# Transactional Actors: Communication in Transactions

Janwillem Swalens
Software Languages Lab
Vrije Universiteit Brussel
Brussel, Belgium
jswalens@vub.be

Joeri De Koster
Software Languages Lab
Vrije Universiteit Brussel
Brussel, Belgium
jdekoste@vub.be

Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
Brussel, Belgium
wdmeuter@vub.be

## Abstract

Developers often require different concurrency models to fit the various concurrency needs of the different parts of their applications. Many programming languages, such as Clojure, Scala, and Haskell, cater to this need by incorporating different concurrency models. It has been shown that, in practice, developers often combine these concurrency models. However, they are often combined in an ad hoc way and the semantics of the combination is not always well-defined. The starting hypothesis of this paper is that different concurrency models need to be carefully integrated such that the properties of each individual model are still maintained.

This paper proposes one such combination, namely the combination of the actor model and software transactional memory. In this paper we show that, while both individual models offer strong safety guarantees, these guarantees are no longer valid when they are combined. The main contribution of this paper is a novel hybrid concurrency model called *transactional actors* that combines both models while preserving their guarantees. This paper also presents an implementation in Clojure and an experimental evaluation of the performance of the transactional actor model.

***CCS Concepts*** • **Software and its engineering → Concurrent programming languages**; **Parallel programming languages**;

***Keywords*** Concurrency, Parallelism, Software Transactional Memory, Actors

**ACM Reference Format:**
Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. 2017. Transactional Actors: Communication in Transactions. In *Proceedings of 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems (SEPS'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3141865.3141866

## 1 Introduction

The multi-core revolution has marked the end of an era in which software developers benefitted from the exponential increase of processor clock speeds to improve the performance of their applications. This revolution has led to an increased interest in concurrency models. Over the past decade, many researchers have revisited old and invented new concurrency models. Today, several of these concurrency models have found their way into modern programming languages. For example, Scala and Clojure provide a plethora of concurrency models. However, these concurrency models are often integrated in an ad hoc way and the semantics of the combination is not always well-defined [19].

This paper argues that developers often combine concurrency models and that these should be carefully integrated such that the properties and guarantees of each model are maintained. Most concurrency models fall into one of two categories: message-passing and shared-memory models [21]. This paper proposes a hybrid programming model in which a concurrency model of both categories, the actor model and software transactional memory respectively, are carefully integrated. This results in a hybrid concurrency model that enables developers to arbitrarily mix concepts and constructs from both models while maintaining the guarantees of both.

The actor model [1] is a message-passing concurrency model that avoids deadlocks and low-level data races by design. Actors are concurrent entities with strict isolation: the state of an actor is fully encapsulated and can only be accessed using asynchronous communication. While the actor model strictly prohibits shared memory, software transactional memory (STM) [17] is at the opposite side of the spectrum. STM introduces transactions: blocks of code in which shared memory locations can be read and modified safely. STM simplifies concurrent access to shared memory, as transactions are executed atomically and are guaranteed to be serializable. Depending on the type of STM, this can

*SEPS'17, October 23, 2017, Vancouver, Canada*

ACM ISBN 978-1-4503-5517-9/17/10…$15.00
https://doi.org/10.1145/3141865.3141866

be implemented by storing modifications locally, and committing them at the end of the transaction. Transactions that want to commit conflicting updates are rolled back and retried.

The fact that transactions can be arbitrarily rolled back and retried makes STM particularly hard to integrate with other concurrency models. Any operation performed within a transaction needs to be either idempotent or a compensating action needs to be available for when the transaction is aborted. This is particularly problematic for several operations of the actor model such as creating a new actor or sending a message from within a transaction.

In this paper, we introduce **transactional actors**. Transactional actors combine the actor model with software transactional memory with a well-defined semantics. Using this hybrid concurrency model, developers can arbitrarily mix concepts from each concurrency model while maintaining their individual guarantees. In particular, we make the following contributions:

- In Section 3, we show that combining a message-passing model and a shared-memory model is useful, and happens in practice. Furthermore, we show that a naïve combination of actors and transactions in existing languages violates the guarantees provided by the separate models.
- We propose transactional actors, by specifying a well-defined semantics for the combination of the actor model and software transactional memory (Section 4).
- Transactional actors maintain the guarantees of the separate models: serializability, freedom from low-level data races, and freedom from deadlocks (Section 5).
- We present an implementation of transactional actors as an extension of Clojure (Section 6), and evaluate its performance using a representative benchmark (Section 7).

## 2 Background: Actors and Transactional Memory

We first describe the actor model and software transactional memory separately. We describe the constructs, use cases, and properties of each model. Throughout this paper, we will use a flight reservation system as a running example.

### 2.1 Actors

The actor model used in this paper is based on Agha et al. [2]. A program consists of multiple actors that run concurrently. An actor has three elements: a unique address, an inbox of messages, and a behavior. A message is a tuple of values, similar to Erlang.

In our language, a behavior is defined as follows:

```
1 (def airline-behavior
2   (behavior [flights] [orig dest n]
3     (let [flight   (search-flight flights orig dest)
4           flight'  (reserve flight n)
5           flights' (replace flights flight flight')]
6       (become airline-behavior flights'))))
```

This behavior specifies an actor that represents an airline. Messages can be sent to this actor to reserve a flight. The behavior of an actor defines how it responds to an incoming message. A behavior is parameterized by two types of parameters: first, the internal state of the actor (here, `flights`, a list of all flights by that airline), second, the values of the received message (here, the details of a reservation: its origin `orig`, destination `dest`, and the number `n` of seats to reserve).

An actor can be spawned using `spawn`, e.g.

```
7 (def airline (spawn airline-behavior
8   [{:orig "LHR" :dest "YVR" :seats-available 211}]))
```

This creates a new actor with `airline-behavior` as initial behavior and the list of flights as internal state. `spawn` returns the address of the new actor.

`(send airline "LHR" "YVR" 2)` sends a message to this actor: it puts a message containing the values `"LHR"`, `"YVR"`, and `2` in the inbox of the actor with address `airline`. When the receiving actor processes the message, it will execute the code in the behavior defined above (lines 3–6), with `flights` bound to the list of flights given when the actor was spawned, and `orig`, `dest`, and `n` bound to the message's values.

An actor can change its behavior and internal state using `become`. On line 6 in the example, `(become airline-behavior flights')` updates the `airline` actor, keeping its behavior identical but updating its internal state to the new list of flights, in which two seats were reserved.

An actor alternates between two states: ready to accept a message, or busy processing a message. A *turn* is the processing of a single message by an actor, that is, the process of an actor taking a message from its inbox and processing that message to completion [5].

The actor model provides the **isolated turn principle**: a turn can be seen as a single, isolated step. This is a result of three properties of the actor model: (1) an actor's state cannot be observed by other actors except through messages, (2) an actor processes its messages one by one, there is no parallelism inside an actor, and (3) the actor model does not contain any blocking operations, guaranteeing that once a turn started, it always runs to completion. The isolated turn principle guarantees that the actor model is *free from low-level data races* and *free from deadlocks*. Thanks to the isolated turn principle, a developer can reason about an actor system at the level of turns. When two actors execute in parallel, it does not matter in which order the instructions in their turns are interleaved, only in which order the turns are executed. This makes the program easier to understand, reason about, and debug.

### 2.2 Transactional Memory

Software Transactional Memory (STM) is a concurrency model that allows multiple threads to safely access shared variables, grouping the accesses into *transactions* [17]. Applications using STM typically contain complex data structures, of which the pieces that are modified by multiple threads are encapsulated in transactional variables. In our case, these *transactional variables* are memory locations that contain a value, and are created using (**ref** v). In Haskell these are called TVars, in Clojure they are refs. A transaction (**dosync** e) encapsulates an expression that can contain a number of *primitive operations* to the shared objects, such as reads (**deref** r) (abbreviated to @r) and writes (**ref-set** r v).

For example, the code below reserves two flights. Lines 1–3 define a map of all flights. Each flight has a number of available seats, encapsulated in a transactional variable. Lines 5–11 contain a transaction that reserves two seats. If enough seats are available on the outbound and the return flight, the number of available seats on those flights is decreased by two; else, no reservation is made.

```
1  (def flights
2    {"AC854" {:orig "YVR" :dest "LHR" :seats (ref 211)}
3     "AC855" {:orig "LHR" :dest "YVR" :seats (ref 211)} …})
4
5  (dosync
6    (let [outbound (get (get flights "AC855") :seats)
7          return   (get (get flights "AC854") :seats)]
8      (if (and (>= @outbound 2) (>= @return 2))
9        (do (ref-set outbound (- @outbound 2))
10           (ref-set return   (- @return   2)))
11       (println "Not enough seats available"))))
```

Transactional systems guarantee **serializability**: transactions appear to execute serially, so that the steps of one transaction never appear to be interleaved with the steps of another [10]. The result of a transactional program must always be equal to the result of *a* serial execution of the program. In the example, this entails that the values of the transactional variables on line 8 and on lines 9 and 10 must be equal, even if another thread modified them in the meantime. Due to serializability, the developer can reason about the program at the level of transactions: when transactions execute in parallel, it does not matter in which order their instructions are interleaved, only in which order the transactions are committed. This too makes the program easier to understand and debug.

In Clojure and Haskell, STM is implemented using multiversion concurrency control (MVCC) [3, 8, 9]. Each transactional variable contains a (limited) history of its values. During a transaction, reading a transactional variable will return the value it had when the transaction started. Writing a transactional variable will store its new value locally. At the end of the transaction, it will attempt to commit, and the new values become visible for new transactions. In case of a conflict during the commit, which occurs when two transactions wrote to the same variable, the later transaction is aborted. This means it discards its changes, and retries.

Because transactions can be retried, a transaction should not contain operations with a side effect, such as I/O. In the next section we will see that this can lead to problems when STM is combined with other concurrency models.

## 3 Motivation and Problem Statement

In this section, we motivate the combination of both actors and transactional memory in one application. On the one hand, it can be useful to add transactional memory to an actor system, to allow memory to be shared safely between the actors (Section 3.1). On the other hand, it can be useful to send messages to actors in a transaction, to distribute and coordinate work between transactions (Section 3.2).

### 3.1 Transactions in Actors, to Share Memory

For some applications, it is desirable to introduce shared memory in an actor system. We discuss two types of actor systems: pure and impure systems [4].

*Pure actor systems* are actor systems that enforce strict isolation between the actors: the state of an actor is fully encapsulated and can only be accessed asynchronously. The developer benefits from strong safety guarantees: low-level data races are prevented by design. However, representing shared state in pure actor systems is complex [4]. The difficulty of preventing race conditions and deadlocks is pushed to the developer.

*Impure actor systems* do not enforce strict isolation. These are often libraries for languages that do not have actors built in. Here, developers can use the underlying shared-memory model of the language when shared memory is the most natural or efficient solution.

Tasharofi et al. [20] performed a study of 15 large, mature, and actively maintained actor programs written in Scala. They found that 80% mix the actor model with another concurrency model. In 6 out of 15 cases (40%), developers circumvent the actor model in the places where it is not a good fit. In some cases, developers introduce shared memory, which they protect using locks. However, the disadvantage of this approach is that the guarantees of the actor model are lost: the isolated turn principle is broken, hence low-level data races and deadlocks become possible.

These guarantees can be re-introduced by sharing memory between actors using transactions. STM guarantees the absence of low-level races by encapsulating atomic sections in a transaction. Moreover, in contrast to traditional locking, STM guarantees the absence of deadlocks. Thus, using STM, memory can safely be shared between actors.

### 3.2 Actors in Transactions, to Distribute and Coordinate Work

Not only are transactions useful to share memory between actors, conversely, actors are also useful to coordinate work between transactions. In this section, we demonstrate how actors can be used to distribute and coordinate work from within a transaction, again using a flight reservation system. This example is inspired by the Vacation benchmark from STAMP [14], a suite of eight applications that use STM.

```
 1 (def flights [(ref {:id "AC855"  :price 499 :orig "London" :dest "Vancouver" :available ["1A" "1B" …]    :occupied []}) …])
 2 (def rooms   [(ref {:id 101      :price 100 :location "Vancouver" :beds 5    :available ["2017-10-01" …] :occupied []}) …])
 3 (def cars    [(ref {:id "ABC123" :price 42  :location "Vancouver" :seats 5   :available ["2017-10-01" …] :occupied []}) …])
 4 (def customers [(ref {:id 0 :orig "London" :dest "Vancouver" :n 3 :start "2017-10-22" :end "2017-10-27" :password nil}) …])
 5
 6 (defn reserve-flight [orig dest date seats]      ; Finds the cheapest flight from orig to dest on date, and reserve seats.
 7   (let [filtered (filter (fn [f] …@f…) flights) ; Filter flights from orig to dest on date, with sufficient seats
 8         cheapest (get-cheapest filtered)]        ; Find the cheapest of these flights
 9     (ref-set cheapest (occupy cheapest seats)))) ; Move seats from :available to :occupied
10 ; Functions reserve-room and reserve-car are similar
11
12 (defn process-customer [c]
13   (dosync
14     (reserve-flight (:orig @c) (:dest @c) (:start @c) (:n @c)) ; Find and update two flights (outbound & return),
15     (reserve-flight (:dest @c) (:orig @c) (:end @c) (:n @c))   ; a room, and a car, using details from the customer.
16     (reserve-room   (:dest @c) (:n @c) (:start @c) (:end @c))
17     (reserve-car    (:dest @c) (:n @c) (:start @c) (:end @c))
18     (ref-set c (assoc @c :password (generate-password)))))     ; Attach a generated password to the customer.
```

**Listing 1.** The Vacation example. (Code has been modified for clarity.)

```
(defn process-customer [c]
  (dosync
   (send (rand-nth sec-workers) :flight (:orig @c) …)
   (send (rand-nth sec-workers) :flight (:dest @c) …)
   (send (rand-nth sec-workers) :room   (:dest @c) …)
   (send (rand-nth sec-workers) :car    (:dest @c) …)
   (ref-set c (assoc @c :password (generate-password)))))

(def sec-worker-behavior
  (behavior [id] [type & args]
    (case type
      :flight (dosync (apply reserve-flight args))
      :room   (dosync (apply reserve-room   args))
      :car    (dosync (apply reserve-car    args)))))
```

**Listing 2.** An adapted version of the Vacation benchmark, with secondary worker actors.

The code of the example is shown in Listing 1. Its input consists of a number of customers, who want to reserve two flights, a hotel room, and a car (line 4). These items are stored in transactional memory, so that multiple customers can reserve them in parallel (lines 1–3). A reservation consists of looking for these items, reserving them, and updating the reservation (lines 12–20).

In the original Vacation benchmark, there are a configurable number of worker actors, over which the customers are evenly distributed. However, we believe better performance may be achieved by processing the items that form a reservation in separate actors. (We will verify this in Section 7.) Hence, we create a variation of Vacation, in which the workers send the reservations of the individual items to one of a configurable number of 'secondary' worker actors.

The code then looks as shown in Listing 2. However, using traditional actors and STM, this code does not work as expected! In the transaction, four messages are sent, and afterwards the transactional variable c is updated. If another thread updates the same variable, this causes a conflict, and the transaction will be aborted. When the transaction retries, the messages are sent again. The problem is that when a transaction aborts, the messages it sends are not rolled back. As a result, in our example multiple items can be reserved for the same customer.

We observe that communicating with actors in a transaction breaks its serializability: the result of a parallel execution is no longer equal to the result of a serial execution.

### 3.3 Problem Statement

We see that combining a message-passing model and a shared-memory model can be useful, and occurs in practice. However, combining actors and transactions leads to issues:

- When memory is shared in an actor system, the isolated turn principle is broken. Consequently, race conditions that were prevented by the actor model can re-appear.
- When sending messages in a transaction, the serializability is broken. Sending messages is a side effect that is not rolled back when the transaction is.

These issues complicate combining both concurrency models in a single application, as the guarantees that developers expect are no longer true. These problems hinder composability: when one model is most suitable for one component of the application, and another model fits another component, the developer cannot safely use both. They also hinder re-usability: including a library that uses one model in an application that uses another may lead to incorrect results.

The ideal solution is a combination of actors and transactional memory that maintains the guarantees of both models. Then, developers can freely mix concepts of both models with a predictable outcome.

## 4 Solution: Transactional Actors

The problems described in the previous section occur because the semantics of the combination of two concurrency models is not well-defined. In this section, we solve this by creating transactional actors. Transactional actors provide the same operations as the actor model and STM, described in Section 2, but also define their semantics when they are combined.

There are two operations which contain nested code: `behavior` and `dosync`. Hence, we study the following combinations:

Transaction in Actor

`behavior` containing:

- `dosync` ①
- `ref` ②
- `deref` ②
- `ref-set` ②

Actor in Transaction

`dosync` containing:

- `behavior` ③
- `spawn` ④
- `send` ④
- `become` ④

We distinguish four categories:

① Using `dosync` in `behavior` executes a transaction in an actor. This transaction is bound to the current actor.

② Manipulating transactional state using `ref`, `deref`, or `ref-set` follows the regular semantics of transactions: these actions are only allowed in a transaction, and operate within its context. No special semantics are needed when this occurs in an actor.

③ A behavior can be defined in a transaction. The behavior is separated from the transaction in which it is defined: when the code in this behavior will run, at a later time, it no longer has access to the transaction in which the behavior was defined. Hence, a behavior can be defined in a transaction as it can anywhere else, and no special semantics is needed.

④ `spawn`, `send`, and `become` have a side effect. Therefore, when they occur in a transaction, their effect should become part of the transaction.

We now discuss these four cases in more detail.

### 4.1 Using Transactional Memory in an Actor

A transaction can run in an actor, inside `dosync` ①. Similar to a thread-based transactional system, where each thread has at most one active transaction, here, each actor has at most one transaction active at a time. Therefore, transactions run in actors as they do in thread-based systems.

Manipulating transactional state ② works within the context of the transaction that is active in the current actor. This is again as in a thread-based system.

This occurs in Listing 1, when the function `process-customer` (lines 26–34) is called in an actor and executes a transaction.

### 4.2 Communicating with Actors in a Transaction

While defining a behavior in a transaction ③ might seem strange, it does not pose any particular difficulties, as it is an idempotent operation. Defining a behavior is similar to defining a function: the code it contains is not executed when it is defined, but at a later time, in a new actor. It does not have access to its encapsulating transaction, instead, any transactional operations it contains run within the context of the transaction that is active when its code is executed. A behavior can refer to variables in its lexical scope though, it is effectively a closure.

Spawning an actor, sending a message, or becoming a new behavior ④ are actions with a side effect. Using these in a transaction requires special semantics, so that they can roll back. Transactional actors make these operations part of the transaction.

Regular transactional systems use two techniques to incorporate side effects into a transaction. The first technique is to *delay* the side effect until it is certain the transaction will commit successfully. For example, an update to a transactional variable is only visible locally at first, the global update is delayed until the transaction commits. In case the transaction aborts, the effect is discarded. The second technique is to perform the side effect immediately, but *roll back* the effect if the transaction aborts, using a compensating action. For example, in transactional systems with open nesting [16], one transaction can be nested in another, and commit separately. If the outer transaction aborts, the inner transaction needs to roll back, which relies on the developer specifying a compensating action.

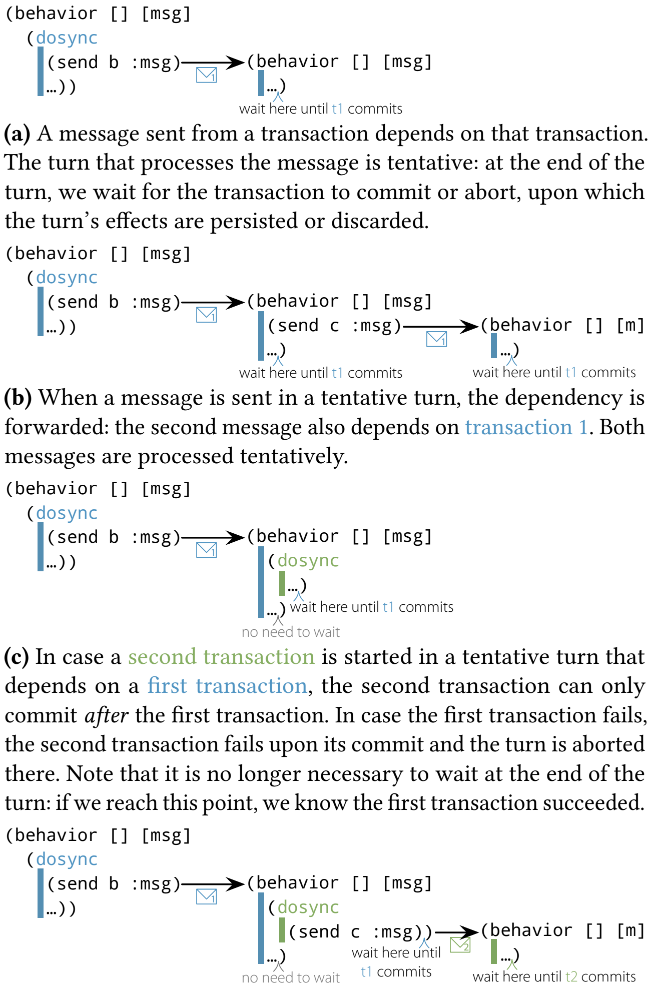
We handle the side effect of each actor operation with the appropriate technique:

**spawn** In our system, spawning a new actor in a transaction is delayed until the transaction commits. Spawning an actor is a costly operation: memory is allocated for the new actor and its inbox, a thread is created, and the actor's execution then starts on that thread. Hence, doing this immediately and rolling back if the transaction aborts is not a good idea: each attempt of the transaction would incur this cost. Delaying the operation until the transaction commits ensures this cost is only paid once.

**become** Become is inherently delayed: its effect only takes place upon the start of a new turn. As a transaction cannot span multiple turns, it will always be committed before the effect of `become` becomes visible. Hence, it does not matter whether we delay or roll back become: both have the exact same cost and result.

**send** In our system, we choose to immediately send out messages, and to roll back their effects if the transaction aborts. Sending a message is not expensive: it only consists of putting a message in the receiver's inbox (although this requires taking a lock). Immediately sending the message increases parallelism though: the receiver can already process the message before the sender's transaction has completed.

However, this implies that a message can now be retracted: when the transaction it was sent in aborts, the message and its effects need to be rolled back. We say that messages sent in a transaction have a *dependency* on the transaction. There are now two types of messages: those sent outside a transaction have no dependency and are *definitive*, those sent in a transaction have a dependency and are *tentative*.



**(a)** A message sent from a transaction depends on that transaction. The turn that processes the message is tentative: at the end of the turn, we wait for the transaction to commit or abort, upon which the turn's effects are persisted or discarded.

**(b)** When a message is sent in a tentative turn, the dependency is forwarded: the second message also depends on transaction 1. Both messages are processed tentatively.

**(c)** In case a second transaction is started in a tentative turn that depends on a first transaction, the second transaction can only commit *after* the first transaction. In case the first transaction fails, the second transaction fails upon its commit and the turn is aborted there. Note that it is no longer necessary to wait at the end of the turn: if we reach this point, we know the first transaction succeeded.

**(d)** When a message is sent in a transaction in a tentative turn, the message depends on this transaction, and not the encompassing turn. The third actor will only proceed when transaction 2 commits, which can only happen when transaction 1 committed as well.

**Figure 1.** Different cases of messages sent in transactions, and their dependencies. Each behavior runs in a different actor. The blue and green lines indicate tentative sections of code: either a transaction (in a dosync block), or a turn that depends on a transaction. A message's dependency is indicated through its color and number (1 or 2).

**Table 1.** How the operations on actors and transactions are executed in the three different contexts. Operations in gray work as before, blue indicates the side effect is delayed until the end of the tentative section (turn or transaction), and green indicates the side effect is performed immediately but rolled back on conflict.

| | Not in a transaction | | In a transaction |
|---|---|---|---|
| | Definitive turn | Tentative turn | |
| behavior | As before | | |
| become | Delayed until end of turn | | |
| spawn | Immediate | Delayed | Delayed |
| send | Immediate | Immediate, dependency forwarded | Immediate, dependency on transaction |
| dosync | Immediate | Immediate, but wait before commit | Immediate with closed nesting |
| ref, deref, ref-set | Not allowed | Not allowed | As before |

This has an impact on the receiver of a tentative message. When an actor takes a tentative message from its inbox, the turn that processes it also becomes tentative: the message is processed, but the effects it causes should not be persisted yet. Even though this turn is not a transaction, it executes in the same 'tentative' manner, as its effects can roll back. When a tentative turn ends, the actor waits until the transaction on which it depends has committed. After a successful commit of its dependency, the actor can continue to its next turn, and we say the turn was successful. After an abortion of its dependency, the tentative turn failed and its effects are discarded. The actor processes the next message in its inbox, as if nothing happened. This is illustrated in Figure 1a.

Now that turns can be tentative and may roll back just like transactions, we need to look at which actions with a side effect can occur in this context, as they can roll back too. Firstly, let us look at the operations on actors. When become and spawn are used in a tentative turn, their effects are handled as above when they appeared in a transaction: their effect is delayed until the turn is successful, or discarded if it fails. On the other hand, send in a tentative turn immediately sends a message, but forwards the current dependency with it, so that its receiver will also depend on the original transaction (Figure 1b).

Secondly, a tentative turn may contain another transaction. In other words, a first actor is executing a first transaction, in which it sends a message to a second actor, and when the second actor processes this message, it starts a second transaction. We say the second transaction depends on the first. There are two serializations of these two transactions: either the first transaction commits before the second, or vice versa. However, because the second transaction is executed as a result of the first, the only valid serialization is the one in which the second transaction is preceded by the first. Therefore, the second transaction needs to wait before it commits, until the transaction it depends on has committed. This is demonstrated in Figures 1c and 1d.

The different cases described in this section are summarized in Table 1. There are three execution contexts: (1) in a transaction, (2) out a transaction but in a tentative turn, and (3) out a transaction in a turn that is not tentative (we call these turns definitive).

## 5 Properties

Transactional actors provide several properties:

**Serializability** Transactional actors maintain the serializability of the transactions. In a naïve combination of transactions and actors, operations on actors inside a transaction cause side effects, breaking serializability (as in the example in Section 3.2). Transactional actors incorporate these operations into the transaction, so that they succeed or fail depending on whether the transaction commits or aborts, as explained in Section 4.2. For other actors and transactions, all effects inside a transaction thus appear to take place at the moment the transaction commits.

Effects of a (second) transaction that depends on a first transaction, will only occur if and when the dependency (first transaction) succeeded *and* the second transaction has no conflicts. These effects also occur in a single atomic step, and this always happens after the dependency committed. Hence, serializability is maintained, in a serialization in which the transaction with a dependency succeeds its dependency.

Effects of a tentative turn that does not contain a transaction (these are the actors it spawned and its become operation) also only occur after the dependency succeeded. Serializability is thus maintained: the order in which effects appear to other actors and transactions is equivalent to a serial execution.

**Freedom from races and deadlocks** Transactional actors break the isolated turn principle. When a turn contains a transaction followed by other code, the effects of the transaction already become visible before the turn has finished. Hence, the turn is no longer 'isolated'. This is a consequence of the fact that transactional actors introduce shared state that can be accessed synchronously from all actors, breaking one of the assumptions of the isolated turn principle (Section 2.1). However, the isolated turn principle had two consequences which are still upheld: freedom from data races and freedom from deadlocks.

Even though the isolated turn principle no longer holds, transactional actors still guarantee **freedom from low-level data races**, as all accesses to shared memory need to be encapsulated in a transaction. While the actor model guaranteed freedom from low-level data races by prohibiting shared memory, transactional actors allow shared memory but require it to be protected using transactions.

Transactional actors also maintain **freedom from deadlocks**. To substantiate this, we need to look at the operations that can block, and show that they cannot lead to a deadlock. We introduced blocking in two instances: at the end of a tentative turn, and at the end of a transaction in a tentative turn. In both cases, the current thread is blocked until the transaction on which it depends commits or aborts. Could this lead to a deadlock? In other words, could it happen that a transaction *A* is waiting for another transaction *B* to finish, and vice versa—a circular dependency between transactions?

Transactional actors prevent deadlocks. Dependencies can never be circular: a transaction or tentative turn can only depend on a transaction that started *before* it started. A dependency can only be introduced at the start of a turn: when a message is received with a dependency on *A*, a dependency from *B* upon *A* is introduced at the *start* of turn *B*. This message was sent from within transaction *A*, which was therefore necessarily already running before the message was received. The inverse dependency from *A* upon *B* is impossible: *A* started without a dependency on *B* because *B* did not exist when it started, and it cannot acquire such a dependency while it is running. This entails that dependencies always point to older transactions, and that time defines an order on dependencies. No circular dependencies can exist, so no deadlocks can occur.

In conclusion, transactional actors maintain the serializability of transactions, and replace the isolated turn principle of actors with freedom from low-level data races and from deadlocks. Using transactional memory, developers can reason about their code at the level of transactions; using the actor model, they can reason at the level of turns. Transactional actors enable developers to reason at the level of transactions *and* turns.

## 6 Implementation

We implemented transactional actors as an extension of Clojure, a Lisp dialect built on top of the Java Virtual Machine.[1] Clojure has support for software transactional memory built in; its implementation is described by Halloway [8, p. 182]. In this section, we first describe how we extended Clojure to support (regular) actors, and next which modifications were made to Clojure's STM and the actor system to support transactional actors.

### 6.1 Actors

We briefly sketch how we extended Clojure to support regular (non-transactional) actors. It is a rather simple implementation, meant to demonstrate the semantics of the model.

We first describe how behaviors are represented. In Section 2.1, a behavior was defined as a piece of code that is parameterized over two types of parameters: the internal state of the actor, and the values of a message (remember that a message is a list of values); e.g. `(behavior [flights] [orig dest n] e)`. Internally, this is translated into nested functions, e.g. `(fn [flights] (fn [orig dest n] e))`. Thus, to execute this code, we first apply the internal state of the actor to it, and next the received message.

Next, we implemented an `Actor` class, containing three fields: its current behavior, its current internal state (list of values), and its inbox (list of messages). The operations on actors are implemented as follows:

- `(become b v...)` changes the behavior and state of the current actor.

[1] Our implementation is available at https://github.com/jswalens/transactional-actors.

- `(send a v...)` enqueues a message consisting of the values `v...` in the inbox of actor `a`.
- `(spawn b v...)` creates a new actor with initial behavior `b` and `v...` as state. Then, a new thread is created in which the actor executes. This consists of an infinite loop that takes a message from its inbox, and calls its current behavior (a function nested in a function) with its state first, and the message's values next.

### 6.2 Modifications Made to Clojure's STM and the Actor System to Support Transactional Actors

We made several changes to these STM and actor systems to support transactional actors:

- Messages can have a dependency. When sending a message in a transaction, the transaction is added as a dependency. In a tentative turn but outside a transaction, the dependency of the turn is used. In a definitive turn the dependency remains null.
- At the start of a turn, if the message has a dependency the turn is marked as tentative.
- In a transaction, the effects of `become` and `spawn` are not performed immediately, but stored in the transaction.
- In a tentative turn but outside a transaction, the effects of `become` and `spawn` are also not performed immediately; they are stored in the actor.
- At the end of a transaction in a tentative turn, it waits for the dependency to complete. If the dependency succeeds, the transaction commits and its delayed effects (spawned actors and new behavior) are performed. If the dependency aborts, the transaction *and the encompassing turn* are aborted.
- At the end of a tentative turn, the actor waits for the dependency to complete. If the dependency succeeds, the delayed effects (spawned actors and new behavior) are performed. If it aborts, the current turn is aborted and the actor proceeds to the next one (that processes the next message in the inbox).

Our implementation is a simple proof of concept, aimed to demonstrate the benefits of our approach. We can think of several optimizations, but have not implemented those (yet). For example, when a message with a dependency is taken from the inbox, we could check whether its dependency has finished already. If it finished and succeeded, the turn does not need to be tentative, and if it aborted, the message can be discarded immediately.

## 7 Evaluation: Vacation Benchmark

In this section we demonstrate the benefits of transactional actors using the Vacation application. We first describe this benchmark, both the original version as well as the one using transactional actors, and then study its performance.

Our Vacation application is inspired by the Vacation benchmark from the STAMP benchmark suite [14]. Its input consists of $c$ customers that want to book a holiday. At the start of the program, $r$ flights, $r$ hotel rooms, and $r$ cars are generated—we call these items—with a random price and random number of seats/beds (between 100 and 500). Each customer will reserve between one and five seats on two flights, a hotel room, and a car. For each of these four items, the customer will select a subset of $q$ random items, pick the cheapest with sufficient available seats, and book it. Additionally, for each customer a password is generated using a cryptographically secure hash. We ran the experiments with $c = 1000$, $r = 50$, and $q = 10$.[2]

Each item and each customer is encapsulated in a transactional variable. Customers are written to five times: four times to update their bill (one for each item), and once to store their password. Each reservation also writes to four items: two flights, a room, and a car.

In the original benchmark, there are $p$ worker actors. The $c$ customers are evenly distributed so that each worker actor processes $c/p$ customers. Each customer reserves four items and generates a password. This is encapsulated in a transaction, hence, there are $c$ transactions each writing to one distinct customer and four items. This is illustrated in Listing 1.[3] There will never be a conflict on the customer, as it is distinct for each reservation, but there can be conflicts on the items.

In the version of the benchmark that uses transactional actors, next to $p$ primary worker actors, there are $s$ secondary worker actors. As before, the $c$ customers are distributed evenly over the $p$ primary worker actors. However, now each customer's reservation sends four messages to randomly selected secondary worker actors, and then generates the customer's password (as in Listing 2). The secondary worker actors will look for and reserve an item of the requested type, in a transaction. Hence, in this application, there are $c$ transactions that write to one distinct customer each and send four messages, and $4c$ transactions that write to one customer and one item. In this version, there can be conflicts on the customers as well as the items.

We benchmark both versions, varying $p$ and $s$, and measure the total execution time. Each variation is repeated 15 times, of which we calculate the median and interquartile ranges. Next, we calculate the speed-up compared to the result for $p = 1$ and $s = 1$, for each experiment separately.

All experiments ran on a machine with four AMD Opteron 6376 processors, each containing 16 cores with a clock speed of 2.3 GHz, resulting in a total of 64 cores. The machine has 128 GB of memory. We implemented transactional actors as a

[2] $r = c \times 5/100$ ensures that there are at least as many available seats as requested. $q = 10$ is as in the original STAMP benchmark.

[3] Note that the code in Listing 1 has been modified for this paper. The full code of the benchmark can be found at https://github.com/jswalens/vacation2.

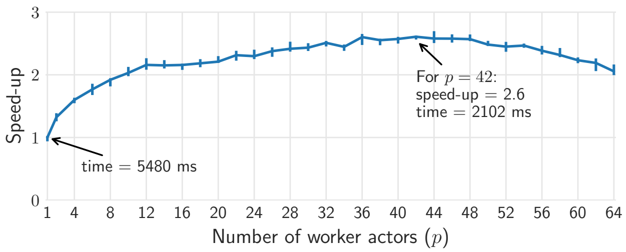


**Figure 2.** Speed-up of original version for an increasing number of worker actors ($p$). Each result is the median of 15 measurements; the error bars depict interquartile ranges.

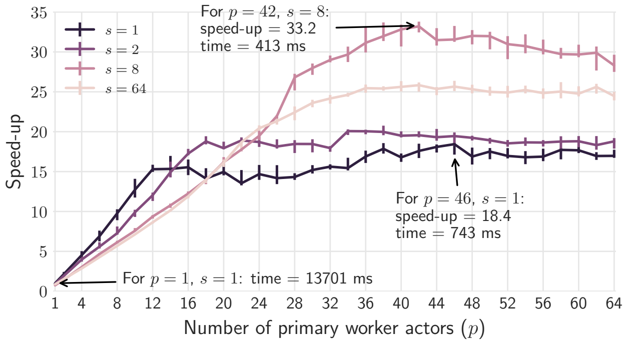


**Figure 3.** Speed-up of version using transactional actors for increasing numbers of primary ($p$) and secondary ($s$) worker actors. Full results are in the appendix.

fork of Clojure 1.8.0, running on the OpenJDK 64-Bit Server VM (build 25.131-b11) for Java 1.8.0.

The results of the original version are shown in Figure 2. Using a single worker actor, the program runs in 5480 ms. As the number of worker actors increases, the run time decreases, reaching a minimum of 2102 ms for 42 worker actors. This corresponds to a speed-up of 2.6. On a machine with 64 cores, this speed-up is very limited. This is a result of STM's optimistic concurrency: as the number of transactions that execute in parallel increases, the chance of conflicts and thus the number of retries increases.

Figure 3 shows a subset of the results for the version that uses transactional actors. (Full results are in the appendix.) In this benchmark, both the number of primary and secondary worker actors are varied. The version using one primary and one secondary worker actor is slower than in the original version, at 13701 ms. However, better performance is achieved when increasing the number of actors.

The black line in Figure 3 shows the results using only one secondary worker actor, so customers are processed in parallel but the reservation of individual items is not. Here, a minimal execution time of 743 ms is reached for 46 primary worker actors, a better result than the original version. This is because there are far fewer conflicts: there is only one secondary actor reserving items, so there can never be any conflicts on the items.

By increasing the number of secondary worker actors (the other lines in Figure 3), a higher speed-up can be achieved. We see that a maximum speed-up of 33.2 is reached for 42 primary and 8 secondary worker actors, on this machine. At this point, the balance between increased parallelism and a low chance of conflicts is optimal. Using more than 8 secondary worker actors will again lower the performance, due to a higher chance of conflicts. The optimal result for this version corresponds to a run time of 413 ms, compared to a minimum of 2102 ms for the original version. This indicates that this application benefits from being parallelized in two places, instead of only parallelizing the processing of customers (as in the original version) or only parallelizing the reservation of items (the results of $p = 1$ in the appendix), the optimum is found by combining both.

This experiment demonstrates another benefit of transactional actors: they allow a transaction to be split up into multiple transactions with dependencies. Every transaction in the original version was split into one primary and three dependent transactions. If the primary transaction fails, the three dependent transactions fail too. However, if a dependent transaction fails, this does not abort any other transaction. Using transactional actors, transactions can be split up, lowering the cost of a conflict in a dependent transaction, as only this part needs to retry.

Finally, we note the high overhead of transactional actors, when $p = 1$. Our implementation is a relatively simple prototype, and we believe further optimizations could improve the performance (some were suggested in Section 6.2).

Overall, this experiment shows that transactional actors can be used to increase the performance of an application that uses transactions. Distributing a transaction over multiple actors allows more fine-grained parallelism and lowers the chance and cost of conflicts. Further, we observe that this requires only limited code changes because the properties of the separate concurrency models are preserved.

## 8 Related Work

There is extensive related work on communication in transactions, or shared memory in actors.

**Communication in transactions** Our work on transactional actors is not the first to advocate communication between transactions. Using *Transactions with Isolation and Cooperation* [18], a transaction's atomicity and isolation can be temporarily suspended, to exchange data between transactions. While this allows communication between transactions, it breaks their serializability.

Similarly, Luchangco and Marathe [13] extend transactional memory with *transaction communicators*, a special type of object through which transactions can communicate. Access to a communicator should be encapsulated in a `txcommatomic` block, which should be nested in a regular transaction. Again, serializability is broken. This system also introduces dependencies between transactions: when a communicator is read by transaction *a* after it was written to by transaction *b*, transaction *a* depends on *b* to commit successfully. Circular dependencies are possible, and lead to a deadlock if both transactions are guaranteed to always abort, e.g. when they both write to the same transactional variable.

Finally, Lesani and Palsberg [12] introduce *communicating memory transactions*: a combination of transactional memory with channel-based message passing. In an `atomic` block, the constructs `send` and `receive` can be used to communicate over a channel. This introduces a dependency from the receiving transaction on the sending transaction, very similar to our transactional actors. In case of circular dependencies, all transactions in the cluster will *attempt* to commit at the same time, while maintaining serializability. If no valid serialization exists, the program is invalid. Communicating memory transactions thus maintain serializability, but disallow programs with circular dependencies.

Transactional actors borrow the idea of dependencies between transactions, and thus guarantee serializability. Deadlocks due to circular dependencies are avoided: there can never be a cycle in a dependency chain (see Section 5), as messages can only be sent in a transaction, and not received. Transactional actors extend the actor model of Agha et al. [2], which does not provide an explicit `receive` statement.

**Shared memory in actors** Our work is also not the first to consider how to safely share memory between actors. De Koster et al. [4] extend the actor model with *domains*, containers that can be accessed from multiple actors. Access must be encapsulated in a `when_shared` or `when_exclusive` block, the former gives shared read-only access while the latter gives exclusive write access. The code in these blocks is executed asynchronously to prevent deadlocks.

*Sharing actors* [11] also share state between a single writer actor and multiple reader actors, by replicating the data. Sharing actors encode the replication pattern discussed in Section 3.1. When the (single) writer updates the shared data, a message is sent to the readers to update their copies.

Morandi et al. [15] separate active processors (similar to actors) into their executing thread and their data. They introduce *passive processors*, which consist of only the data without the thread. Active processors can access this data by assuming the identity of a passive processor, giving them exclusive access to that processor's data.

All of these approaches introduce shared state in actor systems, but they only allow one writer per container at a time. To allow concurrent but safe access, it is important to split data correctly over these containers. For the Vacation example described in this paper, this is not evident: how should the flights be split, so that multiple customers can access all of them consistently at the same time? By using transactional memory, our approach allows multiple actors to write to shared memory, without requiring data to be split into containers: the transactional system ensures consistency.

**Transactional communication** *Transactors* [7] encapsulate changes to actors' local state and their communication in transactions. *Communicating transactions* [6] also coordinate distributed processes using transactions. Both use transactions to ensure that state that is distributed over multiple actors can be updated consistently, but do not share memory.

## 9 Conclusion

Many modern programming languages support a wide variety of concurrency models. This paper shows that, in practice, it can be beneficial for developers to mix these different concurrency models within a single application. Not only to structure the different parts of their application using the concurrency model that feels most natural for that part, but also to exploit some of the additional parallelism within those applications. Unfortunately, current language implementations often integrate these various concurrency models in an ad hoc way without clearly specifying the semantics of the combination.

This paper presents transactional actors, a combination of actors and software transactional memory with well-defined semantics. On the one hand the actor model can be used as a coordination mechanism between various components of an application. Within those actors, software transactional memory can be used to express safe access to shared state, which was otherwise impossible to express between these strictly isolated software entities. On the other hand, software transactional memory can be used to coordinate various tasks sent to different actors. Messages sent from within the same transaction either all succeed, or all fail.

To validate that additional parallelism can be exploited when combining actors and transactions, we presented an experimental evaluation using a representative benchmark. In future work, we plan to extend our evaluation with other benchmarks, include a comparison with related work, and optimize our implementation.

Transactional actors maintain the guarantees offered by the separate models: they maintain the serializability of the transactions, and guarantee freedom from low-level data races and deadlocks. Hence, developers can freely mix both concurrency models, increasing the composability and reusability of concurrent programs.

# A Appendix

Full results are in the table below:

Number of primary worker actors ($p$)

| Number of secondary worker actors ($s$) | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 | 32 | 34 | 36 | 38 | 40 | 42 | 44 | 46 | 48 | 50 | 52 | 54 | 56 | 58 | 60 | 62 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.0 | 2.2 | 4.5 | 6.9 | 9.8 | 12.7 | 15.3 | 15.3 | 15.5 | 14.1 | 15.0 | 13.5 | 14.7 | 14.2 | 14.3 | 15.2 | 15.6 | 15.4 | 16.9 | 17.9 | 16.8 | 17.6 | 18.1 | 18.4 | 16.9 | 17.5 | 17.0 | 16.8 | 16.9 | 17.7 | 17.7 | 17.0 | 17.0 |
| 2 | 0.9 | 1.9 | 4.0 | 5.6 | 7.3 | 9.9 | 12.0 | 14.9 | 17.3 | 18.8 | 17.9 | 18.9 | 18.7 | 18.1 | 18.5 | 18.5 | 17.9 | 20.1 | 20.1 | 20.0 | 19.5 | 19.6 | 19.3 | 19.5 | 19.2 | 18.9 | 18.5 | 18.7 | 18.6 | 18.8 | 18.8 | 18.3 | 18.8 |
| 4 | 0.8 | 1.6 | 3.4 | 5.0 | 6.9 | 8.4 | 10.2 | 12.0 | 14.0 | 15.8 | 18.3 | 20.1 | 21.3 | 22.7 | 23.7 | 23.9 | 28.1 | 28.9 | 27.7 | 26.7 | 27.4 | 27.7 | 27.4 | 26.7 | 26.6 | 26.5 | 25.9 | 25.4 | 26.0 | 26.0 | 25.1 | 25.2 | 24.6 |
| 6 | 0.8 | 1.5 | 3.2 | 4.8 | 6.6 | 7.9 | 9.3 | 11.4 | 13.0 | 14.4 | 16.1 | 18.1 | 20.1 | 22.3 | 24.2 | 29.1 | 30.3 | 31.1 | 31.7 | 31.7 | 31.4 | 32.0 | 30.7 | 30.4 | 30.4 | 30.6 | 29.1 | 29.6 | 28.8 | 28.8 | 28.2 | 28.7 | 28.0 |
| 8 | 0.7 | 1.5 | 3.0 | 4.7 | 6.1 | 7.6 | 9.4 | 10.8 | 12.2 | 14.0 | 16.2 | 17.8 | 19.5 | 22.0 | 26.8 | 28.0 | 29.0 | 29.7 | 31.1 | 32.0 | 32.8 | 33.2 | 31.5 | 31.6 | 32.0 | 32.0 | 31.0 | 30.7 | 30.2 | 29.7 | 29.7 | 29.9 | 28.3 |
| 10 | 0.8 | 1.5 | 3.0 | 4.6 | 6.0 | 7.6 | 9.3 | 10.5 | 12.1 | 13.5 | 15.3 | 17.3 | 18.8 | 23.0 | 25.1 | 27.9 | 29.1 | 31.3 | 31.5 | 32.3 | 32.0 | 31.8 | 32.2 | 31.5 | 31.2 | 31.5 | 31.3 | 30.0 | 30.2 | 30.6 | 30.8 | 29.5 | 29.4 |
| 12 | 0.8 | 1.4 | 2.9 | 4.5 | 6.1 | 7.5 | 8.8 | 10.3 | 11.8 | 13.7 | 15.4 | 16.6 | 19.2 | 21.1 | 25.5 | 27.4 | 29.2 | 30.0 | 30.8 | 31.4 | 31.6 | 31.8 | 32.0 | 29.9 | 29.8 | 30.6 | 30.5 | 30.1 | 29.2 | 30.2 | 28.6 | 28.8 | 29.5 |
| 14 | 0.8 | 1.4 | 2.9 | 4.4 | 5.9 | 7.4 | 8.8 | 10.4 | 11.7 | 13.1 | 14.7 | 16.8 | 18.6 | 22.0 | 23.7 | 26.9 | 28.9 | 31.1 | 31.2 | 31.0 | 30.8 | 30.8 | 30.5 | 30.4 | 30.2 | 30.9 | 30.1 | 28.8 | 29.1 | 29.5 | 28.5 | 28.7 | 28.5 |
| 16 | 0.8 | 1.4 | 2.9 | 4.4 | 5.8 | 7.2 | 8.9 | 10.2 | 11.8 | 13.2 | 15.2 | 16.9 | 19.4 | 21.0 | 25.2 | 28.2 | 29.4 | 29.9 | 30.2 | 30.7 | 30.1 | 30.5 | 30.1 | 30.2 | 30.2 | 29.8 | 29.7 | 28.8 | 28.1 | 28.3 | 26.8 | 28.1 | 26.8 |
| 18 | 0.9 | 1.4 | 2.9 | 4.3 | 5.8 | 7.3 | 8.7 | 10.3 | 11.7 | 13.5 | 14.9 | 17.0 | 18.7 | 21.0 | 24.4 | 26.9 | 29.4 | 29.5 | 30.2 | 30.5 | 30.3 | 30.0 | 30.6 | 30.8 | 29.4 | 29.1 | 28.5 | 28.8 | 28.7 | 28.0 | 27.7 | 27.0 | 27.8 |
| 20 | 0.9 | 1.4 | 2.8 | 4.3 | 5.7 | 7.2 | 8.6 | 10.3 | 11.9 | 13.3 | 15.2 | 16.8 | 18.9 | 21.0 | 25.7 | 27.3 | 28.6 | 29.2 | 29.4 | 30.0 | 29.9 | 29.7 | 29.1 | 29.2 | 29.6 | 29.3 | 28.3 | 27.9 | 27.5 | 28.1 | 26.7 | 27.0 | 26.8 |
| 22 | 0.9 | 1.5 | 2.9 | 4.3 | 5.7 | 7.1 | 8.4 | 10.4 | 11.8 | 12.9 | 15.0 | 17.1 | 19.4 | 21.6 | 25.2 | 27.1 | 28.3 | 28.9 | 28.9 | 28.6 | 28.9 | 28.9 | 29.8 | 29.4 | 28.5 | 28.5 | 28.2 | 28.0 | 28.2 | 27.0 | 27.1 | 26.8 | 27.3 |
| 24 | 0.9 | 1.5 | 2.9 | 4.3 | 5.7 | 7.0 | 8.9 | 10.2 | 11.8 | 13.5 | 15.0 | 17.5 | 20.4 | 23.3 | 25.1 | 26.7 | 27.8 | 28.0 | 28.6 | 28.7 | 28.7 | 29.3 | 29.2 | 27.9 | 27.6 | 28.0 | 28.2 | 28.7 | 27.0 | 27.6 | 26.6 | 25.9 | 26.1 |
| 26 | 0.9 | 1.5 | 2.8 | 4.2 | 5.7 | 7.3 | 8.8 | 10.3 | 11.8 | 13.4 | 15.2 | 17.3 | 19.8 | 22.8 | 24.5 | 26.5 | 28.1 | 28.5 | 28.5 | 28.4 | 28.3 | 29.0 | 28.5 | 27.8 | 28.1 | 28.3 | 27.8 | 27.9 | 26.9 | 27.4 | 26.8 | 26.3 | 27.3 |
| 28 | 0.9 | 1.5 | 2.8 | 4.3 | 5.8 | 7.3 | 8.7 | 10.3 | 11.8 | 13.3 | 15.3 | 16.9 | 20.0 | 21.9 | 25.1 | 26.4 | 27.5 | 28.4 | 28.2 | 28.1 | 28.1 | 28.2 | 27.9 | 27.5 | 26.7 | 28.2 | 27.9 | 27.1 | 27.4 | 26.1 | 26.5 | 26.2 | 25.6 |
| 30 | 0.9 | 1.5 | 2.8 | 4.3 | 5.9 | 7.3 | 8.7 | 10.2 | 11.9 | 13.3 | 15.3 | 17.5 | 20.0 | 22.5 | 25.1 | 26.7 | 27.0 | 27.6 | 28.1 | 28.3 | 27.9 | 28.1 | 28.0 | 28.3 | 28.1 | 28.5 | 27.4 | 27.1 | 28.3 | 27.2 | 26.2 | 26.5 | 25.6 |
| 32 | 0.9 | 1.5 | 2.9 | 4.3 | 5.8 | 7.3 | 8.8 | 10.3 | 11.7 | 13.6 | 15.4 | 17.9 | 20.0 | 23.0 | 25.7 | 26.5 | 26.7 | 27.3 | 28.2 | 27.5 | 28.1 | 27.3 | 27.4 | 28.1 | 27.8 | 27.4 | 28.0 | 27.0 | 26.6 | 26.0 | 26.9 | 26.0 | 25.8 |
| 34 | 0.9 | 1.5 | 2.9 | 4.3 | 5.8 | 7.2 | 8.8 | 10.3 | 11.7 | 13.5 | 15.5 | 17.8 | 19.9 | 23.4 | 24.9 | 26.1 | 27.2 | 27.1 | 27.2 | 28.2 | 27.6 | 27.8 | 27.9 | 27.1 | 27.3 | 28.0 | 26.5 | 27.5 | 26.1 | 26.6 | 26.7 | 26.0 | 25.3 |
| 36 | 0.9 | 1.5 | 2.9 | 4.3 | 5.8 | 7.3 | 8.7 | 10.3 | 11.8 | 14.0 | 16.1 | 17.8 | 21.5 | 24.0 | 25.1 | 26.0 | 26.5 | 26.6 | 26.9 | 27.2 | 27.0 | 27.2 | 26.8 | 26.6 | 26.8 | 26.8 | 26.8 | 25.9 | 26.6 | 26.3 | 25.5 | 25.2 | 25.4 |
| 38 | 0.9 | 1.5 | 2.9 | 4.3 | 5.8 | 7.2 | 8.7 | 10.3 | 11.9 | 13.5 | 15.7 | 18.0 | 21.9 | 23.6 | 25.3 | 25.5 | 26.2 | 26.5 | 26.7 | 27.0 | 27.7 | 27.0 | 27.3 | 26.9 | 26.7 | 26.4 | 26.9 | 26.6 | 26.2 | 26.1 | 26.2 | 26.2 | 25.8 |
| 40 | 0.9 | 1.5 | 2.9 | 4.3 | 5.7 | 7.2 | 8.7 | 10.1 | 12.0 | 13.9 | 15.8 | 18.6 | 22.0 | 23.3 | 24.4 | 25.5 | 25.9 | 27.1 | 26.5 | 27.0 | 26.8 | 26.8 | 26.9 | 26.6 | 26.5 | 27.1 | 26.1 | 26.3 | 25.5 | 26.9 | 25.4 | 25.5 | 25.7 |
| 42 | 0.9 | 1.5 | 2.9 | 4.3 | 5.7 | 7.2 | 8.7 | 10.1 | 11.9 | 13.7 | 16.4 | 18.5 | 20.4 | 23.3 | 24.7 | 24.7 | 26.1 | 26.4 | 26.8 | 27.1 | 27.2 | 26.8 | 26.8 | 26.6 | 26.3 | 26.5 | 26.0 | 26.1 | 25.8 | 25.7 | 25.8 | 25.1 | 25.5 |
| 44 | 0.9 | 1.5 | 2.8 | 4.3 | 5.7 | 7.2 | 8.7 | 10.2 | 11.9 | 13.6 | 16.3 | 18.1 | 20.4 | 22.0 | 22.8 | 23.6 | 24.3 | 26.0 | 26.5 | 26.8 | 26.2 | 26.6 | 27.1 | 27.3 | 27.1 | 26.6 | 25.7 | 26.4 | 25.9 | 24.9 | 25.2 | 25.6 | 26.1 |
| 46 | 0.9 | 1.5 | 2.9 | 4.3 | 5.8 | 7.2 | 8.7 | 10.1 | 11.8 | 13.3 | 15.7 | 17.7 | 19.6 | 21.9 | 22.9 | 23.4 | 23.5 | 24.6 | 25.8 | 25.8 | 27.0 | 26.5 | 26.3 | 26.6 | 26.3 | 26.4 | 25.9 | 25.3 | 26.0 | 26.0 | 25.5 | 25.6 | 25.5 |
| 48 | 0.8 | 1.6 | 2.9 | 4.3 | 5.7 | 7.2 | 8.6 | 10.1 | 11.5 | 13.6 | 15.6 | 18.0 | 20.8 | 21.6 | 23.0 | 23.3 | 24.2 | 24.6 | 24.7 | 25.8 | 25.9 | 25.9 | 26.1 | 26.1 | 25.8 | 26.4 | 26.2 | 26.5 | 26.0 | 26.0 | 25.3 | 25.8 | 25.3 |
| 50 | 0.9 | 1.6 | 2.9 | 4.3 | 5.8 | 7.2 | 8.6 | 10.2 | 11.6 | 13.7 | 15.7 | 18.3 | 19.8 | 21.8 | 22.7 | 23.5 | 23.9 | 24.5 | 24.8 | 25.4 | 26.7 | 26.0 | 26.1 | 26.1 | 26.4 | 26.1 | 26.1 | 26.0 | 25.7 | 25.4 | 25.8 | 24.9 | 24.9 |
| 52 | 0.9 | 1.5 | 2.9 | 4.3 | 5.7 | 7.1 | 8.6 | 10.1 | 11.7 | 13.6 | 15.9 | 18.1 | 20.2 | 21.9 | 22.6 | 23.3 | 24.0 | 24.1 | 25.7 | 25.7 | 25.7 | 25.8 | 25.7 | 26.1 | 26.1 | 25.6 | 26.0 | 25.6 | 25.9 | 25.8 | 25.2 | 24.9 | 25.3 |
| 54 | 0.9 | 1.6 | 2.9 | 4.3 | 5.7 | 7.1 | 8.6 | 10.1 | 11.7 | 13.9 | 15.8 | 18.1 | 20.4 | 21.3 | 22.5 | 22.9 | 24.1 | 25.2 | 25.9 | 25.7 | 25.6 | 25.6 | 26.1 | 26.3 | 25.9 | 26.0 | 25.5 | 25.9 | 25.5 | 25.3 | 25.8 | 25.1 | 24.9 |
| 56 | 0.9 | 1.5 | 2.9 | 4.3 | 5.7 | 7.1 | 8.6 | 10.2 | 11.8 | 13.7 | 16.3 | 18.2 | 20.6 | 21.7 | 22.4 | 23.0 | 23.3 | 25.0 | 25.4 | 25.7 | 25.9 | 25.6 | 26.1 | 25.5 | 25.8 | 25.8 | 26.3 | 25.9 | 25.4 | 25.3 | 25.3 | 25.0 | 24.2 |
| 58 | 0.9 | 1.6 | 2.8 | 4.3 | 5.7 | 7.3 | 8.7 | 10.1 | 12.0 | 13.6 | 15.9 | 17.9 | 19.7 | 21.6 | 22.5 | 22.9 | 23.0 | 24.1 | 25.1 | 25.8 | 25.5 | 25.5 | 26.1 | 25.7 | 25.2 | 25.9 | 25.4 | 25.3 | 25.7 | 25.2 | 25.4 | 25.3 | 24.9 |
| 60 | 0.9 | 1.5 | 2.9 | 4.3 | 5.7 | 7.1 | 8.7 | 10.3 | 12.0 | 13.6 | 16.1 | 18.4 | 19.9 | 21.3 | 22.2 | 23.2 | 23.9 | 24.7 | 25.1 | 25.8 | 25.7 | 25.8 | 25.5 | 25.6 | 26.4 | 25.4 | 25.2 | 25.5 | 25.3 | 25.1 | 24.8 | 24.9 | 25.1 |
| 62 | 0.9 | 1.6 | 2.9 | 4.3 | 5.7 | 7.2 | 8.6 | 10.1 | 11.8 | 13.8 | 16.4 | 18.2 | 19.9 | 21.2 | 22.2 | 23.8 | 24.0 | 24.9 | 25.5 | 25.3 | 25.4 | 25.5 | 25.8 | 25.9 | 25.9 | 26.1 | 25.1 | 25.1 | 25.8 | 25.3 | 24.8 | 24.8 | 24.7 |
| 64 | 0.9 | 1.5 | 2.9 | 4.3 | 5.7 | 7.1 | 8.6 | 10.1 | 11.9 | 14.0 | 16.1 | 18.7 | 20.4 | 21.3 | 22.4 | 23.6 | 24.2 | 24.6 | 25.5 | 25.4 | 25.6 | 25.8 | 25.4 | 25.7 | 25.3 | 25.0 | 24.9 | 25.2 | 24.8 | 25.0 | 24.8 | 25.6 | 24.5 |

# References

[1] G. A. Agha. 1985. *Actors: a model of concurrent computation in distributed systems.* Ph.D. Dissertation. MIT.

[2] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. 1997. A Foundation for Actor Computation. *Journal of Functional Programming* 7, 1, 1–72.

[3] P. A. Bernstein and N. Goodman. 1981. Concurrency Control in Distributed Database Systems. *Comput. Surveys* 13, 2, 185–221.

[4] J. De Koster, S. Marr, T. Van Cutsem, and T. D’Hondt. 2016. Domains: Sharing state in the communicating event-loop actor model. *Computer Languages, Systems & Structures* 45, 132–160.

[5] J. De Koster, T. Van Cutsem, and W. De Meuter. 2016. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *AGERE! 2016.* 31–40.

[6] E. de Vries, V. Koutavas, and M. Hennessy. 2010. Communicating Transactions. In *CONCUR’10.* 569–583.

[7] J. Field and C. A. Varela. 2005. Transactors: A Programming Model for Maintaining Globally Consistent Distributed State in Unreliable Environments. In *POPL’05.* 195–208.

[8] S. Halloway. 2009. *Programming Clojure* (1st ed.). Pragmatic Bookshelf.

[9] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. 2005. Composable Memory Transactions. In *PPoPP’05.* 48–60.

[10] M. Herlihy and J. E. B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. *ACM SIGARCH Computer Architecture News* 21, 289–300.

[11] M. Lesani and A. Lain. 2013. Semantics-preserving Sharing Actors. In *AGERE! 2013.* 69–80.

[12] M. Lesani and J. Palsberg. 2011. Communicating Memory Transactions. In *PPoPP’11.* 157–168.

[13] V. Luchangco and V. J. Marathe. 2011. Transaction Communicators: Enabling Cooperation Among Concurrent Transactions. In *PPoPP’11.* 169–178.

[14] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *IISWC’08.* 35–46.

[15] B. Morandi, S. Nanz, and B. Meyer. 2014. Safe and Efficient Data Sharing for Message-Passing Concurrency. In *COORDINATION’14.* 99–114.

[16] Y. Ni, V. S. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. 2007. Open Nesting in Software Transactional Memory. In *PPoPP’07.* 68–78.

[17] N. Shavit and D. Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2, 99–116.

[18] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. 2007. Transactions with Isolation and Cooperation. In *OOPSLA’07.* 191–210.

[19] J. Swalens, J. De Koster, and W. De Meuter. 2016. Transactional Tasks: Parallelism in Software Transactions. In *ECOOP’16.* 23:1–23:28.

[20] S. Tasharofi, P. Dinges, and R. E. Johnson. 2013. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?. In *ECOOP’13.* 302–326.

[21] P. Van Roy and S. Haridi. 2004. *Concepts, techniques, and models of computer programming.* The MIT Press.