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

Travis Desell
University of North Dakota
3950 Campus Road Stop 9015
Grand Forks, ND 58203, USA
tdesell@cs.und.edu

Carlos A. Varela
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180, USA
cvarela@cs.rpi.edu

ABSTRACT

This paper presents a newly developed implementation of remote message passing, remote actor creation and actor migration in SALSA Lite. The new runtime and protocols are implemented using SALSA Lite's lightweight actors and asynchronous message passing, and provide significant performance improvements over SALSA version 1.1.5. Actors in SALSA Lite can now be *local*, the default lightweight actor implementation; *remote*, actors which can be referenced remotely and send remote messages, but cannot migrate; or *mobile*, actors that can be remotely referenced, send remote messages and migrate to different locations. Remote message passing in SALSA Lite is twice as fast, actor migration is over 17 times as fast, and remote actor creation is two orders of magnitude faster. Two new benchmarks for remote message passing and migration show this implementation has strong scalability in terms of concurrent actor message passing and migration. The costs of using remote and mobile actors are also investigated. For local message passing, remote actors resulted in no overhead, and mobile actors resulted in 30% overhead. Local creation of remote and mobile actors was more expensive with 54% overhead for remote actors and 438% for mobile actors. In distributed scenarios, creating mobile actors remotely was only 6% slower than creating remote actors remotely, and passing messages between mobile actors on different theaters was only 5.55% slower than passing messages between remote actors. These results highlight the benefits of our approach in implementing the distributed runtime over a core set of efficient lightweight actors, as well as provide insights into the costs of implementing remote message passing and actor mobility.

Keywords

Actor model, Lightweight Actors, Distributed Actor Benchmarks, Distributed Computing, Actor Migration

1. INTRODUCTION

As programming environments continue to increase in parallelism in terms of numbers of processors and cores, the need for efficient and effective concurrent and distributed programming languages becomes ever more important. In many ways, the common method of using object, threads and communication over synchronous sockets is not well suited to these environments, as evidenced by the large body of work on detecting, preventing and avoiding deadlocks and other race conditions [23, 28, 18, 49, 27, 31, 16, 1, 26, 17]. Many of these issues arise due to the fact that objects do not encapsulate their state, so member fields must be protected with mutexes or other blocking synchronization constructs to prevent concurrent access. This blocking behavior, coupled with the blocking behavior of using sockets synchronously makes it quite easy for deadlocks and other race conditions to occur.

As threads move from object to object, without any programmatic way of knowing where they came from, in many ways they present a harmful situation similar to the much maligned `GOTO` statement [15, 42]. As Dijkstra eloquently stated, *"The unbridled use of the* **go to** *statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress."* In many ways, the unbridled use of threads presents a similar situation where it becomes terribly hard to find a meaningful set of coordinates in which to describe a *threads* progress.

The Actor model, formalized over 40 years ago [22, 21] and later extended to open distributed systems [2], provides a strong alternative model without these pitfalls. Actors are independent, concurrent entities that communicate by exchanging messages. Each actor encapsulates a state with a logical thread of control which manipulates it. Communication between actors is purely asynchronous. The actor model assumes guaranteed message delivery and fair scheduling of computation. Actors only process information in reaction to messages. While processing a message, an actor can carry out any of three basic operations: altering its state, creating new actors, or sending messages to other actors (see Figure 1). Actors are therefore inherently independent, concurrent and autonomous which enables efficiency in parallel execution [32] and facilitates mobility [3, 44]. In the actor model, a thread of control only operates on a single actor, and activity passes through the system via asynchronous messages, which have sources and targets that enable an easier understanding of program flow. This model, if strictly adhered to, actually makes it challenging to program a system which deadlocks and completely prevents concurrent

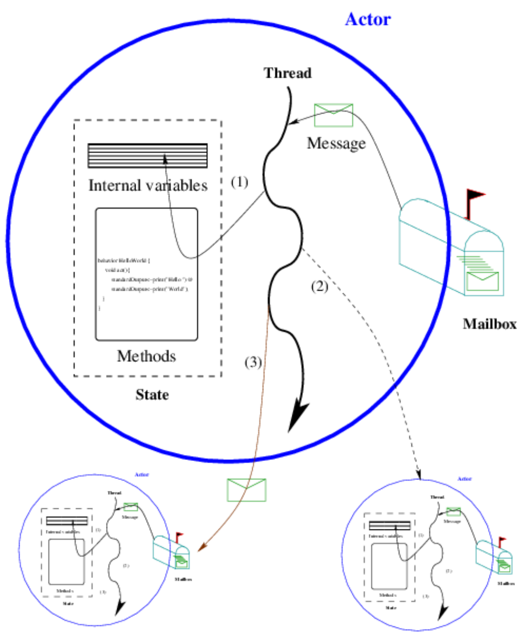


Figure 1: Actors are reactive entities. In response to a message, an actor can (1) change its internal state, (2) create new actors, and/or (3) send messages to other actors (image from [43]).

memory access issues.

Many of the early implementations of actor languages involved heavyweight actors, such as Erlang [7], Scala [19] and SALSA [44, 4], where each actor had an actual thread of control. This type of actor has a fair bit of overhead over objects, and also has low limits on how many actors can exist imposed by the operating system and hardware on how many threads are allowed. Because of this, languages like SALSA, Scala, and Kilim [38] allow the use of both objects and actors, which can unfortunately lead to potential violations of the actor model [14] and the use of a mixture of concurrency models [40].

In part due to these issues, and in part due to a desire to seek higher levels of performance in actor languages, systems using lightweight actors have been developed and are of high interest. In Java based actor libraries and languages, Kilim utilizes lightweight threads [38] and Scala also allows actors to run using a thread pool [19] or as lightweight actors using the Akka framework [5, 9]. Charm++ [29] provides a lightweight actor inspired library based on C++ for use in cluster computing environments, and more recently libcppa [12], has evolved into the C++ Actor Framework (CAF) which utilizes lightweight actors and also enables GPGPU computing [10, 11].

SALSA Lite follows in this path by rebuilding SALSA from the ground up using lightweight actors with a strong emphasis on performance. Previous work has evaluated the performance of SALSA Lite in non-distributed concurrent settings [14]. This work presents how these lightweight actors have been used to develop an efficient distributed computing runtime that enables both remote and mobile actors. Few actor languages and libraries support transparent distribution of actors, and even fewer support mobility. To the authors' knowledge, only the ActorFoundry [8] (based on Kilim), ActorNet [34, 33] (an actor platform for wireless sensor networks), JavAct [6], Actor Architecture [25], SALSA and now SALSA Lite provide transparent actor mobility [30].

2. APPROACH

The design philosophy of SALSA Lite is in essence to practice what we preach in regards to the benefits of the Actor model, *i.e.*, the language should be built using the Actor model as opposed to objects and threads. Further, to borrow from Unix, the common case should be executed fast. As such, SALSA Lite has been rebuilt from the ground up to provide a core of lightweight actors which can send messages and be created extremely quickly. Using this simple efficient core, language semantics and runtime services are then built using these actors as opposed to objects and threads. Other results have shown the efficiency of SALSA Lite actors in concurrent non-distributed settings [14, 11], providing justification for using them as a foundation for the language.

This lightweight actor implementation has been built in a novel way based on hashing to eliminate any synchronization bottlenecks [14]. For example, if actors or messages need a unique identifier, a common way to generate that is to take the host, port and start time of the theater and append a counter. However, that counter needs to be accessed atomically which makes it a singular point of synchronization. This can lead to applications which appear concurrent, but actually are operating sequentially based on those synchronization bottlenecks. SALSA Lite avoids this by using the hashcode of the actor requesting a service and selecting one of N copies of a service by the hashcode modulo the number of services. Collisions are not an issue as mutliple actors are expected to use the limited number of N services. The number of the various services can be specified at runtime, allowing SALSA Lite applications to easily scale to the number of cores available.

Instead of developing a runtime and services for remote and mobile actors using objects and threads, as done in the previous implementation of SALSA, the lightweight core of actors has been used to develop a distributed computing environment which utilizes the Actor model to eliminate deadlocks and achieve high levels of concurrency and performance. Lastly, as SALSA Lite is being developed with performance in mind, we have decided to allow programmers to specify if an actor will be local, remote or mobile, as adding this functionality does come at a cost. This allows programmers to use the type of actor with the best performance for the task at hand.

3. IMPLEMENTATION

SALSA Lite's runtime is based on the concept of *stages* (see Figure 2), which essentially act as a unified mailbox and thread of control for groups of actors assigned to them. This allows local actors to be implemented as simple Java objects with only a reference to the stage they are "performing" on. When a message is sent to an actor, the message is placed in its stage's mailbox using this reference and the stage's `putMessageInMailbox` method. While to the authors' knowledge, SALSA Lite is the only actor langauge to utilize this type of runtime, however others such as E's

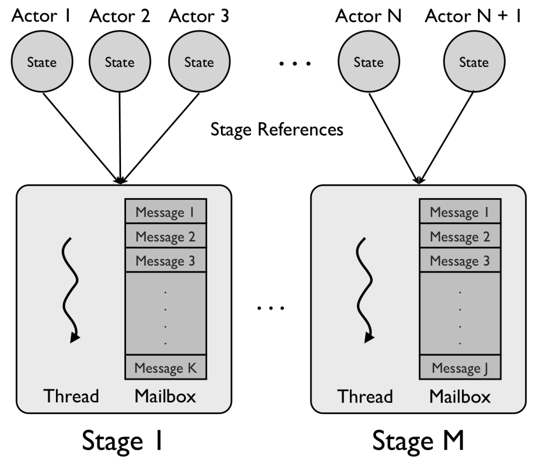


Figure 2: The SALSA Lite runtime environment. Heavyweight actors called *stages* are used to process messages on multiple lightweight actors, simulating their concurrent execution. A stage will repeatedly get the first message from its mailbox and process that message on the message's target actor. Every actor is assigned to a stage. A Message sent to an actor is placed at the end of its assigned stage's mailbox (image from [14]).

```
//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);
```

Figure 3: SALSA Lite has simplified syntax for creating remote and mobile actors. Unique names are specified with the `called` keyword, the host and port of the theater the actor is created on are specified with the `at` keyword, and the name server actor a mobile actor is registered at is specified with the `using` keyword.

vats [36, 35] and SEDA [48, 37] use a similar approach for high efficiency. With this design in mind, remote and mobile actors were implemented in a way to not impact or degrade the performance of these local actors.

3.1 Theaters

Distributed computing in SALSA is done using the concept of *theaters*. Each theater serves as a separate process in which multiple actors perform on a set of stages. The number of stages in a theater can be dynamically specified at runtime, and actors can either be automatically assigned to stages, have new stages generated for them, or be assigned to particular stages programatically (see [14] for further details). Theaters listen on a particular port for incoming connections to other theaters which can be distributed over local area networks or the Internet.

Each theater has a `TheaterActor` repeatedly listening for incoming socket connections. When one occurs, it spawns a `IncomingTheaterConnection` actor who handles receiving messages and migrating actors from that theater. Messages and migrating actors are sent via a theater's `TransportService` which has a set of static methods which put messages in the appropriate `OutgoingTheaterConnection` mailboxes and create new `OutgoingTheaterConnection` actors when necessary. The `TheaterActor`, `IncomingTheaterConnection` and `OutgoingTheaterConnection` actors are all heavyweight, each running on their own stage as to not block the execution of other actors when they are blocked listening for connections or waiting to receive data over a socket.

The `IncomingTheaterConnection` actors put messages in the appropriate stage's mailbox as they are received. Making sure references to actors are correctly kept as messages are serialized and de-serialized is described in Section 3.2.1. The `OutgoingTheaterConnection` actors simply send messages and actors across a socket to the `IncomingTheaterConnection` actor they are paired with. As described in Section 3.3.1, implementing these services as actors also allows for the easy implementation of a protocol to update remote references to mobile actors as they migrate around a set of theaters.

3.2 Remote Actors

As remote actors do not migrate, it is always the case that a reference to a remote actor refers to the actual actor when it is present at the same theater, or that it is a reference to a remote actor on another theater. In the first case, implementation of remote actors is identical to that of a local actor, with the exception that the remote actor needs a unique name so that it can be referred to and looked up by other actors. Figure 3 presents the syntax for creating the various types of actors in SALSA Lite and Figure 4 presents the syntax for generating references to actors using their name and location. The remote actor also needs to be added to a `RemoteActorRegistry` which is a `HashMap` of names to the actual remote actor lightweight actor object, so incoming messages can be directed towards it accordingly. This adds some overhead to the creation of a remote actor, while sending messages to it locally can be performed the same as with local actors as the other actors simply have a reference to the actual remote actor.

In the second case, where it is a reference to a remote actor on another theater, when a message is invoked on that reference (implemented as a local actor), it instead uses SALSA Lite's transport service to put it in the mailbox of the appropriate `OutgoingTheaterConnection` actor, which sends it over a socket to the appropriate theater. When the mes-

```
//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);
```

Figure 4: SALSA Lite has also simplified syntax for referencing remote and mobile actors. Instead of the `new` keyword, the `reference` keyword is used. The actor's names are specified with the `called` keyword, the host and port of the theater the actor is created on are specified with the `at` keyword, and the name server actor a mobile actor is registered at is specified with the `using` keyword.

sage is received by the target theater's `IncomingTheaterConnection`, the actual remote actor is looked up in the `RemoteActorRegistry` and the message is sent to it.

3.2.1 Actor Reference Propagation

Some challenges arise in that a message sent to a remote actor on another theater can contain references to other local or remote actors. If these messages were blindly serialized while being sent to the other theater, this would result in unintended copies of these actors. To overcome this, SALSA Lite uses Java's `readResolve()` and `writeReplace()` methods instead of default object serialization. When a local or mobile actor is serialized, its `writeReplace()` method is called, which creates a serialized reference only containing the hashcode, host and port in the case of a local actor, or the unique name, host and port in the case of a remote actor. For local actors, a reference to the local actor is also placed in a `LocalActorRegistry` so it can be looked up if messages are sent to it from another theater. This also can drastically reduce the size of the messages being sent as only the minimum amount of data required to lookup the actor or generate a reference is sent. It should be noted that the implementation of local actors has remained completely unchanged, apart from now providing `readResolve()` and `writeReplace()` methods for serialization when references to them propagate to remote theaters.

When the serialized reference is received by a theater, the `readResolve()` method is called on the serialized reference. This performs a lookup in either the `LocalActorRegistry` or `RemoteActorRegistry`. If the actor is present, the `readResolve()` method returns the actual reference to that actor, otherwise it returns a remote reference object which sends messages to the `OutgoingTheaterConnection` actor instead of actually processing them. This prevents copies of actors from occurring, and also ensures that there is only one remote reference to an actor at any one theater (which will aid in implementing distributed garbage collection). These registries have been implemented using the hashing strategy described in Section 2, so multiple copies can be made which are selected by the actor's hashcode, preventing the registries from acting as a singular bottleneck.

3.3 Mobile Actors

Unlike local and remote actors, there are significant challenges in implementing mobile actors as a single object (either as a remote reference or the actual actor), as migration would then involve having to update the references to it held by all other actors every time it migrates. Keeping track of these reverse references can lead to significant memory and performance overhead. Similar to how actors are implemented in SALSA, in SALSA Lite, mobile actors are divided into reference and state objects. When a mobile actor is created, its state is placed in a `MobileActorStateRegistry`, which is the only object with a reference to the actor's state. The reference acts in the place of a lightweight actor on a stage. When a message is invoked on the reference, it performs a lookup in the `MobileActorStateRegistry` which invokes the message on the state if the actor is present. When the actor migrates, the state object is put in a message to the `OutgoingTheaterConnection` it is being sent over, and the state object is removed from the `MobileActorRegistry` and is replaced with a reference to the `OutgoingTheaterConnection` actor that sends messages to the theater the actor migrated to. If the lookup returns the connection, the message is placed in the `OutgoingTheaterConnection`'s mailbox to be sent to that theater. In this way, the only time the mobile actor's state is serialized is when it migrates.

This also allows references to mobile actors to propagate in a manner similar to local and remote actors. This propagation is handled the same way by using `readResolve()`, `writeReplace()`, a serialized reference and `MobileActorReferenceRegistries`.

Note that every time a message is invoked on a mobile actor, a lookup in a `MobileActorStateRegistry` needs to be performed, which adds overhead to message passing.

3.3.1 Finding Mobile Actors

In addition to mobile actors requiring a unique name, host and port in their reference and state, mobile actors also need to be registered at a *name server*. The name server is used as a lookup service for getting a reference to a mobile actor (see Figure 4). When the mobile actor is created, it sends a `PUT` message asynchronously to the name server it will be registered at. When the actor migrates, it sends an asynchronous `UPDATE` message to the name server, which updates its location on the name server. When another actor wants to get a reference to a mobile actor, it can contact the name server with a `GET` message which will return a reference to that actor. This is done transparently when the `reference` keyword is used.

In contrast with SALSA, where name servers are run as standalone daemons, in SALSA Lite, name servers are implemented using remote actors and are first class entities within the runtime (see Figure 3 and 4). This makes the use of name servers much easier, as they can be easily created within SALSA Lite programs, and also allows them to use SALSA Lite's remote message sending infrastructure.

Another major difference is that in SALSA Lite, name servers operate asynchronously. In SALSA, whenever an actor migrates, it synchronously performs an `UPDATE` on the name server and only migrates after it completes, as name servers are used synchronously within the protocol for looking up actors if a message arrives at a theater and the actor had migrated away in the meantime. In SALSA Lite, name servers only asynchronously provide a reference to the actor and the run time updates itself as to where the actor is located.

This is done with the following protocol: if a message received by an `IncomingTheaterConnection` actor 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.

All these messages are sent asynchronously using SALSA Lite's remote messaging, to prevent deadlocks and improve performance. Also, this means that the name server responds with a reference to an incorrect theater, as an actor had completed migration before the `UPDATE` message was processed, that reference will be updated to the actor's current location using this protocol without requiring any further calls to the name server.

4. RESULTS

For reproducibility, source code for SALSA Lite is freely available on GitHub.[1] The benchmarks used can be found in the `benchmarks` directory of the repository. This section presents a performance analysis of the newly developed remote and mobile actors and compares them to local actors in SALSA Lite as well as SALSA version 1.1.5.

4.1 Runtime Environment

All results were gathered using a small Beowulf HPC cluster with 4 dual quad-core compute nodes (for a total of 32 processing cores). Each compute node has 64GBs of 1600MHz RAM, two mirrored RAID 146GB 15K RPM SAS drives, two quad-core E5-2643 Intel processors which operate at 3.3Ghz, and run the Red Hat Enterprise Linux (RHEL) 6.2 operating system. All 32 nodes within the cluster are linked by a private 56 gigabit (Gb) InfiniBand (IB) FDR 1-to-1 network. Java version 1.6.0_26 was used, with the Java(TM) SE Runtime Environment (build 1.6.0_26-b03) and the Java HotSpot(TM) 64-Bit Server VM (build 20.1-b02, mixed mode). Runs were performed 10 times each, each with freshly created theaters and name servers, so startup times are included, and the various figures display the mean runtime as well as the standard deviation of the different runs. Runs done with SALSA version 1.1.5 were done with garbage collection turned off by using the `-Dnogc` system property for a more accurate comparison as distributed garbage collection in SALSA Lite remains an area of future work. This is in part due to challenges in correctly and efficiently implementing distributed garbage collection that can also handle pathological cases such as distributed circular references.

[1] https://github.com/travisdesell/salsa_lite

4.2 Local Message Passing Performance

Figure 5 shows the performance of local, remote and mobile actors in SALSA Lite, with all actors running on a single stage.[2] The `ThreadRing` benchmark was identical for all three, except that actors either were local, or extended the `RemoteActor` or `MobileActor` behaviors. One stage was used to avoid introducing effects from thread scheduling which could significantly impact performance. 31 actors were created (as typical for the benchmark) and 500,000 to 1,000,000 messages were passed around the ring.

While this figure shows a fair amount of overhead for using remote and mobile actors, this is mostly due to increased startup times. Remote actors require a theater to be created, which opens a socket and listens for incoming communication from other theaters, mobile actors require this in addition to a name server which they are registered with. A linear regression was performed for each of these runs, with r-values (correlation coefficients) greater than 0.999 for all three actor types. Figure 5 shows the slope and intercept for these regressions, showing that remote actors have a ∼2.5x increase in startup costs and that mobile actors have a ∼3.26x increase over purely local actors. Apart from the increased startup costs, the performance of actual message sending is quite good, with local and remote actors having practically identical message passing performance, and mobile actors having ∼30% overhead.

4.3 Local Actor Creation Performance

Figure 6 shows the performance of creating local, remote and mobile actors using a simple actor creation micro benchmark. As in the previous benchmark, only one stage was used to avoid introducing effects from thread scheduling. The benchmark created between 100,000 and 600,000 actors, with each actor responding to the master actor with an acknowledgement message that it had been created. After an acknowledgement had been received for each actor, the benchmark would terminate. Here, in addition to the startup costs seen in the `ThreadRing` benchmark, there is significantly more overhead for creating remote and mobile actors.

Linear regression on these runs produced an r-value above 0.999 for the local actors, and r-values above 0.977 for remote and mobile actors. Creation of remote actors resulted in ∼54% overhead, while mobile actors had ∼438% overhead. This was expected however, as in SALSA Lite, local actors are little more than objects with a reference to the stage they are running on, while remote actors also need to

[2] Previous results have compared SALSA Lite to SALSA for the `ThreadRing` benchmark, along with a set of other actor based programming languages [14] (Kilim, Scala, and Erlang). SALSA Lite has been found to be three times faster than Kilim and an order of magnitude faster than Erlang, Scala and SALSA on this benchmark. This performance has also been recently reproduced by Charousset *et* al. [11]. As this paper focuses on the performance of remote and mobile actors in SALSA Lite, we refer the reader to those works for further performance comparisons between different actor languages.

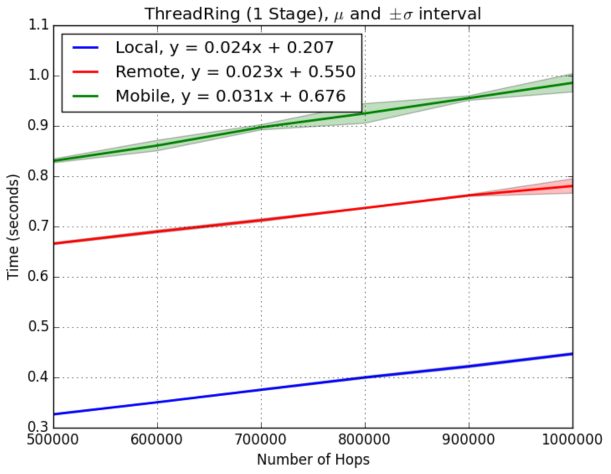


Figure 5: This figure shows the performance of the `ThreadRing` benchmark for local, remote and mobile actors in SALSA Lite. This serves as a measure of the basic amount of overhead in message passing and theater startup costs for using remote and mobile actors. In each benchmark, 31 actors were created and 500,000 to 1,000,000 messages were sent around the ring. The mean and standard deviation of 10 runs for each data point are shown.

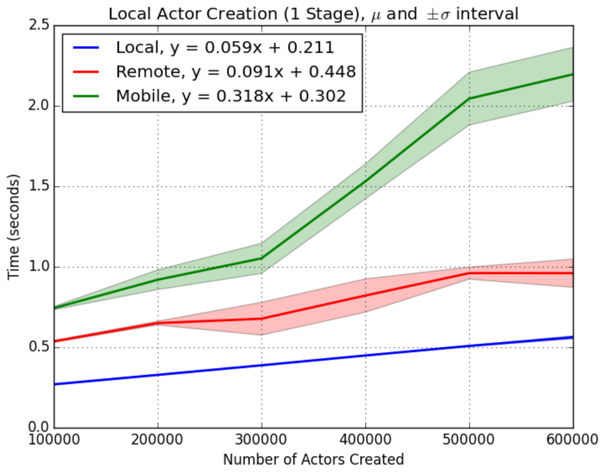


Figure 6: This figure shows the performance of a actor creation micro-benchmark in SALSA Lite. This serves as another measure of theater startup costs, as well as the overhead of creating remote and mobile actors. 100,000 to 600,000 actors were created, each sending an acknowledgement to a master actor. The mean and standard deviation of 10 runs for each data point are shown.

have a name, host and port. In addition to that, mobile actors also require a reference to a name server, and also send a `PUT` message to register at the name server when they are created. Further, both also need to be stored in their respective registries. As the actors created in this micro-benchmark only have a reference to the master actor that created them, these costs can represent significant overhead.

4.4 Remote Actor Creation Performance

Figure 7 shows the performance of creating both remote and mobile actors at a remote theater. Additionally, the cost of creating an actor locally and then migrating it to the remote theater is presented. Only one stage was used, however the theater actors are created on their own stages, so some of the performance differences could be attributed to thread context switching.

This micro-benchmark is identical to the previous local version except for the actors being created on the remote theater. For the version with migration, the acknowledgement message to the master was only sent after the migration had completed. For these tests, 1000 to 5000 actors were created. Linear regression produced r-values greater than 0.994 for the local actors, 0.979 for the remote actors, and 0.983 for the mobile actors. Interestingly, creating the mobile actors locally and migrating them performed better than creating them remotely, however this makes some sense in that creating the mobile actors remotely returns a token (similar to a future) which must be received by a `TokenDirector` actor created by the runtime, so it entails the creation of an extra actor and two extra messages (one from the remote theater to the `TokenDirector` at local theater and another from the `TokenDirector` to any actor which wants to use the reference to the remotely created actor). Even so, remote creation of mobile actors was only ∼6% slower than remote creation of remote actors.

It is also worth mentioning that while local creation and migration performed faster than remote creation of mobile actors, in general this is probably not the case. For example, if an actor generates any large amount of data in its constructor, creating it locally and migrating it could be significantly more expensive due to extra bandwidth required. As such, it is good to be able to have the option to use either method.

Results were also gathered using SALSA version 1.1.5 however these were not added to the figures as for above 100 actors deadlocks as well as issues with reaching a limit on the number of threads available occurred. However, for 100 actors, remote creation took an average of 1.84 seconds with a standard deviation of 0.033 over 10 runs, and local creation and migration took on average 1.24 seconds with a standard deviation of 0.0092 seconds. This is an order of magnitude slower than the time taken for SALSA Lite to create 1,000 actors; so the new implementation shows significant performance gains over SALSA version 1.1.5.

4.5 Remote Message Passing Performance

Figure 8 presents results for a new benchmark called `TheaterRings` which examines the performance of remote message sending. This benchmark operates similarly to the `ThreadRing` benchmark, except in this case a single actor is created at each theater. Additionally, multiple rings of actors are created which send messages concurrently. All messages hop around the ring of actors in the same direction. For this benchmark, 1 to 320 rings were generated, with each ring sending 1000 messages. Each ring had one actor created on one of the four nodes in the Beowulf cluster. Each theater had one stage for these actors, however

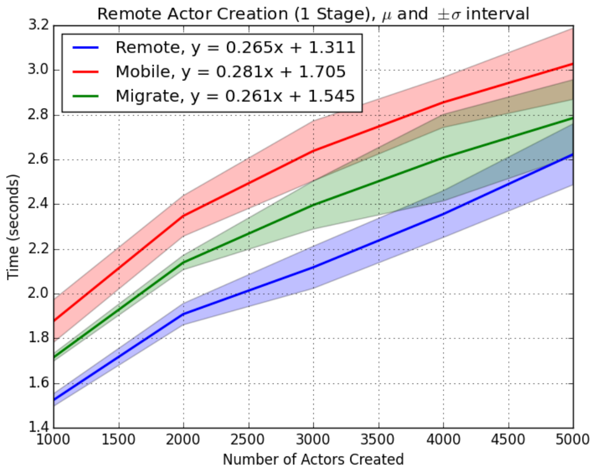


Figure 7: This figure shows the costs of creating remote and mobile actors at a remote theater, along with creating mobile actors locally and migrating them to the remote theater. After each actor was created remotely, or completed its migration, it sent an acknowledgement message back to the master actor. The mean and standard deviation of 10 runs for each data point are shown.

other stages were created for actors in the theater runtime. Figure 8a shows results for 1 to 10 rings, and Figure 8b shows results calculated with 10, 20, 40, 80, 160, and 320 rings. Results were gathered using remote and mobile actors in SALSA Lite, as well as actors in SALSA version 1.1.5.

From 1 to 10 rings some rather interesting behavior occurred. First, once SALSA reached 7 rings, the runtime started varying dramatically and deadlocks started to occur, as is shown by the large increase in the standard deviation of the runtime. Additionally, for 1 and 2 rings, mobile actors exhibited some very poor performance, running almost 5 times slower than remote actors and SALSA. Also, for all three, runtime generally *decreased* as more rings were added. Performance was the fastest for SALSA and remote actors at 6 rings, with an average runtime of 2.001 and 1.518 seconds, respectively, and mobile actors were the fastest at 20 rings, with an average runtime of 1.971 seconds.

For SALSA, after 10 rings, deadlocks occurred even more frequently, however the runtime stabilized for the runs which completed (the times shown are the average runtime of runs which completed). Presumably, from 7 to 9 rings, the issue causing the deadlock could resolve itself resulting in the larger span of run times, however with 10 or more rings the issue was not resolvable. After 10 rings, the runtime increased quite linearly, with the linear regression having r-values of greater than 0.999, 0.998 and 0.999 for remote, mobile and SALSA actors, respectively. Using the values from the linear regression from 10 to 320 stages, mobile actors were $\sim$5.55% slower than remote actors (similar to results from the remote actor creation micro-benchmark). Further, both remote and mobile actors were around twice as fast as SALSA (not counting SALSA deadlocks).

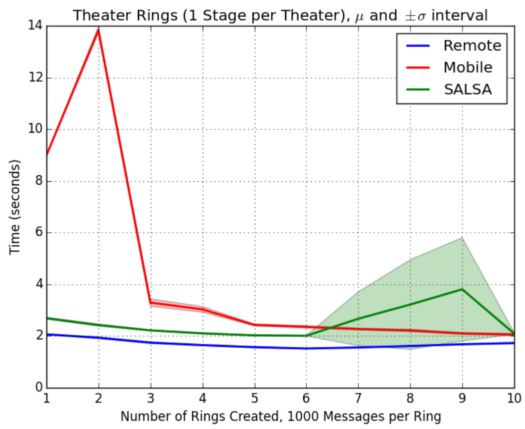
Given these results, it seems that the `TheaterRings` benchmark presents a pathological case for mobile actors when there are 1 or 2 rings. These results are somewhat similar to the case of the `ThreadRing` benchmark where each actor is given their own stage (*i.e.*, their own thread), where SALSA Lite performs similarly to SALSA, at about an order of magnitude slower than having the `ThreadRing` actors entirely on one stage. Because of this, the poor performance seems to be due to the cost of context switching between the stage of the `IncomingTheaterConnection` and the stage of the `TheaterRingWorker`. When there is only one message being passed around, the thread of each stage wakes up from a notification when the message is placed in its mailbox, and after processing the message the thread goes back to sleep waiting for the next message as its mailbox is empty. This behavior would explain how increasing the number of rings improved performance, as this would further reduce context switching time. After 10 or so rings, the latency and bandwidth between theaters became the bottleneck, resulting in the linear performance from there as more rings were added.

4.6 Actor Migration Performance

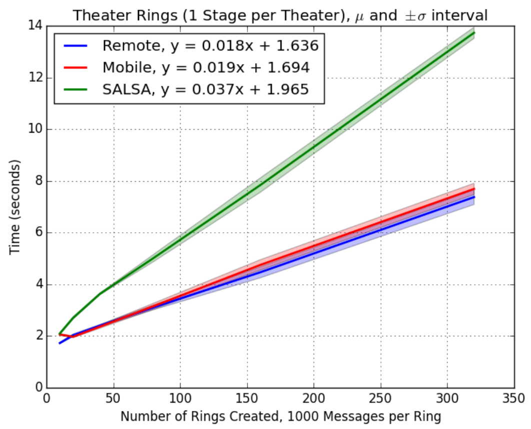
Figure 9 presents results for another new benchmark called `MigrationRings`, which examines the performance of migrating actors in a distributed system. The benchmark creates N actors, and each are given the same list of theaters in the system. Each actor starts at the same origin theater, and then migrates around the theaters in a ring until M migrations have been performed. For this benchmark, 1 to 320 actors were created with results being measured for 10, 20, 40, 80, 160 and 320 actors. Each actor performed 1000 migrations. These actors migrated around four theaters, one on each node in the Beowulf cluster. Each theater was created with one stage for these actors, however additional stages were created for the theater runtime actors. After each actor completed the given number of migrations, an acknowledgement is sent to the master actor, which terminates when it receives an acknowledgement for each actor. Results were gathered using both SALSA version 1.1.5 and mobile actors in SALSA Lite.

Similar to the results for the `TheaterRings` benchmark, mobile actors show weak performance with a single actor, however with more than one ring performance is very good. Additionally, these results also show a similar decrease in runtime when more concurrency is added, presumably due to less context switching. It should also be noted that unlike the remote creation and `TheaterRings` benchmarks, no deadlocks were detected in SALSA version 1.1.5 for these runs. Using the linear regression from results with 10 to 320 actors, with r-values greater than 0.999 for mobile actors and 0.984 for SALSA have mobile actors migrating 17.88 times faster than SALSA, which is a dramatic improvement in performance.

There are many factors in regards to this large performance improvement. First, SALSA actors are heavyweight, each with their own thread, so migration involves starting up and destroying threads as the actor migrates between theaters. Additionally, while the name server in SALSA Lite is used asynchronously as a boot strapping method, in SALSA, migration involves synchronously updating the actor's entry in the name server before performing migration. In many ways, this makes the name server a synchronous bottleneck to performance. While this can be somewhat alleviated by having multiple name servers, it is still a significant performance hit.

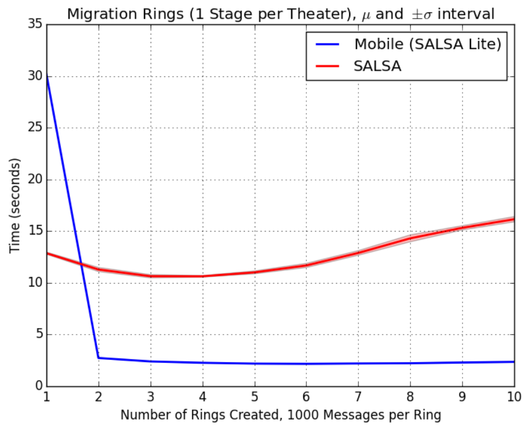


(a) 1 to 10 Rings

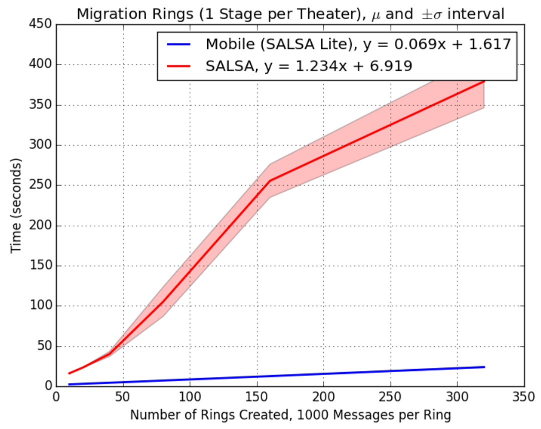


(b) 10 to 320 Rings

Figure 8: This figure shows results from the `TheaterRings` benchmark. 1 to 320 rings of actors were created, one actor per theater, and each ring sent 1000 messages around similar to the `ThreadRing` benchmark. Each ring operated concurrently. Results were gathered using SALSA version 1.1.5 as well as remote and mobile actors in SALSA Lite. The mean and standard deviation of 10 runs for each data point are shown.



(a) 1 to 10 Rings



(b) 10 to 320 Rings

Figure 9: This figure shows results from the `MigrationRings` benchmark. 1 to 320 actors were created, and each migrated 1000 times around four theaters. Results were gathered using SALSA version 1.1.5 and with mobile actors in SALSA Lite. The mean and standard deviation of 10 runs for each data point are shown.

5. CONCLUSIONS AND FUTURE WORK

This paper presents a new distributed runtime to enable remote and mobile actors in SALSA lite. This runtime is built using SALSA Lite's lightweight actors as a foundation to enable high performance and scalability. Performance improvements over SALSA version 1.1.5 are significant: remote message passing in SALSA Lite is 1.94 times faster for mobile actors, and 2.05 times faster for remote actors; migration of mobile actors in SALSA Lite is 17.88 times faster than SALSA; and remote creation of mobile and remote actors in SALSA Lite is two orders of magnitude faster. Additionally, with two new benchmarks, SALSA Lite is shown to have strong scalability in terms of concurrent actor execution. Further, in some of the more complicated benchmarks with lots of actor concurrency, SALSA version 1.1.5 suffered from deadlocks, while SALSA Lite did not, adding more justification for building the runtime using lightweight actors.

The performance overhead of using remote and mobile actors was also compared to SALSA Lite's local actor implementation. For message passing within a theater, remote actors resulted in 0% overhead, and mobile actors resulted in 30% overhead. Local creation of remote and mobile actors was more expensive with 54% overhead for remote actors and 438% for mobile actors. In distributed scenarios, creating mobile actors remotely was only 6% slower than creating remote actors remotely, and passing messages between mobile actors on different theaters was only 5.55% slower than passing messages between remote actors. In general, this overhead is found to be fairly low given the requirements of remote and mobile actors.

This investigation opens up many avenues for future work and analysis. In particular, some pathological cases for message passing were found in SALSA Lite when the cost of thread context switching becomes quite high. SALSA Lite uses Java's `LinkedList` class along Java's `synchronized` keyword to make access to it thread safe.[3] These pathological cases could be potentially eliminated by using lock-free data structures as used by CAF [10], or thread pools as in Scala [20] and Akka [9, 5]. Another potential area for improved performance would be the use of an asynchronous IO framework such as Netty [41] instead of Java's synchronous sockets, which could allow the `IncomingTheaterConnection` and `OutgoingTheaterConnection` actors to be lightweight instead of heavyweight.

Additionally, for a more in depth investigation of SALSA Lite's performance and comparison to other Actor programming languages, we intend to fully implement the Savina benchmark suite [24] which can potentially uncover other areas where performance can be improved and compare the performance remote messaging and actor migration to other modern implementations. Further, it would be beneficial to extend this suite with more benchmarks such as the `MigrationRings` benchmark discussed in this paper which can more fully test and evaluate the performance of actor migration. The syntax for remote actor referencing described in Section 3 can potentially be simplified even further using syntax described in [45]. Lastly, as touched on in Section 3.2.1, using Java's `readResolve()` and `writeReplace()` methods for serialization lays groundwork for investigating efficient implementations of distributed actor garbage collection [47, 13, 46, 39].

6. REFERENCES

[1] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and run-time monitoring. In *Hardware and Software, Verification and Testing*, pages 191–207. Springer, 2006.

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

[3] G. Agha and N. Jamali. Concurrent programming for distributed artificial intelligence. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to DAI.*, chapter 12. MIT Press, 1999.

[4] G. Agha and C. Varela. Worldwide computing middleware. In M. Singh, editor, *Practical Handbook on Internet Computing*, pages 38.1–21. CRC Press, 2004.

[5] J. Allen. *Effective Akka*. " O'Reilly Media, Inc.", 2013.

[6] S. R. J.-P. Arcangeli, F. Migeon, and S. Rougemaille. Javact: a java middleware for mobile adaptive agents. *Lab. IRIT, University of Toulouse, February 5th*, 2008.

[7] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[8] M. Astley. The actor foundry: A java-based actor programming environment. *University of Illinois at Urbana-Champaign: Open Systems Laboratory*, 1998.

[9] J. Bonér, V. Klang, R. Kuhn, et al. Akka library. *http://akka.io*.

[3]Java's concurrent list implementations were experimented with but resulted in poorer performance.

[10] D. Charousset, R. Hiesgen, and T. C. Schmidt. Caf-the c++ actor framework for scalable and resource-efficient applications. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 15–28. ACM, 2014.

[11] D. Charousset, R. Hiesgen, and T. C. Schmidt. Revisiting actor programming in c++. *arXiv preprint arXiv:1505.07368*, 2015.

[12] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch. Native actors: a scalable software platform for distributed, heterogeneous environments. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*, pages 87–96. ACM, 2013.

[13] S. Clebsch and S. Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. *ACM SIGPLAN Notices*, 48(10):553–570, 2013.

[14] T. Desell and C. A. Varela. Salsa lite: A hash-based actor runtime for efficient local concurrency. *Springer Lecture Notes in Computer Science: Concurrent Objects and Beyond*, 8665, 2013.

[15] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[16] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[17] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *ACM SIGPLAN Notices*, 39(1):256–267, 2004.

[18] A. Gosain and G. Sharma. A survey of dynamic program analysis techniques and tools. In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, pages 113–122. Springer, 2015.

[19] P. Haller and M. Odersky. Actors that unify threads and events. In *Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION)*, pages 171–190, 2007.

[20] P. Haller and M. Odersky. Actors that unify threads and events. In *Coordination Models and Languages*, pages 171–190. Springer, 2007.

[21] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.

[22] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[23] S. Hong and M. Kim. A survey of race bug detection techniques for multithreaded programmes. *Software Testing, Verification and Reliability*, 25(3):191–217, 2015.

[24] S. Imam and V. Sarkar. Savina-an actor benchmark suite. In *4th International Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE*, 2014.

[25] M.-W. Jang. The actor architecture manual. *Department of Computer Science, University of*

Illinois at Urbana-Champaign, 2004.

[26] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM, 2010.

[27] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *ACM Sigplan Notices*, 44(6):110–120, 2009.

[28] A. Jyoti and V. Arora. Debugging and visualization techniques for multithreaded programs: A survey. In *Recent Advances and Innovations in Engineering (ICRAIE), 2014*, pages 1–6. IEEE, 2014.

[29] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[30] R. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

[31] N. Kaveh and W. Emmerich. Deadlock detection in distribution object systems. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 44–51. ACM, 2001.

[32] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing'95*, pages 39–48, 1995.

[33] Y. Kwon, K. Mechitov, and G. Agha. Design and implementation of a mobile actor platform for wireless sensor networks. In *Concurrent Objects and Beyond*, pages 276–316. Springer, 2014.

[34] 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*, pages 1297–1300. ACM, 2006.

[35] M. Miller, E. Tribble, and J. Shapiro. Concurrency among strangers. *Trustworthy Global Computing*, pages 195–229, 2005.

[36] M. S. Miller and J. S. Shapiro. *Robust composition: Towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

[37] T. Salmito, A. L. de Moura, and N. Rodriguez. A flexible approach to staged events. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 661–670. IEEE, 2013.

[38] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In J. Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 104–128. Springer, 2008.

[39] C.-H. Tang, T.-Y. Song, M.-F. Tsai, and W.-J. Wang. Collecting mobile-agent garbage using actor-based weighted reference counting. *Advanced Science Letters*, 9(1):157–161, 2012.

[40] S. Tasharofi, P. Dinges, and R. E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *ECOOP 2013–Object-Oriented Programming*, pages 302–326. Springer, 2013.

[41] The Netty Project. Netty. [Accessed Online 2015] http://netty.io.

[42] R. Van Renesse. Goal-oriented programming, or composition using events, or threads considered harmful. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 82–87. ACM, 1998.

[43] C. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, U. of Illinois at Urbana-Champaign, 2001. http://osl.cs.uiuc.edu/Theses/varela-phd.pdf.

[44] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.

[45] C. A. Varela. *Programming Distributed Computing Systems: A Foundational Approach*. The MIT Press, 2013.

[46] W.-J. Wang. Conservative snapshot-based actor garbage collection for distributed mobile actor systems. *Telecommunication Systems*, 52(2):647–660, 2013.

[47] W.-J. Wang and C. A. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. In *Advances in Grid and Pervasive Computing*, pages 360–372. Springer, 2006.

[48] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, 2001.

[49] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *ECOOP 2005-Object-Oriented Programming*, pages 602–629. Springer, 2005.