A Distributed Selectors Runtime System for Cluster-based Java Applications

Arghya Chatterjee[1], Branko Gvoka[2], Bing Xue[1], Zoran Budimlic[1], Shams Imam[3], and Vivek Sarkar[1]

[1]Department of Computer Science, Rice University, Houston, TX, (arghya,bx3,zoran,vsarkar)@rice.edu

[2]Faculty of Technical Sciences, University of Novi Sad, bane92ru@gmail.com

[3]Two Sigma Investments, LP, shams.imam@twosigma.com

ABSTRACT

The demand for mainstream programming models supporting scalable, reactive and versatile distributed computing has grown immensely with the proliferation of manycore/heterogeneous processors on portable devices and cloud computing clusters that can be elastically and dynamically allocated. With such changes, distributed software systems and applications are shifting towards service oriented architectures (SOA) that consist of largely decoupled, dynamically replaceable small components and migrating towards loosely coupled, interactive networks that may exhibit more complex coordination and synchronization patterns. We believe that the fine-grained dependencies and asynchronous processing with state isolation are the central motif in task-level parallelism that fulfill the need for elasticity, performance, and programmability in a modern distributed setting and can pave the road towards a scalable and versatile distributed programming.

In this paper, we propose the Distributed Selector (DS) model, a task-based programming model that supports multiple distributed nodes with complex coordination patterns using a simple syntax, featuring automated system bootstrap and global termination. We focus on the Selector Model (a generalization of the actor model) as a foundation for creating distributed programs and introduce a unified runtime system that supports both shared memory and distributed multi-node execution of such programs. The multiple guarded mailboxes, a unique and novel property of Selectors, allow the programmers to turn on and off the processing of different mailboxes and enable coordination and synchronization patterns such as *a*) synchronous request-reply, *b*) join patterns in streaming application, and *c*) producer-consumer with bounded buffer, which are strictly more general than those supported by the Actor model.

We evaluate the performance of our selector-based implementation using benchmarks from the Savina benchmark suite [13]. Our results show promising scalability performance for various message exchange patterns. We also demonstrate high programming productivity arising from high-level abstraction and consistent location transparency in the HJ 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.

PPPJ '16, August 29-September 02, 2016, Lugano, Switzerland

DOI: http://dx.doi.org/10.1145/2972206.2972215

Keywords

Actor Model, Selector Model, Distributed Selectors, Remote Messaging, Remote Synchronization

1. MOTIVATION

Distributed application for today's cloud and mobile platforms need more than mere computing capacity. Without improvements in scalability and programmability, the ever-growing complexity of interaction patterns in distributed computing can limit us from efficiently exploiting available computational resources. While the need for exploiting multi-node parallelism is widely acknowledged in modern cloud services, there remains a conceptual disparity between programming models for shared-memory parallelism and distributed concurrency and it has not been fully resolved. The *Actor Model* (AM) [1, 8], represented by isolated processes (*actors*) that interact solely via asynchronous message passing, is a natural fit for a unified concurrency model for both multi-core and cluster level parallelism. However, we believe that the traditional AM poses certain limitations on actor coordination and synchronization patterns, and we aim to develop a distributed runtime system based on the more general Selector Model(SM) [15].

In this paper, we introduce the Habanero Java Distributed Selector (HJ-DS) runtime, which uses the Distributed Selector (DS) model as a unified programming model for both single-node and multi-node parallelism. We build the HJ-DS runtime as an extension to the single-node shared-memory Habanero Java Runtime Library (HJlib) [3]. The DS model extends the Actor Model with *multiple guarded mailboxes* and *message priorities* to enable more general coordination and synchronization patterns than those supported by the AM. It offers a promising approach for building distributed concurrent applications with both productivity and scalability.

Unlike other distributed actor libraries, our implementation features light-weight *selector* creation on remote places, automatic system bootstrap, and automatic program termination detection. Our library includes support for *a*) transparent creation of selectors on remote nodes; *b*) transparent message delivery for local or remote recipient selectors; and *c*) distributed global termination when the program becomes quiescent. In contrast to most distributed Actor-based libraries (e.g. Akka [22], SALSA [23]) that use daemon tasks for these system services, our library implements

such services as system-level actors when internal runtime events are triggered.

2. THE DISTRIBUTED SELECTOR MODEL

2.1 The Selector Model

Selectors are an extension of the Actor model. A Selector is an execution unit that has the capability to process incoming messages. Similar to actors, selectors encapsulate their local state and process incoming messages, one message at a time. Figure 1 shows a decomposition of the Selector model. The modularity and data locality of the Actor model are still preserved when using selectors.

Although the Actor Model has been successfully used for concurrent computation, not all concurrent problems are most effectively solved using this model. In some concurrent programming patterns, preserving the integrity of the objects requires synchronization mechanisms to control the order in which messages are processed in the mailbox [19].

Since the Actor model does not permit shared state and forces all communication to be asynchronous, concurrent coordination involving multiple actors might be harder than using some simple constructs such as locks [16]. The Selector model we are proposing in this paper acts as an abstraction to support synchronization and coordination mechanism among multiple actors. Selectors allows an actor to have multiple guarded mailboxes. This is a unique and novel property of Selectors, which distinguishes them from Actors. Messages can be sent to any of the mailboxes, and the processing of messages from a specific mailbox can be managed by turning the processing of a specific mailbox *on* or *off*.

The multiple guarded mailboxes in the Selector model enable coordination and synchronization patterns such as *a*) synchronous request-reply [Section 2.2.1], *b*) join patterns in streaming application [Section 2.2.2], and *c*) producer-consumer with bounded buffer [Section 2.2.3]. We explain each of the motivating examples in the next section, contrasting our Selector based approach with the Actor based approach. We briefly describe our library implementation of the Distributed Selectors [Section 4] and finally present some performance results of our Java based implementation of the Savina benchmarks [Section 5] that show promising strong-scaling results.

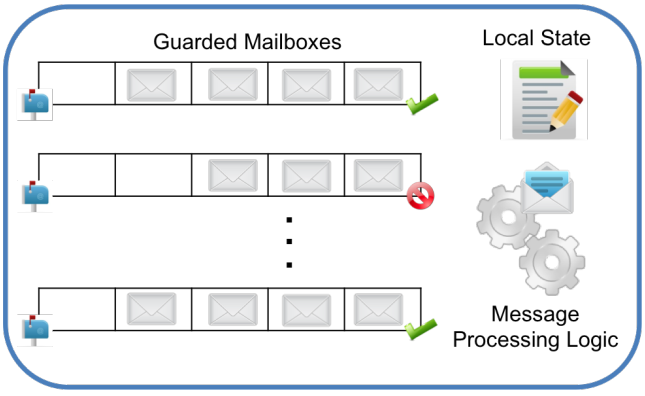

Figure 1: Decomposition of a selector: guarded mailboxes, local state, message processing logic.

Selectors differ from the conventional actor design in two ways: *a*) Selectors have multiple mailboxes to receive messages, which allows messages of different priority or purpose to be concurrently and asynchronously added to different mailboxes, eliminating the need for blocking coordination and reduce contention, *b*) The mailboxes have a boolean condition (guards) which can be used to enable or disable a specific mailbox while processing a message. This *guard* does not affect the mailboxes' ability to receive messages; it only controls whether the messages in the mailbox can be processed or not. Each mailbox is guarded with one or more boolean conditions that control whether message processing is enabled or disabled for the mailbox. An *actor* can be implemented as a *selector* with an always enabled, single mailbox.

The DS life cycle is very similar to its shared-memory counterpart [15], and displays the complete encapsulation and state isolation found in most Actor model interpretations. The selector life cycle consists of the following three stages:

- new: A *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 after processing a set limit of messages.
- terminated: A *selector* is terminated upon the `exit` signal. A Selector in such state will not process any more 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 said selector are observed to have completed and no outgoing messages remain in the local buffer. A Selector in such state cannot be restarted, and a system-wide termination of all selectors will signal the global termination of the application.

Figure 2: Life cycle of a selector.

The multiple guarded mailboxes in a selector allows the programmer to perform the following actions:

- *Mailbox determined by sender*: The message sender can directly specify the mailboxId to send a message to.
- *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 loading and dynamic updates in many interactive or reactive systems.
- *Declarative mailbox guard*: A mailbox may be guarded with explicit declarative expressions, thus separating the mailbox enable/disable logic from regular message processing logic. Such approach can be useful when the enabling/disabling of mailboxes rely on the selector's internal state, such as in many pipeline based applications.

2.2 Coordination with Selectors in Distributed Applications

In this section, we describe how the multiple guarded mailboxes in the Selector model allow efficient coordination and synchronization patterns, and how the patterns are transparently applied to a distributed runtime. We demonstrate the Distributed Selector model's programmability by contrasting the distributed selector patterns with a traditional actor-based solution.

2.2.1 Synchronous Request-Reply Pattern

We can observe a synchronous request-reply pattern [18, 11] when a requestor sends a message to the replier, which receives and processes the message, and eventually returns the message in response to the requestor. When using the Actor-based model for a computation of synchronous request-reply, when an actor sends a message(request), its message processing logic stalls all other computations until it receives the corresponding reply. Since the Actor model relies on asynchronous messages, this pattern would require separate messages for request and reply. Such pattern can be hard to implement efficiently as the requesting actor's single mailbox needs to house both the request and the reply messages. Using a non-blocking method the incoming messages before the response message must be *stashed* and *unstashed* to the mailbox after processing the reply message.

Another approach to avoid complications in processing the existing messages can be implemented using a blocking mechanism but limits scalability. 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 [20]. Such approach, however, produces overhead associated with maintaining the `stashed` messages when the actor is in a *reply blocked* state, and when the messages need to be `unstashed` after the response message is processed, as well as switching context for different processing patterns. Additional overhead is observed if the `unstashed` messages need to be prepended to the head of the mailbox, as the actor can be receiving other messages during the *reply-blocked* stage.

Solution: Request-Reply with Selectors.

Using the Selector-based approach we can define two separate maiboxes, one to receive *regular* messages including all the *request* messages and another mailbox to receive only *synchronous response* messages. Let's say the selector has two mailboxes (see Figure 3): REGULAR and REPLY. Whenever a selector is expected to process a synchronous response message, it disables the REGULAR mailbox, which ensures that the next message to be processed will be from the REPLY mailbox [line 10]. When a responder processes the request message it will send back a response to the REPLY 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 REPLY mailbox, it enables the REGULAR mailbox and starts processing other messages[lines 12 - 17]. Such pattern translates seamlessly to distributed applications, hides away the longer message passing latency, and can be a large contributor to improving efficiency in interactive applications and service-oriented architectures where a request-response pattern is commonly used.

2.2.2 Join Patterns in Streaming Applications

An Actor-based approach can be an excellent choice for streaming applications as it can be used to pipeline messages. Actors can be connected in a data flow chain to form a producer-consumer pair, and ensure messages to be processed in FIFO order. Such a network can effectively exploit parallelism, by propagating data asynchronously from producers to consumers as data becomes available. 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. Join-patterns 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. Figure 4 shows an *aggregator*, where the *Adder actor* is consuming data from the *Source actors* and adding streams of corresponding values.

```
1  public class ReqRespSelector extends ↩
       DistributedSelector {
2    public void process(MessageType theMsg){
3      if( theMsg instanceof SomeMessage){
4          // a case where we want a response
5          SomeRequest req = new SomeRequest(this,
6              new SomeMessage(theMsg));
7          anotherActor.send(req);
8          // move to reply-blocked state
9          this.mailbox.get(REGULAR).disable();
10
11     } else if (theMsg instanceof SomeReply){
12
13         // process the reply (from REPLY mailbox)
14         ...
15         // resume processing regular messages
16         this.mailbox.get(REGULAR).enable();
17
18         ...
19     } else {
20       ...
21     }}}

23 class ResponseSelector extends DistributedSelector{
24   public void process(MessageType theMsg) {
25     if (theMsg instanceof SomeRequest){
26         SomeReply reply = compute(m.data);
27         // send to response mailbox
28         sender().send(REPLY, reply);
29       ...
30     } } }
```

Figure 3: Using Selectors to solve the Request-Response Pattern without blocking. In this example the responding entity is an actor and is sending the reply message to the `REPLY` mailbox (line 28).

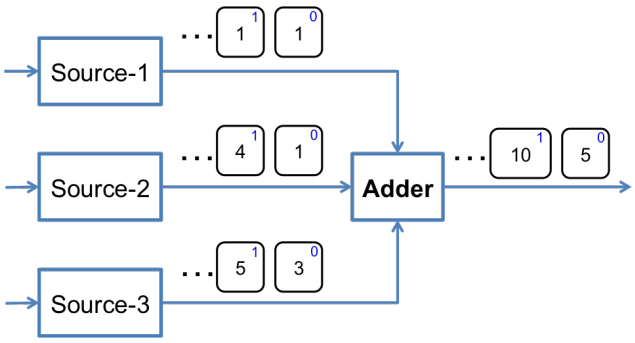

Figure 4: Actor network simulating a join pattern. `Source-1`, `Source-2`, and `Source-3` are producers for data streams. The `Adder` actor aggregates data items from each of the three sources and sums them up.

The order of processing of messages is not guaranteed on the sender Actors, which makes implementing this protocol difficult. Further complexity is added when we need to keep a track of all the *in-flight* messages from various sequence numbers. To aggregate the results from the sources we also need to tag messages with source and sequence number. Only when messages from all the source Actors for the oldest sequence number is received by the aggregator Actor, it can reduce the items into a single value. To avoid any memory leaks the aggregator needs to remove all the processed

sequence numbers.

Akka provides support for the aggregator pattern that allows `match` patterns to be dynamically added to and remove 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.

Solution: Join Pattern with Selectors.

In an actor based solution, we need to tag the messages with source and sequence numbers to support the join pattern. In contrast, using the selector-based approach we need to make sure that the senders send their messages to the correct mailbox of the aggregator. 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 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.

```
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
12         srcMatched += 1;
13         if (srcMatched == numSrcs) {
14           SomeValue joinResult = computeJoin(items);
15
16           nextInChain.send(joinResult);
17           // reset locals
18           items = new int[numSrcs]; srcMatched = 0;
19           // enable all mailboxes for next seq
20            this.mailbox.enableAll();
21       } }}}
22

24 // process items in round-robin order
25 public class AdderRoundRobinOrder(...)
26 extends DistributedSelector{
27   int[] items = new int[numSrcs];
28   int srcMatched = 0;
29   // expect item from first source
30   this.mailbox.disableAllExcept(0);
31   public void process(theMsg: AnyRef) {
32     if(theMsg instanceof ItemMessage){
33       ItemMessage im = new ItemMessage(theMsg);
34         items(im.sourceId) = im.intValue();
35         // disable the current mailbox
36          this.mailbox.disable(im.sourceId);
37         srcMatched += 1;
38
39         if (srcMatched == numSrcs) {
40             SomeValue joinResult = computeJoin(items)↩
                    ;
41             nextInChain.send(joinResult);

43             // reset locals
44             items = new int[numSrcs]; srcMatched = 0;↩

45         }
46         //enable round-robin mailbox for next seq
47                 this.mailbox.get(srcMatched).enable()↩
                ;
48     } } }
```

Figure 5: Using Selectors to solve the Join Pattern problem of Figure 4. The aggregator selector versions (`Adder Any Order` and `Adder Round Robin`) maintain one mailbox for each source. For simplicity we assume sources are identified by consecutive integers starting at 0.

For the first approach (see Figure 5, lines 1-22), 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 13-20].

For the second approach (see Figure 5, lines 24-47), the aggregator selector disables all mailboxes except the first one, which corresponds to messages from the first source[line 30]. As each message is received by a mailbox, that mailbox is disabled and next mailbox is enabled in a round-robin fashion[line 36]. 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 39-44]. The first mailbox, corresponding to the first source is then enabled to process items for the next sequence number.

2.2.3 *Producer-Consumer with Bounded Buffer*

A classic example of a multi-process synchronization problem is the producer-consumer model with bounded buffer [9, 24], where the producer pushes work into the buffer as work is produced and the consumer pulls work from the buffer when they are ready to execute. To model this problem as an Actor-based system, we model the producer, consumer, and the buffer as actors.

```
1  public class BufferSelector extends ↩
       DeclarativeSelector {
2   public void registerGuards() {
3    // disable producer msgs if buffer might overflow
4    guard(MBX_PRODUCER,
5     (theMsg) -> dataBuffer.size() < thresholdSize)
6    // disable consumer msgs when buffer empty
7    guard(MBX_CONSUMER,
8     (theMsg) -> !dataBuffer.isEmpty())
9   }
10  public void doProcess(MessageType theMsg) {
11    if(theMsg instanceof ProducerMsg) {
12        ProducerMsg dm = new ProducerMsg(theMsg);
13        // store the data in the buffer
14        dataBuffer.add(dm);
15        // request producer to produce next data
16        dm.producer.send(ProduceDataMsg.ONLY);
17    }else if(theMsg instance of ConsumerMsg) {
18        ConsumerMsg cm = new ComsumerMsg(theMsg);
19        // send data item to consumer
20        cm.consumer.send(dataBuffer.poll());
21        this.tryExit();
22    }else if(theMsg instance of ProdExitMsg){
23        numTerminatedProducers += 1;
24        this.tryExit();
25    } } }
```

Figure 6: Using Selectors to solve solve the Producer-Consumer with Bounded-Buffer Pattern. The Buffer selector maintains two mailboxes, one to receive messages from producers and another to receive messages from consumers. The use of declarative guards separates the `enable` and `disable` logic of mailboxes into the guard registration method, `registerGuards`.

The bounded buffer actor acts as an intermediary which needs to keep track of these scenarios: *a*) whether the data buffer is empty or full, *b*) when the buffer is empty, and the consumer requests work, then the consumer is placed in a queue until work is available, *c*) when producers are ready to produce data, and the buffer is full, the producer is placed in a queue until the buffer is empty, and finally *d*) notify the producer when the buffer is ready, and more work can be pushed into the buffer. Additional complexity is observed as the buffer actor also needs to maintain queues for the available producers and consumers as there are no ways to disable

processing of particular messages. Pattern matching can be used to implement this scenario, but it is expensive as one would need to search for the next message in the mailbox to be processed.

Solution: Producer-Consumer with Selectors.

Using the Selector-based model we can model the buffer as a selector and the producer and consumers as actors. The buffer selector maintains two mailboxes, one to receive messages from producers and the other to receive messages from the consumer. We can disable processing messages from the producers mailbox when the buffer is full, and disable processing messages from the consumers mailbox when the buffer is empty.

Figure 6 presents a solution that uses declarative guards to isolate the message processing logic from the logic to enable or disable mailboxes. This method avoids maintaining separate sets of available producers and consumers in a purely Actor-based model.

3. DISTRIBUTED SELECTOR: INTERFACE

The design of our Distributed Selector (DS) model is based on the Habanero Java Runtime Library (HJlib) [3]. We expand the shared-memory implementation of the Selector Model to achieve remote message passing, remote selector creating and bounded global termination in a transparent manner. The DS model refers to each single HJ runtime instance as a `place`. A physical computing node can serve as a single or multiple places, given each place have its distinct port. In general, selectors are located at the same `place` to show logical affinity and/or to exploit data locality in both communication patterns and logical functionality. Figure 7 shows the DS library class hierarchy.

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

6  public interface ISelector{...}

8  public class SelectorHandle<MessageType> implements ↩
       ISelector, Serializable {
9    private long _UID;
10   public void send(final int mailboxId, final ↩
         MessageType message);
11   public long getUID();
12   public void start();
13 }

15 public abstract class DistributedSelector<MessageType↩
       > extends Selector<MessageType> implements ↩
       ISelector {
16   private SelectorHandle _handle;
17   public final void send(int mailboxId, final ↩
         MessageType message);
18   public final void start();
19 }
```

Figure 7: The HJ Distributed Selector class hierarchy. The `DistributedSelector` class is not accessible to users

Both `hj.distributed.SelectorHandle` and `hj.distributed.DistributedSelector` inherits from the `hj.distributed.ISelector` interface. The `hj.distributed.SelectorHandle` is the single point of access to a Selector object in user programs, while `hj.distributed.DistributedSelector` extends its shared-memory predecessor but remains exclusive to access internally to the package. An user program can use the factory method `hj.distributedHJSelector.newSelector` to obtain a SelectorHandle instance. The factory method abstracts away the difference between creating a selector locally or on remote location by allowing the user to omit the location of the selector to be created. The introduction of `hj.distributed.SelectorHandle` isolates all Selector states to further eliminate any possible state sharing formerly available in the previous version. More importantly, the separation of Selector object and the access handle gives a lightweight vehicle of communicating Selector object information across the distributed system, as well as routing messages when needed. Given the lightweight handle, programmers will not have the need to deal explicitly with the low-level complexities of distributed coordination.

```
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: 5002,
12  },
13  {
14   place : p2
15   hostname:cn35.davinci.rice.edu,
16   port: 5002,
17  }
18  ]
19 }
```

Figure 8: Sample configuration file, nodes are on Rice University's DAVinCI cluster.

The runtime also features automatic system bootstrap and termination. To set up the DS system, users provide a configuration file (see Figure 8) in which the IP addresses (or host names) 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. The `init` keyword specifies the bootstrap master node, while the `remote` keyword indicates other predefined places in the bootstrap. The runtime reads information from the configuration file and boots up the system. By making sure the master node have the program executable and SSH access to all places specified in the configuration file, the runtime stages all executable on all `remote` nodes and initiates the bootstrap sequence. The runtime will exit the program after all user created selectors have safely terminated. The current implementation requires user to set the username and password for remote hosts as environment variables. The runtime can be extended to support more configurations like memory usage limit and thread count in the bootstrap config files.

4. DISTRIBUTED SELECTOR: WORKFLOW

In this section, we give an overview of the runtime system design and implementation, and the subsections are divided into *system structure* (Section 4.1), *initialization*(Section 4.2), *communication*(Section 4.3), and *termination* (Section 4.4).

4.1 System Structure

Figure 9 shows the decomposition of the Distributed Selector system[1]. Each node is represented as a *place* [15]. A selector runtime system on one node (*place*) consists of multiple user-defined selectors, and two service actors: System Actor and Proxy Actor, and multiple user-defined selectors. Section 4.2 explains how these *actors* are initialized. The user can choose to denote a specific *place* in the configuration file as the *Master Node*.

[1]In general, there can be more than one place per physical node.

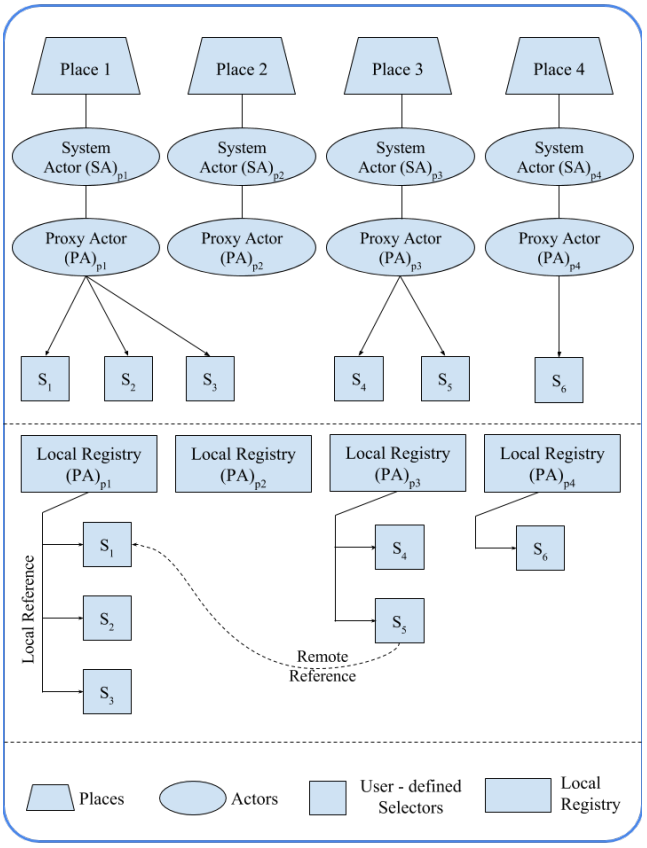

Figure 9: The Distributed Selector System

In this diagram, the *distributed selector system* refers to the system on which an HJDS program is executed. It contains a collection of computing nodes referred to as *places*. *place* refers to an individual HJ runtime instance. This concept is inherited from the *place* abstraction in HJlib to describe affinity among selector objects [3]. Usually, one physical computing node will be denoted as a single *place*, but the runtime puts no restriction on the number of places one computing node may have. When a physical computing node contains multiple *places*, each *place* needs a distinct port for message passing (see Figure 8). The term *selector system* refers to the HJ runtime instance on a single *place*. It contains the System Actor and the Proxy Actor. The System Actor is the service actor that maintains the internal state of its Selector System, as well as communicating such information to the rest of the distributed system. The System Actor on a Master Node maintains the internal state of the entire Distributed Selector system. The System Actor is also responsible for managing the system termination process for the *Global Finish Scope*.

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 user-end view of the Selector System includes instances of local user-defined selectors throughout the program execution as in [15]. Each Proxy Actor maintains a local registry of all the selectors that are located in the same *place* for easy communication between local selectors. Any message to a non-local selector will be sent to the Proxy Actor of the remote selector (whose location encoded in the selector handle), and the Proxy Actor forwards the message directly to local selector instances.

In the distributed selector 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 DS program, including the initiation and termination of the distributed system. Support for multiple master nodes for increased fault tolerance and scalability is a subject of our in-progress work.

The system defines its *Global Finish Scope* as the enclosing join operation around the user program. Derived from the `finish` construct in AFM [3, 14], 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 4.4.

In the user-end view of the distributed selector system, a *User-defined Selector* refers to any DS objects created through user code. 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.

4.2 Initialization and Bootstrap

The HJ runtime treats each user program using DS as a single distributed system, instead of the usual practice of having server daemons on each computing node to host user programs, although the users are not limited to a monolithic approach in their programs and may model each DS program as a single service that can be easily integrated into a larger system. In a single DS program, each computing node is set up with a configuration file (as discussed in Section 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 work-in-progress. 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 computation. Figure 10 demonstrates the bootstrap process in a distributed selector program.

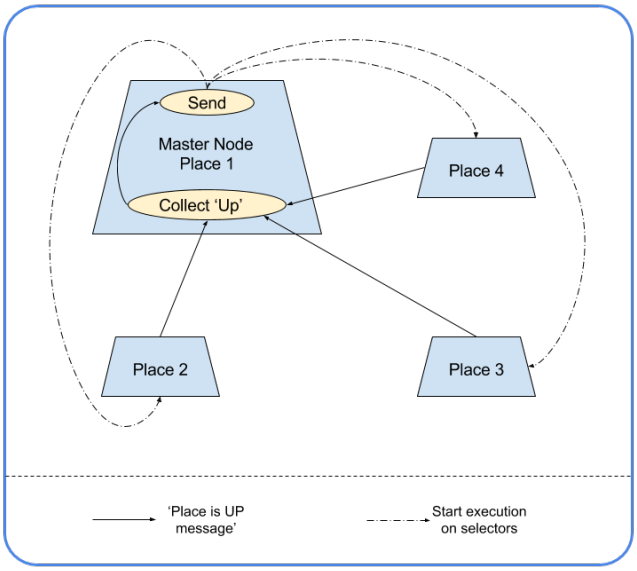

Figure 10: Bootstrap process of a distributed selector program.

Upon bootstrap, the System Actor on Master Node obtains information for all other *places* in the system from the configuration file. The System Actor logs into each remote location through SSH, and starts up an idle HJ selector system there. The system actors on each *place* will identify the Master Node from the configura-

tion file. When initialized successfully, these system actors send periodical heartbeat messages to the Master Node to indicate its 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 each place is correctly setup in accordance with the HJ DS system requirement. This choice does not affect the automated global termination of the system.

4.3 Communication among Selectors

The HJ Distributed Selector interface provides users with a SelectorHandle as the access point to a selector object. To send SelectorHandles across the network; they are designed to be lightweight. It contains a globally unique identifier for 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. A selector object's, globally unique identifier is currently a 32-bit integer that encodes three pieces of information: *1*) an 8-bit value encoding the *place p* on which the selector is created; *2*) an 8-bit value encoding the *place q* on which the selector instance resides; and *3*) a 16-bit integer value representing a unique identifier for the selector on *p*.

We use the Kryo serialization framework [6], which has been shown to be faster than the Java serializer [5]. As a Java–oriented framework, it is better suited for our purpose than other high-performance serialization tools such as Google's Protocol Buffers, Apache Avro, or Apache Thrift, which works across multiple languages and platforms and have more restrictions on the data that can be sent [4]. In theory, a selector can send and receive any message that implements `java.serializable`. Our system does not limit the message types that could be sent across the selectors. However, it will be up to the user's discretion to decide whether to include objects that may result in massive data transfer.

Figure 11 shows how the Proxy Actors 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: *a*) 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. *b*) If local registry doesn't have an entry yet, then the Proxy Actor buffers any message to the non-existing selector. *c*) If the destination *place* is remote then the Proxy Actor forwards the message to that specified *place*. The Proxy Actor at the destination *place* will further use the local registry to forward the message to the specific selector. The topology of the selector system is tightly coupled (i.e., every *place* has a symbolic link to every other *place*).

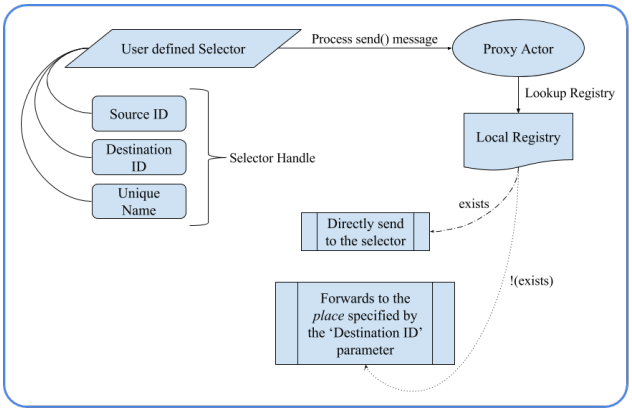

Figure 11: The Proxy Actor helps forward messages between local selectors and selectors across *places*.

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

1. 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.

2. 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*.

4.4 Global Termination

The DS program waits for all the tasks 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*. Each stage is discussed below in detail:

- **Stage 1:** A System Actor detects its place to be *idle* if its local user-defined selectors have all safely been safely 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*.

- **Stage 2:** 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. 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*. 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 cancelling 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 3:** In the final stage of the global termination process, the Master Node assumes all places are ready to exit, and sends a termination signal down the place-ring.

  Each place shuts down after forwarding the message to the next place. The system gracefully exits by terminating each *place* and finally the bootstrap *place*. During this stage, every node is either in *Stage 2*, or it has been shut down.

5. MICRO-BENCHMARKS

Selectors can act naturally as a programming primitive in distributed setting, and more efficiently support the aforementioned coordination patterns than with an actor model explicitly implementing multiple guarded mailboxes [15]. 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 [13]. The benchmarks were run on a 12 core, 2.8GHz Westmere nodes with 48GB of RAM per node (4 GB per core), running Red Hat (RHEL 6.5). For benchmarking we use the number of workers equal to twelve *times* the number of nodes. On each node equal number of selectors are created. Each benchmark was run 20 times, and we report the mean and the best execution times across these runs for a given number of nodes.

The selected benchmarks uses 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 DS system in all benchmarks.

Trapezoidal Approximation

The benchmark approximates the integral function over an interval $[a, b]$ by using the trapezoidal approximation [2, 23]. In the benchmark, we approximate the integral of the function:

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

It achieves parallelism by dividing the approximation into several intervals. The approximation is calculated in a master-worker style parallelism. Each worker is a selector that (*user-defined*) computes the integral approximation in parallel and send their results back to the master selector. The master selector then collects the results from all the worker selectors, adds them up, and displays the final result.

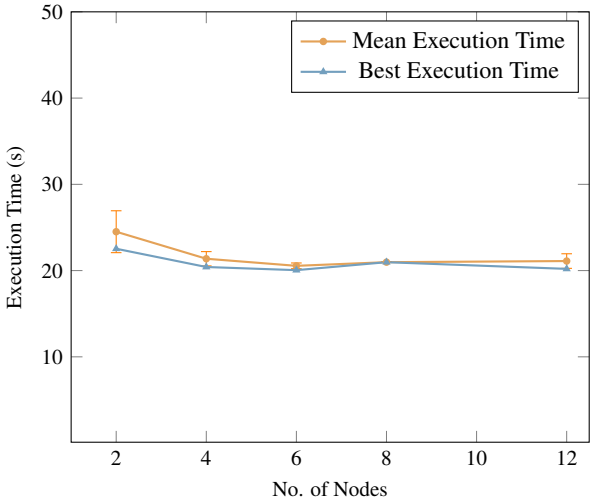

Figure 12: Trapezoidal Approximation : Shows weak scaling computing approximation with 10_000_000 pieces to calculate for each worker. Number of workers per node is constant (12), and as the number of nodes increases we increase the total number of pieces to keep the amount of work on each node constant. Mean Execution time in milliseconds from 20 iterations. The error bar on mean execution time plot shows the standard deviation of the 20 iterations.

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

Precise Pi Computation

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

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

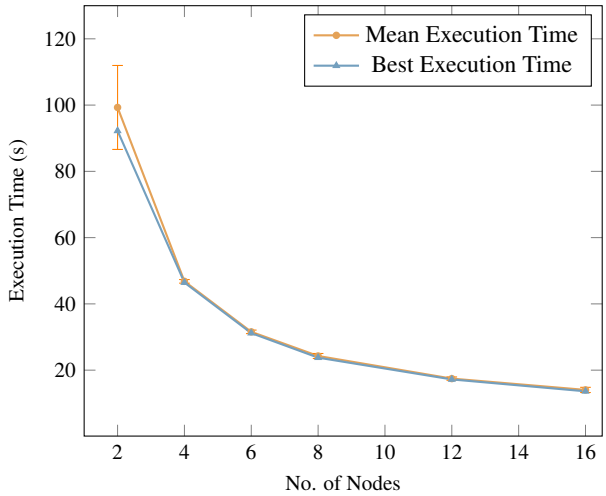

Figure 13: Precise Pi : Shows strong scaling for the calculation of Pi with a precision of 80_000. Number of workers per node is constant (12). Work is evenly distributed among all selectors. Mean Execution time in milliseconds from 20 iterations. The error bar on mean execution time plot shows the standard deviation of the 20 iterations.

Similar to the Trapezoidal Approximation, the Precise Pi Computation represents a master-worker style parallelism. In this scenario, however, the amount of communication between the master and its workers are much more frequent, though still with a small message body. In the Precise Pi 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.

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 13. 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.

NQueens First K Solutions

The NQueens 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. The 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.

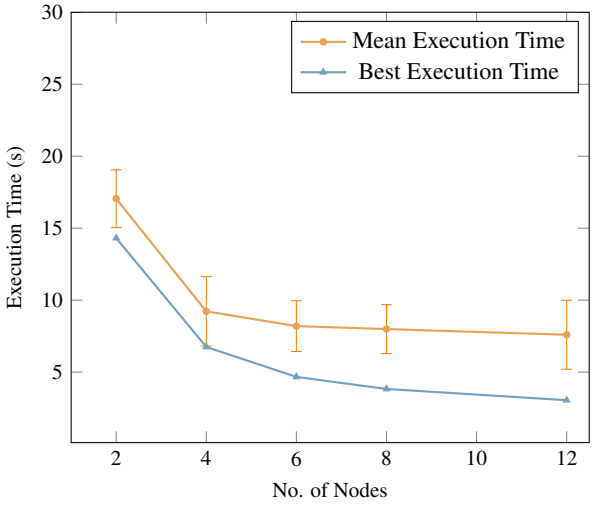

Figure 14: NQueens : Shows strong scaling of computing the NQueens problem on a 17×17 board using the described algorithm. Workers compute solutions sequentially when given a partial solution with six placed queens. The number of workers per node is constant (12). The solution limit is set to 1_477_251, which is a tenth of the size of a complete solution set. Mean Execution time in milliseconds 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 more complicated interaction pattern between the master and its workers than with the previous two, where either the amount of message exchange always stay small or increase linearly with the size of the system. In this algorithm, the master node becomes a bottleneck when the number of workers increases, as all communication about partial results route 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 14 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.

The three micro-benchmarks show how the HJ DS runtime perform under different message exchange patterns. With the Trapezoidal Approximation benchmark, where message exchange grows scarcely with increased number of places in the system, we show that computation resources can be exploited without much overhead in its steady weak scaling. With the Precise Pi 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 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 all three benchmarks take an average of 20ms, consisting of less than 0.001% of program execution time and confirms that the major limitation on scalability comes from the communication patterns.

The adaptation of the micro-benchmarks from their shared-memory counter parts only involved minimal change in the code base. The DS implementation PiPrecision and Trapezoidal Approximation display 5 and 6 lines respectively in difference of source code attributed to API change, and both have 2 extra lines of code attributed to distributed system startup call. The ease to convert between shared-memory and distributed versions of the application also proves useful for debugging: programmers can debug with the behavior of a shared-memory version of a selector program.

6. RELATED WORK

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* [22]. 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 [21]. 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 HJ Distributed Selector, 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 [15].

Microsoft Orleans is an open-source.NET framework built at Microsoft Research that specifically targets actor-based distributed applications [17]. The project was initially developed to aid the development of streaming applications focusing on high scalability

and low latency. With an industrial oriented approach, the project focuses on programmability as well as the availability of service. It implements its serialization layer to maximize low-latency and can occasionally compromises consistency throughout the system. The *Orleans* project contributes an elastic, scalable, and simplified message-passing implementation targeting larger cloud systems, while conforming to a traditional view of the Actor Model.

SALSA is a Java-based actor programming language developed at RPI [23]. 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 sub-components at runtime, SALSA provides daemon programs for host universal actors named *Theaters* and supports universal actor with the *Naming Server*. 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 the HJ distributed selectors 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. 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 HJ Distributed Selectors support selectors with multiple mailboxes in a distributed setting. We were unable to include any performance comparisons with SALSA in this paper because of running into JVM OutOfMemory errors for the SALSA versions of the benchmarks when using the same conïňĄgurations that we use for HJ Distributed Selectors.

The **Communicating Sequential Processes (CSP)** concurrency model proposed by Tony Hoare in 1978 [10], which resembles many key features found in the Actor Model, has also influenced many current distributed programming languages and frameworks whose communication pattern relies on message passing. The CSP model influences the design of both *Go* [7] and *Limbo* [12]. The traditional CSP view is similar to the AM with its emphasis on independent sequential processes only coordinate through message passing. However, the traditional CSP view features anonymous processes while AM provides identifiable actors. The fundamental difference is that CSP focuses on synchronous message passing, while the AM completely decouples the process and all messages are sent asynchronously. The synchronous nature of CSP messages also influenced the use of *explicit* channels in *Go*. Messages are not sent to processes directly, but to named channels that use buffered communication that ensures message delivery. In the DS runtime, all messages are delivered to named selectors asynchronously (with implicit channels), and only system-level messages involving remote selector creation or explicit selector termination will have guaranteed delivery. While asynchronous message passing could be emulated by buffered channels, these explicit channels pose programmers difficulty on loosely coupled distributed systems, where locality needs to be addressed explicitly.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented a Distributed Selector (DS) model, a novel programming model for shared and distributed memory parallel applications. DS model allows programmers to focus on implementing the algorithm for solving the problem their application is trying to solve, without worrying whether their application will run on a shared-memory or distributed-memory system. Our runtime implementation supports Selectors (a strictly more powerful version of Actors) on both shared-memory and distributed-memory systems. This framework provides automated system bootstrap and global termination, unlike any other distributed approaches.

Our experimental evaluation using the Savina benchmark suite shows promising strong scaling results, making a strong case for the DS model as a viable alternative to the existing, much harder to program and port, parallel programming models.

For future work, we will 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. Since Actors represent very light-weight primitives compared to threads, they can be spawned and destroyed with minimal overhead. This makes Actor-model an efficient way to model computation and applications on embedded devices (E.g. any two-way communication application where the sender, the receiver, and the channel can be modeled as Actors and use the message passing paradigm to communicate). Exploring the possibility of using Actors for these kinds of communication on embedded devices is also of future interest. Extensions on the current work is to include dynamic joining and leaving of nodes from the cluster for a better fault tolerance mechanism. We will also study more closely the performance tradeoffs between traditional Actor-based libraries (such as Akka and SALSA) and our DS model implementation.

8. ACKNOWLEDGEMENT

We thank Prof. John Mellor-Crummey for his insightful comments and suggestions on our global termination procedure design. We thank Prof. Carlos Varela and Prof. Travis Desell for their support in understanding SALSA. We are grateful to Max Grossman for his help in our experimental set-up and Prasanth Chatarasi for his valuable feedback on early drafts of this paper.

9. REFERENCES

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

[2] J. Ayres and S. Eisenbach. Stage: Python with Actors. In *Proceedings of IWMSE '09*, pages 25–32, Washington, DC, USA, 2009. IEEE Computer Society.

[3] 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, pages 51–61, New York, NY, USA, 2011. ACM.

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

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

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

[7] I. Google. The Go Programming Language.

[8] 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.

[9] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Commun. ACM*, 17(10):549–557, Oct. 1974.

[10] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

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

[12] V. N. Holdings. The Limbo Programming Language.

[13] 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.

[14] 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, pages 753–772, New York, NY, USA, 2012. ACM.

[15] 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, pages 1–14, New York, NY, USA, 2014. ACM.

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

[17] Microsoft Research. Microsoft Orleans, 2015.

[18] Oracle. Understanding Interaction Patterns, 2011.

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

[20] Typesafe Inc. Actors - Akka Documentation, 2014.

[21] Typesafe Inc. Akka Cluster Documentation, 2014.

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

[23] C. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, Dec. 2001.

[24] Wikipedia, The Free Encyclopedia. Producer Consumer Problem, 2014. [Online; accessed 3-April-2014].