Transactors: Unifying Transactions and Actors

Mohsen Lesani

Abstract

Composability and deadlock-freedom are important properties that are stated for transactional memory (TM). Commonly, the Semantics of TM requires linearization of transactions. It turns out that linearization of transactions that have cyclic communication brings incomposability and deadlock. Inspired from TM and Actors, this work proposes Transactors that provide facilities of isolation from TM and communication from Actors. We define the semantics of Transactors including support for cyclic transactional communication. An algorithm implementing this semantics is offered. The soundness of the algorithm is proved.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel programming

General Term Algorithms, Languages, Theory

Keywords Transactional memory, Actors

1 Introduction

To preserve consistency of data, concurrent operations involving a sequence of accesses (reads and writes) to shared memory should be executed in isolation. To coordinate cooperative tasks, threads need to communicate. Therefore, a concurrent programming model is expected to provide means of isolation and communication.

As a concurrency programming model, Transactional Memory (TM) [9] allows the programmer to declare blocks of code as transactions and the TM runtime system guarantees that transactions are linearized, i.e. run in isolation. In this paradigm, communication among transactions should also be done by reading and writing to shared memory. The sender transaction writes to a memory location. The receiving transaction reads from the memory location after the sender transaction commits. TM is shown to be deadlock-free and composable [8] in provision of isolation. Because of the linearization guarantee, as examples will show, transactions with cyclic communication are incomposable and can deadlock.

Actors concurrency programming model, support communication elegantly by message passing mechanism. Actors provide coarse-grained isolation by the fact that at any point of time at most one thread is scheduled to execute code for each actor instance [6]. This means that operations executed for an actor instance are serialized and hence are done in isolation to each other.

This work aims at merging the strengths of the two paradigms, i.e. isolation from TM and communication from Actors. To this end, we define the semantics of the new model including support for transactional communication and propose an algorithm realizing this semantics. The soundness of the algorithm is proved. Especially, it is proved that every transaction is eventually finalized, i.e. aborted or committed. This guarantees that the algorithm is deadlock-free even while there are cyclic dependencies.

Motivating examples are presented in the next section. In subsequent sections, semantics of Transactors are defined and the algorithm that implements the semantics is explained. Soundness theorems come afterwards. Related works finalize the paper. Some sections refer to the same section numbers of the appendix of the accompanying technical report [11] for details or proofs.

2 Incomposability and Deadlock

To explain the problem of incomposability and deadlock in transactional communication, two examples are presented in the following subsections. First, as a simple example, roundtrip is presented and then barrier is explained as a realistic case.

2.1 Transactions

2.1.1 Roundtrip

Consider the simple roundtrip example that exhibits the problem abstractly. A transaction sends a message to another transaction and then receives a message from it. This is implemented in SSTM (Scala Software Transactional Memory) as follows. (We have implemented SSTM very similar to [9].) The Semantics of `conditionWait` in SSTM matches the semantics of `retry` of Haskell STM [8]. (If the condition fails, the transaction is aborted and not retried until one of the objects that the transaction has read before being aborted is updated.)

Assume that two transactions T_1 and T_2 use respectively two shared variables `m1` and `m2` to pass messages to each other:

```
val m1 = new Tint(0), m2 = new Tint(0)
```

The first transaction, T_1, is

```
atomic {
   m1.value = 1
   conditionWait(m2.value == 1)
}
```

The second one, T_2, is

```
atomic {
   conditionWait(m1.value == 1)
   m2.value = 1
}
```

We show that the execution of the two transactions leads to a deadlock either with a deferred-update or direct-update STM implementation. In a deferred-update STM implementation the following happens. When T_1 is being executed, `m1` is tentatively updated and then the condition `m2.value == 1` is checked. Because `m2` should be updated by T_2 and updates are deferred, condition of T_1 is only satisfied when T_2 commits. When T_2 is being executed, it checks for condition `m1.value == 1`. Since `m1` should be updated by T_1 and updates are deferred, the condition of T_2 is satisfied only when T_1 commits. Therefore, neither of T_1 and T_2 can pass the condition. So both abort and go to the waiting state that results in a deadlock. Explanation for direct-update STM implementation is presented in the technical report [11].

To see where such roundtrips can happen in practice, we present the implementation of a barrier abstraction with transactions.

2.1.2 Barrier

Transactional communication can bring deadlock when classes implemented by transactions are composed. Consider the following example: Barrier, the simplest thread coordination the we adopted from [12]. `Barrier` class is implemented as follows:

```
class Barrier(partiesCount: Int) {
   val count = new TInt(0)
   def await() {
      atomic {
         count.value = count.value + 1
      }
      atomic {
         conditionWait(count.value == partiesCount)
      }
   }
}
class Party(barrier: Barrier) extends Thread {
   override def run {
      // Do some job
      barrier.await
      // Do some other job
   }
}
```

There are two atomic blocks in the `await` method. The first one increments the value of `count`. `count` is a field of `Barrier` class of type transactional integer that counts the number of parties that have called the `await` method. The second atomic block waits for equality of `count` to the number of expected parties, `partiesCount` that is initialized in the constructor. If a condition is not true, the thread is suspended until at least one of the objects that the transaction has read is updated. When the value of `count` is incremented to `partiesCount`, all of the suspended parties retry the atomic block and as the condition is satisfied, pass the atomic block. Effectively, the parties continue together after calling the `await` method.

The implemented `Barrier` works properly if the `await` method is not called inside a transaction. To see if `Barrier` is composable, consider the following `TParty` class that calls `await` inside an atomic block.

```
class TParty(barrier: Barrier) extends Thread {
   override def run {
      atomic {
         // Do some job
         barrier.await
         // Do some other job
      }
   }
}
```

For the purpose of presentation, the nesting can be written syntactically as follows:

```
atomic {
   atomic {
      count.value = count.value + 1
   }
   atomic {
      conditionWait(count.value == partiesCount)
   }
}
```

There are two known semantics for nested atomic blocks: closed nesting and open nesting. `Barrier` is composable neither with closed nor open nesting semantics.

By the closed nesting semantics, the updates of the inner transactions are all committed when the outermost transaction commits. For a `conditionWait` in an inner transaction, there are two approaches. Either the condition is moved to the beginning of the outermost transaction [7] or the condition is evaluated in-place and if the condition fails, the outer transaction aborts and before retrying waits until at least one of the objects that it has read is updated [12]. In both of these approaches, all of the parties that call the `await` method go to deadlock:

- If the condition is checked at the beginning of the outermost atomic block, it is never satisfied. This is because the evaluation result of the condition can only change by the updates that are inside the atomic block itself.
- Also if the outer transaction aborts and goes to waiting state when the condition is failed, the parties go to deadlock. We explain about deferred-update STM implementation here. Explanation for direct-update STM implementation is presented in the technical report [11]. If STM implementation is deferred-update, the first transaction reads a value of zero from `count` and tentatively updates it to one in its own copy. As the condition fails, the transaction ignores its tentative update, aborts and goes to the waiting state. As no transaction commits and updates are deferred, any transaction that reads value of `count` gets zero. Therefore, any later transaction also aborts and goes to the waiting state resulting in a deadlock.

Explanation for open nesting is presented in the technical report [11].

This means that the `await` method of `Barrier` cannot be used inside nested atomic blocks and therefore `Barrier` implemented by STM is not composable.

A solution to this problem based on closed nesting called TIC is offered by Smaragdakis et al. [12]. TIC commits the transaction and starts a new one before the `wait` statement. By their terminology, the transaction is punctuated before the `wait` statement. Committing before the wait statement exposes updates to other transactions and thus provides means of communication. But punctuation of an atomic block breaks its isolation. Furthermore, if an atomic block A_1 is inside method M_1 and M_1 is called by another method M_2 inside a nesting atomic block A_2, punctuating A_1 breaks isolation of not only A_1 but also A_2. To make this break explicit to the programmer, TIC designed a type system that tracks methods that contain punctuated atomic blocks. If the programmer wants to call such methods in an atomic block, the type system forces him to call it inside `expose()` and to write code to compensate the breaking of isolation in `establish{}` block. The barrier case is implemented as follows in TIC:

```
class TICBarrier(partiesCount: Int) {
   val count = new TInt(0)
   def await() {
      atomic {
         count.value = count.value + 1
         wait(count.value == partiesCount)
      }
   }
}
class TParty(barrier: TICBarrier) extends Thread {
   override def run {
     atomic {
         // Do some job
         expose (barrier.await)
         establish { //... }
         // Do some other job
     }
   }
}
```

Even if any compensation is possible, re-establishing local invariants is a burden on the programmer. More importantly, TIC breaks isolation for communication while isolation is the main promise of TM. Actually, TIC treats communication the same as I/O. Side effects caused by I/O operations are out of control of TM runtime system; thus they can break isolation and cannot be rolled back and retried. This is in contrast to communication, for which proper mechanisms can be designed to perform communications tentatively and to discard and retry them on aborts. Our proposal for semantics and implementation of these mechanisms is explained in the following sections. By these mechanisms, Transactors provide the facility for the programmer to send and receive messages inside transactions while composability and isolation are preserved.

2.2 Transactors

A transactor is essentially a thread that can send and receive messages both outside and inside transactions. In fact, Transactor model includes features from both TM and Actor models. An atomic block inside a transactor can not only read from and write to shared memory but also send messages to and receive messages from other transactors. All of the required mechanisms to keep track of messages sent by aborted transactions are maintained by the Transactors runtime system.

```
class MyTransactor(peer: Transactor) extends Transactor {
   val i1 = 0, i2 = 0  //Non-transactional objects
   val ti1 = TInt(), ti2 = TInt() //Transactional objects
   override def act {
      // Outside atomic block
      val v = i1  //Read non-transactional objects
      i2 = v  //Write non-transactional objects
      peer ! new MessageClass  //Send a message
      receive {  //Receive a message
         case MessageClass1 => //...
         case MessageClass2 => //...
      }
      atomic {
         // Inside atomic block
         val tv = ti1.value //Read transactional objects
         ti2.value = tv  //Write transactional objects
         peer ! new MessageClass  //Send a message
         receive {  // Receive a message
            case MessageClass1 => //...
         }
      }
   }
}
```

Similar to actors, each transactor has a mailbox where messages sent to the transactor are enqueued. A transactor can dequeue messages from its mailbox by `receive`. The input parameter to `receive` is a partial function which is defined for a set of message types. When a transactor executes `receive`, if a message of a type that the partial function is defined for is not in the mailbox, it waits until such a message is enqueued.

2.2.1 Roundtrip

The roundtrip example can be coded simply in Scala Transactors as follows:

```
class Transactor1(peer: Transactor) extends Transactor {
   override def act {
      atomic {
         peer ! new Message
         receive { case Message => }
      }
   }
}
class Transactor2(peer: Transactor) extends Transactor {
   override def act {
      atomic {
         receive { case Message => }
         peer ! new Message
      }
   }
}
```

Transactions inside Transactors can send and receive tentative messages from each other. After completion, they are finally committed together. The mechanisms behind transactors are explained in the following sections.

2.2.2 Barrier

This subsection explains implementation of the barrier case by Scala Transactors. Consider the following code snippet. A class called `BarrierActor` that extends base class `Transactor` is defined inside `Barrier` class. Inside an atomic block in its `act` method, `BarrierActor` waits to receive `JoinNotificationRequest` message from the parties and adds the sender transactor of each received message to `parties` set. After receiving the request form `partiesCount` parties, it sends a `JoinNotification` message to all the parties in `parties` set. On construction of a `Barrier`, a new object called `barrierActor` of type `BarrierActor` is created and started. When a party calls the `await` method on a `Barrier` object, an atomic block is executed that sends a `JoinNotificationRequest` message to the `barrierActor` and waits to receive `JoinNotification` message. In Transactor model, transactions can communicate tentatively by the message passing mechanism. Therefore, in contrast to the first barrier implementation using TM transactions, the current implementation does not go to deadlock waiting for messages from others. The `await` method can be called inside a nested atomic block and the implemented barrier is composable. In addition, in contrast to TIC, composable communication is supported without breaking isolation.

```
class Barrier(partiesCount: Int) {
   class BarrierActor extends Transactor {
      override def act {
         atomic {
            val parties = Set[Transactor]()
            for(i <- 0 until partiesCount) {
               val request = receive {
                  case JoinNotificationRequest =>
                     parties += request.sender
               }
            }
            for(party <- parties) {
               party ! new JoinNotification
            }
         }
      }
   }
   val barrierActor = new BarrierActor
   barrierActor.start
   def await() {
      atomic {
         barrierActor ! new JoinNotificationRequest
         self.receive { case JoinNotification => }
      }
   }
}
class TParty(barrier: Barrier) extends Transactor {
   override def act {
      atomic {
         barrier.await
      }
   }
}
```

As will be explained in the next sections, by the semantics, a receiving transaction becomes dependent on the sender transaction. A transaction can be aborted as a result of conflict resolution with another transaction. When a transaction is aborted, abortion is propagated to dependent transactions.

When a party has sent a message to the barrier transactor and is waiting to receive a reply message, the transaction of the barrier is dependent on the transaction of the party. If the transaction of the party aborts, the abort is propagated to the transaction of the barrier. While the transaction of barrier is aborting, the messages from the other parties that are not aborted are restored to the barrier's mailbox. Therefore, on retry, the atomic block of the barrier can receive the same set of request messages as its previous execution other than the request from the aborted party. This means that the barrier transactor effectively ignores the aborted party and waits for another.

When the barrier has received request messages from all the parties and the parties are released after receiving messages from the barrier, the transaction of each party and the transaction of the barrier are interdependent. If the transaction of one of the parties aborts, abort propagates to the transaction of the barrier and then transactions of all of the other parties. In other words, if one of the parties aborts, the barrier and all of the parties are aborted and retried. This matches the expected behavior from the barrier that all of the parties together or none of them should pass the barrier.

3 Semantics

In this section, properties that are expected from a Transactor algorithm are specified the first subsection. The lemmas needed as the background for the operational semantic are established in the second subsection. The last subsection presents the operational semantics.

3.1 Algorithm Specification

A transaction starts from the running state and can change states as shown in Figure 1.

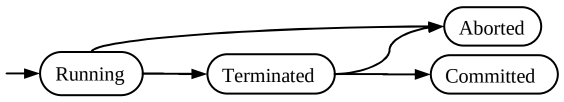

Figure 1. **State transitions of a transaction.**

DEFINITION 1: A transaction is terminated iff it has reached the end of its atomic block (but is not committed yet). A transaction is committed iff its updates to shared memory are committed. A transaction is aborted iff its execution is stopped and its tentative updates to shared memory are discarded.

DEFINITION 2: A transaction is finalized iff it is aborted or committed.

The first property that is expected from a transactor algorithm is finalization that is defined as follows:

PROPERTY 1: Finalization: Every transaction is eventually finalized.

DEFINITION 3: A message is stable iff its sender transaction is committed.

If a transaction T_1 receives a message that is sent by another transaction T_2, as computation of T_1 is reliant on the message, it cannot commit unless the message becomes stable. We say then that the receiving transaction is dependent on the sending transaction. The notion is formalized as follows:

DEFINITION 4: Transaction dependency relation: A transaction T_1 is dependent on transaction T_2, i.e. $T_1 \rightarrow T_2$, iff T_1 can be committed only if T_2 is committed.

DEFINITION 5: A message is pending iff its sender transaction is running or terminated.

DEFINITION 6: If transaction T_1 receives a pending message that is sent by transaction T_2, T_1 becomes dependent on T_2, i.e. $T_1 \rightarrow T_2$.

The second property that is expected from a transactor algorithm is that when a transaction is committed, no dependency is violated. The property is formalized as follows:

PROPERTY 2: Commit Accuracy: For any committed transaction T_1, all transactions T_2 that T_1 is dependent on are also committed.

To satisfy only PROPERTY 1 and PROPERTY 2, a trivial algorithm can abort any transaction. The third property is non-triviality of the algorithm.

DEFINITION 7: A transaction is non-committable iff its commitment even in the future violates commit accuracy.

For example, a transaction that has dependency to an aborted transaction is non-committable but a transaction that has dependency to only running or committed transactions is not non-committable.

PROPERTY 3: Non-triviality: Only transactions that are non-committable are aborted.

Therefore soundness is defined as follows:

DEFINITION 8: A transactor algorithm is sound if it has the following properties: PROPERTY 1: Finalization, PROPERTY 2: Commit Accuracy, PROPERTY 3: Non-triviality.

3.2 Operational Semantics Background

LEMMA 1: Dependency is transitive, i.e. if $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_3$ then $T_1 \rightarrow T_3$. (We use $\rightarrow^*$ to denote transitive dependency.)

PROOF: It is trivial from DEFINITION 4. ■

LEMMA 2: If $T_1 \rightarrow T_2$ and T_2 is aborted, non-triviality is not violated if T_1 is aborted.

PROOF: Aborted is a final state for a transaction. As T_2 is aborted, it can never commit. By $T_1 \rightarrow T_2$ and DEFINITION 4, T_1 can only commit when T_2 is committed. Hence, T_1 is non-committable; thus, aborting it does not violate non-triviality, PROPERTY 3. ■

DEFINITION 9: A transaction T_1 is called a failed transaction if there is a transaction T_2 such that $T_1 \rightarrow^* T_2$ and T_2 is aborted.

LEMMA 3: Abort Propagation: Aborting a failed transaction does not violate non-triviality.

PROOF: Direct from DEFINITION 9, LEMMA 1 and LEMMA 2. ■

DEFINITION 10: Transaction dependency relation $\rightarrow$ for the set of transactions T corresponds to the transaction dependency graph $G_{TD}[V, E]$ defined as follows:

$$V = T \text{ and } ((T_1, T_2) \in \rightarrow) \Leftrightarrow ((T_1, T_2) \in E). \qquad (1)$$

A path in the dependency graph corresponds to a transitive dependency relation. So we use them interchangeably.

It is known that a subgraph of a directed graph is called Strongly Connected Subgraph (SCS) if there is a path from each vertex of the subgraph to every other vertex of it. The Strongly Connected Components (SCC) of a directed graph are its maximal SCSs. A node that is on no cycle is an SCC itself.

LEMMA 4: Any two transactions in an SCS of the transaction dependency graph are interdependent.

PROOF: In an SCS of the dependency graph, for every two transactions T_1 and T_2, there is a path from T_1 to T_2 and a path from T_2 to T_1. By DEFINITION 10, $T_1 \rightarrow^* T_2$ and $T_2 \rightarrow^* T_1$. According to LEMMA 2, $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$. ■

LEMMA 5: To preserve commit accuracy, all of the transactions in an SCS of the transaction dependency graph should only commit together.

Proof: By Lemma 4, for any two transactions T_1 and T_2 in an SCC, $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$. To preserve commit accuracy, by Definition 4, T_1 can only be committed if T_2 is committed and vice versa. This means that T_1 and T_2 can only be committed together. ■

The lemma presents the fact that a sound algorithm should perform collective commit when there are cyclic dependencies. This means that transactions of an SCS should not be linearized to distinct points but all of them should be linearized to a single point.

DEFINITION 11: A dependency to a transaction is resolved if the transaction is committed.

Atomic:	$Atomic1$:	$\tau\ fresh$ $\langle \mathcal{A}[a \mapsto \langle \_, \mathcal{R}[atomic\ t] \rangle] \cdot \mathcal{T} \cdot \_ \rangle \overset{R}{\rightarrow}$ $\langle \mathcal{A}[a \mapsto \langle \_, \mathcal{R}_\tau[t] \rangle] \cdot \mathcal{T}[\tau \mapsto \langle \mathbb{r}, \mathcal{R}[atomic\ t], \{\} \rangle] \cdot \_ \rangle$
	$Atomic2$:	$\langle \mathcal{A}[a \mapsto \langle \_, \mathcal{R}_\tau[v] \rangle] \cdot \mathcal{T}[\tau \mapsto \langle \mathbb{r}, \_, \_ \rangle] \cdot \_ \rangle \overset{R}{\rightarrow}$ $\langle \mathcal{A}[a \mapsto \langle \_, \mathcal{R}_\tau[v] \rangle] \cdot \mathcal{T}[\tau \mapsto \langle \mathbb{t}, \_, \_ \rangle] \cdot \_ \rangle$
Send:	$Send1$:	$\langle \mathcal{A}[a_1 \mapsto \langle \_, \mathcal{R}_\tau[a_2\ send\ v] \rangle][a_2 \mapsto \langle M_2, \_ \rangle] \cdot \_ \cdot \_ \rangle \overset{R}{\rightarrow}$ $\langle \mathcal{A}[a_1 \mapsto \langle \_, \mathcal{R}_\tau[unit] \rangle][a_2 \mapsto \langle M_2 \uplus \{\langle v, \tau \rangle\}, \_ \rangle] \cdot \_ \cdot \_ \rangle$
	$Send2$:	$\langle \mathcal{A}[a_1 \mapsto \langle \_, \mathcal{R}[a_2\ send\ v] \rangle][a_2 \mapsto \langle M_2, \_ \rangle] \cdot \_ \cdot \_ \rangle \overset{R}{\rightarrow}$ $\langle \mathcal{A}[a_1 \mapsto \langle \_, \mathcal{R}[unit] \rangle][a_2 \mapsto \langle M_2 \uplus \{\langle v, \tau_{Committed} \rangle\}, \_ \rangle] \cdot \_ \cdot \_ \rangle$
Receive:	$Receive1$:	$\langle \mathcal{A}[a_1 \mapsto \langle M_1 \uplus \{\langle v, \tau_2 \rangle\}, \mathcal{R}[receive] \rangle] \cdot [\tau_2 \mapsto \langle \mathbb{c}, \_, \_ \rangle] \cdot \_ \rangle \overset{R}{\rightarrow}$ $\langle \mathcal{A}[a_1 \mapsto \langle M_1, \mathcal{R}[v] \rangle] \cdot [\tau_2 \mapsto \langle \mathbb{c}, \_, \_ \rangle] \cdot \_ \rangle$
	$Receive2$:	$\langle \mathcal{A}\left[a_1 \mapsto \langle M_1 \uplus \{\langle v, \tau_2 \rangle\}, \mathcal{R}_{\tau_1}[receive] \rangle\right] \cdot \mathcal{T}[\tau_1 \mapsto \langle \_, \_, B_1 \rangle][\tau_2 \mapsto \langle \mathbb{c}, \_, \_ \rangle] \cdot \_ \rangle \overset{R}{\rightarrow}$ $\langle \mathcal{A}\left[a_1 \mapsto \langle M_1, \mathcal{R}_{\tau_1}[v] \rangle\right] \cdot \mathcal{T}[\tau_1 \mapsto \langle \_, \_, B_1 \uplus \{\langle v, \tau_2 \rangle\} \rangle][\tau_2 \mapsto \langle \mathbb{c}, \_, \_ \rangle] \cdot \_ \rangle$
	$Receive3$:	$\langle \mathcal{A}\left[a_1 \mapsto \langle M_1 \uplus \{\langle v, \tau_2 \rangle\}, \mathcal{R}_{\tau_1}[receive] \rangle\right] \cdot \mathcal{T}[\tau_2 \mapsto \langle \mathbb{a}, \_, \_ \rangle] \cdot \_ \rangle \overset{R}{\rightarrow}$ $\langle \mathcal{A}\left[a_1 \mapsto \langle M_1, \mathcal{R}_{\tau_1}[receive] \rangle\right] \cdot \mathcal{T}[\tau_2 \mapsto \langle \mathbb{a}, \_, \_ \rangle] \cdot \_ \rangle$
	$Receive4$:	$\langle \mathcal{A}\left[a_1 \mapsto \langle M_1 \uplus \{\langle v, \tau_2 \rangle\}, \mathcal{R}_{\tau_1}[receive] \rangle\right] \cdot \mathcal{T}[\tau_1 \mapsto \langle \_, \_, B_1 \rangle][\tau_2 \mapsto \langle \mathbb{r}\ or\ \mathbb{t}, \_, \_ \rangle] \cdot \mathcal{D} \rangle \overset{R}{\rightarrow}$ $\langle \mathcal{A}\left[a_1 \mapsto \langle M_1, \mathcal{R}_{\tau_1}[v] \rangle\right] \cdot \mathcal{T}[\tau_1 \mapsto \langle \_, \_, B_1 \uplus \{\langle v, \tau_2 \rangle\} \rangle][\tau_2 \mapsto \langle \mathbb{r}\ or\ \mathbb{t}, \_, \_ \rangle] \cdot \mathcal{D} \cup \{\tau_1 \rightarrow \tau_2\} \rangle$

Figure 2. Operational Semantics of Transactor Algorithm 1/2

LEMMA 6: To preserve commit accuracy and non-triviality, if a transaction has dependency to no aborted but a running transaction, it cannot be committed or aborted.

PROOF: If it is committed, as it has an unresolved dependency, commit accuracy is violated. If all its dependencies are resolved later, it can be committed; so, it is not non-committable. Hence aborting it violates non-triviality. ■

DEFINITION 12: A set of transactions $\mathcal{C}$ is a cluster iff all its transactions are terminated and any unresolved dependency of them is to each other. Formally, a set of transactions $\mathcal{C}$ is a cluster iff

$$\begin{aligned} &\forall \tau \in \mathcal{C}: \\ &(\tau\ is\ terminated) and \\ &\begin{bmatrix} \forall \tau' \in G_{TD}: \\ (\tau \rightarrow \tau') \Rightarrow ((\tau' is\ committed)\ or\ (\tau' \in \mathcal{C})) \end{bmatrix} \end{aligned} \tag{2}$$

LEMMA 7: Collective Commit: Committing all transactions of a cluster together does not violate commit accuracy.

PROOF: By DEFINITION 12, in a cluster, any dependency of each transaction is either already resolved or will be resolved by committing other transactions in the cluster. Committing all transactions of the cluster together leaves no unresolved dependency for them. Therefore committing them together does not violate commit accuracy. ■

LEMMA 8: A Transactor algorithm is a sound algorithm if it has the finalization property and only aborts failed transactions and only commits transactions of clusters together.

PROOF: By DEFINITION 8, LEMMA 3, LEMMA 7.

3.3 Operational Semantics

The operational semantics is defined for the following language of terms and values:

$t \rightarrow unit\ \mid\ l\ \mid\ x\ \mid\ \lambda x.t\ \mid\ t\ t$ *Terms*
$\quad \mid\ ref\ t\ \mid\ t \coloneqq t\ \mid\ !t$
$\quad \mid\ "atomic"\ t\ \mid\ t\ "send"\ t\ \mid\ "receive"\ \mid\ "abort"$
$v \rightarrow unit\ \mid\ l\ \mid\ \lambda x.e\ \mid\ A$ *Values*
$A = \{a_1, a_2, a_3, \ldots\}$ *Actor names*

Figure 2 and Figure 3 show the operational semantics. It essentially represents abort propagation for failed transactions and collective commit for transactions of clusters together. Therefore by LEMMA 8, an algorithm is sound if it has the finalization property and it satisfies the operational semantic of Figure 3 and Figure 2. Before explaining the transition rules, we establish the notational conventions.

3.3.1 Notational Conventions

Tuple: "$\langle\ \ \rangle$" denotes a tuple and "$\cdot$" is used to separate elements of a tuple. For instance $\langle a \cdot b \cdot c \rangle$ denotes a tuple of elements a, b and c. Sets and Multisets: Union and multiset union is denoted by $\cup$ and $\uplus$ respectively. For a multiset S, $S\ /\ x$ is a multiset that is the same as S except that an instance of x is removed. Map: $\mapsto$ is used to denote a mapping. For instance, $a \mapsto b$ represents a mapping from a to b. A map is a set of mappings. For a map M, $M(x)$ denotes the element to which M maps x. For a map M, the set of elements that it maps from is called its domain and is denoted by $dom(M)$. $M\ /\ x$ denotes a map M' that is the same as M except that $x \notin dom(M')$.

Pattern matching: Pattern matching is used to determine the applicability of a particular rule, and to match components of terms to variables. Applying term constructors to variables makes simple patterns. For instance $\langle x, y \rangle$ matches tuples of two elements where variables x and y match the first and second elements respectively. The underscore character "_" matches any term. The pattern $M[a \mapsto y]$ matches any map M' where $a \in M'$, y matches $M'(a)$ and M is bound to $M'\ /\ a$. The pattern $S \uplus \{x\}$ matches any multiset S' where x is bound to an element of S' and S is bound to $S\ /\ x$. $a\ or\ b$ matches either a or b.

Transaction States: $\mathbb{r}$, $\mathbb{a}$, $\mathbb{t}$ and $\mathbb{c}$ denote running, aborted, terminated and committed states of a transaction. Reduction Context: In each transition rule, a particular *redex* term is reduced. The redex is considered in a *reduction context*. $\mathcal{R}_\tau[\ ]$ denotes reduction context for terms that are evaluated inside transaction τ and $\mathcal{R}[\ ]$ denotes reduction context for terms that are evaluated outside transactions. We call $\mathcal{R}_\tau[\ ]$ and $\mathcal{R}[\ ]$ transactional and non-transactional reduction contexts respectively.

$\mathcal{R} \rightarrow [\ ]\ \mid\ \mathcal{R}\ t\ \mid\ v\ \mathcal{R}\ \mid\ ref\ \mathcal{R}\ \mid\ !\mathcal{R}\ \mid\ \mathcal{R} \coloneqq t\ \mid v \coloneqq \mathcal{R}$
$\quad \mid\ \mathcal{R}\ send\ t\ \mid\ v\ send\ \mathcal{R}\ \mid\ "atomic"\ \mathcal{R}_\tau$
$\mathcal{R}_\tau \rightarrow [\ ]\ \mid\ \mathcal{R}_\tau\ t\ \mid\ v\ \mathcal{R}_\tau\ \mid\ ref\ \mathcal{R}_\tau\ \mid\ !\mathcal{R}_\tau\ \mid \mathcal{R}_\tau \coloneqq t\ \mid\ v \coloneqq \mathcal{R}_\tau$
$\quad \mid\ \mathcal{R}_\tau\ send\ t\ \mid\ v\ send\ \mathcal{R}_\tau$

Configuration: A configuration describes state of a program at a single point at runtime. A configuration is a triple of the form $\langle \mathcal{A} \cdot \mathcal{T} \cdot \mathcal{D} \rangle$. $\mathcal{A}$ is a mapping from transactor ids $\{a_i\}$ to pairs of the form $\langle M_i \cdot \mathcal{R}[redex]_i \rangle$ where M_i denotes the mailbox of transactor a_i and $\mathcal{R}[redex]_i$ denotes the current reduction context

Abort:	$Abort1$:	$\langle \mathcal{A}[a_1 \mapsto \langle \_, \mathcal{R}_{\tau_1}[abort\ \tau_2]\rangle] \cdot \mathcal{T}[\tau_2 \mapsto \langle \mathbb{a}, \_, \_\rangle] \cdot \_\rangle \overset{R}{\rightarrow}$ $\langle \mathcal{A}[a_1 \mapsto \langle \_, \mathcal{R}_{\tau_1}[unit]\rangle] \cdot \mathcal{T}[\tau_2 \mapsto \langle \mathbb{a}, \_, \_\rangle] \cdot \_\rangle$
	$Abort2$:	$\langle \mathcal{A}[a_1 \mapsto \langle \_, \mathcal{R}_{\tau_1}[abort\ \tau_2]\rangle][a_2 \mapsto \langle \_, \mathcal{R}_{\tau_2}[\_]\rangle] \cdot \mathcal{T}[\tau_2 \mapsto \langle \mathbb{r}\ or\ \mathbb{t}, \_, \_\rangle] \cdot \_\rangle \overset{R}{\rightarrow}$ $\langle \mathcal{A}[a_1 \mapsto \langle \_, \mathcal{R}_{\tau_1}[unit]\rangle][a_2 \mapsto \langle \_, \mathcal{R}_{\tau_2}[abort]\rangle] \cdot \mathcal{T}[\tau_2 \mapsto \langle \mathbb{r}\ or\ \mathbb{t}, \_, \_\rangle] \cdot \_\rangle$
	$Abort3$:	$\dfrac{\exists \tau' \colon\ ((\tau \rightarrow \tau') \in \mathcal{D})\ and\ (\mathcal{T}(\tau') = \langle \mathbb{a}, \_, \_\rangle))}{\langle \mathcal{A}[a \mapsto \langle \_, \mathcal{R}_{\tau}[\_]\rangle] \cdot \mathcal{T}[\tau \mapsto \langle \mathbb{r}\ or\ \mathbb{t}, \_, \_\rangle] \cdot \mathcal{D}\rangle \overset{R}{\rightarrow}}$ $\langle \mathcal{A}[a \mapsto \langle \_, \mathcal{R}_{\tau}[abort]\rangle] \cdot \mathcal{T}[\tau \mapsto \langle \mathbb{r}\ or\ \mathbb{t}, \_, \_\rangle] \cdot \mathcal{D}\rangle$
	$Abort4$:	$\langle \mathcal{A}[a \mapsto \langle M, \mathcal{R}_{\tau}[abort]\rangle] \cdot \mathcal{T}[\tau \mapsto \langle \_, \mathcal{R}[atomic\ t], B\rangle] \cdot \_\rangle \overset{R}{\rightarrow}$ $\langle \mathcal{A}[a \mapsto \langle M \uplus B, \mathcal{R}[atomic\ t]\rangle] \cdot \mathcal{T}[\tau \mapsto \langle \mathbb{a}, \mathcal{R}[atomic\ t], \{\}\rangle] \cdot \_\rangle$
Commit:	$Commit$:	$\dfrac{\forall i = 1..n\colon\ [(\exists \tau\colon (\tau_i \rightarrow \tau) \in \mathcal{D})) \Rightarrow ((\mathcal{T}(\tau) = \langle \mathbb{c}, \_, \_\rangle)\ or\ (\exists j = 1..n\colon \tau = \tau_j))]}{\langle \mathcal{A}[a_i \mapsto \langle \_, \mathcal{R}_{\tau_i}[v_i]\rangle]_{i=1..n} \cdot \mathcal{T}[\tau_i \mapsto \langle \mathbb{t}, \_, \_\rangle]_{i=1..n} \cdot \mathcal{D}\rangle \overset{R}{\rightarrow}}$ $\langle \mathcal{A}[a_i \mapsto \langle \_, \mathcal{R}[v_i]\rangle]_{i=1..n} \cdot \mathcal{T}[\tau_i \mapsto \langle \mathbb{c}, \_, \_\rangle]_{i=1..n} \cdot \mathcal{D}\rangle$

Figure 3. Operational Semantics of Transactor Algorithm 2/2

and redex of a_i. A mailbox M is a multiset of $\langle m, \tau\rangle$ pairs where m is a message and τ is the sender transaction of m. $\mathcal{T}$ is a mapping from transaction names $\{\tau_i\}$ to triples of the form $\langle state_i, \mathcal{R}[redex]_i, B_i\rangle$. $state_i$, a value from the set $\{\mathbb{r}, \mathbb{a}, \mathbb{t}, \mathbb{c}\}$ denotes the current state of τ_i. $\mathcal{R}[redex]_i$ denotes the reduction context and redex just before the atomic block of τ_i is started. As will be explained, at the beginning of each atomic block, reduction context and redex are saved in the second element of the triple for the new transaction instance and it is restored when the transaction is aborted and the atomic block needs to be retried. B_i is a multiset of $\langle m, \tau\rangle$ pairs. B_i is the multiset of messages received by τ_i. As will be explained, if τ_i is aborted, B_i is added to the mailbox multiset. $\mathcal{T}$ contains the dummy entry $\tau_{Committed} \mapsto \langle \mathbb{c}, \_, \_\rangle$ from the beginning. As will be explained, $\tau_{Committed}$ is set as the sender transaction of messages that are sent outside transactions. $\mathcal{D}$ is the set of dependencies between transactions of $dom(\mathcal{T})$.

3.3.2 Reduction Rules

The relation $\overset{R}{\rightarrow}$ is a single-step transition on configurations.

Atomic: $Atomic1$: Reduction of $atomic\ t$ adds a new mapping to $\mathcal{T}$. The new mapping $\tau \mapsto \langle \mathbb{r}, \mathcal{R}[atomic\ t]\rangle$ is from a fresh transaction id τ. The state of the new transaction is set to running $\mathbb{r}$ and the current reduction context and redex are also saved. The reduction context and redex are restored later if the transaction is aborted and the atomic block is to be retried. $atomic\ t$ is reduced to t in the context of transaction τ. $Atomic2$: When a transaction reaches the end of the atomic block, i.e. it evaluates to a value, the state of the transaction is changed from running $\mathbb{r}$ to terminated $\mathbb{t}$.

Send: Sending messages is asynchronous, i.e. nonblocking. A sent message is enqueued to the recipient transactor's mailbox and can be received later. $Send1$: When transactor a_1 $send$s message m to transactor a_2 inside the reduction context of transaction τ, the pair $\langle m, \tau\rangle$ is added to the mailbox of a_2. The $send$ statement itself is then reduced to $unit$. The sender transaction that is saved here is checked not to be aborted when m is being received. $Send2$: When transactor a_1 $send$s message m to transactor a_2 inside a non-transaction reduction context, the pair $\langle m, \tau_{Committed}\rangle$ is added to the mailbox of a_2. The dummy transaction $\tau_{Committed}$ is a member of $\mathcal{T}$ and the status of it is always committed $\mathbb{c}$ from the beginning. As the sender transaction of messages that are sent outside transactions is set to $\tau_{Committed}$, these messages are immediately stable.

Receive: $Receive1$: If transactor a_1 $receive$s inside a non-transactional reduction context, it receives only stable messages. If transactor a_1 $receive$s inside the reduction context of transaction τ_1 and $\langle m, \tau_2\rangle$ is an arbitrary member of its mailbox, three different reductions can happen based on the state of the sender transaction τ_2. $Receive2$: If τ_2 is committed, i.e. if the message is stable, the pair is eliminated from the mailbox and $receive$ is reduced to m. $Receive3$: If τ_2 is aborted, the pair is dropped from the mailbox and $receive$ reduces to itself, i.e. $receive$ing should be retried. $Receive4$: If τ_2 is running or terminated, the pair is eliminated from the mailbox and $receive$ is reduced to m, the same as when τ_2 is committed, but also a dependency from τ_1 to τ_2, i.e. $\tau_1 \rightarrow \tau_2$ is added to $\mathcal{D}$. When a message is received inside a transactional context, the pair of the message and its sender transaction is added to the multiset B of the transaction. The elements of the multiset are added to the mailbox if the transaction aborts.

Abort: A transaction τ can be aborted in three ways. It can be aborted by another transaction due to a shared memory conflict resolution ($Abort1$ and $Abort2$). It can be aborted following abortion of a transaction that τ is dependent on ($Abort3$). Also, it can be aborted by a user programmed abort statement inside the atomic block ($Abort4$). $Abort1$ and $Abort2$: It is notable that $abort\ \tau$ is not a term of the language but is executed by the transactors runtime system when a shared memory conflict is to be resolved. $Abort1$: Aborting a transaction that is already aborted has no effect. $Abort2$: If transactor a_2 is running transaction τ_2 and τ_2 is running or terminated then evaluating $abort\ \tau_2$, changes the redex of a_2 to $abort$ statement regardless of its current redex. This reduces $Abort2$ to $Abort4$. $Abort3$: This rule encodes abort propagation. If a transactor a is running transaction τ, τ is running or terminated, and the state of a transaction τ' that τ is dependent on is aborted, τ is also aborted. This case is also reduced to $Abort4$ by changing the current redex of a to $abort$ statement. $Abort4$: Reduction of $abort$, restores $\mathcal{R}[atomic\ t]$ that is saved at the beginning of the atomic block. Restoring $\mathcal{R}[atomic\ t]$ effectively restarts the atomic block. Besides, messages that are received throughout execution and are in the multiset B of the aborted transaction are added to the mailbox.

$Commit$: The commit rule encodes collective commit of a cluster. If there is a set of terminated transactions that their dependencies are either to committed transactions or to each other, they are committed together and transactor reduction contexts return back to non-transactional contexts.

4 Transactor Algorithm

4.1 Sending and Receiving Messages

When an atomic block starts, a new transaction descriptor is created and stored in a thread local variable. Descriptor of a transaction is a data structure that stores all the information regarding that transaction. To get the descriptor of the current transaction, this thread local variable is checked. If the value of the variable is null, evaluation is out of atomic blocks and if it is not null, it is the descriptor of the current transaction. The value of the variable is set to null after a transaction commits.

Transactors provide the facility for the programmer to send and receive messages inside transactions. When a transaction aborts, all its tentative effects should be discarded. Particularly, if it has sent a message, the message should be discarded. A message can have three different states: stable, annihilated and pending. Stable is the state of messages sent by committed transactions or sent outside transactions. Annihilated is the state of messages sent by aborted transactions. Pending is the state of messages that are sent by transactions that are running or terminated, i.e. not committed or aborted yet.

When a message is being sent, instead of only the message itself, a cell containing the message is enqueued to the mailbox. The information that a receiving transaction needs later are stored in the cell. Besides the message, the cell contains a reference to the descriptor of the sender transaction, the state of the message and a reference to a notifiable object. We explain about this data as we proceed. Figure 4 and Figure 5 depict data structures and their relations while sending and receiving a message.

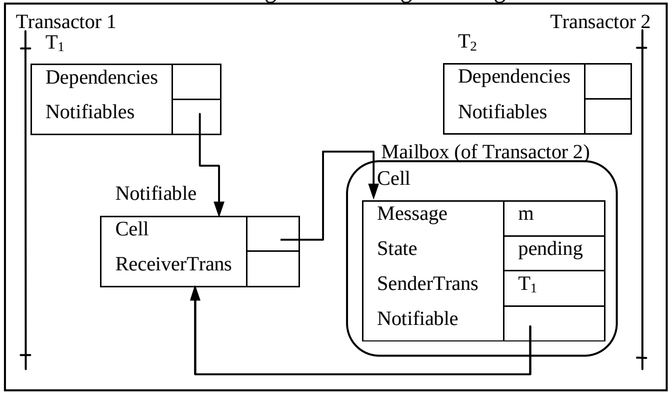

Figure 4. **Sending Message**

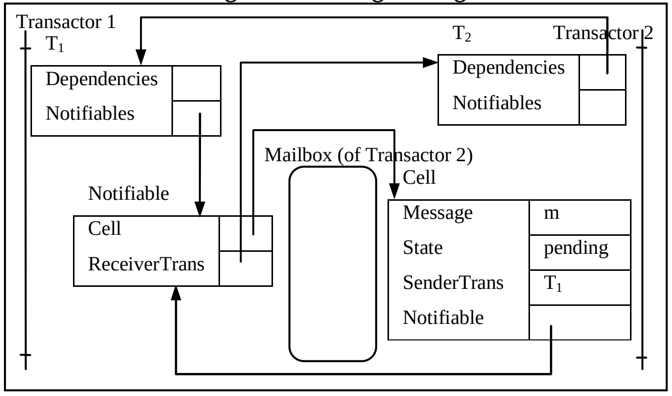

Figure 5. Receiving Message

The state of a message sent respectively outside and inside a transaction is stable and pending at the beginning. If a transaction commits, the state of the messages that it has sent should be changed to stable and if it aborts, the state should be changed to annihilated. This is done by notifiable objects. When a transaction sends a message, a new cell containing the message is enqueued to the recipient's mailbox and furthermore a notifiable object having a reference to the new cell is created and registered to the descriptor of the sender transaction. On abort or commit of a transaction, all of the registered notifiables are notified of abortion or commitment. Notifiables, when notified, update the state of the cells that they reference. When a notifiable is notified of abortion, it sets the state of the cell to annihilated. When it is notified of commitment (also called dependency resolution), it sets the state of the cell to stable.

If a message is to be received outside a transaction, a stable message is required. If it is to be received inside a transaction, a non-annihilated message is required. When a message is being received, cells are dequeued from the mailbox and any annihilated message is dropped until the required message is found. The thread suspends if no required message exists in the mailbox until one is enqueued.

The dependencies of each transaction are kept inside its transaction descriptor. To track dependencies, when a transaction T_R receives a pending message, a reference to the descriptor of the sending transaction T_S should be added to the dependency set of descriptor of T_R. Hence, when a pending message is being received, a reference to the descriptor of the sending transaction is needed. To have this reference, when a transaction is sending, it saves a reference to the descriptor of itself in the new cell.

Finalization process of transactions is described in the following subsection but for the purpose of completing the explanation of this subsection, assume T_R to be a transaction that is terminated and $Dep = \{T_{S_{i=1..n}}\}$ to be the set of transactions that it is dependent on. T_{S_i}'s are transactions that T_R has received pending messages from. By LEMMA 6, if there is no aborted but running transactions in Dep, T_R cannot be committed or aborted. Therefore T_R goes to the waiting state to get notified of abortion or commitment of T_{S_i}'s. Hence, a T_{S_i}, i.e. a sender transaction, when aborted or committed, should notify the transaction that has received its sent message. Notifying waiting dependent transactions is done by the same notifiables that update the state of cells. After a transaction T_R receives a pending message, the notifiable object, when notified, should notify T_R. Therefore, when the message is received, the receiving transaction should be subscribed to the notifiable object as a notification sink. This means that a reference to the notifiable object is needed when the message is being received. To have the notifiable while receiving, it is saved in the cell when the message is being sent. When a pending message is being received, the notifiable object that is previously registered in the descriptor of the sender transaction is obtained from the cell that contains the message and a reference to the receiving transaction is subscribed to it. Pseudo code of the send and receive methods is presented in the technical report [11].

When a transaction aborts, its effects should be rolled back. While aborting, a transaction that has received messages from the transactor mailbox should put the messages back. Therefore, to track received messages, when a message is being received inside a transaction, the cell that the message is obtained from is added to a backup set in the transaction descriptor. The set is iterated while the transaction is being aborted and any cell that is not annihilated is put back to the mailbox. As the cell of a received message may be later put back to the mailbox, a cell should be notified to become stable or annihilated by its corresponding notifiable object not only when it is in the mailbox but also when the message is received and the cell is dequeued from the mailbox. Thus, the notifiable object notifies the cell even after the receiving transactor is subscribed.

4.2 Finalization

4.2.1 Abort Propagation

Consider a transaction T_S and the set of transactions $\{T_{R_{i=1..n}}\}$ that are dependent on T_S. The state of a transaction T_S can be set to aborted in three different ways that were explained in the previous section. In any of the ways that T_S is aborted, T_S propagates abortion to dependent transactions $\{T_{R_{i=1..n}}\}$ by notifying notifiables $\{N_{R_{i=1..n}}\}$ corresponding to $\{T_{R_{i=1..n}}\}$. Every notifiable N_{R_i}, in turn, sets the status of the transaction descriptor of T_{R_i} to aborted state. By DEFINITION 9, T_{R_i}s are failed transactions and by LEMMA 3, aborting them does not violate non-triviality.

The same situation recurs on abortion of the T_{R_i}s, i.e. each of them notifies their own notifiables. Therefore, abortion is propagated by an implicit traversal of the transaction descriptors that are (transitively) dependent on T_S. It is notable that by notifiable objects, the traversal is done in reverse direction of dependencies. Setting the status of an aborted transaction descriptor to aborted returns without any action. Hence, the traversal avoids infinite loops by terminating at previously aborted transaction descriptors.

As previously explained, any non-annihilated cells of the backup set are put back to the mailbox. Finally, after a transaction is aborted, its atomic block is restarted as a new transaction.

4.2.2 Termination

Every transactor that reaches the end of the atomic block sets the status of its descriptor to terminated. Then it starts the cluster search to check if it is possible to commit at this time. If the cluster search succeeds finding a cluster, it commits all of its transactions together. Cluster search is explained in the next subsection. If the cluster search returns failure, the transaction goes to the waiting state. There are three different events that wake up a transaction from the waiting state: Abort, Dependency Resolution and Commit events.

- An Abort event is raised when the transaction descriptor is set to aborted. On this event, the transaction aborts as explained before.
- A Dependency Resolution event is raised when the transaction descriptor is notified of a dependency resolution. As will be explained in the next subsection, a transaction that commits notifies all of the transactions that are dependent on it about the dependency resolution. On this event, as a dependency of the current transaction is known to be resolved, it may be able to commit; therefore, the cluster search algorithm is retried.
- A Commit event is raised when the transaction is committed by the cluster search of another transaction. On a Commit event, the atomic block successfully returns.

4.2.3 Collective Commit

The dependencies of transaction descriptors can in general form a cyclic graph. If the transactions in a cycle obliviously wait until all of their dependencies are resolved, they may wait forever. Therefore, without a cycle detection mechanism, deadlocks occur. Cluster search tries to find cycles containing the current transaction that make a cluster and to commit them collectively.

Cluster search employs the Tarjan algorithm [13] that given a graph and a starting node, finds the set of SCCs of the graph reachable from the starting node. For each SCC, Tarjan algorithm obtains the set of vertices of the SCC. It performs a depth first traversal of the graph to traverse all the reachable nodes. For each present node, it gets the adjacent nodes and continues traversal by moving to one of them. Getting the set of adjacent nodes of a node is where cluster search hooks to Tarjan algorithm:

- If an adjacent transaction is aborted, the search is left and the current transaction aborts itself. In this case, the current transaction has a path to and hence is transitively dependent on an aborted transaction. Therefore, by DEFINITION 9, it is a failed transaction and by LEMMA 3, aborting it does not violate non-triviality.
- If an adjacent transaction is running, the search is left and the current transaction goes to the waiting state. In this case, since the current transaction is transitively dependent on a running transaction, by LEMMA 1 and LEMMA 6, it cannot be aborted or committed. Therefore it goes to waiting state to get notified by other transactions.
- As the dependency to committed transactions is previously resolved, it is as if they didn't exist. Therefore, adjacent committed transactions are ignored.
- Any adjacent transaction that is terminated is returned as an adjacent transaction.

The pseudo code of getting adjacent nodes is presented in the technical report [11].

Let G_{TD} denote the transaction dependency graph. The search is left when an aborted or running transaction is reached and also it ignores committed transactions. This means that Tarjan algorithm effectively searches on a subgraph of G_{TD} that is induced by terminated transactions. Let G_T denote this subgraph. As G_T is a subgraph of G_{TD}, any SCC of G_T is an SCS of G_{TD}. Therefore, if Tarjan algorithm finds an SCC of G_T, the cluster search has found an SCS of G_{TD}.

LEMMA 9: If the cluster search finds one SCS of the dependency graph, it is a cluster.

PROOF: The cluster search is left when an aborted or running transaction is reached. Therefore, none of the transactions of the found SCS can be aborted or running and also they cannot have dependency to any aborted or running transaction. The cluster search also ignores committed transactions. Hence, all transactions of the found SCS are terminated. As only one SCS is found, any dependency from transactions of the SCS is either to other transactions of the SCS or to the committed transactions that are ignored in the traversal. Hence, by DEFINITION 12, the SCS is a cluster. ■

If the cluster search finds only one SCS, the algorithm commits all of its transactions together. As proved above, the found SCS is a cluster. Therefore, by LEMMA 7, committing all its transactions together does not violate commit accuracy.

If the cluster search finds more than one SCS, it is possible to commit SCSs in the order that they are found. But for simplicity, the transaction goes to the waiting state. (Please see the technical report [11] for more explanation.)

According to LEMMA 5, all of the transactions of the SCS should be committed together, i.e. committed atomically. Status of the transaction descriptors is changed to committed after the locks of the status of all of them are acquired. To prevent deadlock, the locks are acquired in the order of the unique transaction descriptor numbers.

When a transaction T_S is set to committed, it sends dependency resolution notification to all its registered notifiables $\{N_{i=1..n}\}$. When a notifiable N_i is notified of dependency resolution, it performs two actions. It sets the status of the cell C_i that it references to stable. In addition, if a receiver transaction descriptor T_{R_i} is subscribed to N_i, N_i notifies T_{R_i} about the dependency resolution. This makes a Dependency Resolution event for T_{R_i}.

5 Algorithm Soundness

The reader is invited to see the same section number in the appendix of the technical report [11] for full proofs.

LEMMA 10: The Transactor algorithm has Commit Accuracy.

LEMMA 11: The Transactor algorithm has Non-triviality.

THEOREM 1: The Transactor algorithm has Finalization, i.e. every transaction eventually finalizes.

Theorem 2: The Transactor algorithm is sound.

PROOF: Direct from DEFINITION 8, LEMMA 10, LEMMA 11 and THEOREM 1.

6 Related Works

Argus [10] language provides programming with objects called Guardians which implement a number of procedures that are run in response to remote handler calls from other guardians. Calling the handler of a procedure of a Guardian sends a message to the Guardian. In addition to Guardians, the programmer is provided with Actions that are essentially isolated and failure-atomic transactions. If a handler is called inside an Action, Argus runs the handler call as a subaction. It is guaranteed that none or all of the topaction and its subactions are committed.

Sinfonia [1] provides support for a subset of distributed transactions called minitransactions. A minitransaction is a one-level distributed transaction that can be decomposed to independent subcomputations on participant nodes. The computation on each participant node is a number of condition checks, reads and writes. This constraint on transactions allows Sinfonia to piggyback sending requests to nodes and getting results from them into the roundtrip of the first phase of the two-phase commit protocol. Sinfonia provides various mechanisms for fault tolerance.

A Reactor [4] consists of a collection of relations and rules which constitute a stateful, reactive and atomic unit of distribution. A reaction begins when an update bundle is received. An update bundle is a map from the set of relations of the reactor to sets of tuples to be added to or deleted. Evaluation of the rules of the reactor according to current and tentatively updated state of relations specifies the future state of the local relations and update bundles for other relations. Update bundles initiate subsequent reactions; thus they play a role similar to messages in message-passing models. In the Reactor terminology, the scope of the reaction is extruded to include subsequent reactions, i.e. the reactions are interdependent. A whole reaction is committed when each of the involved reactors reaches a state that satisfies its rules. From the view of external reactors, a reaction is executed atomically.

In fact, inside Argus actions, Sinfonia minitransactions and Reactor reactions, messages can be sent, but cannot be received. A message is always received at the beginning of each subtransaction. In models that a message can only be received at the beginning of a transaction, the distributed transaction takes the form of a tree. To finalize tree-shaped distributed transactions, hierarchical commit can be employed or it can be flattened to a two phase commit as in Argus. But if messages can be received inside transactions, dependencies can form a graph. Finalization of transactions with a dependency graph gets more complicated than with a dependency tree. We proposed cluster search, collective commit and abort propagation for finalization of transactions with dependency graphs.

Field and Varela [5] has proposed tau-calculus which extends lambda calculus with facilities for getting and setting, checkpointing and rolling back the state of transactors and also sending and receiving messages. A transaction of a transactor is started when the first message is received, commits on checkpointing and aborts on rolling back. In this model, a receiver is dependent on the sender. Transitive dependencies of a sender transactor to other transactors are propagated with the sent message. On arrival of a message, the dependencies in the message D_M and dependencies of the receiving transactor D_R are compared. If D_M and D_R do not invalidate each other, the dependencies of the transactor are updated by D_M. If D_M is invalidated by D_R, the message is dropped. If D_R is invalidated by D_M, the transactor rolls back. By the semantic of tau-calculus, checkpointing in a transactor succeeds only when all the transactors that it is dependent on are checkpointed or are ready to checkpoint. To make each participating transactor able to checkpoint or rollback, the programmer should program transactors so that each participant receives messages to know about the state of other participants. Briefly, Tau transactors provide the programmer with features to program but does not automatically support distributed state atomicity, i.e. all-or-none state update of the participating transactors. It also does not support isolation of local concurrent transactions in each transactor.

With Stabilizers [14], the programmer can mark locations of code as stable checkpoints. Threads can send and receive messages synchronously on definite channels. The sender and the receiver of a message become interdependent. A dependency graph is maintained throughout the program execution. The checkpoint, sends and receives locations are nodes of the dependency graph. Edges of the graph are of three different types: 1. Edges between corresponding send and receive nodes of two threads. For each thread: 2. Edges from each send and receive node to the latest passed checkpoint node, 3. Edges from each node to the first node after it. On a transient fault, the programmer calls stabilize in the fault experiencing thread. When stabilize is called, the runtime system reverts back the current thread and each of its dependent threads to their latest possible stable location. This is done by finding the furthest reachable checkpoint node of each thread from the latest node of the thread that calls stabilize. The dependency graph maintained by Stabilizers is interestingly in correspondence with the call stack of nested atomic blocks where stable checkpoints correspond to the beginning of atomic blocks. Essentially, Stabilizers support program location recovery. Assume a transaction T_n that is nested inside transactions $T_{i=1..n-1}$ and is dependent on aborted transaction T_2. Program location recovery is defined as follows: For every such transaction T_n, the thread executing T_n is reverted back to the beginning of the latest possible enclosing nested transaction T_i where $T_i = T_1$ or T_{i-1} is not dependent on T_2.

TE [2] provides the user with a sequencing combinator to combine two events such as synchronous sending or receiving of messages into one compound event. The combination essentially makes a transaction in the sense that synchronizing on the resulting event either performs both or neither of the events. Therefore, TE supports isolation for a sequence of communications but not for shared state manipulations. Throughout the execution, the sender and the receiver events of a message get interdependent. To try different synchronization possibilities, a new search thread is spawned for each message that a receiving event can receive from a channel and a message sent to a channel can be received by several search threads receiving on the same channel. In addition to sequencing, TE supports the choice combinator, chooseEvt. Synchronizing on a choice event succeeds if synchronizing on the event of either of its branches succeeds. Employing chooseEvt, guarded (or conditional) receive can be programmed. To support chooseEvt, two search threads are spawned to tentatively try each branch of chooseEvt. Each search thread maintains a path recording the path of communication partners at points where it sends or receives

Table 1. Related Works

1. Features:	2. Guarantees:
1.1. Local state isolation	2.1. Finalization
1.2. Asynchronous sends	2.2. Non-triviality
1.3. Receive in transactions	2.3. Commit Accuracy
	2.3.1. Distributed state atomicity
	2.3.2. Program location atomicity

+: Supported, –: Not Supported, *: Semi-automatically Supported	1			2			
	1	2	3	1	2	3	
						1	2
Separate Memory Space (Distributed)							
Argus Actions [10]	+	+	–	+	+	+	+
Sinfonia [1]	+	+	–	+	+	+	+
Reactors [4]	+	–	–	+	+	+	–
Tau Transactors [5]	–	+	+	*	*	*	–
Shared Memory Space							
Stabilizers [14]	–	–	+	–	+	–	+
TE [2]	–	–	+	+	+	–	+
TE in ML [3]	–	–	+	+	+	–	+
TIC [12]	–	+	+	+	+	–	–
Current work	+	+	+	+	+	+	+

messages and also the alternatives it takes at chooseEvts. The transitive dependencies of the path of each search thread specify the set of threads that the search thread is dependent on and an expected path for each of them. A set of search threads are committable if all of the threads of the set are completed, the set is closed under the transitive dependency and the path that each of them expects from the others is consistent with the current path of the them. The synchronizations of a set of committable search threads are committed together. There is a nontrivial runtime overhead to spawn search threads to match different senders and receivers and chooseEvt branches, to track paths and to search for committable search threads.

TE in ML [3] extends TE to support mutation of shared memory in transactional events and also nested synchronizations. A transactional event is logically divided into sections called chunks. Chunks are delimited by sends and receives inside the transactional event. Isolation of a synchronization is broken at the end of chunks. At these points, i.e. before sends and receives, the mutations done in the chunk can be seen by other synchronizations. This semantics seems counterintuitive as it is expected that all of the shared memory mutations of a transaction be executed in isolation. Similar to TE, several search threads are spawned to support nondeterministic choices of sender and receiver matchings and chooseEvt branches. To support mutation, chunks mutate heaps called search heaps tentatively. To let a chunk read a value written by another chunk, chunks of different synchronizations are allowed to interleave. To allow interleaving of chunks, first, when a chunk is finished, its heap is entered to the pool of search heaps and second, when a chunk is to be started, a heap from the pool of search heaps is selected. The non-determinism in choice of the heap from the heap pool leads to spawn of a search thread for each of the possible heaps. For each search heap, a path is maintained that records the path of search threads that contributed toward producing it. When a set of search threads are to be committed, not only consistency of their dependencies to each other but also to the dependencies of the path of the heap that is going to be committed is checked. Thus, the runtime cost of TE in ML is even more than the cost of TE. Semantics of nested synchronizations is similar to the semantics of closed nesting. TIC [12] was explained in section 2.1.2.

It is elicited from each of the related works if they support each of the features and guarantees defined as follows. The results are presented in Table 1. Local state isolation: Intermediate state updates of each transaction are hidden from other transactions. Asynchronous (non-blocking) sends: Sending a message is non-blocking. Receive inside transactions: Messages from other transactions can be received inside a transaction. Distributed state atomicity (consistency): The state updates of a transaction T are committed only if the state updates of transactions that T is dependent on are committed. Program location atomicity: We define that a transaction is passed through, if its executing thread has started executing the code after the transaction code. By this definition, program location atomicity is defined as follows: Every transaction T is passed through only if the transactions that T is dependent on are passed through.

7 Conclusion and Future Works

This work proposes Transactors that provide the programmer with facilities of isolation from TM and facilities of communication from Actors. In the Transactors model, asynchronous messages can be sent and received inside transactions while the guarantee of transaction isolation is still preserved. The semantics of the model is defined, an algorithm implementing the semantics is proposed and proven sound.

Our preliminary performance evaluations in [11] suggest that Transactors perform competitive to TM for isolation and to Actors for communication. Our future work is to program more case studies and gain more performance evaluations.

References

[1] Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. 2007. Sinfonia: a new paradigm for building scalable distributed systems. In Proc. of SOSP '07. 159-174.

[2] Donnelly, K. and Fluet, M. 2008. Transactional events. J. Functional Programming. 18, 5-6 (Sep. 2008), 649-706.

[3] Effinger-Dean, L., Kehrt, M., and Grossman, D. 2008. Transactional events for ML. In Proc. of ICFP '08. 103-114.

[4] Field, J., Marinescu, M., and Stefansen, C. 2009. Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications. Theor. Comput. Sci. 410, 2-3, 168-201.

[5] Field, J. and Varela, C. A. 2005. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In Proc. of POPL '05. 195-208.

[6] Haller, P. and Odersky, M. 2009. Scala Actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. 410, 2-3 (Feb. 2009), 202-220.

[7] Harris, T. and Fraser, K. 2003. Language support for lightweight transactions. SIGPLAN Not. 38, 11 (Nov. 2003), 388-402.

[8] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. 2005. Composable memory transactions. In Proc. of PPoPP '05. 48-60.

[9] Herlihy, M., Luchangco, V., and Moir, M. 2006. A flexible framework for implementing software transactional memory. In Proc. of OOPSLA '06. 253-262.

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

[11] Unifying Transactions and Actors, Tech. Report LAMP-REPORT-2009-003, IC, EPFL. http://infoscience.epfl.ch/record/139381

[12] Smaragdakis, Y., Kay, A., Behrends, R., and Young, M. 2007. Transactions with isolation and cooperation. In Proc. of OOPSLA '07. 191-210.

[13] Tarjan, Robert, 1971. Depth-first search and linear graph algorithms. In Proceedings of the 12th Annual Symposium on Switching and Automata Theory (13-15 Oct. 1971).,114-121.

[14] Ziarek, L., Schatz, P., and Jagannathan, S. 2006. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In Proc. of ICFP '06. 136-147.

Technical Report Appendix

The material presented in each section complements the section of the paper with the same number.

2 Incomposability and Deadlock

2.1 Transactions

2.1.1 Roundtrip

Second, for direct-update STM implementation, consider the following execution schedule. T_1 updates `m1` and then before checking the condition, T_2 executes. T_2 checks the condition that is satisfied because of direct updates and then updates `m2`. Thus condition in T_1 is satisfied because of the direct update. Both transactions can reach the end of the atomic block but because of reading tentative updates of each other, each transaction waits for the other one to commit. This leads to deadlock. Any other schedule (i.e. if T_1 updates `m1` and checks its condition before T_2 executes or if T_2 executes first), the same situation as deferred-update implementation of STM happens.

2.1.2 Barrier

- If the outer transaction aborts and goes to waiting state when the condition is failed,
  - If STM implementation is direct-update, since `count` is written by the transaction of each party, the write/write conflict lets only one of them run at a time. Therefore transactions cannot see direct updates of each other. Thus, similar to the previous argument, each transaction reads a value of zero from `count`, updates it to one and checks the condition. Since the condition fails, the transaction aborts and the update rolls back and the transaction goes to the waiting state. As the value of `count` rolls back to zero, the same happens to any later transaction.

By open nesting, the updates of an inner transaction are committed on completion of the transaction itself. If we change the first atomic block of the `await` method to an open transaction, then parties can see the updates of each other to `count` field and therefore, on completion of the first atomic block of the last party, all of the suspended parties can retry and pass the condition check of the second atomic block. It may seem that open nesting provides the required behavior. But consider that after the barrier releases the parties, if the outer transaction of a party aborts, on its retry, `count` is incremented once more and becomes equal to `partiesCount + 1`. This does not satisfy the condition and the retrying party is blocked forever. Even if we change the condition to

```
conditionWait(count.value >= partiesCount)
```

to let retrying transactions pass the condition, the problem is that retrying transactions pass the barrier later than other transactions that have passed the barrier and are not aborted. This contradicts the expected behavior from a barrier to release all of the parties at once.

3 Semantics

3.2 Operational Semantics Background

LEMMA 12: A terminated transaction with no unresolved dependencies is a cluster.

PROOF: For a terminated transaction τ with no unresolved dependency

$$(\forall \tau' \in G_{TD}:\ (\tau \rightarrow \tau') \Rightarrow (\tau' is\ committed)) \qquad (3)$$

Therefore, by DEFINITION 12, it is a singleton cluster. ■

4 Transactor Algorithm

4.1 Sending and Receiving Messages

Pseudo code of send and receive methods are as follows:

Send:

```
def send(msg: T) {
  val senderTransDesc =
    thread local variable for transaction descriptor
  val cell = new Cell(msg, senderTransDesc)
  if (senderTransDesc == null) //outside of atomic
    cell.setStable
  else { //inside atomic
    cell.setPending
    val notifiable = new Notifiable(cell)
    cell.setNotifiable(notifiable)

    senderTransDesc.addNotifiable(notifiable)
  }

  if (isReceiverSuspended) {
    cellForSuspendedReceiver = cell
    desuspendReceiver
  } else
    mailbox.enqueue(cell)
}
```

Receive:

```
def receive(): T = {
  val currentTransDesc =
    thread local variable for transaction descriptor
  if (currentTransDesc == null) //outside of atomic
    a stable cell is required
  else //inside atomic
    a non-annihilated cell is required

  iterate the mailbox to find a required cell
  while (a required cell is not found) {
    suspend
    cell = cellForSuspendedReceiver
    if (the cell is not a required cell)
      mailbox.enqueue(cell)
  }

  val msg = cell.message

  if (currentTransDesc == null) //outside of atomic
    return msg

  val senderTransDesc = cell.senderTransDesc
  val notifiable = cell.notifiable

  if (!cell.isStable) {
    currentTransDesc.addDependency(senderTransDesc)
    notifiable.addTransAsSink(currentTransDesc)
  }

  currentTrans.backupCell(cell)

  msg
}
```

The fact that pending messages in addition to stable messages are also received inside transactions is to support cyclic communication. As an instance, consider the roundtrip case: two transactors running two transactions that one performs a send and then a receive and the other performs a receive and then a send. If a message could not be received until it became stable, the two transactions would wait for each other for ever.

The reader may have noticed that a push mechanism is used to update cell state in the sense that the cell is notified whenever its state should change. This could be implemented by a pull mechanism as well. The cell could check the state of the sender transaction to determine its own state. But as the receiving transaction should be notified by the sender transaction, the sender pushes the update information anyways. Therefore, updating the state of cells is also implemented by a push mechanism benefiting the same notification.

4.2 Finalization

4.2.1 Abort Propagation

As an implementation detail, some messages may be sent in the short period between when the transaction descriptor is set to aborted and when executing the atomic block is stopped. The notifiables corresponding to these messages were not notified of the abortion when the transaction descriptor was being set to aborted. They are notified after execution of the atomic block is stopped.

4.2.3 Collective Commit

The pseudo code of getting adjacent nodes is as follows. As an implementation detail, Tarjan algorithm stores two values for each graph node. These two values are stored in the graph nodes themselves by Tarjan algorithm. As multiple instances of the cluster search from different transactions can be active simultaneously on the dependency graph, the two values cannot be stored in the transaction descriptors. Therefore, each instance of the cluster search maintains a map from transaction descriptors to search nodes containing the values.

```
val nodes = Map[TransactionDescriptor, Node]()

def getNeighbors = {
  val deps = transDesc.getDependencies
  val neighbors = Set[Node]()
  for (depTransDesc <- deps) {
    if (n.transDesc.isActive)
      throw new WaitException(n.transDesc)
    if (n.transDesc.isAborted)
      throw new AbortException(n.transDesc)
    if (!depTransDesc.isCommitted)
      if (nodes.contains(depTransDesc))
        neighbors += nodes(depTransDesc)
      else {
        val node = new Node(depTransDesc)
        nodes += (depTransDesc -> node)
        neighbors += node
      }
  }
  neighbors
}
```

The Tarjan algorithm finds all of the SCCs that are reachable from the starting node. It outputs SCCs in the sequence where any later SCC can only reach earlier SCCs. The last SCC that is found is the SCC containing the starting node. As explained before, any SCC found by Tarjan algorithm is an SCS of the dependency graph.

Consider the case when the cluster search finds more than one SCS. Let SCS_0 denote the last SCS that is found. The current transaction is a member of SCS_0. Cluster search has found an least an SCS before SCS_0. This means that the current transaction can reach an SCS other than SCS_0. Thus, at least one of the transactions of SCS_0 is dependent on a terminated transaction T that is not a member of SCS_0. Therefore all of the transactions of SCS_0 are dependent on T. To preserve commit accuracy, transactions of SCS_0 cannot be committed before T is committed. Therefore, for simplicity, the current transaction goes to the waiting state. But, it is possible to commit SCSs in the order that they are found.

LEMMA 13: Committing SCSs in the order that they are found by the cluster search does not violate commit accuracy.

PROOF: Induction is on the position of the SCS in the found sequence.
Base case: Transactions of the first SCS have dependency to only transactions within the SCS itself or committed transactions. Thus, by DEFINITION 12, the first SCS is a cluster. Therefore, by LEMMA 7, its transactions can be committed together.
Induction case: If all of the SCSs before the current SCS are committed, we show that the current SCS can be committed. Any later SCS can only reach earlier SCSs in the sequence. All of the earlier SCSs are already committed. Therefore, transactions of the current SCS only have dependencies to other transactions of the current SCS or committed transactions. Therefore, by DEFINITION 12, the current SCS is a cluster. Thus, by LEMMA 7, transactions of it can be committed together.

5 Algorithm Soundness

5.1. Commit Accuracy

LEMMA 10: The Transactor algorithm has Commit Accuracy property.

PROOF: As explained in the collective commit subsection, the Transactor algorithm only commits when the cluster search finds one SCS of the dependency graph and it commits all transactions of the SCS together. By LEMMA 9, if the cluster search finds one SCS of the dependency graph, it is a cluster. By LEMMA 7, committing all transactions of a cluster together does not violate commit accuracy. Therefore the Transactor algorithm has commit accuracy. ■

5.2. Non-triviality

LEMMA 11: The Transactor algorithm has Non-triviality property.

PROOF: Throughout the algorithm explanation, whenever a transaction is aborted by the algorithm, it is shown that it is a failed transaction. The Transactor algorithm only aborts failed transactions. Therefore, by LEMMA 3, it is non-trivial. ■

5.3. Finalization

The presented algorithm waits at some points for notification. We show that this suspension cannot incur deadlocks. We prove that each transaction is eventually finalized, i.e. it is eventually aborted or committed. It is assumed that we do not have user programmed deadlocks; thus every transaction is eventually terminated if not aborted sooner.

5.3.1. Algorithm operations

Some operations of the algorithm are highlighted as ALGOPS in this subsection. They are used in the following subsections for the proof of finalization.

ALGOP 1: For any transaction T_R that is dependent on T_S, a notifiable referencing T_R is registered to T_S.

Explanation: If T_R is dependent on T_S, T_R has received a pending message from T_S. By the algorithm, when the pending message is being received, T_R is subscribed to the notifiable object. The notifiable object is previously registered to T_S since the message has been sent. Therefore, a notifiable referencing T_R is registered in T_S. ■

ALGOP 2: If a transaction T_S is aborted, any transaction T_R such that $T_R \rightarrow T_S$ is eventually aborted.

Explanation: By ALGOP 1, for any transaction T_R that is dependent on T_S, a notifiable referencing T_R is registered in T_S. By the algorithm, all notifiables registered to T_S are notified of abortion when T_S is aborted. So T_S notifies the notifiable that references T_R that in turn aborts T_R. Therefore, any T_R that is dependent on T_S is eventually aborted. ■

ALGOP 3: Any failed transaction is eventually aborted.

Explanation: By DEFINITION 9, A transaction T_R is called a failed transaction if there is a transaction T_S such that $T_R \overset{*}{\rightarrow} T_S$ and T_S is aborted. That any failed transaction is eventually aborted is evident from the implicit traversal that was explained but it can also be shown by induction on length of transitive dependency. If the length is one, by ALGOP 2, any T_R such that $T_R \rightarrow T_S$ is eventually aborted. If any T_R that is transitively dependent on T_S with a length of n is aborted, again by ALGOP 2, any transaction that is dependent on T_S with a length of $n+1$ is also eventually aborted.■

AlgOp 4: If a transaction in an SCC is aborted, all of the transactions in that SCC are eventually aborted.

Explanation: By LEMMA 4 and ALGOP 3. ■

ALGOP 5: If the cluster search starts from a transaction in an SCC $\mathcal{C}$, it commits all of the transactions of $\mathcal{C}$ if

- All of the transactions of $\mathcal{C}$ are terminated and
- If there is any dependency from transactions inside $\mathcal{C}$ to transactions outside of it, the dependency is to a committed transaction.

Explanation: In this setting, the cluster search traverses in an SCC of terminated transactions and the only edges out of the SCC are to committed transactions. Thus, no running or aborted transaction can be reached; therefore, the search is not prematurely terminated. As all of the transactions of an SCC are reachable from each other, the search can reach all of the transactions of the SCC. As any dependency from a transaction inside the SCC to outside transactions is to committed transactions and committed transactions are not traversed, the search can only traverse within the SCC. Therefore the cluster search finds only this SCC. Hence, as explained, the algorithm commits all of the transactions of the SCC. ■

ALGOP 6: If a transaction is committed, all of the transactions that are directly dependent on it are notified about the dependency resolution. Formally

$$T_S \; is\; committed \;\Longrightarrow \begin{bmatrix} \forall T_R \in V^{G_{TD}}\; (T_R, T_S) \in E^{G_{TD}}: \\ T_R \; is\; notified\; of\; dependency\; resolution \end{bmatrix}. \qquad \text{EQ. 4}$$

Explanation: From ALGOP 1, for any transaction T_{R_i} that is dependent on T_S, a notifiable, N_i, referencing T_{R_i} is registered in T_S. By the algorithm, when a transaction T_S is set to committed, it sends dependency resolution notification to all its registered notifiables $\{N_{i=1..n}\}$. N_i notifies T_{R_i} about the dependency resolution. Therefore, any transaction T_R that is dependent on T_S is notified about the dependency resolution. ■

5.3.2. Background

DEFINITION 13: For every directed graph $G[V^G, E^G]$, its condensation (or component) graph $\mathbb{C}(G)[V^{\mathbb{C}(G)}, E^{\mathbb{C}(G)}]$ is defined as follows:
Assuming that $\{SCC_{i=1..n}\}$ is the set of strongly connected components of G, there is a bijective function f (or a one to one correspondence) between $V^{\mathbb{C}(G)} = \{v_{i=1..n}^{\mathbb{C}(G)}\}$ and $\{SCC_{i=1..n}\}$, i.e.

$$\begin{aligned} &\forall i = 1..n: \\ &f(SCC_i) = v_i^{\mathbb{C}(G)} \; and \; f^{-1}\left(v_i^{\mathbb{C}(G)}\right) = SCC_i \end{aligned} \qquad \text{EQ. 5}$$

(contracting each SCC_i into a supervertex $v_i^{\mathbb{C}(G)}$) and

$$\forall i, j = 1..n, i \neq j: \left\{ \begin{matrix} \left[\left(v_i^{\mathbb{C}(G)}, v_j^{\mathbb{C}(G)}\right) \in E^{\mathbb{C}(G)}\right] \Leftrightarrow \\ \begin{bmatrix} \exists v_k^G, v_l^G \in V^G: \\ v_k^G \in SCC_i \; and \; v_l^G \in SCC_j \; and \; (v_k^G, v_l^G) \in E^G \end{bmatrix} \end{matrix} \right\} \qquad \text{EQ. 6}$$

[15].

THEOREM 3: Condensation graph is a DAG (directed acyclic graph). [15]

DEFINITION 14: The reverse (or transpose) graph of a directed graph $G[V, E]$ is a directed graph $\mathbb{R}(G)[V, E^{\mathbb{R}(G)}]$ such that

$$\forall v, u \in V: \left[(v, u) \in E^{\mathbb{R}(G)} \Leftrightarrow (u, v) \in E\right] \qquad \text{EQ. 7}$$

[15].

DEFINITION 15: A topological ordering (or topological sort) of a DAG $G[V, E]$ is a permutation π of V (a bijective function from $\{1..|V|\}$ to V) such that

$$\forall v, u \in V: \left[\pi^{-1}(v) < \pi^{-1}(u) \Longrightarrow (u, v) \notin E\right] \qquad \text{EQ. 8}$$

or equivalently

$$\forall v, u \in V: \left[(u, v) \in E \Longrightarrow \pi^{-1}(u) < \pi^{-1}(v)\right] \qquad \text{EQ. 9}$$

[15].

5.3.3. Finalization Theorems

THEOREM 1: The Transactor algorithm has the Finalization property, i.e. every transaction eventually finalizes.

PROOF: The dependencies of transaction descriptors form a directed graph $G[V^G, E^G]$. Let $SCCs$ denote the set of strongly connected components of G. Obviously, For every $v^G \in V^G$, there is an $SCC \in SCCs$ that $v^G \in SCC$. Therefore the theorem is reduced to the following theorem. ■

The theorem shows that communicating transactions never go to deadlock.

DEFINITION 16: We say that an SCC of the dependency graph is aborted iff all of its transactions are aborted and we say that it is committed iff all of its transactions are committed. An SCC is finalized iff it is aborted or committed.

THEOREM 4: All SCCs of the transaction dependency graph eventually finalize.

PROOF: If one of the nodes in an SCC eventually aborts, by AlgOp 4, all of the transactions of the SCC eventually abort, i.e. finalize.

If none of the transactions in the SCC abort, the theorem reduces to the following theorem. ■

THEOREM 5: If none of the transactions in an SCC of the transaction dependency graph abort, the SCC eventually finalizes.
PROOF: If none of the transactions of an SCC abort, then all of them eventually go to the terminated state. We prove that all of them eventually finalize.
Let $G[V^G, E^G]$ be the dependency graph of transaction descriptors. Let G' be the condensation graph of G, i.e. $G' = \mathbb{C}(G)[V^{\mathbb{C}(G)}, E^{\mathbb{C}(G)}]$. Let G'' be the reverse of the condensation graph, i.e. $G'' = \mathbb{R}(\mathbb{C}(G))[V^{\mathbb{C}(G)}, E^{\mathbb{R}(\mathbb{C}(G))}]$. By THEOREM 3, G' is a DAG, therefore G'' is also a DAG. Let π be a topological order of G'' and for $i = 1..|V^{\mathbb{C}(G)}|$, let $v_i^{\mathbb{C}(G)} \in V^{\mathbb{C}(G)}$ be the ith element in π, i.e. $\pi(i) = v_i^{\mathbb{C}(G)}$. For $i = 1..|V^{\mathbb{C}(G)}|$, let SCC_i denote the SCC of G that corresponds to $v_i^{\mathbb{C}(G)} \in V^{\mathbb{C}(G)}$.
We prove that for $i = 1..|V^{\mathbb{C}(G)}|$, if none of $v^G \in SCC_i$ abort, SCC_i is eventually finalized.

Proof is by induction on i.

1. Base case: $i = 1$:
$v_1^{\mathbb{C}(G)}$ is the first node in π.
$$\forall i = 2..|V^{\mathbb{C}(G)}|: \pi^{-1}\left(v_1^{\mathbb{C}(G)}\right) < \pi^{-1}\left(v_i^{\mathbb{C}(G)}\right) \qquad \text{EQ. 10}$$
By DEFINITION 15,
$$\forall i = 2..|V^{\mathbb{C}(G)}|: (v_i^{\mathbb{C}(G)}, v_1^{\mathbb{C}(G)}) \notin E^{\mathbb{R}(\mathbb{C}(G))} \qquad \text{EQ. 11}$$
By DEFINITION 14,
$$\forall i = 2..|V^{\mathbb{C}(G)}|: (v_1^{\mathbb{C}(G)}, v_i^{\mathbb{C}(G)}) \notin E^{\mathbb{C}(G)} \qquad \text{EQ. 12}$$
By DEFINITION 13,
$$\begin{aligned}&\forall i = 2..|V^{\mathbb{C}(G)}|:\\ &\nexists v_k^G, v_l^G \in V^G:\\ &v_k^G \in SCC_1 \; and \; v_l^G \in SCC_i \; and \; (v_k^G, v_l^G) \in E^G\end{aligned} \qquad \text{EQ. 13}$$
That is equivalent to:
$$\begin{aligned}&\forall i = 2..|V^{\mathbb{C}(G)}|:\\ &\forall v_k^G, v_l^G \in V^G:\\ &v_k^G \notin SCC_1 \; or \; v_l^G \notin SCC_i \; or \; (v_k^G, v_l^G) \notin E^G\\ &\equiv\\ &\forall i = 2..|V^{\mathbb{C}(G)}|:\\ &\forall v_k^G, v_l^G \in V^G \; and \; v_k^G \in SCC_1 \; and \; v_l^G \in SCC_i:\\ &(v_k^G, v_l^G) \notin E^G\end{aligned} \qquad \text{EQ. 14}$$
This means that nodes in SCC_1 have no dependency to any node in other SCCs.
By the assumption of the theorem, none of the transactions in SCC_1 abort; therefore, all of them eventually go to terminated state.
When cluster search is started from the last terminated transaction in SCC_1, all transactions in it are already terminated. Besides, as there is no dependency from SCC_1 to any other SCC, this cluster search can only reach nodes in SCC_1. Hence by ALGOP 5, the cluster search started from the last terminated transaction commits all of the nodes in SCC_1. Hence, by DEFINITION 2, all of them are eventually finalized.
2. Inductive step:
If for $i = 1..j$, all SCC_i are eventually finalized, i.e. aborted or committed, we prove that SCC_{j+1} is eventually finalized. Formally
$$\begin{aligned}&If\\ &\forall p = 1..|V^{\mathbb{C}(G)}|:\\ &\left[p < j+1 \Rightarrow \begin{pmatrix}(SCC_p \; is \; eventually \; aborted) \; or\\ (SCC_p \; is \; eventually \; committed)\end{pmatrix}\right]\\ &then\\ &\qquad SCC_{j+1} \; is \; eventually \; finalized.\end{aligned} \qquad \text{EQ. 15}$$
By DEFINITION 15, for $v_{j+1}^{\mathbb{C}(G)} \in V^{\mathbb{C}(G)}$:
$$\begin{aligned}&\forall v_p^{\mathbb{C}(G)} \in V^{\mathbb{C}(G)}, p = 1..|V^{\mathbb{C}(G)}|:\\ &\begin{bmatrix}\left(v_p^{\mathbb{C}(G)}, v_{j+1}^{\mathbb{C}(G)}\right) \in E^{\mathbb{R}(\mathbb{C}(G))} \Rightarrow\\ \pi^{-1}(v_p^{\mathbb{C}(G)}) < \pi^{-1}(v_{j+1}^{\mathbb{C}(G)})\end{bmatrix}\end{aligned} \qquad \text{EQ. 16}$$
By DEFINITION 14,
$$\begin{aligned}&\forall v_p^{\mathbb{C}(G)} \in V^{\mathbb{C}(G)}, p = 1..|V^{\mathbb{C}(G)}|:\\ &\begin{bmatrix}\left(v_{j+1}^{\mathbb{C}(G)}, v_p^{\mathbb{C}(G)}\right) \in E^{\mathbb{C}(G)} \Rightarrow\\ \pi^{-1}\left(v_p^{\mathbb{C}(G)}\right) < \pi^{-1}\left(v_{j+1}^{\mathbb{C}(G)}\right)\end{bmatrix}\\ &\equiv\\ &\forall v_p^{\mathbb{C}(G)} \in V^{\mathbb{C}(G)}, p = 1..|V^{\mathbb{C}(G)}|:\\ &\left[\left(v_{j+1}^{\mathbb{C}(G)}, v_p^{\mathbb{C}(G)}\right) \in E^{\mathbb{C}(G)} \Rightarrow p < j+1\right]\end{aligned} \qquad \text{EQ. 17}$$
By induction hypothesis,
$$\begin{aligned}&\forall v_p^{\mathbb{C}(G)} \in V^{\mathbb{C}(G)}, p = 1..|V^{\mathbb{C}(G)}|:\\ &\begin{bmatrix}\left(v_{j+1}^{\mathbb{C}(G)}, v_p^{\mathbb{C}(G)}\right) \in E^{\mathbb{C}(G)} \Rightarrow\\ \begin{pmatrix}(SCC_p \; is \; eventually \; aborted) \; or\\ (SCC_p \; is \; eventually \; committed)\end{pmatrix}\end{bmatrix}\end{aligned} \qquad \text{EQ. 18}$$
2.1. If there is a $v_p^{\mathbb{C}(G)} \in V^{\mathbb{C}(G)}$ that $\left(v_{j+1}^{\mathbb{C}(G)}, v_p^{\mathbb{C}(G)}\right) \in E^{\mathbb{C}(G)}$ and $SCC_p \; is \; eventually \; aborted$:
From $\left(v_{j+1}^{\mathbb{C}(G)}, v_p^{\mathbb{C}(G)}\right) \in E^{\mathbb{C}(G)}$ and DEFINITION 13:
$$\begin{aligned}&\exists v_k^G, v_l^G \in V^G:\\ &v_k^G \in SCC_{j+1} \; and \; v_l^G \in SCC_p \; and \; (v_k^G, v_l^G) \in E^G\end{aligned} \qquad \text{EQ. 19}$$
From the fact that SSC_p is eventually aborted, $v_l^G \in SCC_p$ and DEFINITION 16, we have that v_l^G is eventually aborted. From the fact that v_l^G is eventually aborted, $(v_k^G, v_l^G) \in E^G$ and ALGOP 1, we have that v_k^G is eventually aborted. From the fact that v_k^G is eventually aborted, $v_k^G \in SCC_{j+1}$ and AlgOp 4, all of the nodes in SCC_{j+1} are eventually aborted. So by DEFINITION 16, SCC_{j+1} is eventually finalized.
2.2. If for none of $v_p^{\mathbb{C}(G)} \in V^{\mathbb{C}(G)}$ that $\left(v_{j+1}^{\mathbb{C}(G)}, v_p^{\mathbb{C}(G)}\right) \in E^{\mathbb{C}(G)}$, $SCC_p \; is \; eventually \; aborted$, i.e.:
$$\begin{aligned}&\forall v_p^{\mathbb{C}(G)} \in V^{\mathbb{C}(G)}, p = 1..|V^{\mathbb{C}(G)}|:\\ &\begin{bmatrix}\left(v_{j+1}^{\mathbb{C}(G)}, v_p^{\mathbb{C}(G)}\right) \in E^{\mathbb{C}(G)} \Rightarrow\\ not(SCC_p \; is \; eventually \; aborted)\end{bmatrix}.\end{aligned} \qquad \text{EQ. 20}$$
By EQ. 18, we have:
$$\begin{aligned}&\forall v_p^{\mathbb{C}(G)} \in V^{\mathbb{C}(G)}, p = 1..|V^{\mathbb{C}(G)}|:\\ &\begin{bmatrix}\left(v_{j+1}^{\mathbb{C}(G)}, v_p^{\mathbb{C}(G)}\right) \in E^{\mathbb{C}(G)} \Rightarrow\\ (SCC_p \; is \; eventually \; comitted)\end{bmatrix}\end{aligned} \qquad \text{EQ. 21}$$
On the other hand, by DEFINITION 13, for all $p = 1..|V^{\mathbb{C}(G)}|$,

$$\begin{bmatrix} \exists v_k^G, v_l^G \in V^G: \\ v_k^G \in SCC_{j+1}\ and\ v_l^G \in SCC_p\ and (v_k^G, v_l^G) \in E^G \end{bmatrix} \Rightarrow \left[\left(v_{j+1}^{\mathbb{C}(G)}, v_p^{\mathbb{C}(G)}\right) \in E^{\mathbb{C}(G)}\right] \quad \text{Eq. 22}$$

that is equivalent to

$$\forall p = 1..\left|V^{\mathbb{C}(G)}\right|: \begin{bmatrix} \left(\exists v_k^G \in SCC_{j+1}, \exists v_l^G \in SCC_p: (v_k^G, v_l^G) \in E^G\right) \\ \Rightarrow \\ \left(v_{j+1}^{\mathbb{C}(G)}, v_p^{\mathbb{C}(G)}\right) \in E^{\mathbb{C}(G)} \end{bmatrix} \quad \text{Eq. 23}$$

By Eq. 21:

$$\forall p = 1..\left|V^{\mathbb{C}(G)}\right|: \begin{bmatrix} \left(\exists v_k^G \in SCC_{j+1}, \exists v_l^G \in SCC_p: (v_k^G, v_l^G) \in E^G\right) \\ \Rightarrow \\ (SCC_p\ is\ eventually\ comitted) \end{bmatrix} \quad \text{Eq. 24}$$

By Definition 16,

$$\forall p = 1..\left|V^{\mathbb{C}(G)}\right|: \begin{bmatrix} \left(\exists v_k^G \in SCC_{j+1}, \exists v_l^G \in SCC_p: (v_k^G, v_l^G) \in E^G\right) \\ \Rightarrow \\ (\forall v_r^G \in SCC_p:\ v_r^G\ eventually\ commits) \end{bmatrix} \quad \text{Eq. 25}$$

By AlgOp 6:

$$\forall p = 1..\left|V^{\mathbb{C}(G)}\right|: \begin{bmatrix} \left(\exists v_k^G \in SCC_{j+1}, \exists v_l^G \in SCC_p: (v_k^G, v_l^G) \in E^G\right) \\ \Rightarrow \\ \begin{pmatrix} \forall v_r^G \in SCC_p: \\ \forall v_s^G \in V^G, (v_s^G, v_r^G) \in E^G: \\ v_s^G\ is\ eventually\ notified\ of \\ dependency\ resolution \end{pmatrix} \end{bmatrix} \quad \text{Eq. 26}$$

Since $v_k^G \in SCC_{j+1}$, obviously $v_k^G \in V^G$. $v_l^G \in SCC_p$ and $v_k^G \in V^G$; therefore, v_l^G and v_k^G are proper substitutions for respectively v_r^G and v_s^G. After substitution, we have

$$\forall p = 1..\left|V^{\mathbb{C}(G)}\right|: \begin{bmatrix} \left(\exists v_k^G \in SCC_{j+1}, \exists v_l^G \in SCC_p, (v_k^G, v_l^G) \in E^G\right) \\ \Rightarrow \\ v_k^G\ is\ eventually\ notified\ of \\ dependency\ resolution \end{bmatrix} \quad \text{Eq. 27}$$

This means that if there is any dependency from a transaction v_k^G inside SCC_{j+1} to a transaction in any other SCC, v_k^G is eventually notified of dependency resolution. By the assumption of the theorem, none of the transactions in SCC_{j+1} abort; hence all of them are eventually terminated. Consider when the last dependency resolution notification of the transactions inside SCC_{j+1} is received. At this time, transactions inside SCC_{j+1} have no dependency other than dependencies to other transactions inside SCC_{j+1}. When the transaction that receives the last notification performs the cluster search, by AlgOp 5, all of the transactions in SCC_{j+1} are committed. Therefore by Definition 16, SCC_{j+1} is eventually finalized. ■

6 Related Works

A CAA [17] is essentially a set of processes that start together, can send messages to and receive messages from each other, can access shared memory and finally commit atomically together. The limitation of CAA is that the processes are statically interdependent from the beginning. Dependencies are not tracked dynamically. So, all of the processes of a CAA are always committed together, even if they do not communicate at runtime.

7 Performance Evaluations

To compare performance of our Scala implementation of Transactors with Scala Actors and STM in provision of isolation and communication, experimentations are conducted with two fundamental cases.

Each case is implemented with Scala Actors, Scala STM, Scala Transactors and Locks. The code snippets of the implementations of the two cases with each paradigm are presented in the following subsections.

Cases

Bank Account Credit Transfer

To compare performance for isolation, the classical case of transferring credit between bank accounts is experimented. To transfer credit from an account to another, both balances should be read and written in isolation.

Coarse-grained locking

In the coarse-grained locking, all of the transfers even if they are not conflicting are serialized by the bank intrinsic lock.

```
this.synchronized {
   account1.withdraw(amount)
   account2.deposit(amount)
}
```

Fine-grained locking

In the fine-grained locking, instead of having a lock for the whole bank, each account has a lock. It is notable that locks are always acquired in the same order.

```
if (accNo1 <= accNo2) {
   account1.lock.lock
   account2.lock.lock
} else {
   account2.lock.lock
   account1.lock.lock
}

account1.withdraw(amount)
account2.deposit(amount)

account1.lock.unlock
account2.lock.unlock
```

Actors

Each account is modeled as an actor that handles withdraw and deposit requests. As messages are handled one at a time by actors, withdraw, deposit and balance requests are done is isolation.

```
// From Account actor:
def act() {
  react {
    case Withdraw(amount) =>
      b -= amount
      sender ! WithdrawDone
      act
    case Deposit(amount) =>
      b += amount
      sender ! DepositDone
      act
    case BalanceRequest =>
      sender ! Balance(b)
      act
    case TerminateRequest =>
  }
}
```

The transfer operation of the bank sends withdraw and deposit requests to account actors and wait for their acknowledgments before returning.

```
// Transfer operation from Bank class
accounts(accNo1) ! Withdraw(amount)
accounts(accNo2) ! Deposit(amount)
receive {
  case WithdrawDone =>
    receive {
      case DepositDone =>
   }
}
```

This implementation provides an eventual guarantee. Finally, the sum of all of the account balances is the same as the sum before the transfers. We also experimented with an implementation that provides isolation for each transfer but it turned out to be very inefficient. This implementation is more efficient and hence, it is used in performance comparisons.

Transactions and Transactors

The code for Transactions and Transactors is the same: Credit transfer is simply an atomic block. Basically, only the "Transaction part" of Transactors in used for this case.

```
atomic {
   accounts(accNo1).withdraw(amount)
   accounts(accNo1).deposit(amount)
}
```

Token Ring

To compare performance for communication, token ring is simulated. In token ring LAN DLL protocol, stations are organized in a ring topology with a control token being passed sequentially from one station to the next. The token ring simulation essentially employs the communication mechanism of each paradigm.

Locks and Conditions

The station waits on the intrinsic condition of the incoming port while the token is not inside the port yet. When the station finds the token inside the incoming port (maybe after being notified by the neighbor station), it takes the token from the incoming port and puts it inside the outgoing port. The next station may have been suspended after a wait on the outgoing port. To awake the next station, the station notifies on the outgoing port after putting the token in it.

```
inPort.synchronized {
  while (inPort.value == null)
    inPort.wait
  outPort.synchronized {
    outPort.value = inPort.value
    outPort.notify
    inPort.value = null
  }
}
```

Actors

The token ring is very straightforward with Actors. The actor reacts to receiving of the token by sending it to the next station.

```
def act {
  if (currentRound != roundCount)
    react {
     case Token =>
       nextStation ! Token
       currentRound += 1
       act
   }
}
```

Transactions

Port is defined as a transactional object. Inside an atomic block, the station reads the value of the incoming port. If its value is null, i.e. there is no value inside it, `conditionWait` aborts the transaction. The transaction is retried only after the incoming port object is updated. When a (retrying) transaction succeeds in reading the token from the incoming port, it updates values of incoming and outgoing ports to null and the token respectively.

```
atomic {
  conditionWait (inPort.value != null)
  outPort.value = inPort.value
  inPort.value = null
}
```

Transactors

Each station is modeled as a transactor which receives the token and sends it to the next station. Essentially, only the "Actor part" of Transactors is employed for this case.

```
def act {
  while (currentRound != roundCount) {
    val token = receive
    nextStation ! token
    currentRound += 1
  }
}
```

Experiments

The experiments are done on Dell Latitude E6400 Intel® Core™2 Duo CPU P8600 @2.40GHz.

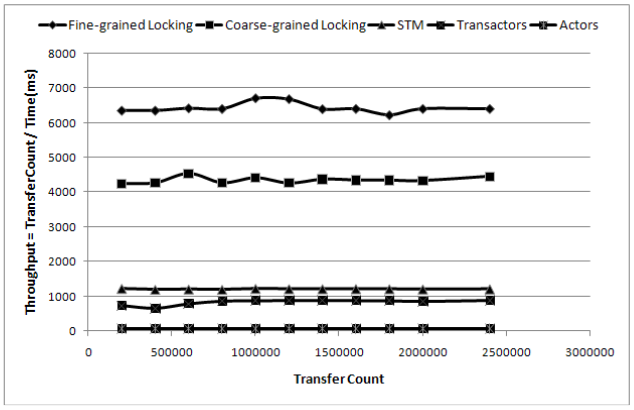

Figure 6. Bank Credit Transfer

Figure 6 depicts performance evaluations for the credit transfer case where the number of transferers is 20 and the number of accounts is 100. For each paradigm, throughput is shown against number of transfers where throughput is transfer count per milliseconds. The plot shows that aside from locking, STM outperforms the other paradigms for isolation. Besides, it shows that performance of Transactors is close to STM for isolation.

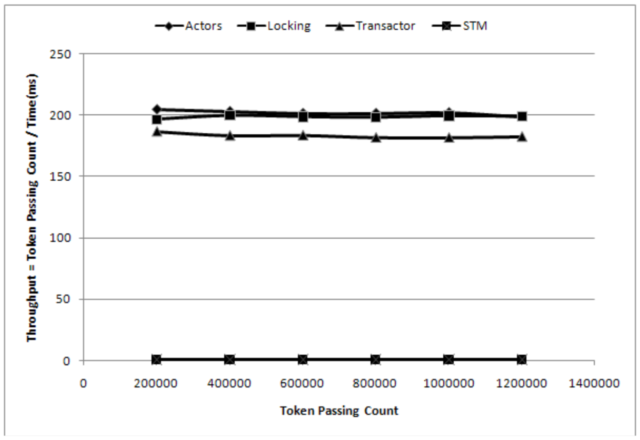

Figure 7. Token Ring

Figure 7 depicts performance evaluations for the token ring case where number of stations is 20. Throughput is number of token passings per milliseconds. The throughput of each paradigm is shown against number of token passings. This plot shows that Actors have almost the same performance as locking for communication. It also shows that Transactors perform very close to Actors for communication.

The experiments suggest that Transactors merge performance benefits of STM and Actors.

More information about comparison of paradigms can be found in [20].

References

[15] Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L.. Introduction to Algorithms. MIT Press and McGraw-Hill., Section 22.5, pp.552–557, ex. 22.1–3, p. 530, pp.549–552.

[16] Donnelly, K. and Fluet, M. 2006. Transactional events. SIGPLAN Not. 41, 9 (Sep. 2006), 124-135. DOI= http://doi.acm.org/10.1145/1160074.1159821

[17] Gallina, B., Guelfi, N., and Romanovsky, A. 2007. Coordinated Atomic Actions for Dependable Distributed Systems: the Current State in Concepts, Semantics and Verification Means. In *Proceedings of the the 18th IEEE international Symposium on Software Reliability* (November 05 - 09, 2007). ISSRE. IEEE Computer Society, Washington, DC, 29-38. DOI= http://dx.doi.org/10.1109/ISSRE.2007.5

[18] Harary, Frank; Norman, Robert Z.; Cartwright, Dorwin (1965), Structural Models: An Introduction to the Theory of Directed Graphs, John Wiley & Sons, p. 63.

[19] Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. N. 2003. Software transactional memory for dynamic-sized data structures. In Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing (Boston, Massachusetts, July 13 - 16, 2003). PODC '03. ACM, New York, NY, 92-101. DOI= http://doi.acm.org/10.1145/872035.872048

[20] Lesani, M., Odersky, M. and Guerraoui, R. Concurrent Programming Paradigms, A Comparison in Scala. LAMP-REPORT-2009-002, School of Computer and Communication Sciences, EPFL.

[21] Schholliers, C., Van Cutsem, T., and De Meuter, W. 2008. Ambient transactors. In Proceedings of the 6th international Workshop on Middleware For Pervasive and Ad-Hoc Computing (Leuven, Belgium, December 01 - 05, 2008). MPAC '08. ACM, New York, NY, 49-53. DOI= http://doi.acm.org/10.1145/1462789.1462798

[22] Ziarek, L., Schatz, P., and Jagannathan, S. 2007. Modular Checkpointing for Atomicity. Electron. Notes Theor. Comput. Sci. 174, 9 (Jun. 2007), 85-115.