A Performance and Scalability Analysis of Actor Message Passing and Migration in SALSA Lite

Travis Desell
tdesell@cs.und.edu

Department of Computer Science
University of North Dakota

Carlos A. Varela
cvarela@cs.rpi.edu

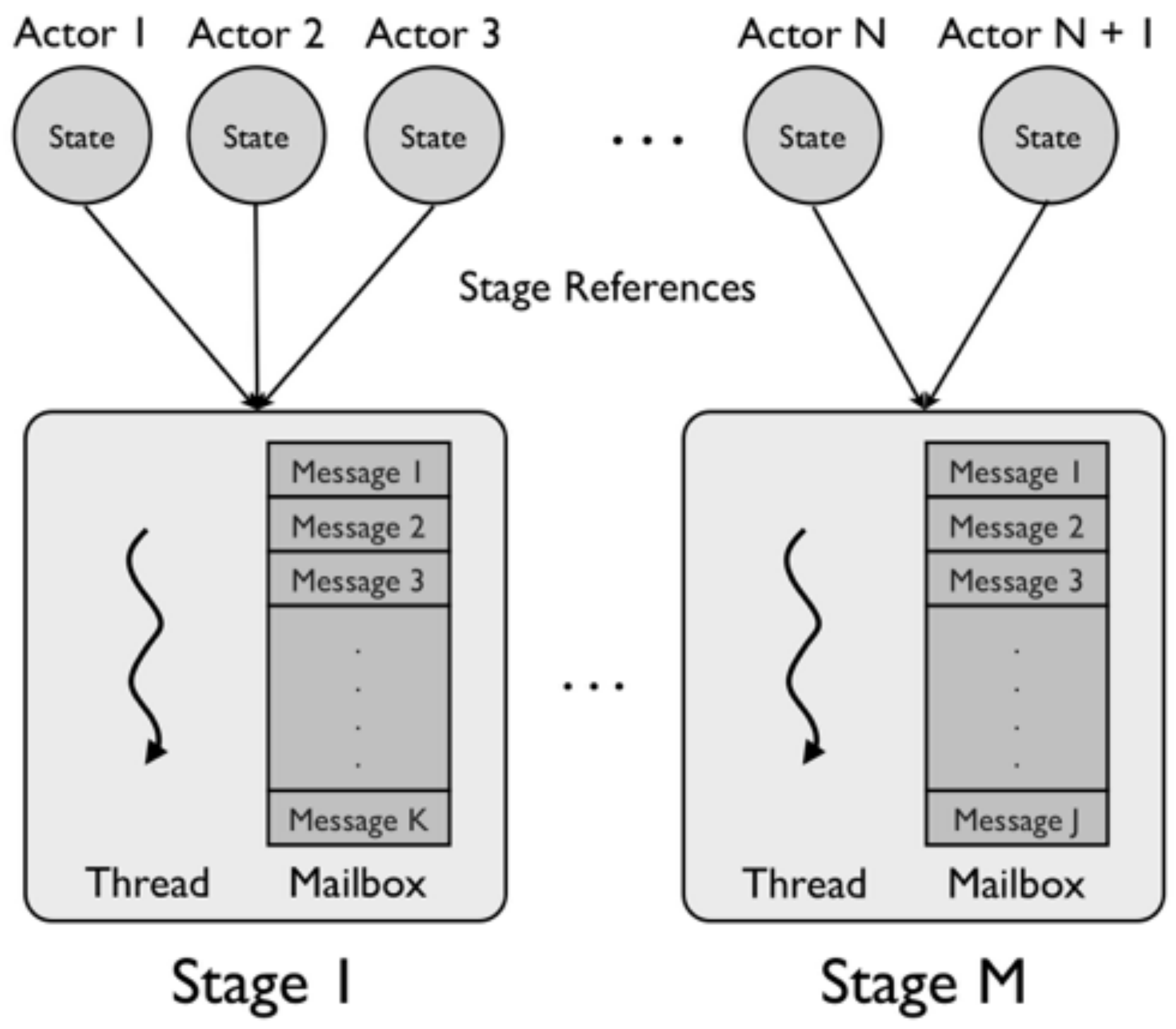
Department of Computer Science
Rennselaer Polytechnic Institute

October 26, 2015
Pittsburgh, Pennslyvania, USA

The 5th International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE! 2015)
Held in conjunction with the ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)

Overview

SALSA Lite Implementation



SALSA Lite uses the concept of *stages* to host lightweight actors. These have many similarities to E's *vats* or SEDA.

Essentially, stages are heavyweight actors, which have a mailbox and process messages one after the other. Each actor is assigned to a stage, and only that stage will invoke messages on that actor.

The default number of stages can be specified dynamically at runtime, and new stages can be created as needed by actors. Actors can either be assigned to a random stage (by default) or select the stage they are created on.

Assigning Actors to Stages

```
//create on a default stage
MyActor a = new MyActor();

//create b on a's stage
MyActor b = new MyActor() on (a);

//create c on stage 3
MyActor c = new MyActor() on (3);

//create d on its own new stage
MyActor d = new MyActor()
    on (StageService.getNewStage());
```

The code above shows the different ways an actor can be created on different stages.

The initial number of stages in a SALSA application can be specified at runtime by using the:

```
-Dnstages=<number of stages>
```

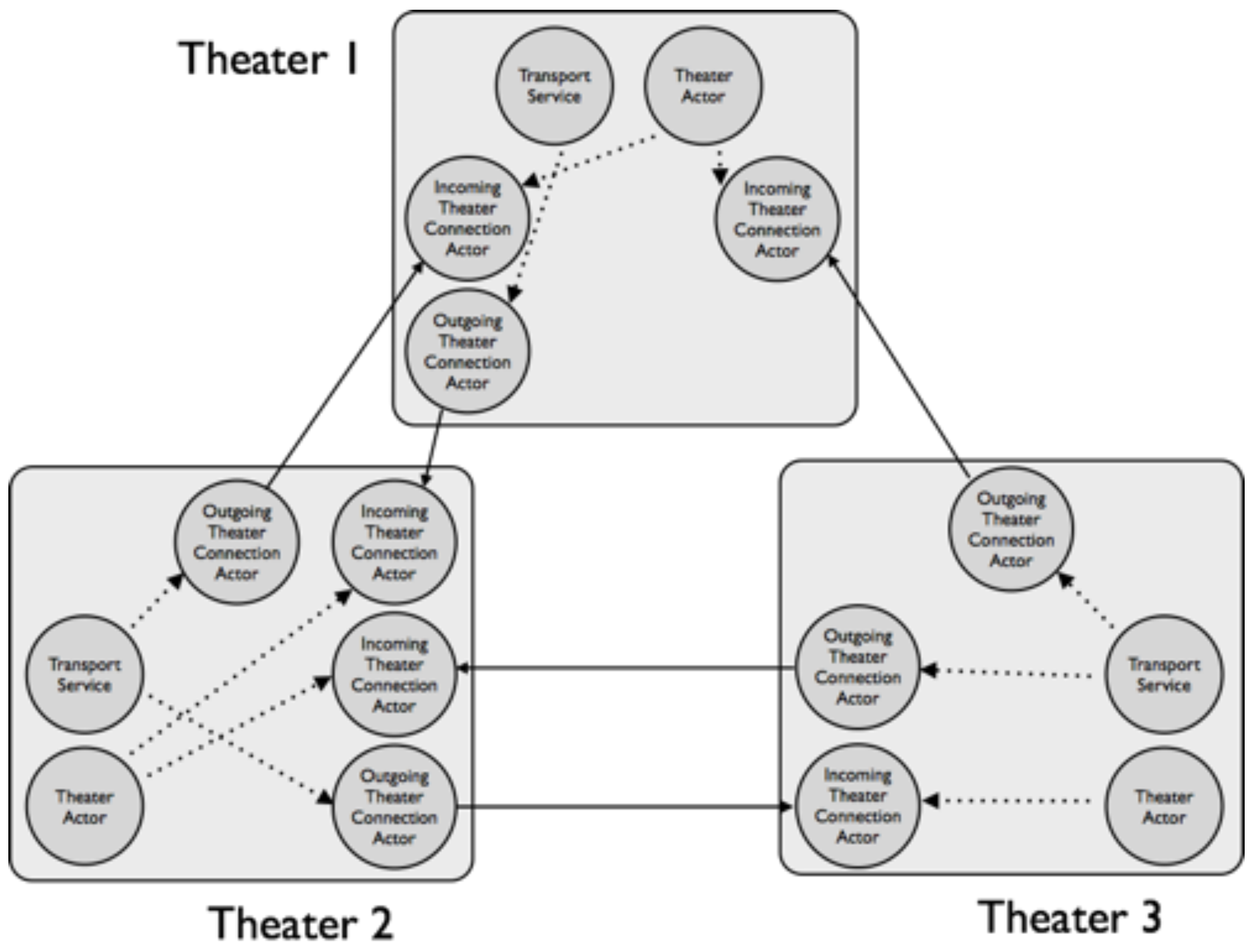
system property.

Eliminating Bottlenecks via Hashing

SALSA Lite uses a series of services which potentially could act as singular bottlenecks as they perform actions (e.g., generating a unique ID for a message) which must be done sequentially and in a thread safe manner.

SALSA Lite overcomes this by allowing multiple similar services to run which are accessed by actors/stages by hashing the actor's stage ID. This minimizes blocking on other stages to access these services. The number of service copies can be specified dynamically at runtime.

Implementation: Theaters


Theater 1
Transport Service
Theater Actor
Incoming Theater Connection Actor
Incoming Theater Connection Actor
Outgoing Theater Connection Actor
Outgoing Theater Connection Actor
Incoming Theater Connection Actor
Transport Service
Incoming Theater Connection Actor
Theater Actor
Outgoing Theater Connection Actor
Theater 2
Outgoing Theater Connection Actor
Outgoing Theater Connection Actor
Transport Service
Incoming Theater Connection Actor
Theater Actor
Theater 3

The runtime environments in which distributed SALSA Lite applications run on are called *Theaters*. Theaters listen on a particular port for incoming connections to other theaters which can be distributed over local area networks or the Internet.

Theaters consist of two main types of actors:

1. OutgoingTheaterConnectionActors
2. IncomingTheaterConnectionActors

Both operate on their own stage to prevent starving other actors during synchronous calls or long data transfers.

IncomingTheaterConnectionActors are spawned by the theater when a connection is established on it's listening port. They repeatedly receive messages (and actors in the case of migration) and forward them on the the appropriate stage to be processed by their target.

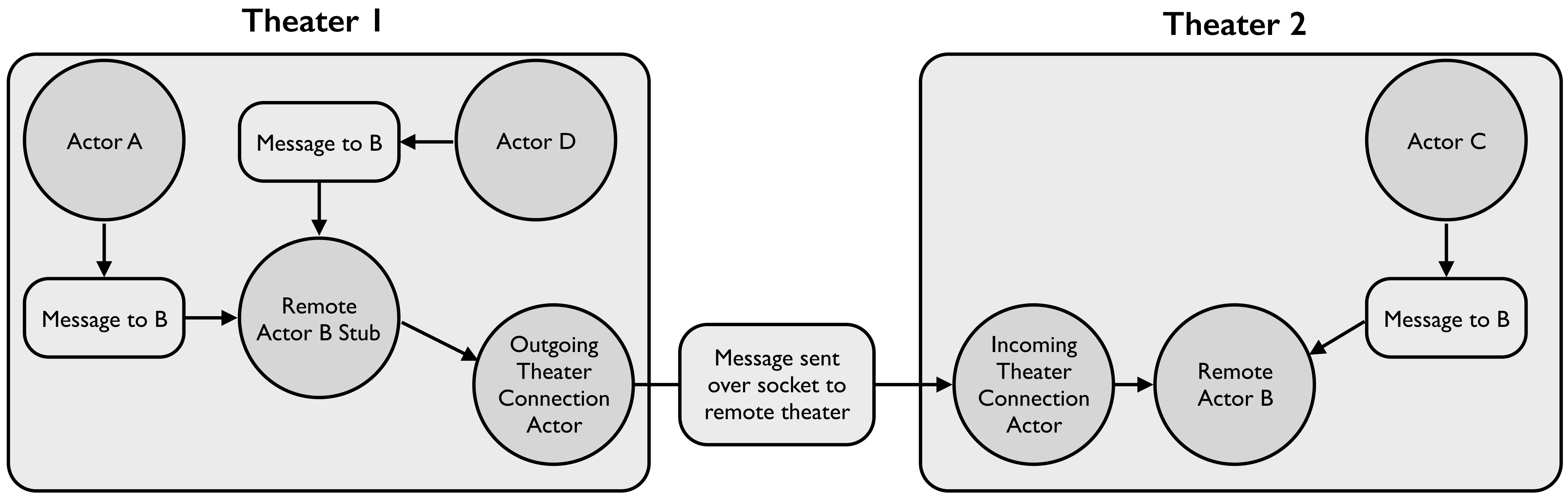
OutgoingTheaterConnectionActors connect to outgoing theaters and then send messages to remote actors over the connection to their corresponding IncomingTheaterConnectionActor.

Implementation: Remote Actors

When an actor holds a reference to a remote actor, that remote actor is either present locally or at a remote theater.

In the case of locally present remote actors, the reference is a reference to the actual actor, and messages are sent identically as if they were being sent to a local actor.

In the remote case, the reference is actually a stub actor which simply places any messages invoked on it by a stage in the mailbox of the appropriate OutgoingTheaterConnectionActor's stage. In this way, no changes were required to be made to the simple and efficient stage runtime of SALSA Lite.

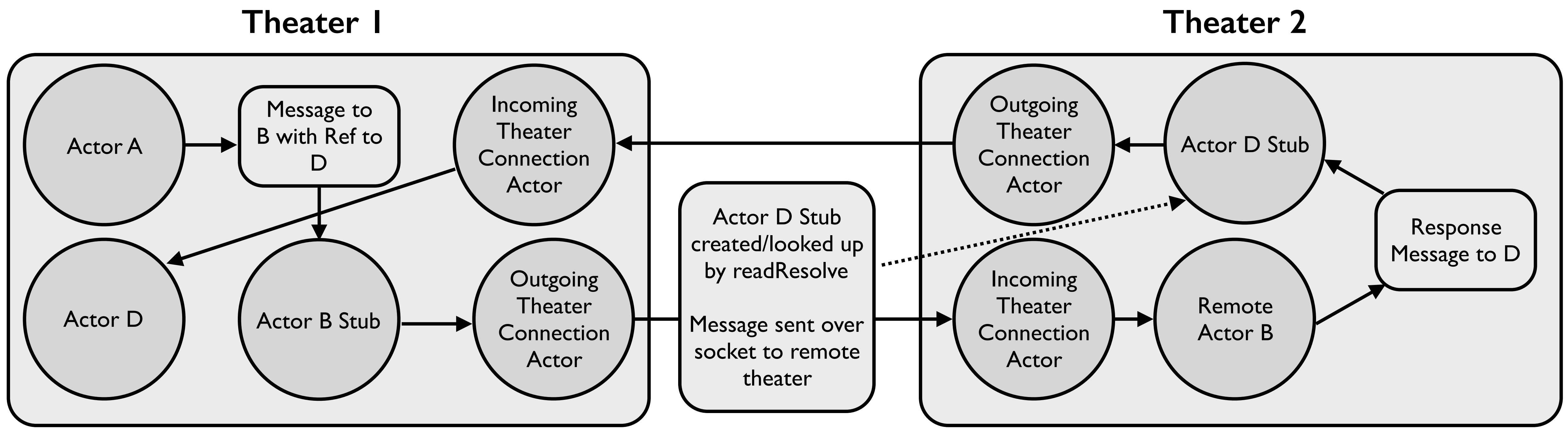

Theater 1
Theater 2
Actor A
Message to B
Actor D
Message to B
Remote Actor B Stub
Outgoing Theater Connection Actor
Message sent over socket to remote theater
Incoming Theater Connection Actor
Remote Actor B
Actor C
Message to B

Implementation: Reference Propagation 1

When a remote actor is created it is placed in a **RemoteActorRegistry**, which is a **HashMap** of remote actor names the respective reference to it's lightweight actor object. In the case of remote actors on other theaters, the registry contains the stub actor instead.

Reference lookup is performed whenever a message or actor is received over the socket of an IncomingTheaterConnectionActor. As opposed to using the standard Java serialization methods, SALSA Lite instead uses the **readResolve** and **writeReplace** serialization methods to instead simply write a minimal object containing the actor's hash code, host, port and name. These are then used to look up the lightweight actor or stub actor in the registry.

If there is no entry in the registry either the remote actor is going to be created here but hasn't yet, or the reference is an unknown stub which can be created from the hash code, host, port and name of the remote actor.



Theater 1
Theater 2
Actor A
Message to B with Ref to D
Incoming Theater Connection Actor
Actor D
Actor B Stub
Outgoing Theater Connection Actor
Actor D Stub created/looked up by readResolve
Message sent over socket to remote theater
Outgoing Theater Connection Actor
Actor D Stub
Incoming Theater Connection Actor
Remote Actor B
Response Message to D

Implementation: Reference Propagation 2

Messages sent to remote theaters can also contain references to local actors. As such, another **LocalActorRegistry** is used to handle the case when these references escape a theater. When a local actor is serialized an entry is placed in the LocalActorRegistry, and stubs can be created/looked up on other theaters by the local actor's hash code, host and port.

This adds no overhead to non-distributed message passing and actor creation as the LocalActorRegistry already existed to prevent serialization of actors when objects sent as message parameters contain references to actors and are copied to enforce state encapsulation.

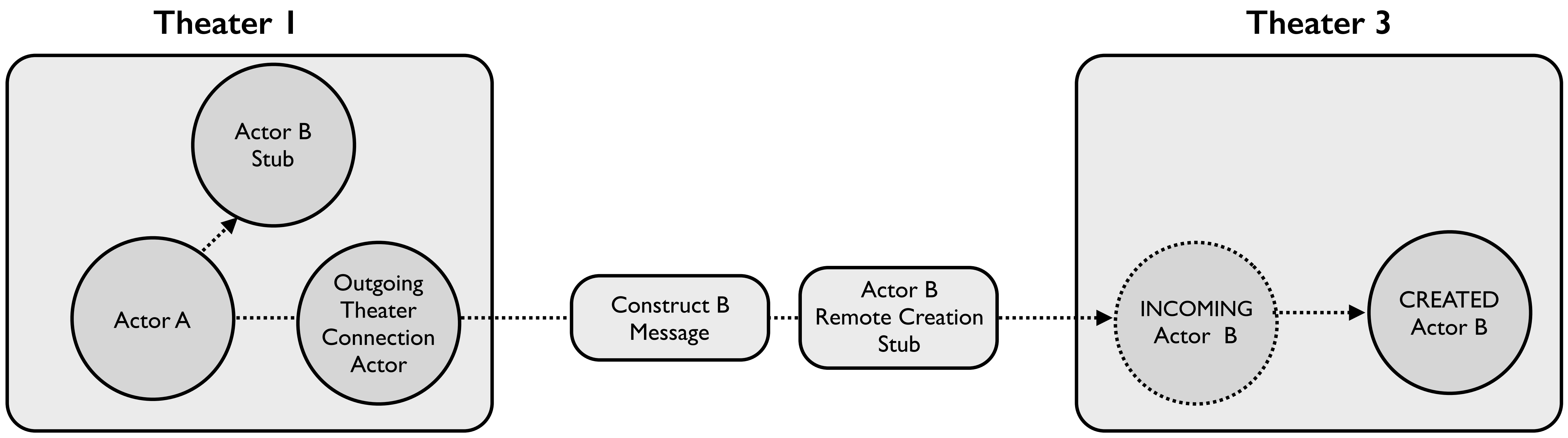
For example, if an ArrayList of actors is passed as an argument to another local actor, the ArrayList should be copied but the actors (and actor references within it) should not. This also uses the **readResolve** and **writeReplace** serialization methods done locally over a fast deep copy stream.

Using this registry scheme is highly important as it prevents unwanted duplication of actors and actor references.

Implementation: Remote Actor Creation 1

The SALSA compiler replaces calls to create new remote actors with a static **construct** method, which returns the actor if it is to be created locally, and forwarding stub otherwise.

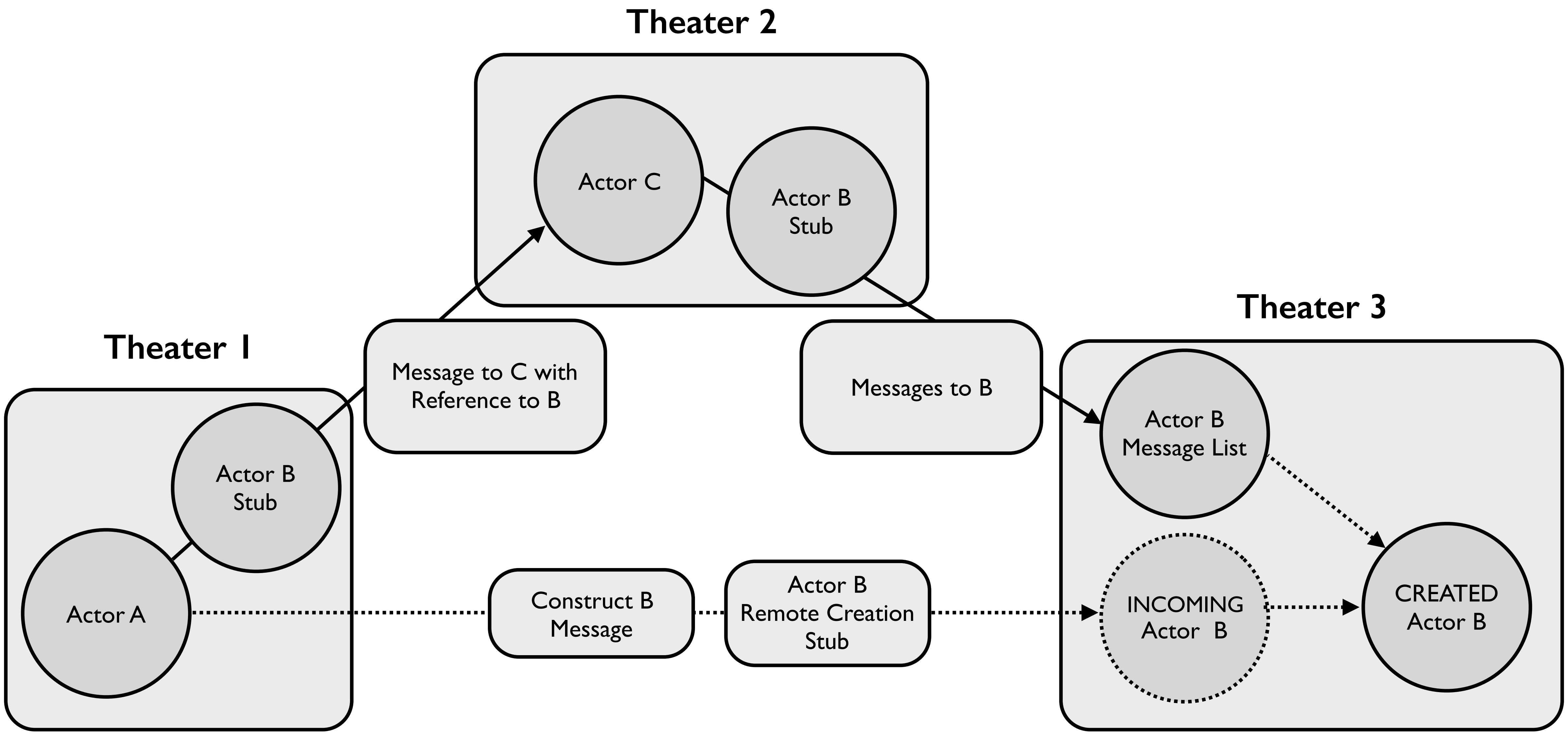
This static method places two messages in the appropriate OutgoingTheaterConnectionActor's mailbox. The first contains a stub for the remote actor to be created, and the second creates a construct method which actually invokes the constructor on the new remote actor. As messages in any actors mailbox are not reordered in SALSA Lite, and the OutgoingTheaterConnectionActor sends these two messages, as well as any subsequent messages in order over the Java socket to the remote IncomingTheaterConnectionActor.



Implementation: Remote Actor Creation 2

It is possible, however, that the returned stub could be sent in a message to an actor at a different theater, which then sends a message to the remote actor whose construct messages haven't yet reached the theater where the actor is to be created.

In this case, when the actor is looked up in the **RemoteActorRegistry** it will not be found even though the host and port specify that it should be at that theater. In this case, an ArrayList is placed in that registry to hold any incoming messages to it. Which are resent after the actor is received and constructed.

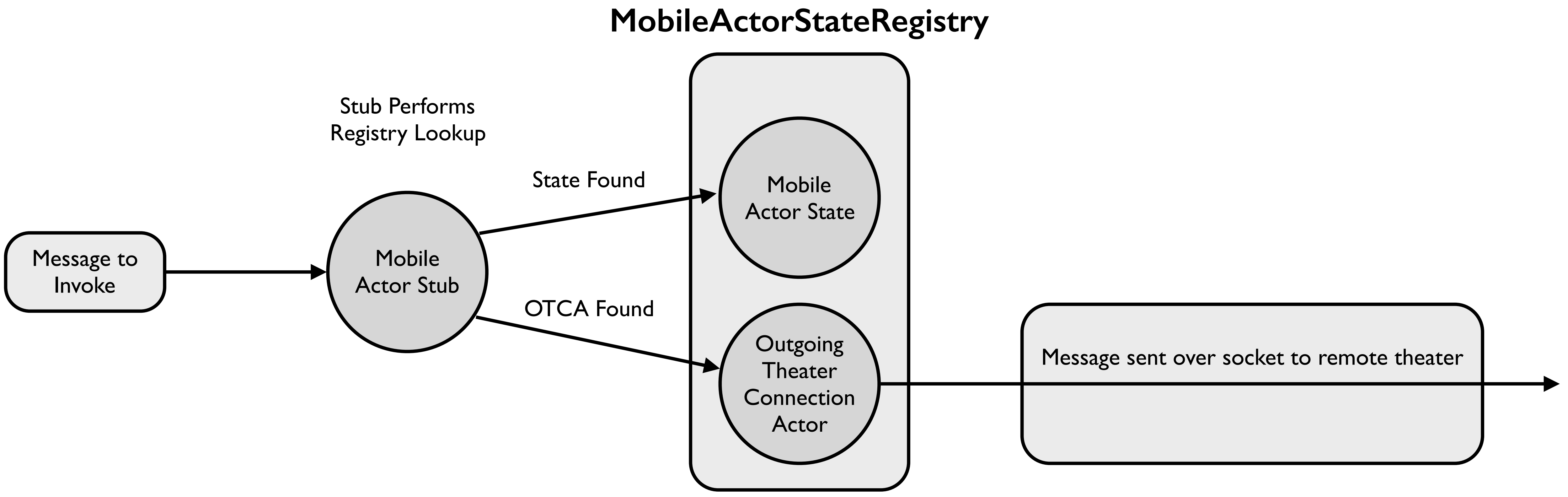


Implementation: Mobile Actors

Whereas local and remote actors can be taken care of with lightweight actor objects and message forwarding stubs, mobile actors always require a stub actor which performs a lookup every time a message is invoked on it to determine if the actor is at the current theater or if it has moved.

This is accomplished with both a **MobileActorRegistry** and **MobileActorStateRegistry**. The **MobileActorRegistry** holds the stub actors which are resolved to when references to the mobile actor are received over a socket. The **MobileActorStateRegistry** holds a reference to the object representing the actor's state when it is present, and a reference to the appropriate OutgoingTheaterConnectionActor when it is not.

When a message is invoked on a mobile actor's stub, it performs a lookup and invokes the message on the actor if it is present, otherwise it places it in the appropriate OutgoingTheaterConnection's mailbox.


MobileActorStateRegistry
Stub Performs Registry Lookup
State Found
OTCA Found
Message to Invoke
Mobile Actor Stub
Mobile Actor State
Outgoing Theater Connection Actor
Message sent over socket to remote theater

Implementation: Migration

The locations of mobile actors are tracked using a **name server**, which is updated whenever a mobile actor is created or migrates. Unlike SALSA 1.5 and earlier, name servers are first class actors in SALSA Lite which are communicated with asynchronously.

When a mobile actor migrates, it sends an UPDATE message to it's corresponding name server. After this the actor's state is placed as a message to the target theater's OutgoingTheaterConnectionActor, and the MobileActorStateRegistry is updated to that theater's OutgoingTheaterConnectionActor. When the actor's is received by the IncomingTheaterConnectionActor, it is placed in that theater's **MobileActorStateRegistry**.

If a message received by an IncomingTheaterConnectionActor has mobile actor as its target, and that mobile actor is not present at the theater, it performs a lookup as to where the actor had migrated using the **MobileActorStateRegistry**. It sends the message on to the theater the actor had migrated to, but also sends an *updateActorLocation* message to the theater actor at the source of the message. It keeps a list of actors it has sent *updateActorLocation* messages to and has not yet heard an acknowledgement back from yet, to prevent spamming the source theater with multiple *updateActorLocation* messages. In this way, as an actor migrates and messages are sent to it, the theaters update their **MobileActorStateRegistry** with references to where the mobile actors have moved to.

Creating and Referencing Distributed Actors

```
//Create a remote actor at the local theater
MyRemoteActor a = new MyRemoteActor()
    called ("a");

//Create a remote actor at a remote theater
MyRemoteActor b = new MyRemoteActor()
     called ("b") at (host, port);

//Create a name server
NameServer ns = new NameServer()
    called ("my_nameserver");

//Create a mobile actor at the local theater
MyMobileActor c = new MyMobileActor()
    called ("c") using (ns);

//Create a mobile actor at a remote theater
MyMobileActor c = new MyMobileActor()
    called ("c") using (ns)
    at (host, port);
```

```
//Reference a remote actor at the local theater
MyRemoteActor a = reference
    MyRemoteActor called ("a");

//Reference a remote actor at a remote theater
MyRemoteActor b = reference
    MyRemoteActor called ("b")
    at (host, port);

//Get a reference to a remote name server
NameServer ns = reference NameServer
    called ("my_nameserver")
    at (host, port);

//Reference a mobile actor registered at
//that name server
token MyMobileActor c = reference
    MyMobileActor called ("c")
    using (ns);
```

Creating and referencing remote and mobile actors is very straightforward.

The **called** keyword specifies the name of the actor, the **at** keyword specifies the host and port, and the **using** keyword specifies the name server tracking a mobile actor.

Actors are referenced in a similar manner, except the **reference** keyword is used instead of **new**.

Result Environments

Initial Results Environment:

Small Beowulf HPC Cluster

Red Hat Enterprise Linux 6.2

4 dual quad core 3.3Ghz E5-2643 Intel processors

56 gigabit Infiniband FDR 1-1 network

64 GB 1600 MHz RAM per node

Java 1.6.0_26

Extended Results Environment:

Mid 2010 Mac Pro

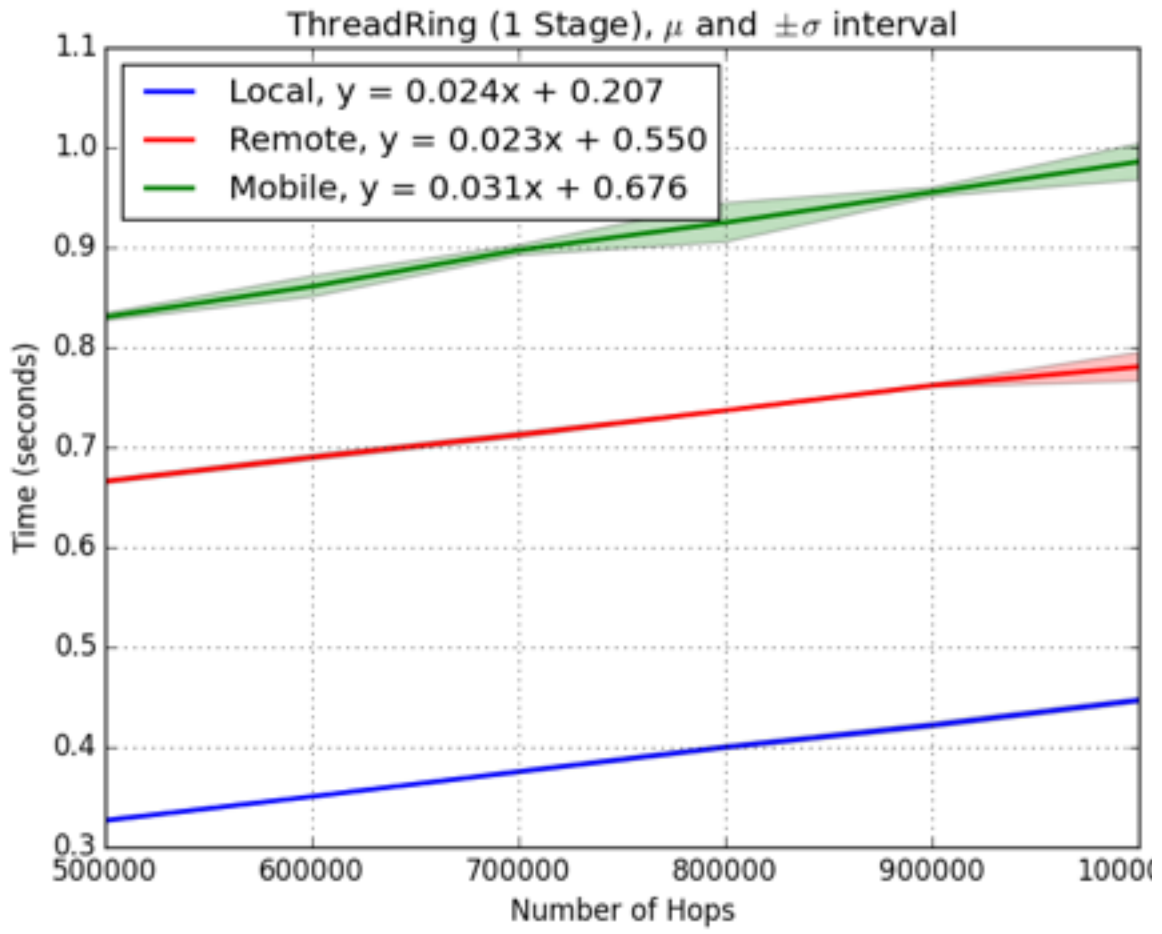
OSX Yosemite 10.10.5

2 2.4 Ghz Quad Core Intel Xeon

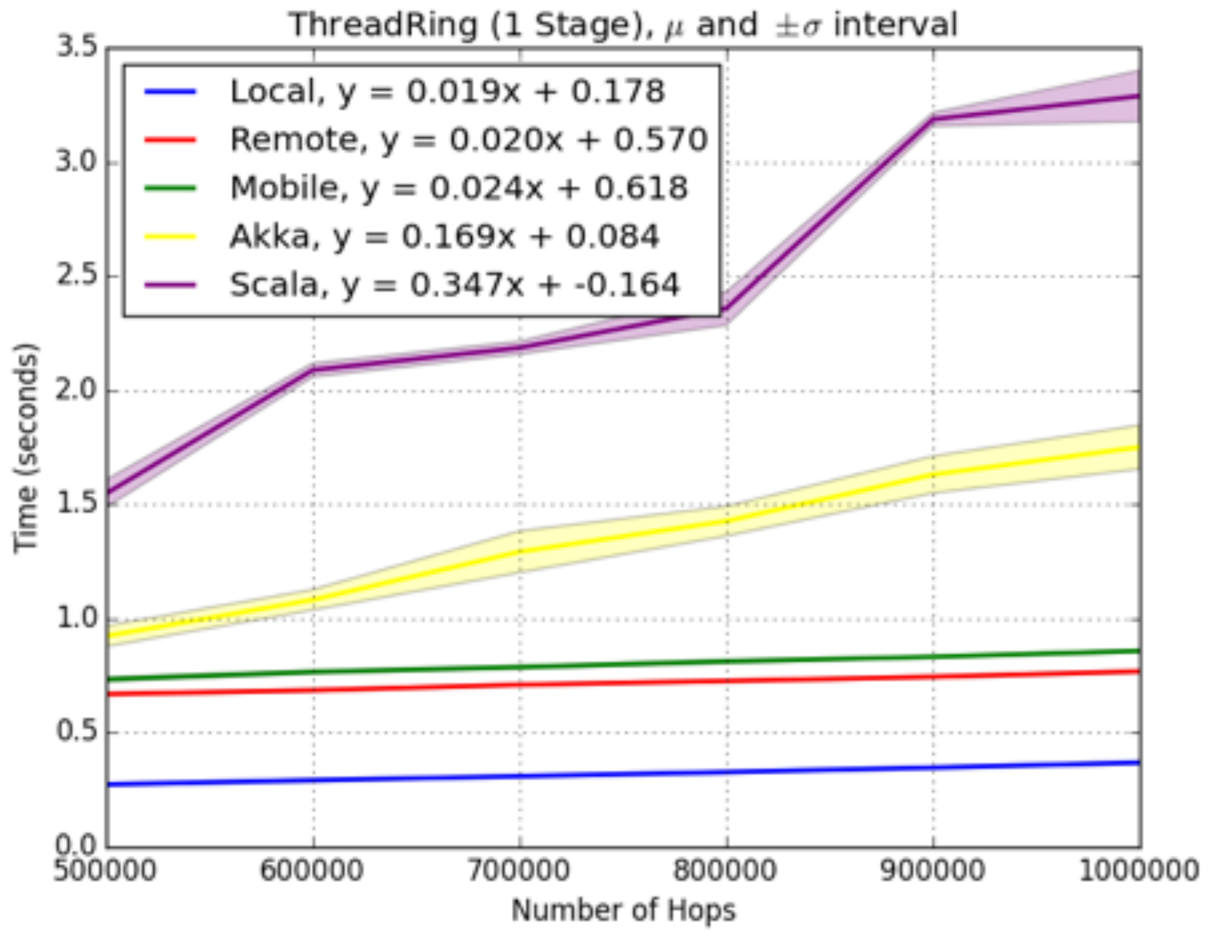
64 GB 1066 Mhz DDR3 EEC RAM

Java 1.8.0_66, Scala 2.11.7, Akka 2.4.0

Thread Ring



Linux cluster, Java 1.6



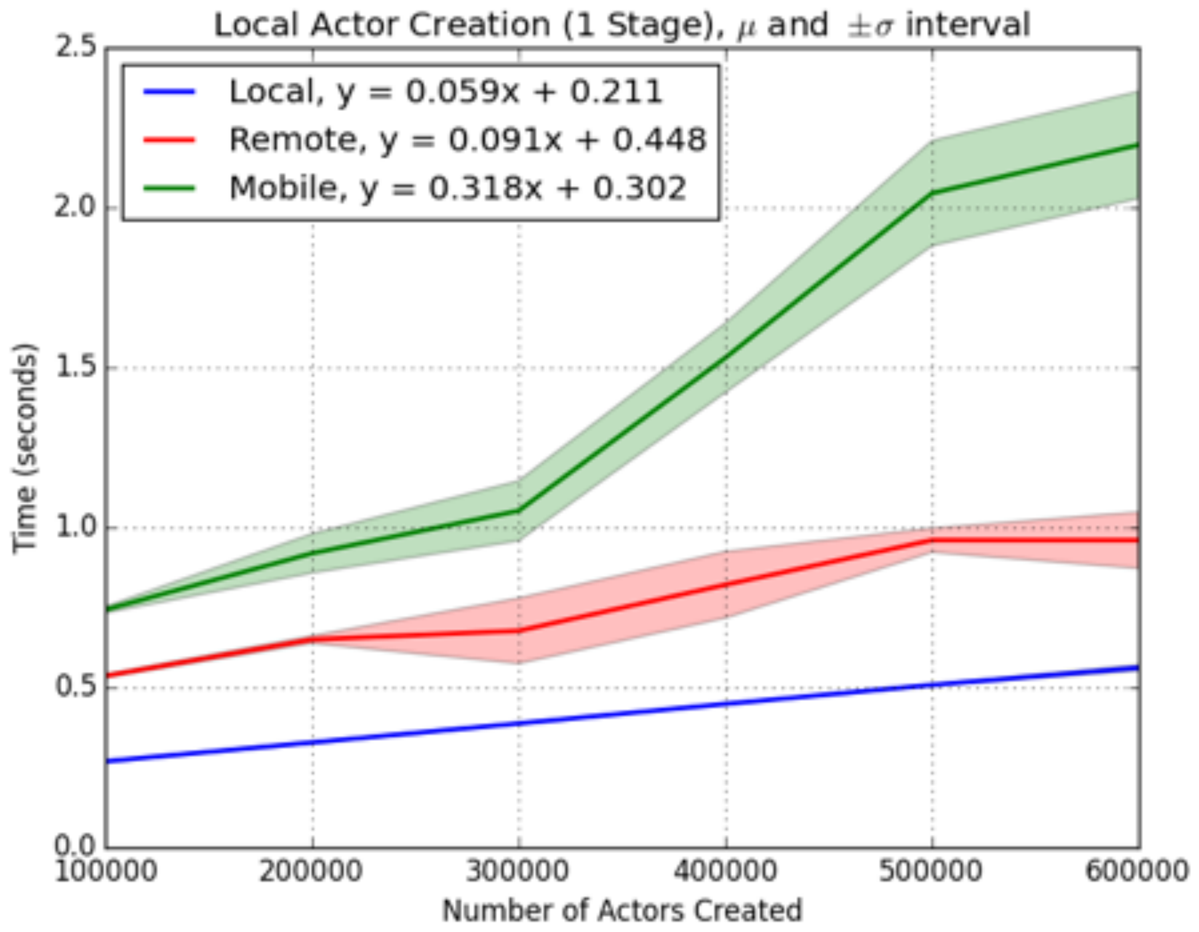
OSX, Java 1.8

Scala and Akka from Savina benchmark suite. Scala and Akka do not include start up time and re-use JVM. SALSA includes start up time and uses new JVMs per run.

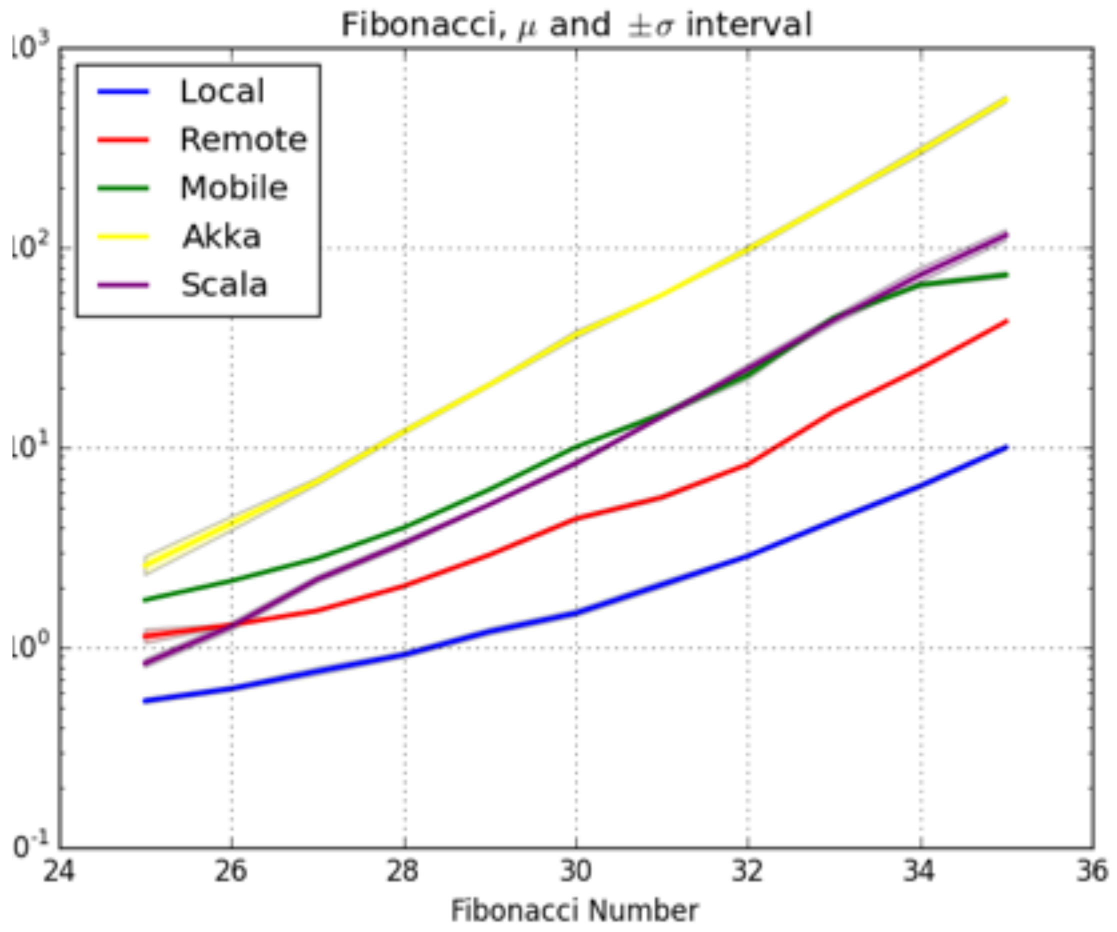
Tests message passing overhead. Mobile actors are approximately 30% slower excluding startup costs.

Negligable overhead (on message passing) for Remote Actors. Startup costs are higher.

Local Actor Creation / Fibonacci



Linux cluster, Java 1.6



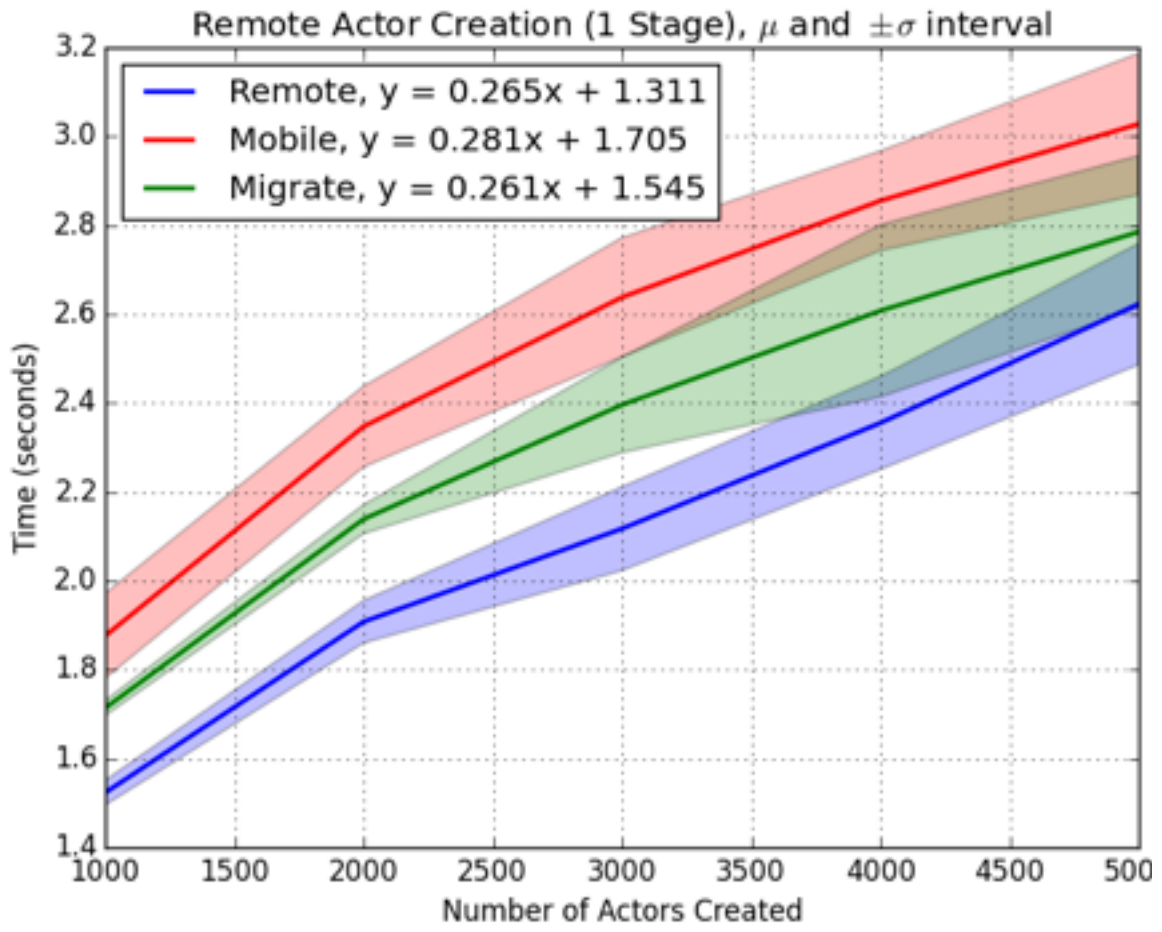
OSX, Java 1.8

Scala and Akka from Savina benchmark suite. Scala and Akka do not include start up time and re-use JVM. SALSA includes start up time and uses new JVMs per run.

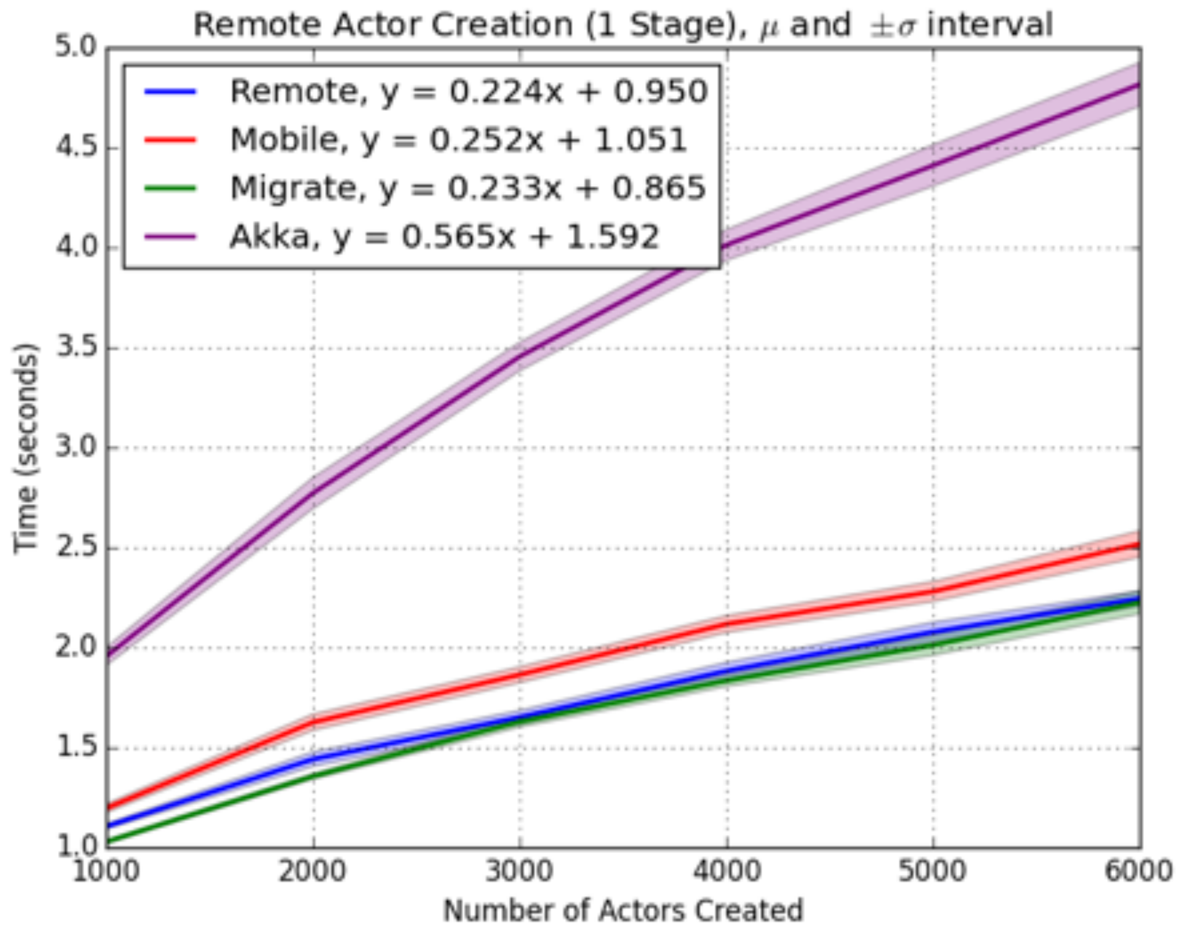
Tests creation of a large number of actors. Note that Fibonacci has a log scale y-axis.

Remote and mobile actors show a fair amount of overhead (54% and 438%) over local actors when being created locally.

Remote Actor Creation



Linux cluster, Java 1.6



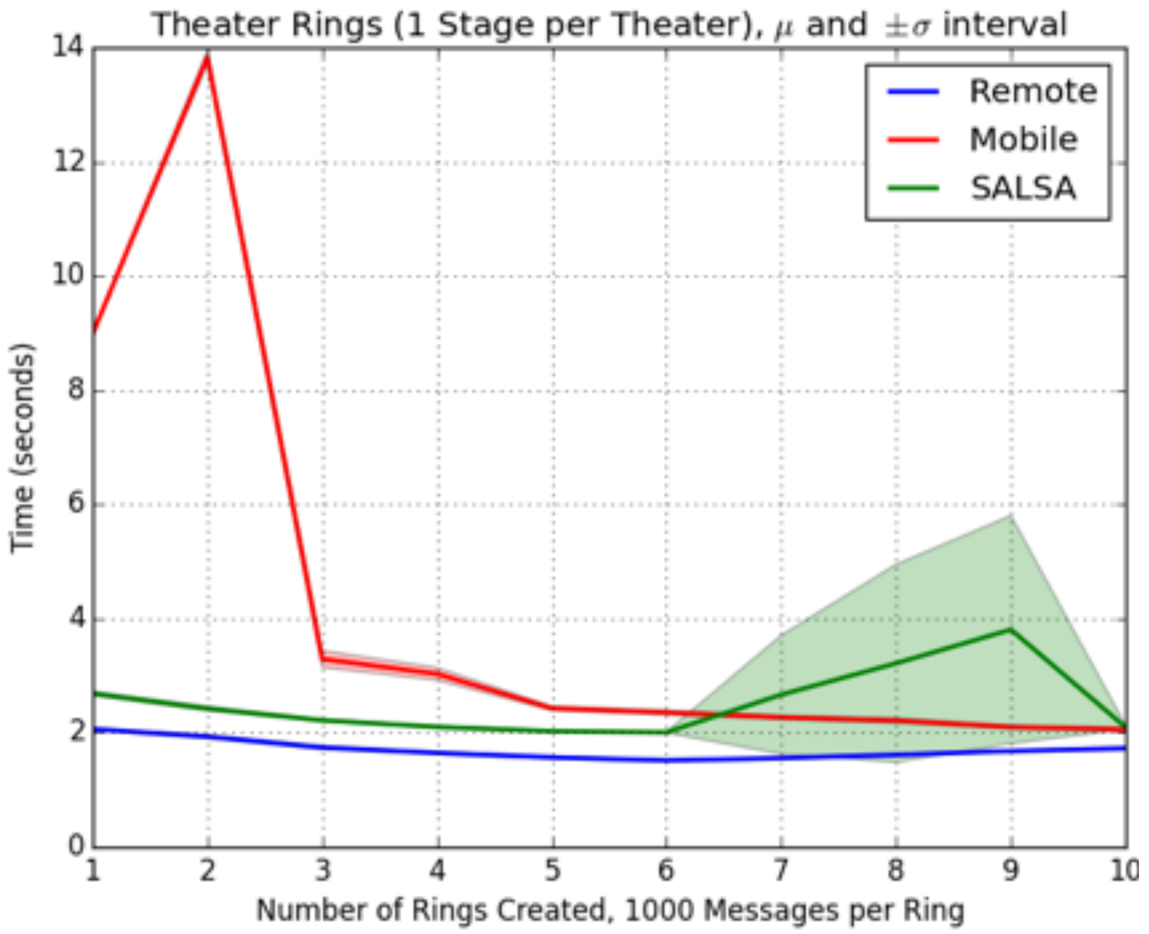
OSX, Java 1.8

Akka does not include start up time, however it does not reuse a JVM. SALSA includes start up time and uses new JVMs per run.

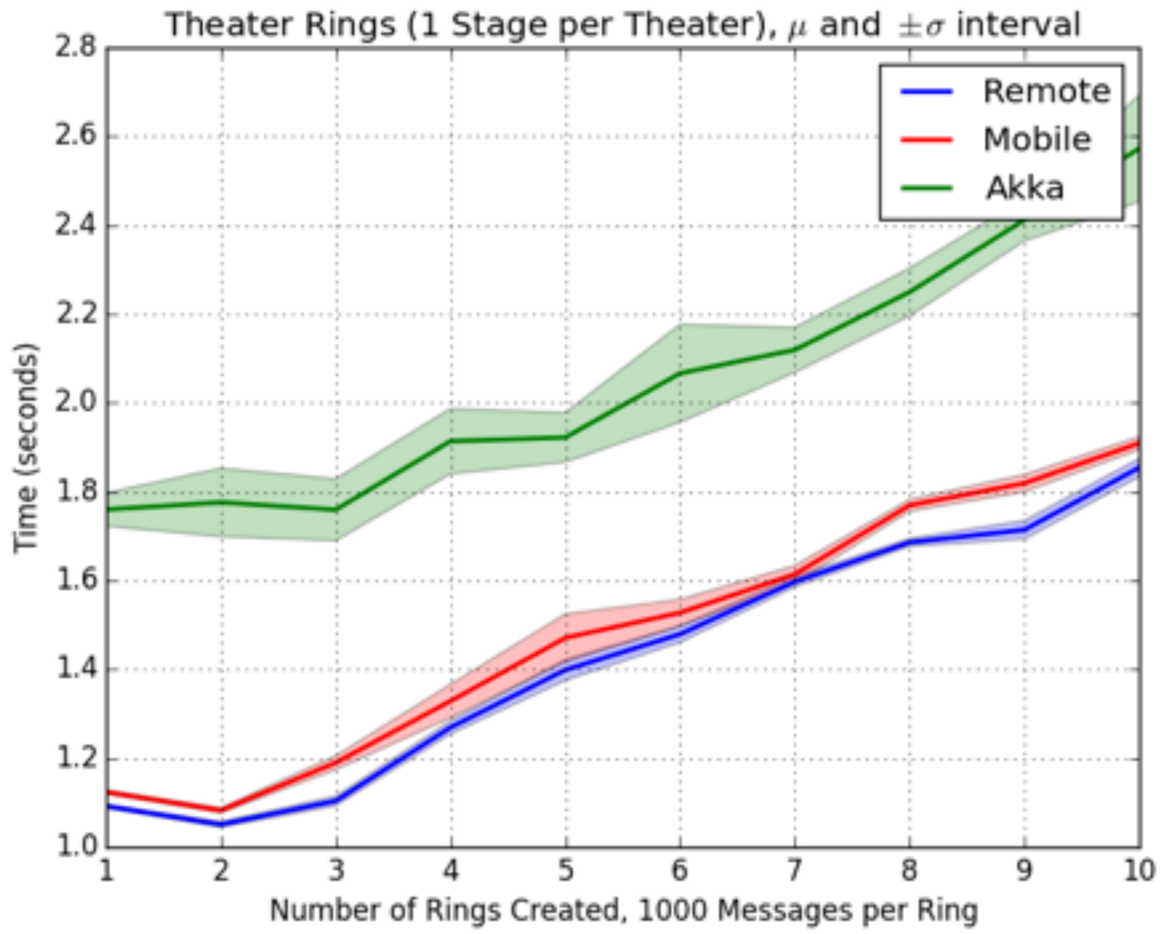
Tests creation of actors on other Theaters. Mobile actor creation is ~6% slower than remote actor creation. Remote actors were created on a different node in the Linux cluster.

Migrating actors is faster (at least for this microbenchmark) than remote creation. Akka crashed with > 6000 actors (with an error 'resend buffer capacity reached').

Theater Rings (1 - 10 rings)



Linux cluster, Java 1.6



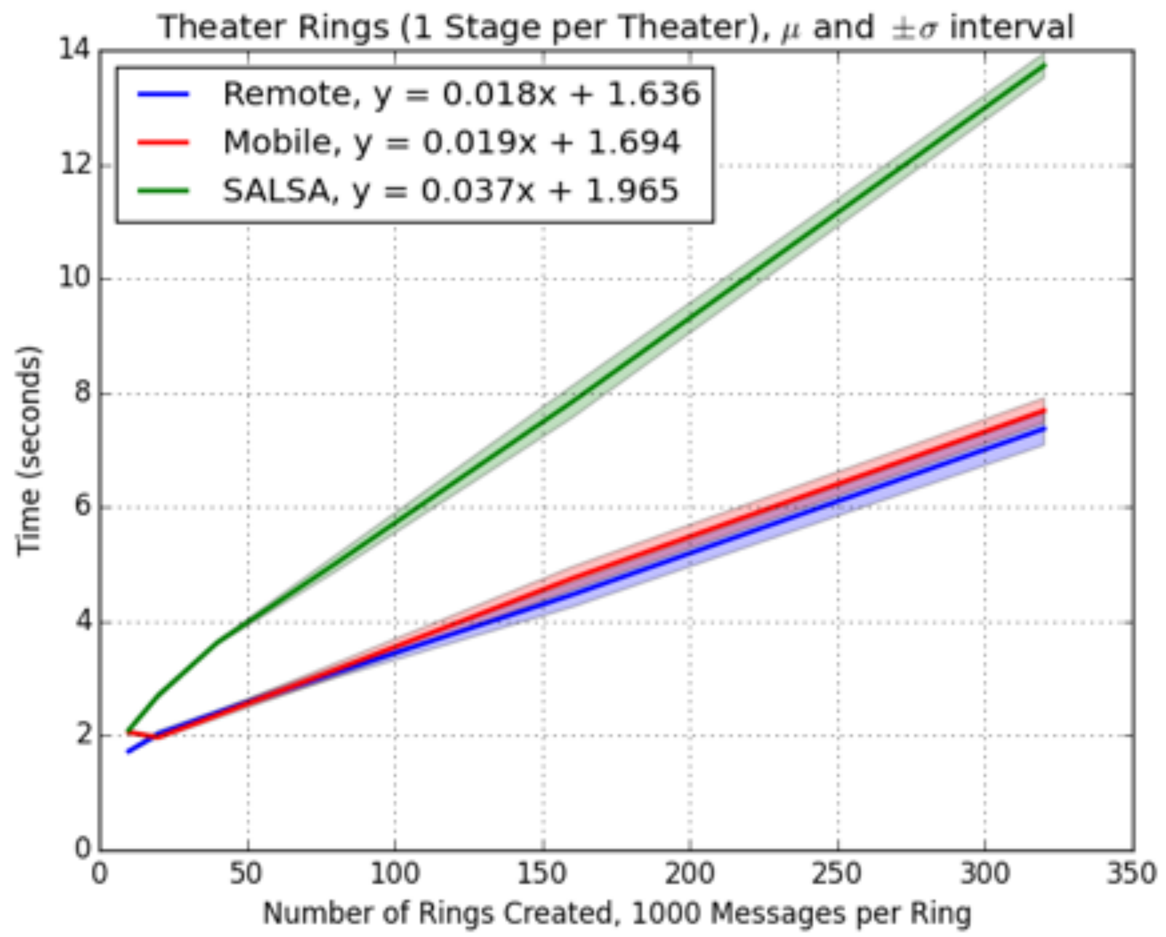
OSX, Java 1.8

Akka does not include start up time, however it does not reuse a JVM. SALSA includes start up time and uses new JVMs per run.

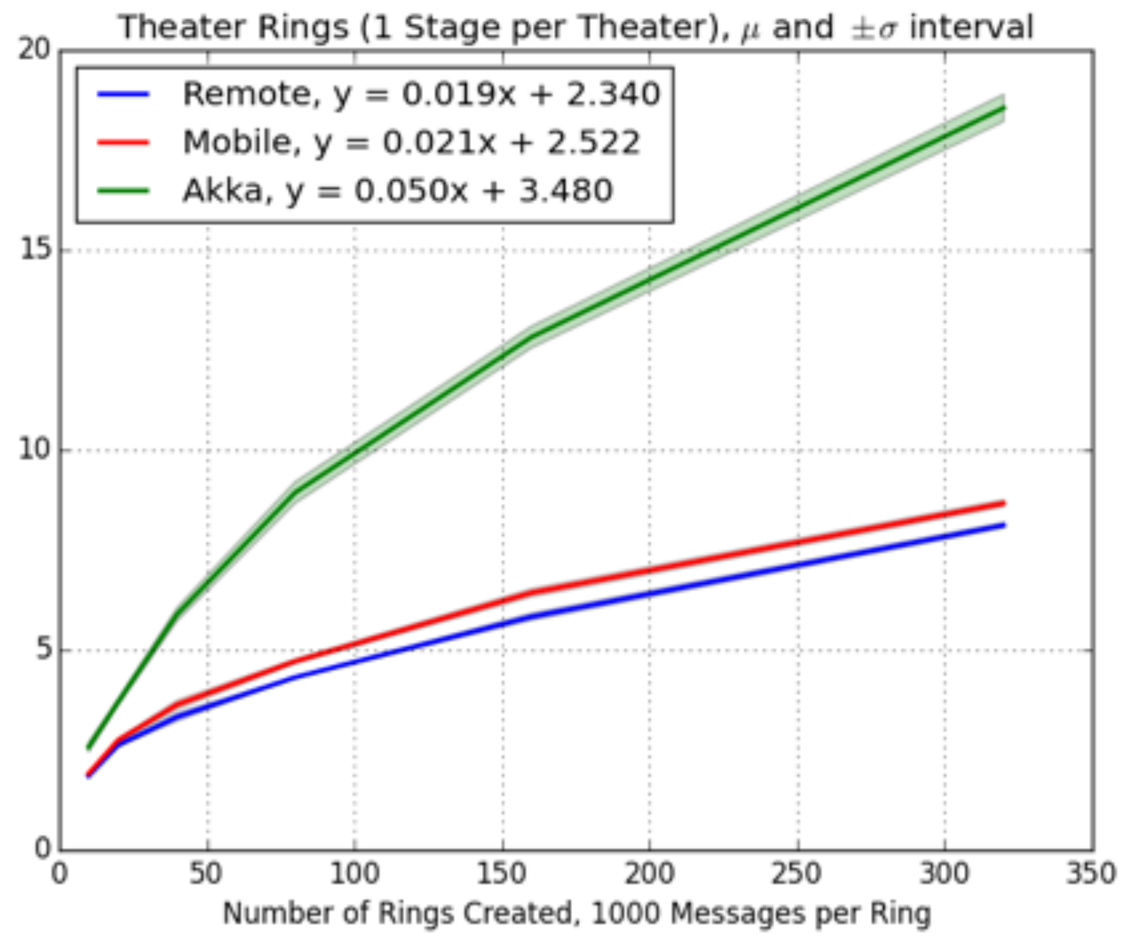
Benchmark is similar to ThreadRing, except multiple rings are created. 4 Runtimes (each on a different node in the Linux clsuter) + 1 Startup runtime were created, and there is an actor in each of the 4 runtimes which host the rings.

The spike in poor performance for 1 and 2 rings found in the paper actually turned out to be a Java issue, most likely due to poor thread scheduling/context switching time.

Theater Rings (10 - 320 rings)



Linux cluster, Java 1.6

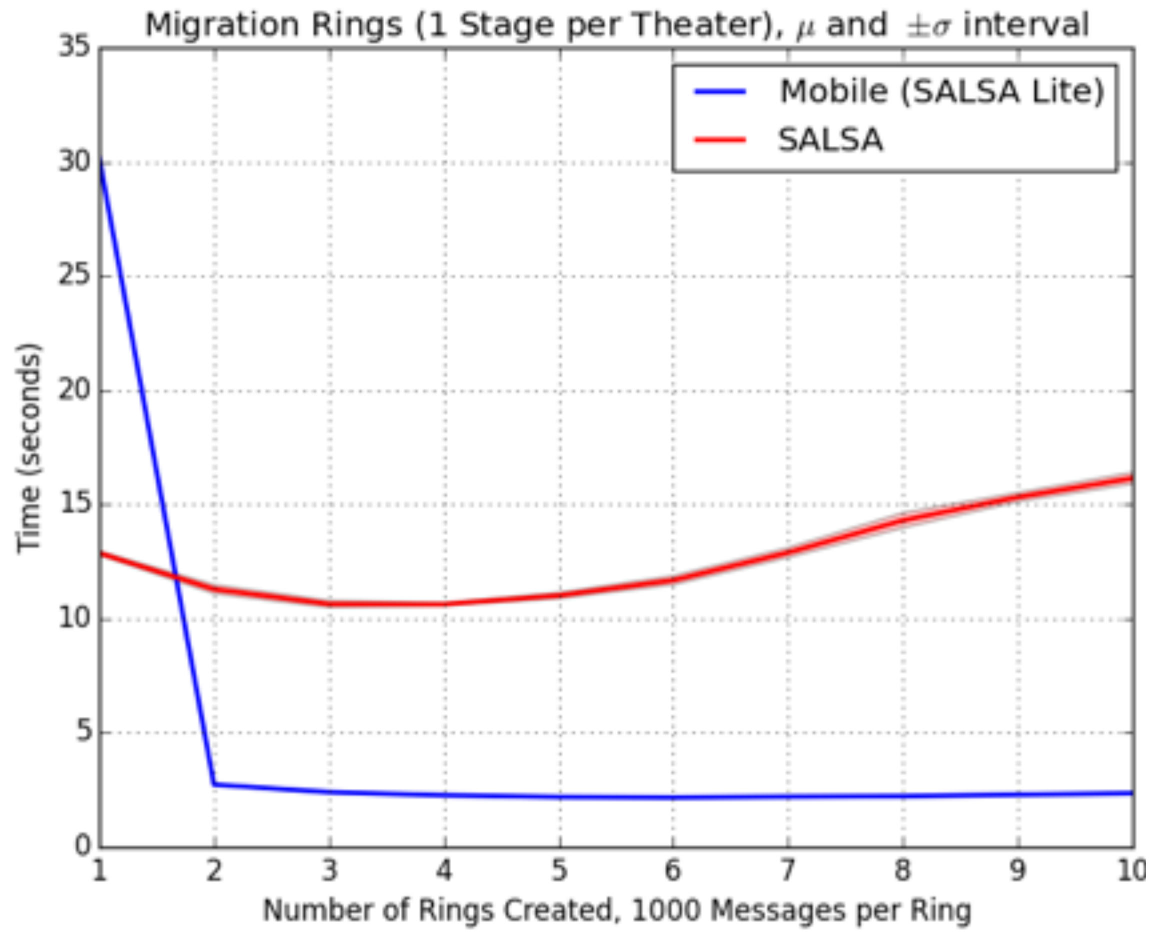


OSX, Java 1.8

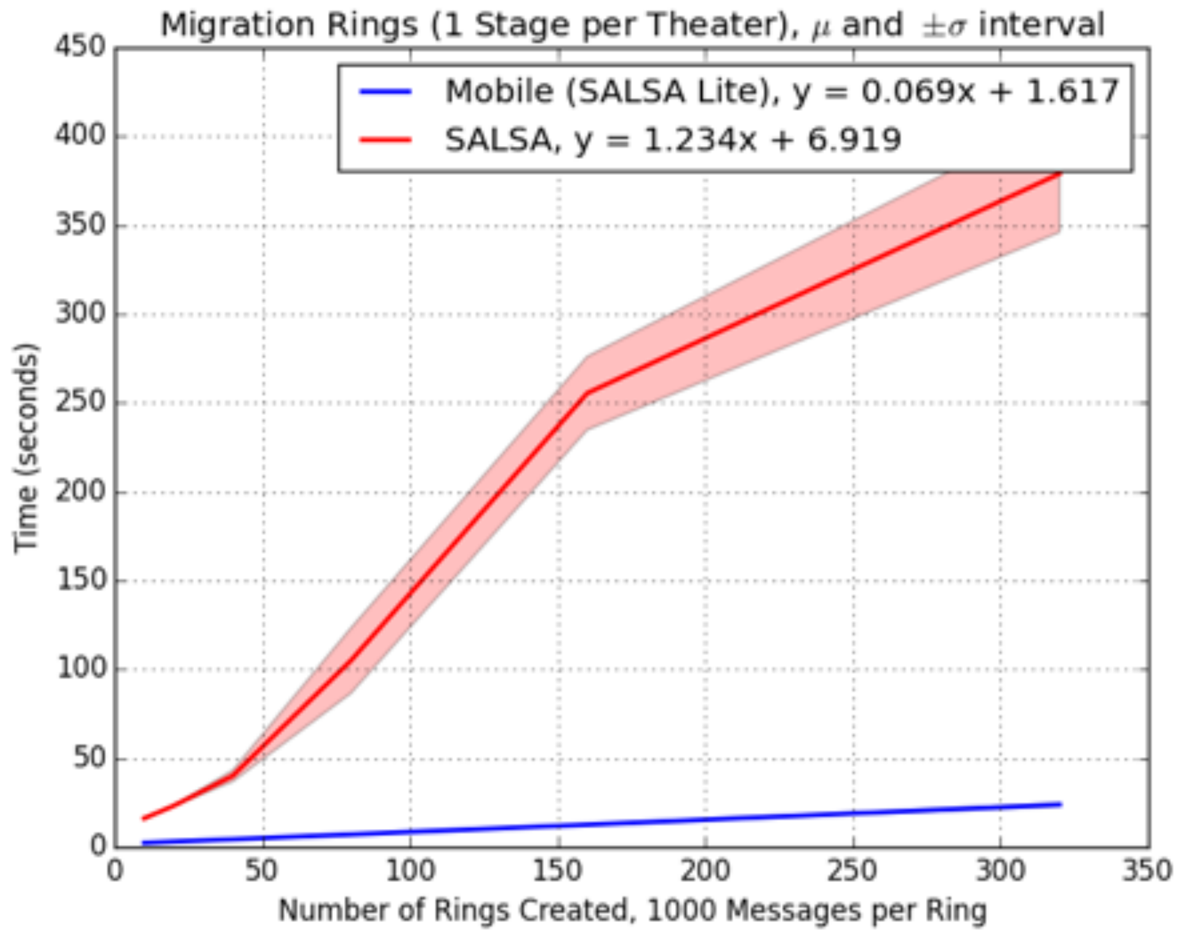
Akka does not include start up time, however it does not reuse a JVM. SALSA includes start up time and uses new JVMs per run.

When scaled up to 320 rings, the overhead of message passing becomes apparent, ~5%-10% for mobile actors over remote actors. Remote actor communication in SALSA Lite in either case is ~2.5x faster than Akka.

Migration Rings



Linux cluster, Java 1.6



Linux cluster, Java 1.6

SALSA includes start up time and uses new JVMs per run.

N actors are created, each of which then migrates 1000 times in a ring.

There is an initial poor performance of SALSA Lite with 1 ring (due to the Java 1.6 issue), however after that performance is significantly better, ~18x faster.

Conclusions

Many benefits to practicing what we preach about the actor model.

Described protocols for remote actor creation, messaging and actor mobility - built using actors.

SALSA Lite's implementation of remote actor creation, messaging and mobility are very efficient, however mobility does come at a non-insignificant cost.

To allow developers to implement the fastest applications, it makes sense to provide RemoteActor and MobileActor classes so only actors requiring that functionality pay the price in performance.

Future Work

- Implementing full Savina Benchmark Suite.
- Developing new distributed actor benchmarks for migration, message passing, remote creation, scalability, garbage collection, etc.
- Mobility between stages.
- Revisiting the Internet Operating System (IOS) - transparent load balancing by actor profiling and migration.
- Examining and implementing efficient approaches to distributed garbage collection.
- Making SALSA Lite to SALSA 2.0.
- A C/C++ implementation to take advantage of MPI and accelerator cards.

Thanks!

Questions?

http://people.cs.und.edu/~tdesell/salsa.php
https://github.com/travisdesell/salsa_lite

tdesell@cs.und.edu

Example: Fibonacci

```
behavior Fibonacci {
  int n;

  Fibonacci(int n) {
    self.n = n;
  }

  Fibonacci(String[] arguments) {
    n = Integer.parseInt(arguments[0]);

    self<-finish(self<-compute());
  }

  int compute() {
    if (n == 0)      pass 0;
    else if (n <= 2) pass 1;
    else             pass new Fibonacci(n-1)<-compute() + new Fibonacci(n-2)<-compute();
  }

  ack finish(int value) {
    System.out.println(value);
    System.exit(0);
  }
}
```