workers to filter through work items they've worked on, but the master node still receives more items as the system size grow. Observing the results in Figure 5.1 showing, we can see that as the amount of (duplicate) partial solutions grows super-linearly with the number of nodes deployed, the bottleneck of processing messages takes over, resulting in decreased speedup as the number of nodes increase from 4 to 12.

5.1.2.2 Pi Precision

This benchmark computes the value of π to a pre-configured precision using a digit extraction algorithm. The following formula can be used to compute π:

$$\pi = \sum_{n=0}^{\infty} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^{n}$$

The *Pi Precision* Computation represents a master-worker style parallelism. In this scenario, the amount of communication between the master and its workers are frequent, though still with a small message body. In the Pi Precision implementation, the master selector incrementally finds work and allocates fragments of the work to the worker selectors, while it collects partial results until reaching the desired precision.

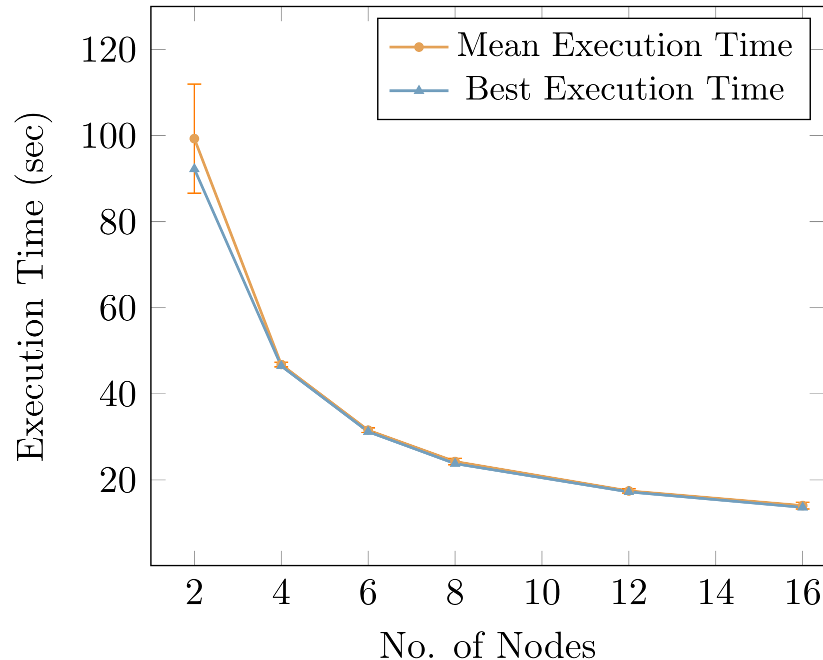


Figure 5.2 : Precise Pi : Shows strong scaling for the calculation of Pi with a precision of 80_000. Number of workers per node is constant (12). Work is evenly distributed among all selectors. Mean Execution time in seconds from 20 iterations. The error bar on mean execution time plot shows the standard deviation of the 20 iterations.

In this benchmark, we explore distributed selector scalability when message exchange grows with the expansion of the system. We observe a general linear strong scaling trend, with a slow decrease in speedup as the number of nodes used increase, as shown in Figure 5.2. Both the increased message amount from having more workers and the increased average message delivery time from the physical span of the computing nodes the system uses can attribute to overheads that prevent linear scalability.

Both the micro-benchmarks show how the HJDS runtime perform under different message exchange patterns. With the *Pi Precision* benchmark, where the amount

of message exchange between master and workers grow with the number of places in the system, we see the runtime perform with near linear strong scaling. With the more complicated *NQueens K Solutions* benchmark, where the program exhibits a bottleneck, we observe sub-linear strong scaling as expected from the super-linear growth of message exchange as the number of places increase. Moreover, the NQueens K benchmark displays a programming pattern less easily achievable with a general actor model and shows better efficiency can be achieved with a traditional implementation. The system *bootstrap* and *termination* sequence for both the benchmarks take an average of 20ms, constituting of less than 0.001% of program execution time. This also confirms that the major limitation on scalability comes from the communication patterns.

5.2 Android based implementation (DAMMP)

Our Android-based implementation of DAMMP currently supports two communication layers, one with standard Wi-Fi and the other with Wi-Fi Direct. We are using the Android operating system as our research vehicle because of its open source software stack and Linux kernel roots, as well as the availability of the Android JVM — Android RunTime (ART) [44], an efficient and low memory footprint virtual machine that provides a high-level managed runtime that is well suited for Actor implementations. Our implementation was undertaken on Android 5.1.1 and complies with available Wi-Fi and Wi-Fi Direct APIs at level 22. Under the Wi-Fi Direct based communication layer, one device acts as group owner and broadcasts provided service(s), while nearby devices may join through service discovery to act as group member(s).

5.2.1 Hardware Setup

Our tests platform includes five Nexus 5 devices, with a Quad-core 2260 MHz Krait 400 processor and a Qualcomm Snapdragon 800 MSM8974 system chip, and three Nexus 4 devices, with a Quad-core 1500 MHz Krait processor and a Qualcomm Snapdragon S4 Pro APQ8064 system chip.

5.2.2 Benchmarks

We provide experimental results and analysis for two benchmarks:

1. A micro-benchmark that measures message passing throughput and the impact of communication overhead on application scalability in different communication environments (Section 5.2.2.1)

2. A distributed actor benchmark that measure scalability of our distributed mobile platform (Section 5.2.2.2)

Since these are standard actor benchmarks, they all use a single mailbox as in the standard actor model. The benchmark execution times exclude Android application startup and termination times. To reduce variability due to system and environmental factors, we minimize log output, disable background processes, and utilize a temperature controlled testing environment (a refrigerator freezer) for all results.

5.2.2.1 Trapezoidal Approximation

The *Trapezoidal* benchmark approximates the integral function over an interval $[a, b]$ by using the trapezoidal approximation [42, 18].

We approximate the integral of the function:

$$f(x) = \frac{1}{x+1} \times \sqrt{1 + e^{\sqrt{2x}}} \times \sin\left(x^3 - 1\right)$$

The parallelism is achieved by dividing up the integral approximation into a fixed number of intervals, by using a master-worker pattern. The original algorithm is obtained from [15], in which each worker is remotely created by the master actor, work is sent to each worker, and the completed results are returned to the master.

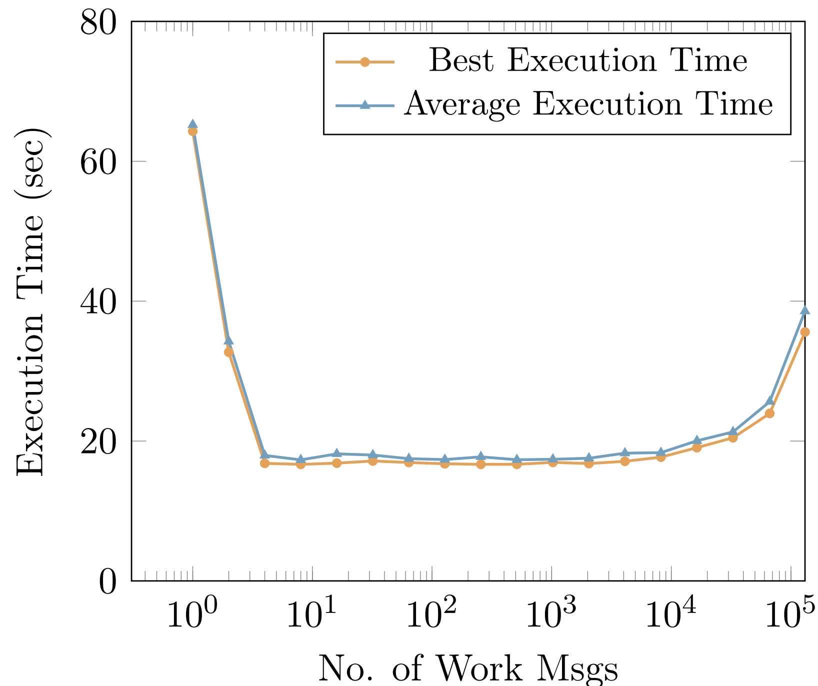


Figure 5.3 : Trapezoidal Approximation: Shows the best and average time (over 20 executions) with 4 Nexus 5 devices to compute an approximation with $100,000,000$ intervals for the integration. The x axis (in log scale) shows the number of work messages sent by the master to workers, and the y axis shows the best and average times. The total work remains the same for all experiments. For each work message, a reply message is sent back to the master with the result.

The DAMMP implementation has been modified to use a request-reply model, where work is sent to workers piece by piece, and upon getting a result, the master will generate another piece of work and send to the replying worker when work is

available. The workload distribution in this benchmark remains fixed. The reason for a fixed distribution is because we use this benchmark to evaluate the impact of the number of messages on performance, without considering the impact of dynamic load balancing. Since the workload is statically distributed, this benchmark is an excellent candidate for examining the tradeoff between the number of messages and workload size in each message. Note that each interval results in two messages exchanged between the master and a worker, a *work message* sent from the master to the worker and a *reply message* sent from the worker to the master.

In our experiments, we use a static network of four Nexus 5 devices, and increase the total number of work messages by powers of two, starting with one work message (serial execution). Figure 5.3 shows the number of work messages on the x-axis on a log scale, and shows the corresponding execution time in its y-axis.

We can observe that for a constant workload, the execution time becomes consistent once all four devices are involved (4 or more work messages are sent). The communication overhead does not affect the 4x speedup until the point when 10^4 to 10^5 work messages are sent. This benchmark demonstrates the robustness of the system to effectively hide latency from communication and shows that ideal parallelism can be achieved even with a relatively large number of communication messages.

5.2.2.2 Pi Precision

The *Pi Precision* benchmark computes the value of π to a specified precision using a digit extraction algorithm. We use the same formula as in Section 5.1.2.2 to calculate the value of π:

$$\pi = \sum_{n=0}^{\infty} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^n$$

Unlike the static work distribution in Section 5.2.2.1, this benchmark uses a master-worker pattern with dynamic work distribution, where the master sends more work (if available) to a worker that sends a reply with results.

Figure 5.4 shows the execution time of calculating Pi to the 15,000 decimal place for an increasing number of devices. Each device contains two worker actors, while the total amount of work remains constant as the number of devices is increased (strong scaling). 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.

The data-point in Figure 5.4 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 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 because 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.

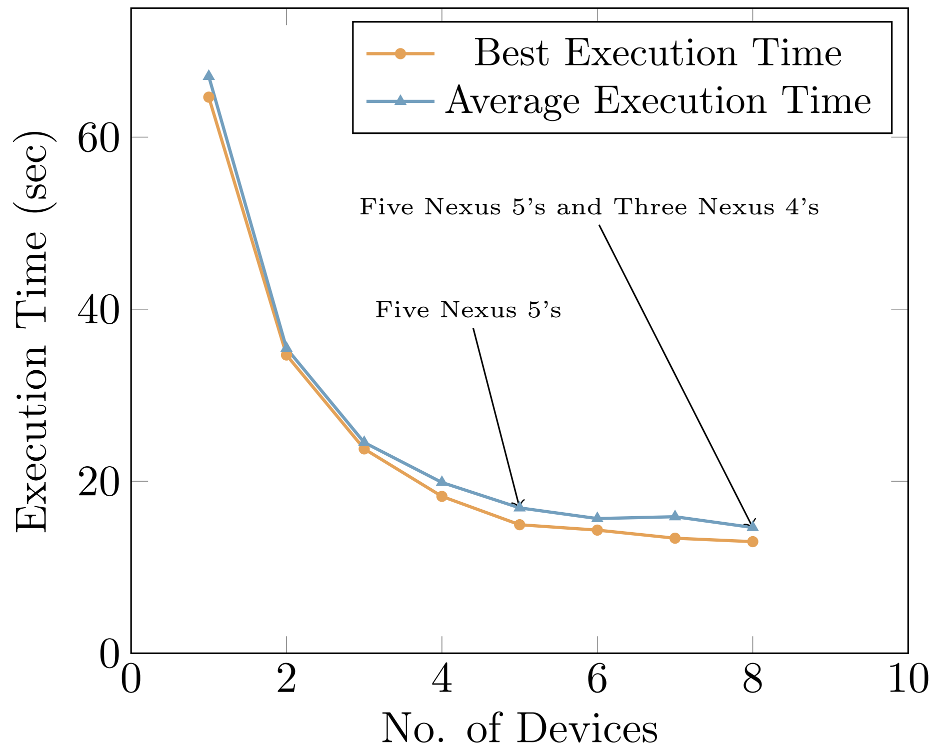


Figure 5.4 : Pi Precision Computation: Shows the best and average time (over 20 executions) to compute the value of Pi to 15,000 decimal points. The x-axis shows the number of devices used for the computation, the y-axis shows the execution time. Each device runs two worker actors, and only one of the devices 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.

Chapter 6

Related Work

Ever since its inception, the logical isolation, and asynchrony inherent in the actor model have made it an attractive candidate for distributed computing for decades, with more recent explorations of the actor model for distributed mobile platforms. We briefly summarize some related work on using the actor model for both cluster based computing and distributed mobile computing in recent years.

6.1 Akka

Akka is an open-source toolkit and runtime for building highly concurrent, fault-tolerant and distributed systems on the JVM based on the *Actor Model* [17]. The toolkit can be used as library similar to our HJ library. The Akka runtime arranges the user-defined actors in an ancestral tree, mainly for the purpose of recovery from single point failures. The Akka runtime requires users to explicitly shutdown a system of actors and relies on the user to ensure termination of the whole system.

Akka.cluster is a module dedicated to aiding actor-based distributed application programming and its significant contribution is to maintain location transparency that follows its previous strict adaptation of AM [45]. Using their gossip protocol [25] and the accrual failure detector [26] Akka provides a fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure/bottleneck.

Akka actors support priority-enabled mailbox to some extent: the Akka prioritized mailboxes associate a specific message class or value to a predefined priority. Although still maintaining a single mailbox, Akka runtime arranges the received message order based on these predefined priorities. The HJDS, on the other hand, is more flexible and allows the user to pass the same message type with different priorities by making no predefined association between mailboxes and the messages it hold. The distributed selector model is capable of more efficient and easier implementation of complex synchronization patterns [3].

6.2 SALSA: Simple Actor Language, System and Architecture

SALSA is a Java-based actor programming language developed at RPI [18]. The language targets open, dynamically re-configurable Internet and mobile applications. It focuses on the mobility of actors in distributed system and features universal naming, active objects, and actor migration. SALSA introduces three language mechanisms to aid coordination between actors: *token-passing continuation, join continuation,* and *first-class continuation*. With a major focus on providing reconfigurable subcomponents at runtime, SALSA provides daemon programs for host universal actors named *Theaters* and supports universal actor with the *Naming Server* [35]. Together these universal actors can host distributed *SALSA* programs and provide services such as migration and message forwarding for remote actors.

The *SALSA* programming model differs from HJDS in several ways. By allowing migration of actors, the SALSA runtime directs all remote reference lookups to the *Naming Server*. The HJ runtime encodes places for remote selectors in their references and deals with message forwarding locally. Exiting an actor can be explicitly called

in HJ while being implicit in SALSA. HJ allows system boot-up and termination by the program, unlike the background daemon servers that make up the distributed SALSA system. In HJ the user decides the duration of keeping the server running. The distributed implementation of SALSA Lite provides no support for the proper termination of the runtime. The user is responsible for terminating all the actor instances at a stage, and once the stage is no longer hosting any active actors the user may terminate the stages and the theaters they are running on.

HJ does not have an explicit construct to wait on multiple actor message returns like the join continuations in SALSA, but the functionality can be easily achieved by using the join pattern with selectors. Finally, a key difference between SALSA and HJ is that our HJDS model supports selectors with multiple mailboxes in a distributed setting. We were unable to include any performance comparisons with SALSA in this dissertation because of running into JVM OutOfMemory errors for the SALSA versions of the benchmarks when using the same coniïňĄgurations that we use for HJDS.

6.3 AmbientTalk

The language AmbientTalk is an actor-based programming language designed specifically for `mobile ad hoc networks` [46]. It features λ calculus based functional elements with local and remote actor-based reductions, and object-oriented elements with both parameter pass-by-value (isolated objects) and pass-by-reference "regular" objects) semantics. It utilizes the Actor model for its concurrency and distributed computation. Actors in the `AmbientTalk` VM are used as containers to hold a set of regular objects, rather than regular "active objects". The virtual machine hosts

multiple actors that can execute concurrently, while each actor itself represents a communicating event loop that uses the *run-to-completion* semantics for method invocation on its host objects.

While the `AmbientTalk` model has similarities with the traditional actor model, one difference is that it can break the pure message-passing model through the use of *far references* introduced in the the E language [47]. The `AmbientTalk` language has both cluster-based and early Android implementations that focus on high-level abstractions for distributed programming with both message passing mechanisms and remote accesses through far references. In our work, on the other hand, the focus is on supporting a pure actor/selector model at the high-level, with distributed mechanisms supported in configurations that are decoupled from the programming application logic. Further, our model, DAMMP does not limit ourselves to *mobile ad-hoc networks* since it can also support communications within and across mobile devices and server devices with a single model. Finally, to the best of our knowledge, `AmbientTalk`'s implementation targets an outdated version of Android (prior to Android 4.0) that is no longer supported on current mobile devices, thereby preventing us from performing experimental comparisons with `Ambient Talk`.

6.4 ActorDroid

The ActorDroid project is based on SCALA actors and focuses on a distributed application framework which follows the stream processing paradigm inherited from SCALA [48]. The framework's basic units are *services*, each independently a SCALA actor that runs in its own execution environment. It uses a publish-subscribe model for service discovery and join, while each mobile device can host one-to-many services.

The `ActorDroid` work described an implementation based on external Wi-Fi Access Points with dynamic network maintained by a Master-Worker scheme. We also use a decentralized structure in our runtime to enable support for dynamic topologies. However, unlike `ActorDroid`, the hardware dependent communication layer is completely decoupled from the application-level actor communication in our approach. Instead of relying on a pre-defined strategy for dynamic topological changes, we expose a minimal amount of information to application-level actors that can provide the user-level application with the ability to adapt to runtime events. As with `Ambient Talk`, to the best of our knowledge, `ActorDroid`'s implementation targets an outdated version of Android (prior to Android 4.0) that is no longer supported on current mobile devices, thereby preventing us from performing experimental comparisons with `ActorDroid`.

6.5 ActorNet

The ActorNet project is an actor-based mobile agent platform for wireless sensor networks (WSNs) [49]. This project aims to create a high-level abstraction for concurrent and asynchronous programming for WSNs to adapt to the limited hardware resources available on mobile sensors. The ActorNet project implements a `Scheme` interpreter that is assumed to be better suited to the limited processing power and memory available on wireless sensors than stack-based virtual machines and introduces new language primitives that enable actor-based message passing, queries, and access to program continuation. The `ActorNet` project also focuses on higher level language abstractions to aid parallel and asynchronous programming on mobile networks. Compared to the emphasis on optimization for limited hardware specifically

for wireless sensors, our work focuses more on adaptability on a wider range of mobile devices by creating a more general distributed runtime system.

Our design DAMMP goes beyond sensor networks and supports rich combinations of mobile devices and cluster-based services. Compared to `ActorNet`, our design is better suited for use with more powerful consumer mobile devices such as high-end tablets and smartphones. Finally, since `ActorNet` provides an implementation specific to wireless sensors, we were unable to perform an experimental comparison with `ActorNet`.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

In this dissertation, we address the dynamic reconfiguration challenges that arise in distributed implementations of the Selector Model, by providing two implementations of distributed selectors, one for distributed servers and another for distributed Android devices.

The Habanero Java Distributed Selector (HJDS) model is a novel programming model for shared memory and distributed memory parallel applications. The Distributed Selectors on the clusters 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.

To address reconfiguration challenges on Android devices, we have developed a mobile platform based Java runtime library, DAMMP (Distributed Actors Model on Mobile Platforms) by extending the HJDS implementation. Using the DAMMP runtime we focus 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 Habanero Java Distributed Selectors for shared-memory and distributed parallelism. We presented a task offloading pattern based on the selector model and the Master-Worker paradigm.

We evaluated the performance of our selector-based distributed implementation on both clusters and Android devices using benchmarks from the Savina benchmark suite [43].

- For the cluster-based implementation our results show promising scalability performance for various message exchange patterns. We also demonstrate high programming productivity arising from high-level abstraction and location transparency in the Habanero Java Distributed Selector runtime library (as evidenced by minimal differences between single-node and multi-node implementations of a selector-based application), as well as the contribution of automated system bootstrap and global termination capabilities. Our experimental evaluation makes a strong case for the HJDS model as a viable alternative to the existing, much harder to program and port, parallel programming models.

- For the Android based implementation we evaluated the DAMMP framework under ideal usage conditions to show promising scalability and performance, and analyze the communication overhead of both Wi-Fi Soft AP and Wi-Fi Direct when used as the communication layer for DAMMP. 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 (Wi-Fi Soft AP and Wi-Fi Direct).

Our empirical results expose some of the limitations of the current state-of-the-art in device-to-device wireless connectivity. Our DAMMP runtime provides network researchers an intuitive and easy to use platform for connectivity experiments.

7.2 Future Work

For future directions on our cluster-based platform, we plan to explore automated program quiescence detection that does not rely on the user explicitly exiting each user-level Selector. We plan to look into dynamic load-balancing by allowing features like the migration of Actors. Extensions on the current work is to include dynamic joining and leaving of nodes from the cluster for a better fault tolerance mechanism.

With the empirical results in mind for our DAMMP runtime, we plan to explore real-world heuristics in thermal-aware dynamic distribution on heterogeneous mobile networks that involve devices with various computing powers, including wireless sensors, tablets, and laptops. We plan to explore dynamic load-balancing across devices by having the runtime automatically migrate Actors when needed for various reasons (maximizing overall performance, maximizing combined system battery life etc.).

We also plan to explore high-level abstraction for distribution by supporting shared-memory parallel construct (DataDriven Futures, Phasers) in the Habanero Execution model in distributed mobile peer-to-peer networks. Support for multiple master nodes for increased fault tolerance and scalability is also of future interest. We will also study more closely the performance tradeoffs between traditional Actor-based libraries (such as Akka and SALSA) on traditional static networks and our DAMMP model implementation.

Bibliography

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

[2] 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.

[3] 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.

[4] 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.

[5] C. Tomlinson and V. Singh, "Inheritance and Synchronization with Enabled-Sets," in *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, (New York, NY, USA), pp. 103–112, ACM, 1989.

[6] I. A. Mason and C. L. Talcott, "Actor Languages: Their Syntax, Semantics, Translation, and Equivalence," *Theoretical Computer Science*, vol. 228, 1999.

[7] C. Lindemann and O. P. Waldhorst, "A distributed search service for peer-to-peer file sharing in mobile applications," in *Proceedings. Second International Conference on Peer-to-Peer Computing,*, pp. 73–80, 2002.

[8] R. K. Lomotey, Y. Chai, A. K. Ahmed, and R. Deters, "Distributed Mobile Application for Crop Farmers," in *Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, MEDES '13, (New York, NY, USA), pp. 135–139, ACM, 2013.

[9] C. Doukas and I. Maglogiannis, "A fast mobile face recognition system for android OS based on Eigenfaces decomposition," in *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pp. 295–302, Springer, 2010.

[10] A. Mutholib, T. S. Gunawan, and M. Kartiwi, "Design and implementation of automatic number plate recognition on android platform," in *Computer and Communication Engineering (ICCCE), 2012 International Conference on*, pp. 540–543, IEEE, 2012.

[11] M. B. Kjærgaard, J. Langdal, T. Godsk, and T. Toftkjær, "EnTracked: Energy-efficient Robust Position Tracking for Mobile Devices," in *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, (New York, NY, USA), pp. 221–234, ACM, 2009.

[12] M.-H. Wang Ph D *et al.*, "Feasibility of using cellular telephone data to determine the truckshed of intermodal facilities," 2012.

[13] S. E. Wiehe, A. E. Carroll, G. C. Liu, K. L. Haberkorn, S. C. Hoch, J. S. Wilson, and J. Fortenberry, "Using GPS-enabled cell phones to track the travel patterns

of adolescents," *International journal of health geographics*, vol. 7, no. 1, p. 22, 2008.

[14] C. Ratti, D. Frenchman, R. M. Pulselli, and S. Williams, "Mobile landscapes: using location data from cell phones for urban analysis," *Environment and Planning B: Planning and Design*, vol. 33, no. 5, pp. 727–748, 2006.

[15] 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.

[16] A. Chatterjee, B. Xue, S. Milakovic, Z. Budimlic, and V. Sarkar, "DAMMP: A Distributed Actor Model on Mobile Platforms," In preparation.

[17] Typesafe Inc., "Akka Documentation," 2014. [Online; accessed 9-April-2014].

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

[19] V. Sarkar, "Habanero-Java." https://wiki.rice.edu/confluence/display/HABANERO/Habanero-Java.

[20] 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.

[21] Oracle, "Understanding Interaction Patterns," 2011.

[22] G. Hohpe and B. Woolf, "Enterprise Integration Patterns - Request-Reply," 2003. [Online; accessed 3-April-2014].

[23] "Akka: Stash Interface." Available at http://doc.akka.io/japi/akka/2.3.1/akka/actor/Stash.html.

[24] "Akka implements a unique hybrid," 2015. Avaialble at http://doc.akka.io/docs/akka/2.4/intro/what-is-akka.html#Akka_implements_a_unique_hybrid.

[25] "Akka: Gossip Protocol," 2015. Avaialble at http://doc.akka.io/docs/akka/current/common/cluster.html#Gossip_Protocol.

[26] "Akka: Cluster Failure Detector," 2015. Avaialble at http://doc.akka.io/docs/akka/rp-current/common/config-checker.html#Cluster_Failure_Detector.

[27] "Akka: Lifecycle Monitoring aka DeathWatch," 2015. Avaialble at http://doc.akka.io/docs/akka/2.4.17/scala/actors.html#deathwatch-scala.

[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[29] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic Algorithms for Replicated Database Maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, (New York, NY, USA), pp. 1–12, ACM, 1987.

[30] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The φ accrual failure detector," in *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pp. 66–78, IEEE, 2004.

[31] "Akka: Poison Pill," 2015. Avaialble at http://doc.akka.io/docs/akka/current/scala/persistence.html#Safely_shutting_down_persistent_actors.

[32] "Akka: Graceful Stop," 2015. Avaialble at http://doc.akka.io/docs/akka/current/scala/actors.html#Graceful_Stop.

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

[34] "World-Wide Computing in SALSA," 2005. Avaialble at http://wcl.cs.rpi.edu/salsa/salsa101/node24.html.

[35] T. Desell and C. A. Varela, "Salsa lite: A hash-based actor runtime for efficient local concurrency," in *Concurrent Objects and Beyond*, pp. 144–166, Springer, 2014.

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

[37] Google, "Protocol Buffers," 2017. Available at https://developers.google.com/protocol-buffers/docs/overview.

[38] Apache, "Apache Avro^{TM} 1.8.2 ," 2017. Available at http://avro.apache.org/docs/1.8.2/.

[39] Apache, "Apache Thrift^{TM} v0.10.0," 2017. Available at https://thrift.apache.org/.

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

[41] S. M. 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.

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

[43] 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.

[44] "ART and DALVIK." Available at https://source.android.com/devices/tech/dalvik/.

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

[46] T. Van, C. Christophe, S. D. Harnie, and W. D. Meuter, "An Operational Semantics of Event Loop Concurrency in AmbientTalk."

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

[48] P.-A. Mudry and R. Cherix, "ActorDroid - A distributed computing framework for mobile devices based on SCALA actors.," *ScalaDays*, 2012.

[49] 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.