Fault Tolerant Distributed Computing using Asynchronous Local Checkpointing

Phillip Kuang
Dept. of Computer Science
Rensselaer Polytechnic Institute
Troy, NY USA
kuangp@rpi.edu

John Field
Google
New York, NY USA
jfield@google.com

Carlos A. Varela
Dept. of Computer Science
Rensselaer Polytechnic Institute
Troy, NY USA
cvarela@cs.rpi.edu

Abstract

The transactor model, an extension to the actor model, specifies an operational semantics to model concurrent systems with globally consistent distributed state. The semantics formalizes tracks dependencies among loosely coupled distributed components to ensure fault tolerance through a two-phase commit protocol and to issue rollbacks in the presence of failures or state inconsistency. In this paper, we introduce the design of a transactor language as an extension of an existing actor language and highlight the capabilities of this programming model. We developed our transactor language using SALSA, an actor language developed as a dialect of Java. We first develop a basic transactor SALSA/Java library, which implements the fundamental semantics of the transactor model following the operational semantics' transition rules. We then illustrate two example programs written using this library. Furthermore, we introduce a state storage abstraction known as the *Uniform Storage Locator* following the *Universal Actor Name* and *Universal Actor Locator* abstractions from SALSA that uses a storage service to maintain checkpointed transactor states. The transactor model guarantees safety but not progress. Therefore, to help develop realistic transactor programs that make progress, we introduce the *Consistent Distributed State Protocol* and *Ping Director* that improve upon the *Universal Checkpointing Protocol* to aid transactor programs in reaching globally consistent distributed states.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—concurrent programming structures; D.4.5 [*Operating Systems*]: Reliability—checkpoint\restart, fault-tolerance; D.1.3 [*Programming Techniques*]: Concurrent Programming—distributed programming

General Terms Design, Languages, Reliability

Keywords Actor; Distributed state; SALSA; Transactor

AGERE! 2014, October 20, 2014, Portland, OR, USA..

http://dx.doi.org/10.1145/10.1145/2687357.2687364

1. Introduction

The transactor model introduced by Field and Varela is defined to as a "fault tolerant programming model for composing loosely-coupled distributed components running in an unreliable environment such as the Internet into systems that reliably maintain globally consistent distributed state"[4, 7]. Therefore, transactors allow for guarantees about consistency in a distributed system by introducing semantics on top of the actor model that allows it to track dependency information and establish a two phase commit protocol in such a way that a local commit succeeds only if local state is globally consistent. This allows a transactor to recognize reliance on other transactors and how they directly influence its own current state. As an extension of the actor model [1], transactors inherit the core semantics of encapsulating state and a thread of control to manipulate the state as well as communication through asynchronous messaging. In addition to these, transactors introduce new semantics to explicitly model node failures, network failures, persistent storage, and state immutability.

This paper presents a working implementation of the transactor model as a step toward developing a language to compose programs that follow an actor oriented programming paradigm that inherently maintains global state [9]. To do this we used the SALSA actor language [12–14] as a base on which we overlay transactor semantics. This is similar to how transactors naturally extend the actor model. This allows users to build loosely coupled distributed systems without a need for central coordination and takes into consideration the high latencies of a wide area network where node and link failures are common occurrences. Our imple-

mentation can also serve as a basis for further research on the transactor model as well for reasoning about composing transactor programs and fault tolerance.

The remainder of this paper is structured as follows: Section 2 provides some background information and important definitions from the transactor model. Section 3 describes our implementation of transactors. Section 4 illustrates a simple reference cell example that highlights our implementation. Section 5 describes how our implementation handles persistent state storage. Section 6 introduces a useful transactor abstraction known as the `Proxy`. Section 7 presents the *Consistent Distributed State Protocol* and the *Ping Director* that allow for creating programs that maintain globally consistent state. Section 8 shows new syntax added to SALSA that encodes the transactor semantics. Section 9 describes a detailed house purchase program example that makes use of our protocol. Section 10 presents related work. Finally, Section 11 concludes with a discussion and future work.

2. Background

In this section we provide a very brief summary of the transactor model and describe important terms used in the rest of the paper. We refer the reader to [4, 7, 9] for a formal definition of the model, which includes a complete operational semantics.

A transactor is composed of three key components: state, a thread of control that represents its behavior, and a *worldview*. The state of a transactor consists of two components: *persistent* and *volatile*. Persistent state has been committed to stable storage and is able to survive failures. Volatile state is vulnerable to failures until it has been committed and holds all changes that differ from a previously committed state. A transactor itself is said to be *permanent* if it has made an initial commit to obtain a persistent state, otherwise it is regarded as ephemeral. Ephemeral transactors are removed on failure to ensure consistency with global persistent state. A transactor's behavior defines its response to incoming messages. Similar to an actor, when a transactor receives a message, it may create new transactors, send messages or modify its own state. In addition to actor primitives, it also has the option to *stabilize*, *checkpoint*, and *rollback*. Stabilization is considered the first step of a two-phase commit protocol and makes a transactor immutable until a checkpoint or rollback occurs. The second step of the two-phase commit is a checkpoint which, if successful, commits the transactor's current state and guarantees consistency among peer transactors. That is, the current transactor state does not have a dependence on any other volatile transactor states. Lastly, rollback brings a transactor back to a previously checkpointed state.

The worldview abstracts over currently known dependency information and has three components: a history map, dependency graph, and root set. The history map is a collection of mapping from transactor names to transactor histories. The history of a transactor abstracts over how many times it has checkpointed and rolled back in the past. A history has three defining properties: a volatility value, incarnation value, and incarnation list. A history's volatility value indicates whether the current transactor is stable. Its incarnation value is a zero based numerical value which is incremented every time a rollback occurs. A checkpoint appends the current value to the history's incarnation list and reset its incarnation to maintain a record of past checkpoints and rollbacks. A dependency graph is a set of transactor dependencies represented as directed edges on transactor names. The root set captures dependencies of message payloads.

Dependency information is tracked by passing worldviews along with messages to other transactors. On reception of a message, a worldview union algorithm is applied to the current and received view, which reconciles these two views into an up-to-date view. Through this algorithm, the transactor model is able to propagate dependency information among interacting transactors. Dependencies are inherited and created by recognizing state mutations as a consequence of evaluating messages and are recorded appropriately by the worldview.

3. Implementation

Our language is first developed as a transactor library on top of the actor library used by SALSA compiled programs [12]. Figure 1 shows the class hierarchy diagram of our transactor library. A transactor is encoded in the `transactor.language.Transactor` class that extends and inherits from the `salsa.language.UniversalActor` class. In addition to the semantics inherited from a SALSA actor we create Java classes that encapsulate the semantics of a transactor worldview and history. Each transactor instantiates a `Worldview` but dependency semantics are meant to be transparent to the user. Similar to how SALSA implements a mailbox to handle message reception transparently, worldview operations are handled internally and the user cannot directly access such information except with supplied transactor primitive operators.

3.1 Message Passing

Message sending is inherited from SALSA as potential method invocations. We leverage the existing actor message handling implementation and add to the payload dependency information to accommodate the transactor model. Just as in SALSA, message sending is asynchronous and message processing is sequential though the ordering of messages is not guaranteed. Message parameters are passed by value to ensure there is no shared memory between transactor states. We provide two methods to the `Transactor` class that implements transactor message handling:

```
void sendMsg(String method, Object[] params,
             Transactor recipient)
```

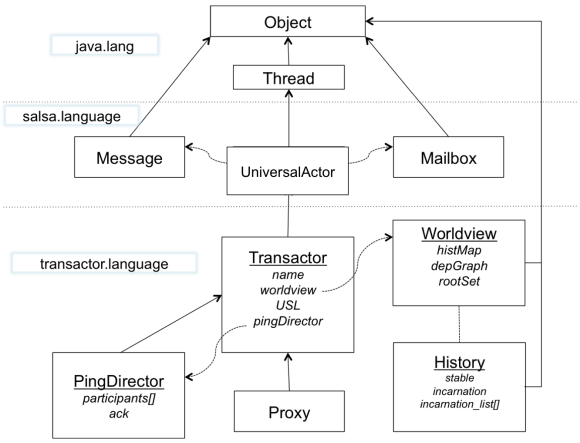

Figure 1. Class hierarchy diagram for the transactor library

```
void recvMsg(Message msg, Worldview msg_wv)
```

`sendMsg(...)` implements a message send by taking as arguments a string, `method`, that represents the type of message being sent which should match an appropriate message handler on the recipient transactor and an array of message parameters, `params`, which should match the arguments of the message handler signature. With these arguments we instantiate a `salsa.language.Message` object the same way messages are created in SALSA but instead we actually create a new message that wraps this requested message with the sender's worldview and call the corresponding `recvMsg(...)` message handler on the recipient.

On reception of a message, `recvMsg(...)` first evaluates the message dependencies by invoking the worldview union algorithm defined in [4, 9] on the recipient's worldview along with the message's worldview. Afterward we branch off to one of four receive transition rules [4, 9] in the transactor model by analyzing the reconciled dependency information. Invalidation of the recipient causes a rollback, annihilating it if it is ephemeral; invalidation of the message discards the message before it is processed; and if no invalidations are detected the message is placed in the transactor's mailbox to be processed normally by the SALSA actor library. In the cases where the recipient does not become invalidated, its worldview is updated to reflect the unionized worldview.

3.2 State Maintenance

State mutation and retrieval is done with the following two transactor methods:

```
boolean setTState(String field, Object newVal)
Object getTState(String field)
```

`setTState(...)` takes as arguments a string that represents the `field` being modified and the `newVal` to mutate the state with. Java reflection is used to reference the appropriate field in its state and mutation is done by replacing the value with the new value. State fields are therefore inherently immutable so set states are actually creating new states similar to mutating references to a Java `String`. Similarly `getTState(...)` uses Java reflection to de-reference and return the value of the requested `field`. Dependencies are created on the current worldview root set [4, 9] if a set state occurs and the transactor name is appended to its root set if a get state occurs.

3.3 Transactor Creation

Transactor creation is done with the following transactor method:

```
Transactor newTActor(Transactor new_T)
```

We use this method to extend the usual call to the `new` keyword in order to instantiate the newly created transactors worldview to reflect dependence on its parent. The `new_T` argument is an instantiated object of the transactor class to be created. The new transactor inherits the history map and dependency graph of the parent augmented with the new transactor's name and dependencies associated with the new transactor by the names in the parent's root set. Both parent and new child transactor will reflect the same history map and dependency graph but the parent will append the new transactor's name in its root set while the child starts with a fresh root set. This method returns the same reference to the new instantiated transactor with an updated worldview. The returned reference must then be type casted back to the constructed transactor class. The following code sample shows use of this method to create a new Foo transactor:

```
Foo FooActor = (Foo) newTActor(new Foo())
```

3.4 Fault Tolerance

Stabilization, checkpointing and rollbacks are provided in the form of the following three transactor operator methods:

```
void stabilize()
void checkpoint()
void rollback(boolean force, Worldview update)
```

`stabilize()` updates the transactor history volatility value to be stable and stores the current transactor state in stable storage if it is not already stable. `checkpoint()` marks the stored stable state as persistent, overwriting previous persistent states, if the transactor is independent and stable and clears its worldview. `rollback(...)` performs state reversion to the most recent checkpoint. The arguments `force` and `update` are used when an implicit rollback is caused by being invalidated by a received message. By passing a `true` to the first argument we can force the transactor to rollback under this scenario even if it is stable. The second argument represents the updated worldview obtained by the

worldview union algorithm so the rolled back state reflects this information.

Implementation of a state rollback is inspired by SALSA actor migration. Each transactor is inherently a SALSA actor, which encapsulates state in a thread of control so we handle rollbacks by halting the current thread and starting a new thread from a preserved checkpointed state and attaching the transactor name to it. However, before doing so, we create a special placeholder transactor, defined by the `transactor.language.Rollbackholder` class, to buffer incoming messages while the rollback operation is taking place. We register this placeholder state with the current transactor name under the SALSA naming service so messages can be routed correctly. We then read the checkpointed state from stable storage and tell the local system to start the transactor state as a new thread and reassign its name with the naming service. All buffered messages from the placeholder are then forwarded to the newly reloaded transactor's mailbox and normal processing resumes.

4. Reference Cell Example

```
behavior Cell extends Transactor {
  private int data = 0;

  public Cell(int data) {
    super(self);
    this.setTState("data", data);
  }

  public void set(int val) {
    this.setTState("data", val);
  }

  public void get(Transactor customer) {
    Object[] args = {((int)this.getTState("data"))};
    this.sendMsg("data", args, customer);
  }
}
```

Figure 2. Simple reference cell implementation

The example in Figures 2, 3, and 4 shows three different implementations of a reference cell written in SALSA using our transactor library. These three different versions highlight the semantics of the transactor model by providing progressively more refined notions of consistent state under different failure and interaction assumptions. It also illustrates application of the semantics implemented in our library. The first version (`Cell`) defines an unreliable reference cell that is volatile and never checkpoints so it does not tolerate failures. Any transactor that depends on the cell's value will not be able to checkpoint and a failure will cause it to be annihilated. This version of the reference cell is simply the actor implementation. The second version (`PCell`) presents a persistent reference cell that performs a checkpoint on initialization so it will be able to survive failures. This cell will also stabilize and checkpoint after each set to ensure the new value is persistent but this makes the assump-

```
behavior PCell extends Transactor {
  private int data = 0;

  public PCell(int data) {
    super(self);
    this.setTState("data", data);
  }

  public void initialize() {
    this.stabilize();
    this.checkpoint(); return;
  }

  public void set(int val) {
    this.setTState("data", val);
    this.stabilize();
    this.checkpoint(); return;
  }

  public void get(Transactor customer) {
    this.stabilize();
    Object[]args = {((int)this.getTState("data"))};
    this.sendMsg("data", args, customer);
    this.checkpoint(); return;
  }
}
```

Figure 3. Persistent reference cell implementation

```
behavior PRCell extends Transactor {
  private int data = 0;

  public PRCell(int data) {
    super(self);
    this.setTState("data", data);
  }

  public void initialize() {
    this.stabilize();
    this.checkpoint(); return;
  }

  public void set(int val) {
    this.setTState("data", val);
    if (this.isDependent()) {
      this.rollback(false, null); return;
    }
    else {
      this.stabilize();
      this.checkpoint(); return;
    }
  }

  public void get(Transactor customer) {
    this.stabilize();
    Object[] args = {((int)this.getTState("data"))};
    this.sendMsg("data", args, customer);
    this.checkpoint(); return;
  }
}
```

Figure 4. Persistent reliable reference cell implementation

tion that the sender is stable and independent, which may not be the case. It also stabilizes before responding to `get` to ensure no dependencies are incurred in the cell's customer and checkpoints to preserve the invariant of being checkpointed and volatile. The last version (`PRCell`) ensures that the cell is reliable in that it will only recognize set messages if the sender is independent and stable so no outstanding dependencies are created. A volatile sender will cause the cell to rollback and discard any changes to its state.

5. Persistent State Storage

5.1 Uniform Storage Locator

In order to handle persistent state storage, we introduce the *Uniform Storage Locator* (USL) to represent the location where checkpoints will be made. This location can be the local system, a remote server, or even the cloud, allowing the user to specify the optimal location to create persistent storage. The USL is inspired from the *Universal Actor Name* (UAN) and *Universal Actor Locator* (UAL) in SALSA and is a simple uniform resource identifier. Some examples of USLs are shown below. The first USL indicates local storage, the second indicates remote storage on a specified FTP server, and the last USL specifies storage on Amazon's Simple Storage Service (S3) cloud storage [3].

```
file://path/to/storage/dir/
ftp://user:pw@domain.com:123/path/to/dir/
http://s3.amazonaws.com/bucket/
```

Transactors are instantiated with a USL and if none is specified, checkpoints are made locally in the current directory. Specifying a USL is similar to specifying UAN and UAL in SALSA:

```
HelloWorld hwActor = new HelloWorld ()
at (new UAN("uan://nameserver/id") ,
    new UAL("rmsp://host1:4040/id"),
    new USL("file://path/to/storage/dir/");[1]
```

When a transactor checkpoints it will reference its USL to serialize its state and store a `<transactor-name>.ser` file at the location given by its USL. A rollback will reference the same file at the USL location to retrieve and de-serialize its state. The implementation of a USL also allows the possibility of mobile transactors similar to how a SALSA actor's UAN and UAL allow it to perform migration. Separating a transactor's storage location makes it location independent, allowing it to migrate as opposed to a locally checkpointing transactor. However further research still needs to be done on modeling mobile transactors whose state may be location dependent.

[1] This example is written with syntactic sugar which compiles to the `newTActor(Transactor new_T)` method.

5.2 Storage Service

Here we introduce the *Transactor Storage Service*, a service class that handles performing serialization/de-serialization of a transactor's state and storing/retrieving it at the transactor's USL. We implement this service as an interface shown in Figure 5. This simple interface has two methods for storing and retrieving state. We chose to create an interface to give the user the ability to implement his or her own desired serialization technique and USL protocol. Doing so gives the user the flexibility to define the optimal implementation that best caters to the given program specifications and performance requirements. For example a user might wish to use

```
public interface TSTorageService {
  public void store(Object state, URI USL);
  public void Object get(URI USL);
}
```

Figure 5. Transactor storage service interface

a FTP server to handle checkpoints and will create USLs with the `ftp://` scheme and implement the `store(...)` and `get(...)` methods to handle the FTP protocol with authentication. A high performance program can implement the use of cloud storage that has many benefits to program performance such as data redundancy and locality. Another high performance example is an implementation that utilizes memory storage instead of persistent storage to achieve fast checkpoint and rollback calls in a program that disregards the possibility of node failures.

6. Proxy Transactors

The *proxy transactor* is a special transactor whose task is to pass along messages it receives without affecting the dependencies of those messages. Proxy transactors implement a useful design pattern, but don't introduce any new semantic concepts, therefore they are special only in the sense that they implement a special case of general transactor semantics. Similar to a network proxy, a proxy transactor routes messages to other transactors and in doing so must not introduce any new dependencies on that proxy. Proxy transactors can prove useful in order to provide privacy for a certain resource or perform message filtering. We implement this abstraction by creating a `Proxy` transactor behavior that extends the `Transactor` behavior. By doing so we inherit all the semantics of a traditional transactor, however we will override the message send and receive implementations to prevent inserting volatile dependencies. We do so by simply issuing an explicit call to `stabilize` prior to sending or processing a new message. By stabilizing before sending a message, we guarantee that the recipient transactor remains independent with respect to the proxy. This affects situations where the proxy may perform a get state introducing its name to the message root set and the recipient subsequently performing a set state creating new dependencies on

the names in the message root set. If the recipient wishes to perform a checkpoint in the future then its worldview would have knowledge of the proxy being stable and therefore not impede it from doing so. We perform stabilization before processing a message upon reception to guarantee new dependencies are not introduced to the proxy. Transactor semantics have the property that any set state calls while processing a message become no-ops and therefore the proxy will not inherit any new dependencies from the message. Lastly, to eliminate any transitive dependencies that stem from a proxy, we restrict proxy creation only to transactors who meet two conditions: the transactor must be independent and stable and the names in the transactor's root set must also be independent and stable. We reason that this is logical because any invalidation of the parent transactor or transactors whose state resulted in the creation of the proxy will also invalidate the proxy and possibly any recipients of the proxy messages. This would be inconsistent with the semantics of a proxy transactor.

7. Consistent Distributed State

7.1 Consistent Distributed State Protocol

To aid in composing transactor programs, we introduce the *Consistent Distributed State Protocol* (CDSP)[2]. This protocol draws inspiration from the *Universal Checkpointing Protocol* (UCP) presented in [7]. The UCP guarantees progress under a set of preconditions. If these preconditions are met then global checkpoints are established through this protocol. However, a strict precondition of the UCP states that no failures can occur while the UCP is taking place and no transactors will rollback during the UCP. This assumes previous application dependent communication and a fault resistant system to guarantee these conditions are met. While it is proven that global checkpoints are possible in this type of situation, any failure would render the UCP useless. Such failures may halt program progress if the rest of the program is unaware of the failure without extra communication. Therefore we have introduced this new protocol to ensure global consistent states can be reached even in the presence of failures. From a theoretical perspective, the CDSP guarantees the same progress as the UCP under the same preconditions. However, from a practical perspective, the CDSP does not stale when a participant fails or rolls back thereby improving over the UCP.

We abstract over a consistent distributed state (CDS) update defined as a set of participating transactors communicating with each other that results in a global checkpoint or rollback. CDS updates are started by a *trigger message* sent by an outside agent whose recipient is defined to be the *coordinator*. On reception of the *trigger message* the message handler behavior defined in the *coordinator* starts the CDS update by starting the "conversation" among the participating transactors. During this "conversation", states may be altered and new dependencies might be created. Eventually the CDS update ends when all "conversations" have been completed and ideally we wish to reach a globally consistent state by issuing a global checkpoint or a rollback in the case of failure, to ensure consistency.

In order to reach a consistent state each participant must be made aware of the state of the transactors it has become dependent on and more importantly it must be made aware of any failures that may have invalidated itself. The UCP handles relaying this information though the use of `ping` messages whose main purpose is to carry dependency information and tell the recipient to attempt a checkpoint if it has enough information to be aware of its independence, otherwise it is a no-op and the cycle of ping messages continue until a checkpoint can be made. These ping messages can also be used to inform others of failure and cause rollbacks to invalidated transactors. The UCP only permits the use of ping messages to reach checkpoints while our protocol will also allow invalidation information to be carried along in ping messages.

In the CDSP protocol we define 5 preconditions:

1. The transitive closure of all participants and their dependencies accrued during the CDS update must be known ahead of time and each participant must be able to receive and issue ping messages.
2. There must be isolation of the participating transactors during the CDS update; i.e., communication only within the set of participants and messages may not be received from an outside transactor that would introduce new dependencies.
3. Each participant starts from a state that is independent of any transactors other than those among the participants. We also assume the outside agent who sends the *trigger message* will not introduce any dependencies on itself or any other outside transactors.
4. Each participant must be stable at the end of the CDS update unless it has rolled back at some point during the CDS update.
5. The *coordinator* must be able to recognize a CDS update has come to an end and indicates start of the consistency protocol.

Once a CDS update completes, each participant will send ping messages to all other participants and attempt a checkpoint if it is independent. On reception of a ping message, the transactor will also attempt a checkpoint.

Since each transactor arrives at a stable state at the end of a CDS update if it has not rolled back, a checkpoint succeeds if it is independent or has received enough ping messages to know it is independent. In the case of failure, a rolled back transactor is volatile at the end of a CDS update so all checkpoint calls will be no-ops. On the other hand, ping

[2] In [9] this is called the Consistent Transaction Protocol.

messages sent out from the failed transactor will alert all those who were dependent on it and invalidate them, causing them to also rollback. Therefore a globally consistent state is reached at the end of the CDS update through this protocol. This protocol also exemplifies eager evaluation of dependencies as opposed to the natural lazy evaluation of the transactor model. It is also important to note that the CDSP (and its UCP precursor) illustrate one possible way to achieve progress in transactor programs. By no means do we claim that it is the only way, or the way with the weakest preconditions. It may be entirely possible to create a protocol that allows for dynamic addition of participants in an open distributed system where knowing all involved participants ahead of time is not possible.

7.2 Ping Director

In order to accommodate the CDSP we introduce a new abstraction know as the *Ping Director* shown in Figure 6. The *Ping Director* is responsible for triggering the CDSP by requesting all participants to ping each other. We also extend the transactor with a new operator:

```
void startCDSUpdate(Transactor[] participants,
                    Transactor coordinator,
                    String msg,
                    Object[] msg_args);
```

and three additional message handlers native to all transactors:

```
void CDSUpdateStart(String msg,
                    Object[] msg_args,
                    PingDirector director);
void pingreq(Transactor[] pingreqs);
void ping();
```

The last two message handlers, `pingreq(...)` and `ping()`, give transactors the ability to send and receive ping messages. `pingreq(...)` takes an array of transactors and issues ping messages to each one, and `ping()` handles the reception of ping messages to attempt a `checkpoint`. The `startCDSUpdate(...)` method is a new transactor operator that is invoked by the outside agent who triggers the CDS update. This method takes as arguments the array of participants, the coordinator transactor to receive the trigger message, the trigger message, and the trigger message arguments. Internally this method will obtain an instance of the `PingDirector` that will handle the current CDS update and send a `pingStart(...)` message to the `PingDirector` instance with the array of participants, coordinator transactor reference, trigger message and its arguments. The `PingDirector` will then record in its state the array of participants and then send a `CDSUpdateStart(...)` message with the trigger message and its arguments and a reference to itself to the coordinator. The `PingDirector` also sends itself a `ping()` message to be described later. The `CDSUpdateStart(...)` message handler records the `PingDirector` instance reference in its state and sends the trigger message to itself to be processed and start the CDS update. Since messages from the `PingDirector` affect the state of the coordinator, we need the `PingDirector` to be independent so it will not affect the dependencies of the CDS update. We do so by having the system create an instance of the `PingDirector` through a new service known as the `CDSUpdateDirector`. We access the `CDSUpdateDirector` through the `salsa.language.ServiceFactory` and request a new instance of the `PingDirector` instead of explicitly creating one in the `startCDSUpdate(...)` method.

```
behavior PingDirector extends Transactor {
  private Transactor[] participants;

  public PingDirector();
  public void pingStart(Transactor[] participants,
            Transactor coordinator, String msg,
            Object[] msg_args);
  public void ping();
  public void endCDSUpdate();
}
```

Figure 6. `PingDirector`

When the coordinator recognizes the completion of the CDS update it will send an `endCDSUpdate()` message to the `PingD-irector` causing the `PingDirector` to stabilize. This stabilization alerts the `PingDirector` that the CDS update is complete. The `PingDirector` recognizes this alert through the `ping()` it sent itself at the start of the CDS update. On reception of a `ping()` message the `PingDirector` inspects its volatility value as an indicator of if the CDS update has completed. Before the CDS update has completed, the `PingDirector` will be volatile so we have the `PingDirector` resend the `ping()` message to itself until it recognizes it has stabilized in a polling manner. At that point the `PingDirector` will send `pingreq` messages to every participant and pass to each one the array of participants for them to ping.

The process of preparing and completing a CDS update is shown in Figure 7. Fortunately, this protocol is simplified by our abstraction and the user only needs to worry about indicating the start and end of a CDS update. An example of this abstraction being utilized is shown in the example in Section 9. We also note that a proxy transactor, described in the previous section, cannot be designated as a coordinator since it cannot alter its state to record a reference to the `PingDirector`. Semantically, proxies have no effect on the global dependency so therefore they do not participate in the CDSP, being that they will always be consistent with the global state.

We note that currently the CDSP assumes that the coordinator and ping director are resistant to failure. However if one of these agents failed then the CDSP would not be able

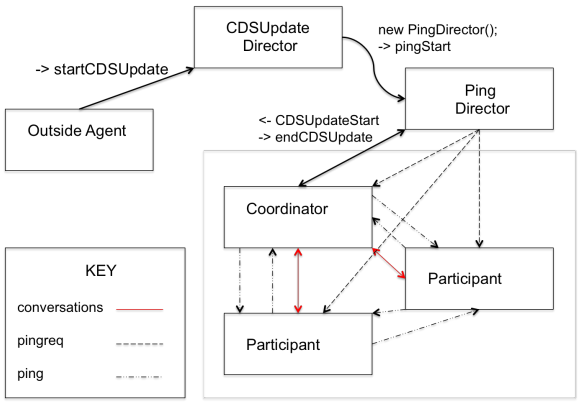

Figure 7. Consistent distributed state protocol using the `PingDirector`

to be triggered. To accommodate for this possibility we propose extending the protocol to provide fault tolerance in the form of redundancy. This can be done by assigning multiple coordinators where each would be able to recognize a CDS update completion and trigger the protocol if one fails. The same can be done with creating multiple ping directors for a CDS update and supplying a reference to each one to the coordinator. The exact details of implementing a fault tolerant CDSP are left as future work.

8. Language Syntax

Similar to SALSA, transactor programs are written as actor behaviors that are compiled into Java classes that extend the `transactor.language.Transactor` class. Through this inheritance chain, behaviors have access to an augmented set of operators that include both actor and transactor primitives. These operators can only be called by the transactor itself and are not explicit message handlers; therefore other transactors cannot directly issue a `stabilize`, `checkpoint`, or `rollback` on another transactor. These operators must be placed in message handlers inside the transactor's behavior. These operations are also sequential in nature, unlike message sends, which are concurrent. We define here our proposed syntax changes for our new transactor language that extends the SALSA/Java syntax.

The following statements are added to SALSA's syntax along with the compiled transactor library code:

```
stabilize;   ≡ this.stabilize();
checkpoint;  ≡ this.checkpoint(); return;
rollback;    ≡ this.rollback(false, null);
               return;
dependent;   ≡ this.dependent();
self;        ≡ this.self();

behavior <Identifier>
≡ behavior <Identifier> extends Transactor

behavior proxy <Identifier>
≡ behavior <Identifier> extends Proxy

startCDSUpdate(<ArgumentList>);
≡ this.startCDSUpdate(<ArgumentList>);

endCDSUpdate;
≡ this.sendMsg("endCDSUpdate", new Object[0],
(PingDirector)this.getTState("pingDirector"));

new <Transactor-Behavior>
≡ (<Transactor-Behavior>)this.newTActor(
                  new <Transactor-Behavior>);

<State-Identifier>:=<Expression>;
≡ this.setTState("<State-Identifier>",
                   <Expression>);

~!<State-Identifier>;
≡ ((<State-Identifier-Class>)this.getTState(
                       "<State-Identifier>"));
```

9. House Purchase Example

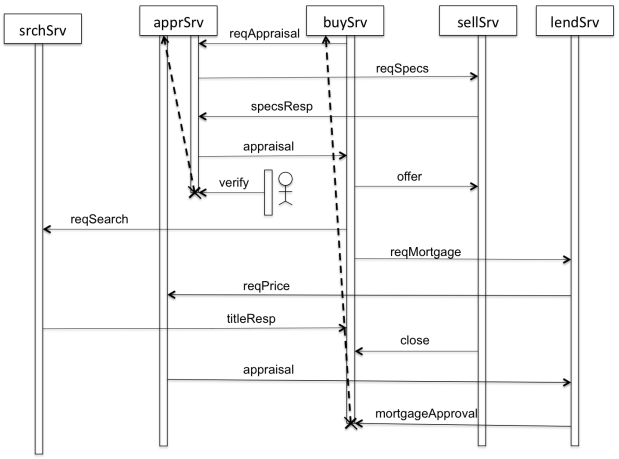

Figure 8. House purchase scenario involving semantic failure [7]

This example simulates the subset of operations that might be performed by a collection of web services involved in the negotiation of a house purchase. Traditionally, a house purchase is a complex task that involves multiple parties and back and forth communication. Some steps required include appraising the desired house, searching for the title, applying for a mortgage, and making negotiations. We represent these operations using five services: the `buySrv` representing the buyer, the `sellSrv` representing the seller, the `apprSrv` representing the appraisal service, the `lendSrv` representing the mortgage lender, and the `srchSrv` representing the

title search service. Our example defines the following steps taken to complete a house purchase:

1. The buyer chooses a candidate house and initiates the buySrv to manage the house purchase process.
2. The buySrv contacts the appraisal service, apprSrv, in order to obtain the market value of the house.
3. The apprSrv contacts the sellSrv and requests basic information about the house.
4. The apprSrv combines the house specifications with other reference information to compute a tentative market price. This tentative market price is only an estimate, which is not a definite appraisal until an on-site visit is made to the house to verify the accuracy of the original specifications.
5. The buySrv makes an offer to the sellSrv based on the appraisal. The buySrv also contacts the srchSrv to perform a title search and the lendSrv to obtain a mortgage.
6. The lendSrv contacts the apprSrv to confirm the appraisal information that is given after an on-site verification is completed.
7. The lendSrv approves the mortgage after a credit check and the buySrv will close the house purchase once it receives a response from the srchSrv and the sellSrv accepts the offer.

The steps above describe a scenario where every step runs accordingly without any semantic failures. However, one possible way this house purchase may fail can be observed in step 6 in the case of the verification discovering inaccurate information. Upon this discovery the apprSrv voluntarily rolls back its state in order to reprocess the verified specifications. This in turn causes the mortgage information to be inconsistent with the information the buySrv has. As a result, the buySrv must also be caused to rollback due to this invalidated dependency where it may choose to renegotiate the sale price. Figure 8 depicts this failure scenario.

Figures 9, 10, 11, 12, 13, 14, and 15 show our implementation of this example written in our proposed language syntax. Figure 12 is an implementation of the on-site verification process and Figure 14 represents a credit database contacted by the lender in order to obtain the buyers credit history used to calculate the requested mortgage. searchSrv, verifySrv, creditDB are implemented as proxies because they only provide access to a resource in order to obtain information and thus will not have an effect on the global dependency. This implementation also allows other types of failures to occur such as an offer rejection and mortgage denial. We make use of our *Consistent Distributed State Protocol* and *Ping Director* in this example to manage the house purchase transaction to notify all participants of a failure or issue a global checkpoint so we arrive at a globally consistent state. This transaction is started by the following call by an outside agent:

```
Transactor[] participants = {<buySrv>,
                    <sellSrv>,<apprSrv>,
                    <lendSrv>, <searchSrv>,
                    <verifySrv>, <creditDB>};
startCDSUpdate(participants, <buySrv>,
              "newHousePurchase", <houseid>);
```

An important observation can be made from this example highlighting how the transactor model tracks fine-grained dependencies. Though the use of the CDSP promotes atomicity of a transaction, its primary purpose is to guarantee consistency, as the name suggests. The atomicity aspect of the CDSP and the transactor model only applies to participants who are strictly invalidated by a dependency on a failed component. In that regard, other participants, such as the srchSrv who remains independent throughout the transaction, will not rollback even if another participant encounters failure. This key feature separates the transactor model from other traditional transaction methodologies that have an ”all or nothing” approach. Like the srchSrv, any participant who is semantically not affected by the overall result of the transaction will not have its operations reverted. This offers benefits in terms of preventing unnecessary rollbacks and not having to redo the same task if the transaction is attempted again allowing it to reuse results without having to recompute them. The srchSrv is a highly simplified implementation of an actual title search service that would involve a much more complex process. This process locates the required information for the title to the house, and this result would have to be re-computed if the search service were to rollback. If the overall transaction does fail and is reattempted, that title information will still be persistent, allowing us to reuse resources. The fact that the srchSrv is implemented as a proxy also ensures us that it has no effect on the global dependency of the transaction and will not incur any upon itself. Similarly, the verifySrv and creditDB both being proxies have no effect on the rest of the transaction and will not be caused to rollback.

10. Related Work

Though there already exists previous work that aims to support distributed state, ours is the first that provides a working implementation of the transactor model. Other types of systems include Liskov’s Argus [10] programming language. Argus provides an abstraction known as a *guardian* that is very much akin to a SALSA actor. Like an actor, guardians are meant to encapsulate a resource and permit access to its resources through *handlers*. Fault tolerance in Argus is provided with stable objects implemented as *atomic objects*, which allocate access through the use of locks to resolve concurrency. Similar to a transactor persistent and volatile state, atomic objects use versioning to handle recovery from

```
behavior lendSrv {
  buySrv buyer;
  String house;
  int price = 0;
  creditDB creditAgency;

  lendSrv() {}

  void initialize() {
    stabilize;
    checkpoint;
  }

  void reqMortgage(String houseid, buySrv buyr,
              int reqPrice, apprSrv appraiser,
              creditDB creditHistory) {
    house := houseid;
    price := reqPrice;
    buyer := buyr;
    creditAgency := creditHistory;
    appraiser<-reqPrice(self);
  }

  void appraisal(int newPrice) {
    price := newPrice;
    ~!creditAgency<-getCreditApproval(~!house,
                      ~!buyer, ~!price, self);
  }

  void approvalResp(String approvalid) {
    if (approvalid != null) {
      stabilize;
      ~!buyer<-mortgageApproval(approvalid);
    } else {
      ~!buyer<-mortgageDeny();
      rollback;
    }
  }
}
```

Figure 9. lendSrv implementation

```
behavior proxy searchSrv {
  HashMap titlesDB;

  searchSrv(HashMap titlesInfo) {
    titlesDB := titlesInfo;
  }

  void initialize() {
    checkpoint();
  }

  void reqSearch(String houseId,
              Transactor customer) {
    customer<-titleResp(~!titlesDB.get(houseId));
  }
}
```

Figure 10. srchSrv implementation

```
behavior sellSrv {
  HashMap minPrices, specs;
  int offeredPrice = 0;

  sellSrv(HashMap specsInfo, HashMap mins) {
    specs := specsInfo;
    minPrices := mins;
  }

  void initialize() {
    stabilize;
    checkpoint;
  }

  void reqSpecs(String houseId,
              Transactor customer) {
    customer<-specsResp(~!specs.get(houseId),
                  ~!minPrices.get(houseId));
  }

  void offer(String houseId,
           int price, buySrv buyer) {
    offeredPrice := price;
    if (price >= ~!minPrices.get(houseId)) {
      stabilize;
      buyer<-close();
    } else {
      buyer<-rejectOffer();
      rollback;
    }
  }
}
```

Figure 11. sellSrv implementation

```
behavior proxy verifySrv {
  HashMap specs, prices;

  verifySrv(HashMap newSpecs, HashMap newPrices) {
    specs := newSpecs;
    prices := newPrices;
  }

  void initialize() {
    checkpoint;
  }

  void verifySpecs(String houseid,
                String reqSpecs,
                Transactor customer) {
    if (reqSpecs.equals(~!specs.get(houseid))) {
      customer<-verify(true, ~!prices.get(houseid));
    } else {
      customer<-verify(false, ~!prices.get(houseid));
    }
  }
}
```

Figure 12. verifySrv implementation

```
behavior buySrv {
  searchSrv searcher;
  apprSrv appraiser;
  sellSrv seller;
  lendSrv lender;
  verifySrv verifier;
  creditDB creditHistory;
  int price = 0;
  String title, mortgage, houseid;

  buySrv(searchSrv srchr, apprSrv appr,
      sellSrv sellr, lendSrv lendr,
      verifySrv verifr, creditDB cHistory) {
    searcher := srchr;
    appraiser := appr;
    seller := sellr;
    lender := lendr;
    verifer := verifr;
    creditHistory := cHistory;
  }

  void initialize() {
    stabilize;
    checkpoint;
  }

  void newHousePurchase(String newHouseId) {
    houseid := newHouseId;
    ~!appraiser<-reqAppraisal(~!houseid, self,
                         ~!seller, ~!verifier);
  }

  void appraisal(int newPrice) {
    price := newPrice;
    ~!seller<-offer(~!houseid, ~!price, self);
    ~!searcher<-reqSearch(~!houseid, self);
    ~!lender<-reqMortgage(~!houseid, self,
                          ~!price, ~!appraiser,
                          ~!creditHistory);
  }

  void titleResp(String newTitle) {
    title := newTitle;
  }

  void mortgageApproval(String approvalid) {
    mortgage := approvalid;
  }

  void close() {
    if (~!title != null && ~!mortgage != null) {
      stabilize;
      endCDSUpdate;
    } else {
      self<-close();
    }
  }

  void rejectOffer() {
    endCDSUpdate;
    rollback;
  }

  void mortgageDeny() {
    endCDSUpdate;
    rollback;
  }
}
```

Figure 13. buySrv implementation

```
behavior proxy creditDB {
  creditDB() {}

  void initialize() {
    checkpoint;
  }

  void getCreditApproval(String houseid,
                   buySrv buyer, int price,
                   lendSrv requester) {
    requester<-approvalResp("approval-" + houseid);
  }
}
```

Figure 14. creditDB implementation

```
behavior apprSrv {
  String house, specs;
  int price = 0;
  buySrv buyer;
  Transactor requester;
  verifySrv verifier;

  apprSrv() {}

  void initialize() {
    stabilize;
    checkpoint;
  }

  void reqAppraisal(String houseid, buySrv buyr,
           sellSrv seller, verifySrv verifr) {
    buyer := buyr;
    house := houseid;
    verifier := verifr;
    seller<-reqSpecs(~!house, self);
  }

  void specsResp(String newSpecs, int newPrice){
    specs := newSpecs;
    price := newPrice;
    ~!buyer<-appraisal(newPrice);
  }

  void reqPrice(Transactor customer) {
    requester := customer;
    ~!verifier<-verifySpecs(~!house,
                            ~!specs,
                            self);
  }

  void verify(boolean ok, int verifiedPrice) {
    if (ok) {
      stabilize;
      ~!requester<-appraisal(verifiedPrice);
    } else {
      ~!requester<-appraisal(verifiedPrice);
      rollback;
    }
  }
}
```

Figure 15. apprSrv implementation

failures. Unlike transactors, Argus does not directly track dependencies and takes an "all or nothing" approach to determining if a set of operations should be committed.

Another system is Atomos [6], introduced by Carlstrom et al. to be a transactional programming language with implicit transactions, strong atomicity, and scalable multiprocessor implementation. Atomos relies on the transactional memory model, which executes read and write instructions in an atomic way. Unlike Argus, but comparable to transactors, Atomos provides open nested transactions, which immediately commit child transactions at completion. Like transactors where independent agents of a failed transaction can still checkpoint, the rollback of a parent transaction is independent from completed open nested transactions.

Stabilizers [16] introduced by Ziarek et al. is a linguistic abstraction that models transient failures in concurrent threads with shared memory. These abstractions enforce global consistency by monitoring thread interactions to compute the transitive closure of dependencies. Like transactors, any non-local action such as thread communication or thread creation constitutes state dependency; however, these dependencies are recorded even if there is no state mutation. In the presence of transient failure, rollbacks are performed that revert state to a point immediately preceding some non-local action. Unlike transactors, there is no predefined concrete checkpoint to rollback to since stabilizers perform thread monitoring instead of state captures.

Some other relevant pieces of work worth mentioning include Orleans [5], Ken [15], and Sinfonia [2]. Orleans is an actor framework for .Net that allows for creating distributed transactions. Actors are known as *grains* and also internally track dependencies. Similar to transactors, state can be persisted to durable storage like checkpoints and a reconciliation mechanism handles lazily merging state changes. In Orleans, grains pass messages in a way very similar to SALSA where a *promise* is returned to the sender as a pending future result the same way *tokens* are used in SALSA. Transactions are also isolated where one transaction cannot access data modified by another pending transaction. Ken is a protocol that coordinates a set of processes, which are like actors. Fault tolerance is tackled with asynchronous local checkpointing. When a process handles a computation, all actions within the computation are committed as a single atomic unit and any outbound messages are buffered until the computation succeeds. Messages are re-transmitted until an ACK is received which indicates receipt and success of the resulting computation at the recipient end. This guarantees all computations and their consequences can tolerate failure. Lastly, Sinfonia is a service that seeks to mitigate the complexity of two phase commit protocols. In a manner similar to the CDSP, Sinfonia reasons that the two-phase protocol can be made more efficient by piggybacking actions on the first phase. This has some similarity to transactor stabilization at the end of its actions (following the CDSP) without waiting for an explicit trigger to start the first phase.

Transactions that are modeled under object-oriented paradi-gms with concurrent threads usually interact through shared memory. As a result, maintaining the integrity of a transaction has largely relied on issuing locks on objects to prevent race conditions. However the biggest problem with such techniques is the possibility of deadlock causing it to be relatively difficult to compose transactional programs correctly. One remedy to this problem is introduced in the Software Transactional Memory [11] model which logs reads and writes within a thread that accesses shared memory. Transactions are then validated once complete and committed or aborted. This is a similar alternative to the transactor model but still assumes a shared memory model whereas transactors assume a distributed memory model with message passing. Therefore, STM model operates on a different domain whose semantics are somewhat orthogonal to that of transactors. The message passing and state encapsulating nature of actors allows them to naturally model atomicity and isolation of message execution, thereby eliminating the need for object-level locks. The semantics of the transactor model provides a much cleaner and more robust building block to model transactions.

11. Discussion and Future Work

A reliable transaction is commonly defined by its *ACID* properties. While the transactor model only guarantees consistency and durability, transactors break down a transaction into its fundamental elements. Atomicity and isolation can be coded into the model if desired, however as shown in our example, transactors provide a looser form of atomicity that we call *selective rollback*. This means that we only undo what is known to be inconsistent. Full isolation is also not a strict requirement for transactor programs as stated in one of the preconditions of the CDSP that requires that obtaining new dependencies on outside transactors not be allowed. We refer to this as *selective state access*. State accesses that create backward dependencies are perfectly legal since it does not prevent the participating transactor from checkpointing. Therefore, lack of full *ACID* properties is a design feature allowing for the creation of lightweight and modular transactions.

Though our language is currently a working implementation of the transactor model, it is still in a developmental stage and there is much work yet to be done. As a consequence of its development it also opens up new directions in the study of transactors. Our next objective would be to develop a compiler similar to the SALSA compiler to produce SALSA/Java code that can be compiled and run on a JVM. This compiler would greatly simplify writing transactor programs with the proposed syntax, which inherits much of the familiar SALSA and Java grammar.

Another key future goal is implementing node failure semantics. Following the transition rules of the transactor model, a transactor system needs to be able to recognize node failures and reload transactors from persistent storage. A record of previously running transactors on the node would be required, perhaps as an extension of the naming service. The program would then proceed normally as if a rollback has occurred. This also opens up concerns on how to bootstrap programs and restart the network of messages. Along with bootstrapping programs there is an open question of whether to initially checkpoint the startup transactor to prevent total program annihilation if the startup node fails before it becomes persistent.

An improvement can also be made to the CDSP to guarantee full isolation among participants and satisfy one of its preconditions. One possible technique is to apply a two-phase CDS update initialization protocol similar to the two-phase commit protocol. The necessity of a two-phase process is due to the message passing nature of transactors where there is no guarantee of when messages will arrive or even be received. Such a protocol could involve the use of synchronization constraints such as *Synchronizers* [8] that handle message dispatching to disable messages arriving from outside transactors. Achieving isolation would be valuable so the user would only have to reason about the specifics of a CDS update rather than consider its reliability.

A future direction to the study of transactors is modeling migration. SALSA has built in support for actor migration and our transactor language allows transactors to be initialized in different SALSA theaters. However, there are concerns over whether location is represented by a transactor's state where an implementation would have to perform reverse migrations should a transactor ever rollback. Migration also becomes a factor in implementing node failure where each node would have to track which transactors would have to be recovered. Transactor USL was developed to permit the possibility of mobile transactors so persistent state storage would not become a limiting factor.

Lastly, interaction between transactors can be simplified by implementing continuations. Currently, in order to retrieve information from another transactor, the sender's name nee-ds to be passed along with the message so the recipient knows where to send a reply. Continuations would make it easier to compose transactor programs by emulating serialized execution among asynchronous transactors. SALSA provides this in the form of *tokens*. However, research needs to be done to consider how to model tokens in the τ calculus which formalizes the transactor model so that dependencies can be maintained correctly.

References

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

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(6):159–174, Oct. 2007.

[3] Amazon Web Services. Amazon simple storage service documentation. http://aws.amazon.com/documentation/s3/.

[4] B. Boodman. Implementing and verifying the safety of the transactor model. Master's thesis, Rensselaer Polytechnic Institute, May 2008.

[5] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 16:1–16:14, New York, NY, USA, 2011. ACM.

[6] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. *SIGPLAN Not.*, 41(6):1–13, June 2006.

[7] J. Field and C. A. Varela. Transactors: A programming model for maintaining globally consistent distributed state in unreliable environments. *SIGPLAN Not.*, 40(1):195–208, Jan. 2005.

[8] S. Frølund. *Coordinating Distributed Objects: An Actor-based Approach to Synchronization*. MIT Press, Cambridge, MA, USA, 1996.

[9] P. Kuang. Implementation of the transactor model: Fault tolerant distributed computing using asynchronous local checkpointing. Master's thesis, Rensselaer Polytechnic Institute, July 2014.

[10] B. Liskov. Distributed programming in Argus. *Commun. ACM*, 31(3):300–312, Mar. 1988.

[11] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

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

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

[14] C. A. Varela, G. Agha, W. Wang, T. Desell, K. E. Maghraoui, J. LaPorte, and A. Stephens. The SALSA programming language: 1.1.2 release tutorial. Technical Report 07-12, Dept. of Computer Science, R.P.I., Feb. 2007.

[15] S. Yoo, C. Killian, T. Kelly, H. K. Cho, and S. Plite. Composable reliability for asynchronous systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[16] L. Ziarek, P. Schatz, and S. Jagannathan. Stabilizers: A modular checkpointing abstraction for concurrent functional programs. *SIGPLAN Not.*, 41(9):136–147, Sept. 2006.