Compas'2022 : Parallélisme/ Architecture/ Système
MIS/UPJV - Amiens France, 5-8 juillet 2022

Shared memory for the actor model

Benoit Martin `benoit.martin@lip6.fr`

Marc Shapiro `marc.shapiro@acm.org`

Sorbonne Université, CNRS, INRIA, LIP6. Paris, France

Résumé
Modern distributed applications are difficult to develop. The actor model is helpful as it is designed for modularity and composition. However, the actor model does not support shared state. For this, developers often use an external database, which integrates poorly and causes anomalies. We argue that some form of native shared memory is useful for a large class of distributed applications. To integrate cleanly with the actor environment, this shared memory should support the transactional causal consistency model. This model provides the strongest consistency compatible with high availability and is compatible with the actor model and its common optimizations.

Mots-clés : actor model, shared memory, transactional causal consistency

1. Introduction

Distributed applications are difficult to build and programmers often find themselves having to write and rewrite the same complex functional components, such as replication mechanisms, deployment strategies or object serialization methods. These repetitive, tedious and error prone tasks are often solved by using boilerplate code that do not integrate well together. For instance, containers that are deployed on the same node pay for the cost of data serialization when a shared memory may be used instead.
Certain properties such as modularity, scalability and performance are requirements that developers want to build a distributed system. These same properties motivate the use of microservices and serverless applications but come at the expanse of performance, expressiveness and overall control as programmers lose the ability to express fine-grained behaviors.
In the actor model [7], actors communicate using asynchronous message-passing. This model is designed for modularity and scalability. Furthermore, the actor model allows fine-grain control of individual actors. Thus making it possible to build complex distributed applications in a very flexible way.
However, by design, the actor's state is isolated from other actors. In practice however, distributed applications require that some form of state is shared between its distributed components. It is common for developers to combine multiple memory models in a single program. In a sample of 15 GitHub projects in Scala that use the actor model, 12 combined it with another model [12]. Developers justify this mix by the weaknesses in the actor library implementations and limitations of the actor model. Furthermore, if an actor communicates both by messages and through a third-party shared memory, there is no guarantee that the two views are mutually consistent.

We argue that the lack of shared memory is limiting for modern distributed applications, and that an integrated, consistent sharing mechanism will be beneficial. Our work aims to allow the use of shared states between actors by providing a natively available shared memory that is causally consistent with inter-actor messages. We believe that this approach will ease how distributed application are expressed. For example, a collaborative text editor, such as Google Docs or Overleaf, where participating editors can share their mouse cursor position and chat alongside their edits, will benefit from causally consistent shared memory between its distributed instances. The shared document corresponds to shared memory while the mouse position and the chat messages are sent through messages. Unified causal consistency between shared memory and messages guarantees the happened-before relation between the two memory models.
In a first section we will present the background to cleanly integrate a consistency mechanism into the actor model. Then, in a second section, we describe how a unified consistency model is implemented into the actor model. Finally, an online collaborative text editor application is studied.

2. Background

2.1. Actor model

The actor model [7] defines an actor as the universal primitive of concurrent computation. An actor responds to messages it receives by making a local decision, creating other actors, sending messages and determining how to respond to the next message. An actor may modify its own private state, but can only affect an other indirectly through messages. Modern implementations of the actor model follow the Reactive Manifesto [5] (responsive, resilient, elastic and message driven) by using asynchronous message-passing to send messages between actors.
Actors do not share their state. Message-passing is the only mechanism that an actor can use to interact with other actors. Shared memory between actors is thus considered incompatible with the actor model, as it would violate isolation and thus cause inconsistencies.
A data-flow graph is formed when actors (nodes) send messages to other actors through message-passing (edges). The resulting graph may be dynamic as actors spawn and terminate, and cyclic as actors communicate with any actor.
The actor model encourages code reuse, leveraging modularity without sacrificing fine-grain control for the application developer. Any application logic can be expressed within an actor.
The actor model features the following properties:

— an actor encapsulates application logic
— actors communicate using asynchronous message-passing
— an actor is identified by a unique address, which is location transparent, enabling its deployment on any node
— an actor's state can only be modified through message-passing. Its state is isolated.

2.2. Glitches

A common approach is to string actors in a data flow to handle data stream updates and the propagation of change. As updates are asynchronous, the values within such a system might get out of sync, resulting in the application accessing obsolete data. Such a temporary inconsistency in an observable state is called a glitch [6].
For instance, consider the following dataflow, where each actor sends the variables that it updates to the actors that use it:

```
actor1 = { every 3600 seconds do {tick = tick+1} }
actor2 = { while true do {hour = tick % 24 } }
actor3 = { while true do {day = (tick/24) % 7 +1 } }
actor4 = { while true do {show (hour, day)} }
```

Computing *hour* and *day* use the latest value of *tick*. Unfortunately, depending on the order of evaluation, the values of *hour* and *day* may not reflect the same value of *tick*. This may result in a glitch, an inconsistency between *hour* and *day*; for instance, just after midnight, *day* might indicate the new day while *hour* is still at 23:59:59.
Glitches result in incorrect program state and wasteful recomputations. Note that FIFO channels are not sufficient to endure glitch-freedom. What is needed is to ensure that every expression is evaluated after those it depends upon, i.e., *causal ordering*. To solve this problem, current practice eliminates glitches by arranging actors in a topologically sorted graph [6, 9], which does ensure causal order if the message graph is a DAG.
Glitches are a result of a missing global consistency model. The lack of shared memory does not avoid inconsistencies. Of course, inconsistencies would only be worse if using an external database.

3. Unified causal consistency for actors

In this section, we propose a model that solves the glitch problem, while extending its expressiveness with shared memory, without breaking the fundamental benefits of the actor model.
Let us augment an actor system with shared memory; its events are sending and receiving messages, and reads and updates to the shared memory. We consider that updating some object in the shared memory creates a new version of that object.
Informally, a system is causally consistent if events are observed in an order consistent with the order in which they happened, without any causal gaps (ie: all causally related dependencies of an event is satisfied).
Formally, we say that event e happened-before f, noted $e \rightarrow f$ if: (1) some actor executes e followed by f; (2) e is the send of a message and f is the reception of the same message; (3) e is an update to the shared memory and f is a read of the corresponding version; (4) there exists an event g such that $e \rightarrow g \wedge g \rightarrow f$. The system is *causally consistent* if: $e \rightarrow f$, and some actor observes both e and f, then it observes e before observing f; and if there exists updates $x1$ and $x2$ to some object x, such that $x1 \rightarrow x2 \rightarrow e$ for some event e, then an actor that both observes e and reads x must observe version $x2$.
In addition to causal consistency (safety), let us require convergence, the liveness property that the views of the shared memory observed by different actors (called replicas) eventually converge [11]. Finally, we allow an actor to group events into *transactions*, such that a transaction observes a causally-consistent snapshot, and its message sends and memory updates occur atomically (i.e., all or nothing).
Let us call this model *Actor Transactional Causal Consistency* (ATCC). We claim that ATCC remains compatible with the actor vision. Indeed, thanks to the snapshot property, each actor remains encapsulated and sequential. ATCC is asynchronous and non-blocking, as it does not require locking or consensus between actors. Furthermore, ATCC avoids glitches, and ensures that messages and memory are mutually consistent (in contrast to the use of an external data store).

4. Shared memory implementation

We implement our shared memory model on top of the Akka actor framework.[1] Akka is open source and has a limited but working *DistributedData* extension [1] that enables actors to share data with eventual consistency guarantees. An actor accesses data in the shared store through a replicator actor that provides a key-value store API and that handles data replication (See Figure 1). The replicator actor spreads object updates to its neighbours via direct replication and gossip-based dissemination.
A key is the unique identifier of the underlying data value. Values are CRDTs. A CRDT supports updates from any node without coordination. Concurrent updates to the same value eventually converge using CRDT logic [11]. CRDTs provide high read and write availability, with low latency.
Akka's *DistributedData* extension enables the sharing of data between actors but is limited to a weak consistency model. Furthermore, transactions are not supported.
We extend *DistributedData* to implement ATCC without breaking any of the native properties of the actor model. This allows actors to share states with strong guarantees by ensuring that all events are causally ordered end-to-end. Furthermore, snapshots ensure that actors remain mutually isolated, and improve concurrency of reads with updates. Our modifications to Akka include additional protocols for the replicator actor, and additional message metadata that is required to guarantee transitive causal delivery of messages.
The following code snippet illustrates the user API that exposes transactions to an Akka actor in Scala.

```
val flagKey = FlagKey("flag")
var flag = Flag()
val counterKey = PNCounterKey("counter")
var counter = PNCounter()

val t = new Transaction((context) -> {
    flag = flag.switchOn                // toggle flag on
    context.update(flagKey, flag)       // update flag on replicator
    counter = context.get(counterKey)   // retrieve counter value
})
t.commit()
```

In this example, a transaction starts by instantiating a *Transaction* object and by defining an anonymous function that represents the transaction's content. The *context* argument encapsulates the causal context and is accessible within the transaction body to interact, in a causally correct way, with the replicator actor using the *update* and *get* methods. The replicator actor guarantees the correctness of the data by implementing ATCC.

4.1. Causal consistency

Causal consistency captures the causal relationships between operations which allows components in a distributed system to locally agree on the order of the causally-related operations. They may disagree on the order of operations that are causally unrelated.
Akka ensures, by default, that messages between two actors are FIFO. However, this property is not transitive. Messages between two actors may be received in a non-causal order if they

1. https://akka.io

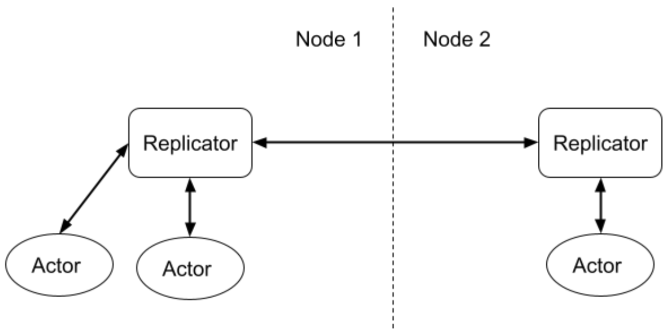

Figure 1 – Akka Replicator

transit through a third actor. Thus, to guarantee transitive causal delivery of messages, the use of additional information (metadata) helps maintain causality between actors (see section 4.1.1).
Furthermore, an replication protocol is added to handle causal delivery of replicated objects. On a successful transaction commit, changes are broadcasted to all replicator actors [4].
ATCC requires that mergeable data types are used. Fortunately, Akka offers several CRDT types that can be shared between actors using the replicator actor: GCounter, PNCounter, GSet, ORSet, ORMap, ORMultiMap, LWWMap, PNCounterMap, LWWRegister and Flag.

4.1.1. Metadata

The network does not natively provide causal guarantees, even if all channels are FIFO. This requires piggy-backing extra information, called *metadata*, to be able to check if an event is received out of order, and if so, to delay its delivery.
Replication messages between replicator actors are modified to include a version vector [10] with one entry per replicator. This extra information is necessary to implement causal consistency with multi-version snapshots.
A new *CausalDelivery* trait is available for Akka developers to use. This enables the runtime to detect which messages are delivered in causal order with respect to other messages and transactional shared memory. In addition, the delivery of unordered messages are delayed using a custom causal mailbox.

4.1.2. Multi-version snapshots

In order to provide each transaction with an appropriate snapshot, our system maintains multiple versions of the same data object. This mechanism is called multi-version snapshot isolation.
We modified Akka's replicator actor to maintain a causally consistent and transactional view of the data. When a transaction starts, the system assigns it the most recent available causally-consistent snapshot. For this, the replicator maintains a mapping between version vectors and the corresponding snapshots. This structure of the data permits fast creation of a snapshot and rapid pruning of unused snapshots.
Updating an object in the store causes the replicator actor to send a notification message to any subscribed actors. The actor's custom causal mailbox uses this information to causally delivery pending messages. Naturally, this message is also delivered in a causally-consistent way.

4.2. Transactions using One-Phase commit

Transaction requires the use of a commitment protocol between the actor that starts the transaction and the local replicator that stores and replicates the data. One-phase commit [3] (1PC) is a protocol that enables atomic transaction. The replicator actor receives *commit* and *abort* messages that respectively apply and discard transaction operations.

The replicator actor requires that a transaction is uniquely identifiable. This is necessary to define the causal context and the transaction's operations. To achieve this, each new transaction generates a unique transaction identifier by generating a Universally Unique Identifier (UUID) [8] [2].

Furthermore, to maintain atomicity, transaction operations are temporarily stored in an auxiliary in-memory data structure. On commit, the transaction is identified by it's UUID which helps select a causally correct snapshot version. Transaction operations move from their temporary store to a permanent store with a corresponding commit version vector (see Appendix, Figure 3). After a successful commit, updated objects are replicated using broadcast. If an error occurs during commit, the failing transaction stops and the result of it's operations roll back. A commit success or failure message is sent to the initiating actor.

The abort phase discards any uncommitted operations that are pending for the given transaction id.

5. Usecase: online collaborative text editor

A collaborative online text editor allows users to concurrently edit a shared text document. Additionally, some editors such as Google Docs or Overleaf provide additional features such as viewing a collaborator's mouse cursor or a chat. The document content is a shared memory between users, while mouse positions and chat messages are not persistent and are thus sent using messages. Without a unified consistency model between shared memory and messages, inconsistencies happen. It is common to see a text selection that is incoherent with the actual text. For instance, Alice and Bob are both working on a shared document. Alice, after having made edits to the document, highlights (using her mouse cursor) a paragraph then sends a chat message to Bob asking him to check the content of her highlighted text. Bob, receives Alice's chat message and scrolls to her mouse position. Unfortunately, he does not understand why Alice's highlighted text is showing him an old paragraph that Alice did not even write !

ATCC implements a unified causal consistency model where shared memory and messages are mutually causally consistent. This unification makes Alice's mouse selection and chat message causally consistent with her text edit. Using ATCC, Bob would have correctly seen Alice's highlighted text, saving him time and frustration.

6. Conclusion

We implemented the strongest consistency model (ATCC) compatible with high-availability using the Akka actor framework. This makes it possible to use shared objects within the actor model using transactions and while sending message between actors with causal data consistency guarantees. We studied an online collaborative text editor to show that edits to the shared document can be causally consistent with additional non-persistent messages (mouse cursor and chat).

2. A possible optimisation candidate is to use a better globally unique identifier algorithm, such as Twitter's Snowflake [2].

Bibliographie

1. Distributed data • akka documentation.
2. Snowflake ID. Page Version ID: 1070234294.
3. Abdallah (M.), Guerraoui (R.) et Pucheral (P.). – One-phase commit: does it make sense? – In *Proceedings 1998 International Conference on Parallel and Distributed Systems (Cat. No.98TB100250)*, pp. 182–192, 1998. – ISSN: 1521-9097.
4. Akkoorath (D. D.), Tomsic (A. Z.), Bravo (M.), Li (Z.), Crain (T.), Bieniusa (A.), Preguica (N.) et Shapiro (M.). – Cure: Strong semantics meets high availability and low latency. – In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pp. 405–414. IEEE, 2016.
5. Boner (J.), Farley (D.), Kuhn (R.) et Thompson (M.). – The reactive manifesto, 2014.
6. Cooper (G. H.) et Krishnamurthi (S.). – Embedding dynamic dataflow in a call-by-value language. – In *Proceedings of the 15th European conference on Programming Languages and Systems, ESOP'06*, ESOP'06, pp. 294–308. Springer-Verlag, 2006.
7. Hewitt (C.), Bishop (P.) et Steiger (R.). – A universal modular ACTOR formalism for artificial intelligence. – In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI'73*, IJCAI'73, pp. 235–245. Morgan Kaufmann Publishers Inc., 1973. – event-place: Stanford, USA.
8. Leach (P. J.), Salz (R.) et Mealling (M. H.). – *A Universally Unique IDentifier (UUID) URN Namespace*. – Request for Comments nRFC 4122, Internet Engineering Task Force, 2005.
9. Meyerovich (L. A.), Guha (A.), Baskin (J.), Cooper (G. H.), Greenberg (M.), Bromfield (A.) et Krishnamurthi (S.). – Flapjax: a programming language for ajax applications. *ACM SIGPLAN Notices*, vol. 44, n10, 2009, pp. 1–20.
10. Parker (D.), Popek (G.), Rudisin (G.), Stoughton (A.), Walker (B.), Walton (E.), Chow (J.), Edwards (D.), Kiser (S.) et Kline (C.). – Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, vol. SE-9, n3, 1983, pp. 240–247. – Conference Name: IEEE Transactions on Software Engineering.
11. Shapiro (M.), Preguiça (N.), Baquero (C.) et Zawirski (M.). – Conflict-free replicated data types. – In Défago (X.), Petit (F.) et Villain (V.) (édité par), *SSS 2011 - 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Lecture Notes in Computer Science*, volume 6976, pp. 386–400. Springer, 2011.
12. Tasharofi (S.), Dinges (P.) et Johnson (R. E.). – Why do scala developers mix the actor model with other concurrency models? – In Castagna (G.) (édité par), *ECOOP 2013 – Object-Oriented Programming*, *Lecture Notes in Computer Science*, Lecture Notes in Computer Science, pp. 302–326. Springer, 2013.

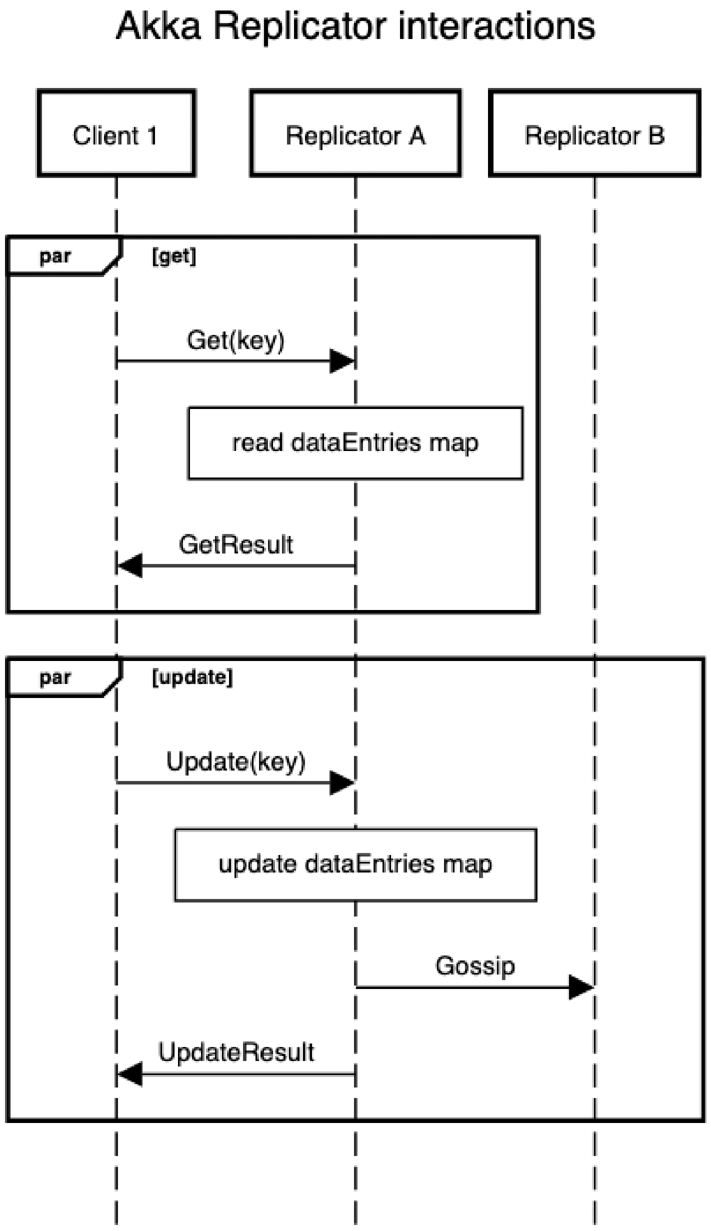

Figure 2 – Akka replicator interactions

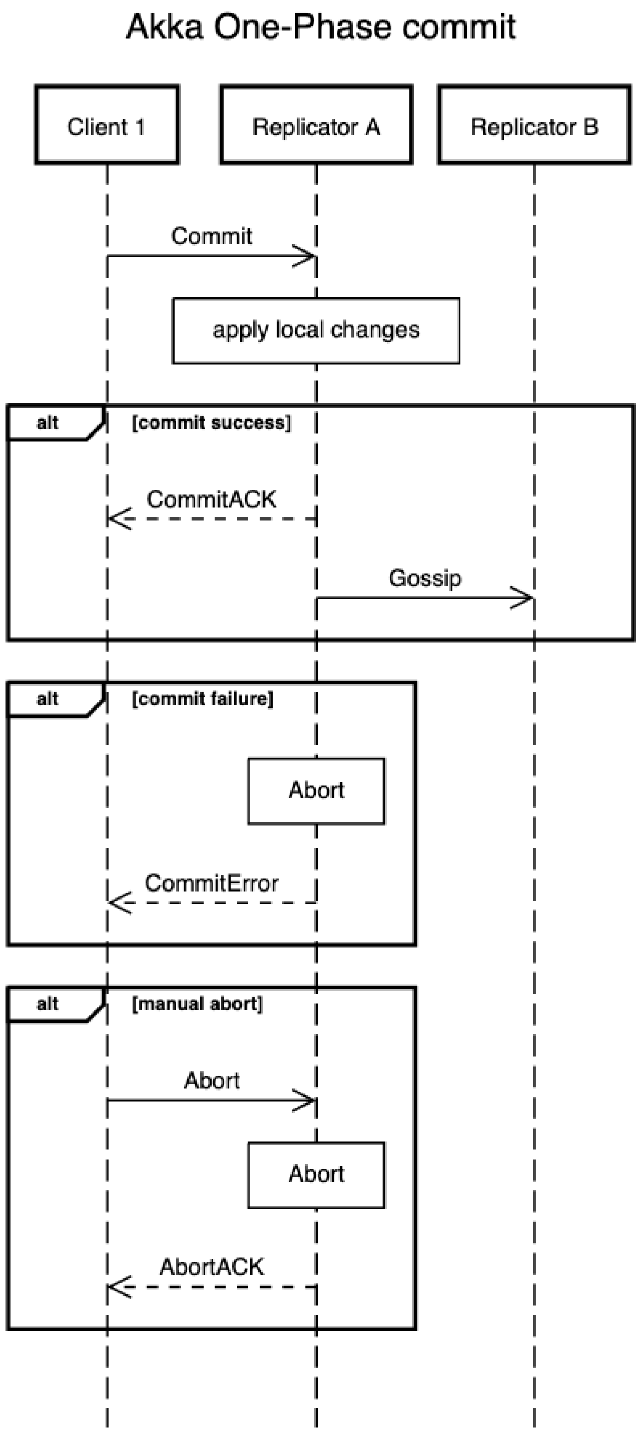

Figure 3 – Akka One-Phase commit