Transactional-Turn Causal Consistency

Benoît Martin, Laurent Prosperi, and Marc Shapiro

Sorbonne-Université, CNRS, Inria, LIP6.
firstname.lastname@lip6.fr

Abstract. Function-as-a-Service (FaaS, serverless) computing systems use an actor-like model that executes a function asynchronously, atomically and in an isolated context. However, a function must often also access state, e.g., memory or a database. This mixed model can break the actor guarantees, leading to bugs, crashes and data loss. To avoid this, we define Transactional-Turn Causal Consistency (TTCC). TTCC unifies the Turn of the actor model with the Transaction of the database model, under asynchronous, atomic and isolated execution, and guarantees mutual consistency of messages and memory. We define the model formally and present a reference implementation, along with preliminary experimental evaluation.

Keywords: causal consistency · actor model · message-passing · shared-memory · serverless

1 Introduction

This paper studies the issues that occur in a system that combines event- (or message-)based and shared-memory communication, and proposes a solution.

For instance, in Function-as-a-Service (FaaS, serverless) computing, a computation is a set of functions that execute following the actor model [2, 10]. When an actor receives an event or a message, this triggers a computation called a *turn*, to run the function being called. A turn runs in parallel with other actors, executes in the actor's separate memory space, and is uninterrupted until it terminates. Its results become visible only by sending more messages. We say an actor is *asynchronous, isolated* and *atomic*. These features are pleasing for developers, who can leverage concurrency without having to worry about memory interference, locking or deadlocks.

However, business logic often requires state; examples include video encoding, file conversion or collaborative workspaces [1, 16]. For instance, a turn may observe the memory state left by the previous turn in the same actor.[1]

Frameworks such as Orleans, Cloudflare Durable Objects, Lightbend Akka Serverless or Azure Durable Entities allow an actor to store application state in a database (Figure 1). A database computation, called a transaction, runs in

[1] This may create consistency anomalies known as *glitches* [12]. Although not well known, they are an indication that the actor model is underspecified.

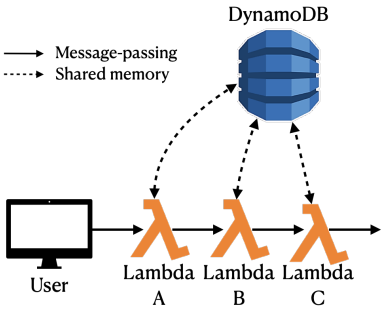

Fig. 1. A stateful serverless construct.

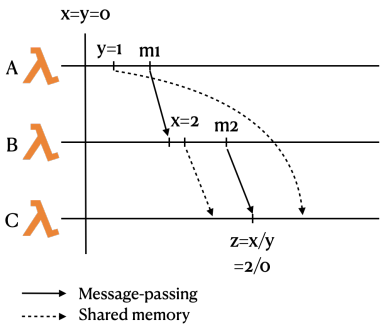

Fig. 2. An inconsistency leading to a crash.

isolation and is atomic, i.e., its results become visible at once when the transaction commits. Transactions may be or not be asynchronous, depending on the database's *isolation level* (a.k.a. consistency model): under *serializability,* transactions execute (logically) in lockstep; whereas under *snapshot isolation* (SI), a transaction does not block but operates upon a private snapshot of the database [4]. The lesser-known *Transactional Causal Consistency* (TCC) is fully asynchronous, as it also supports concurrent writes [3, 14]. In summary, TCC is also asynchronous, isolated and atomic; developers may leverage concurrency without having to worry about memory interference, locking or deadlocks.

Unfortunately, even though the buzzwords align, actors and database remain different worlds. The guarantees of one do not extend to the other. For instance, despite a turn accessing isolated local memory, it can still suffer interference via the database; and conversely, messages between transactions can defeat the consistency guarantees of the database.

To illustrate, consider Figure 2, the timeline representation of Figure 1. Database data items x and y are initially set to 0, and replicated at all nodes. Node A updates y to 1, and notifies B with message $m1$. Node B updates x to 2, notifies node C with message $m2$. In response, C computes $z = x/y$. Unfortunately, in existing systems, nothing stops $m2$ from being delivered before y is replicated on node C. Because the message view and the database view are mutually inconsistent, C computes $z = 2/0$, leading to a crash. Even if messages are delivered in order, and even if the database guarantees strong consistency, maintaining separate consistency guarantees fails to maintain mutual consistency and violates the fundamental *causality* assumption.

To avoid such anomalies, we unify the actor/message-passing and the database/shared-memory views of the world with *Transactional-Turn Causal Consistency* (TTCC). TTCC combines an actor-style execution model with shared-memory access, and *equates turns with transactions.* A transactional turn is *isolated* and *atomic*, and executes *asynchronously.* TTCC ensures that information remains consistent, whether carried in messages or in shared memory.

This paper contains the following contributions:

- The design and formalization of TTCC, a unified transaction-turn and memory-message model, in Section 3.
- Algorithms for TTCC for actors accessing a shared database, in Section 4.
- Reference implementations thereof, in Akka (Section 5).
- An experimental evaluation, showing that TTCC in addition to providing superior guarantees, TTCC can perform better than a non-unified algorithm.

2 Background

In summary, existing FaaS environments provide a mixture of asynchronous, isolated computation execution models, and of inter-actor communication models. What is lacking is a unified, consistent view across them. Therefore, this work defines a common asynchronous and isolated execution model, and a common consistent communication model.

2.1 Groundwork

A (distributed) system consists of any number of sequential processes, called actors. Actors execute in parallel, and communicate via messages and shared memory. A message may be point-to-point (from one actor to another, or to itself) or multicast (from one actor to several). Our current treatment does not consider failures.

A system may become inconsistent if events are observed in the wrong order. Intuitively, *causal consistency* is the property that if some event e might influence (cause) some event f, no actor could observe f before observing e.[2] For instance, in Figure 2, message $m2$ should not be delivered until after the update to y is replicated to C.

Borrowing from Burckhardt [6] and Viotti and Vukolić [15], we model a system execution using a multi-graph $A = (\mathcal{E}, vis)$ built on a set $\mathcal{E}$ of *events*.[3] Events comprise send, receive, read and write operations. More specifically:

Program-order $\xrightarrow{\text{PO}}$ is a binary relation over $\mathcal{E}$ that expresses the natural execution order of operations by a process.

Visibility vis is a binary relation over $\mathcal{E}$ that describes the propagation of information through the system. It satisfies the following rules:

(1) $\xrightarrow{vis}$ is acyclic.

(2) It is transitive: $\forall e, f, g \in \mathcal{E} : e \xrightarrow{vis} f \wedge f \xrightarrow{vis} g \implies e \xrightarrow{vis} g$

(3) Program order implies visibility: $\xrightarrow{PO} \subseteq \xrightarrow{vis}$

For instance, a is visible to b (i.e., $a \xrightarrow{\text{vis}} b$) means that the effects of a are visible to the process invoking b. Two operations are said *concurrent* if they are not ordered by vis.

[2] Lamport [11] calls the relation between e and f "happened-before;" recent literature uses the term "visible." [6, 7, 15].

[3] Burckhardt also defines a total arbitration order, but it is not necessary for our purpose.

2.2 Actor execution model

The classical actor model describes processes communicating only via messages. An actor alternates between being ready to accept a message, and busy processing a message. An actor responds to a message by doing local computation, creating actors, and sending messages. A *turn* is the processing of a single message. Actors conform to the following "Isolated-Turn Principle" of de Koster et al. [9]:

- Continuous message processing: An actor's turn terminates without interruption.
- Consecutive message processing: An actor processes messages from its own inbox, and processes them one by one. Within a single actor, turns do not interleave.
- Isolation: An actor can only access its own memory.

Thus, the actor is isolated, and the processing of a turn is free from low-level data races. The programmer can reason about the application as a sequence of isolated, functional turns.

2.3 Message-based communication model and causal delivery

We note messages m, n (messages are assumed unique); message-related events are send and receive, noted $send(m)$ and $recv(m)$ respectively. A message is *causally delivered* if and only if it satisfies the common rules ①–③, as well as the following:

④ A received message must be sent: $rcv(m) \in \mathcal{E} \implies send(m) \in \mathcal{E}$
⑤ A send precedes the corresponding receive: $send(m) \xrightarrow{vis} rcv(m)$
⑥ A message does not overtake another message:

$$send(m) \xrightarrow{vis} send(n) \implies \neg(rcv(n) \xrightarrow{PO} rcv(m))$$

Rule ⑤ states that m is visible when it is received, which is after it was sent. Rule ⑥ defines the order in which messages m and n are made visible (delivered). If an actor sends m, and later an actor sends n, a destination actor must observe m before n.[4]

2.4 Shared-memory transactional execution model

We borrow our shared-memory execution model from Cerone et al. [7]. They consider a database consisting of *objects* $Obj = \{x, y, \dots\}$. Events consist of $wr(x, v)$, writing version v to object x, and $rd(x, v)$, reading v from x; a write associates a new, unique version to the object being updated.

[4] We use negation ($\neg$) because a destination might receive only one of the messages.

Reads and writes are grouped into *transactions*. A transaction is a sequential and isolated execution. Its writes become visible, *atomically*, to other transactions only after it *commits*. Formally, we say transaction T is atomic iff: $\forall e, f \in T \wedge g \in T' \neq T \implies (e \xrightarrow{\text{vis}} g \Leftrightarrow f \xrightarrow{\text{vis}} g)$, i.e., either all of T's effects are visible (T is committed), or none are (T hasn't terminated yet or aborted). In what follows, we consider only committed transactions.

A transaction operates on its own *snapshot* [5], which is a copy of the state of the database at a given point in time. The snapshot ensures the transaction executes without interference from concurrent transactions.

To formalize this intuition, we define the predecessors for x in transaction T as $pred_T(x) = \{y \mid y \xrightarrow{\text{vis}} x \wedge y \notin T\}$. T has the snapshot property iff: $x \in T \wedge y \in T \implies pred_T(x) = pred_T(y)$. In other words, all the reads of a transaction come from the same set of committed transactions.

2.5 Shared-memory communication and causal consistency

Transactions communicate through the shared memory. A transaction's committed updates can be transmitted asynchronously to another transaction's snapshot, without waiting; this might cause inconsistency. An execution is *causally consistent for shared memory* if and only if it satisfies the common rules (1)–(3), as well as the following:

(7) A version read must be written: $rd(x, v) \in \mathcal{E} \implies wr(x, v) \in \mathcal{E}$
(8) A write precedes the corresponding read: $wr(x, v) \xrightarrow{vis} rd(x, v)$
(9) An update does not overtake another update:

$$wr(x, v_1) \xrightarrow{vis} wr(x, v_2) \xrightarrow{vis} wr(y, w) \implies \neg(rd(y, w) \xrightarrow{PO} rd(x, v_1))$$

Rule (8) states that an update to object x with version v, is visible before reading x. Rule (9) states that once an update, tagged with version v_2, is visible, then no subsequent operation can see a version prior to v_2. In other words, only the latest version of an object is visible.

3 Transactional-Turn Causal Consistency: unifying messages and shared memory

To avoid the inconsistency in Figure 2, while maintaining a familiar execution model, we propose to unify the asynchronous, isolated and causally consistent properties of the message and memory models. We call this model *Transactional-Turn Causal Consistency* (TTCC).

3.1 TTCC unified execution model

Our execution model equates an (actor) turn with a (database) transaction. When an actor receives a message, this triggers a transactional turn. It reads

from a snapshot that is causally consistent with the message received. When it terminates, its writes and its sends become visible atomically.

The model allows a transaction to send no more than a single message per destination actor. Otherwise, the result would not be atomic, as sending multiple messages to the same actor would cause multiple sequential turns, each one observing only a subset of the transaction's commit. If an actor must send multiple message to the same destination, it can do so in multiple sequential turns.

3.2 TTCC unified causally-consistent communication model

In the unified model, actors communicate through any mixture of message-passing and shared-memory access. An execution is *causally consistent for shared memory and messages* if and only if it satisfies the common, message-passing, and memory rules above (1)–(9), as well as the following *interaction rules*:

(10) An update does not overtake a message:

$$send(m) \xrightarrow{vis} wr(x, v) \implies \neg(rd(m, v) \xrightarrow{PO} rcv(m))$$

(11) A message does not overtake an update:

$$wr(x, v_1) \xrightarrow{vis} wr(x, v_2) \xrightarrow{vis} send(m) \implies \neg(rcv(m) \xrightarrow{PO} rd(m, v_1))$$

These rules define visibility when messages interact with shared-memory operations. Rule (10) states that if an actor writes version v to x knowing $send(m)$, then the receiving actor must receive m before observing version v for key x. Conversely, Rule (11) states that if an actor sends m while knowing $wr(x, v_2)$, then the destination actor must no longer observe the earlier v_1 after receiving m. Indeed, upon m reception, the receiving actor sees the $send(m)$ causal dependencies, i.e., $wr(x, v_1) \xrightarrow{vis} wr(x, v_2)$. Hence, the read must return v_2, the freshest visible version of x.

4 Unified message-memory protocol

In this section, we present a reference protocol that uses a unified version vector (with one entry per node) to track causal dependencies for both messages and shared objects. We present the causal delivery mechanism for messages as well as replication for shared objects.

Our protocol assumes that values in shared memory are conflict-free data structures (CRDTs) [13], which is helpful to resolve conflicts in concurrent updates without coordination.

4.1 Overview

Our protocol executes in two phases: in an actor (when a transaction is executed and when a message is received) and in a *replicator* actor that is unique per node. Replicators of different nodes communicate with each other and are responsible for maintaining transactions, snapshots and replication. A transaction operation (read, update, send message) runs inside an actor, and accesses an isolated snapshot version that is managed by the local replicator. The replicator provides the latest *local*, causally consistent snapshot to new transactions. A transaction originating from the local node is immediately visible to local actors when it commits, as local actors share the latest local snapshot. However, a transaction arriving from a remote node is visible to local actors only after the preceding transactions have committed locally.

Causal message delivery To implement causal message delivery, TTCC delays messages until all its causal dependencies are satisfied. Conversely, sending a message is non-blocking. Causal dependencies are propagated by piggy-packing metadata to messages. For instance, if an actor sends m then n, the metadata of n indicates that n causally depends on m.

Causal shared-memory To maintain causal consistency for shared memory, TTCC maintains multiple versions of objects and exposes them through isolated snapshots. Write operations are non-blocking and replication is done asynchronously. When reading an object, TTCC materializes only the requested value for the given object, as opposed to all objects in the snapshot, to reduce compute and memory consumption.

Memory-message interactions TTCC unifies causal consistency for shared memory and causal message delivery, by considering the interactions between the two memory models. Messages are delayed until causally dependent messages are delivered (Rule (6)) *and* shared-memory is up-to-date (Rule (11)). A snapshot is causally visible, when causally dependent snapshots are available (Rule (9)). Visibility of a snapshot is not delayed by causally dependent messages as the reception of a message triggers an actor's turn, which exposes a causally consistent snapshot.

4.2 Notation and definitions

Table 1 introduces the notation followed in this section to describe the execution of our protocols on an actor and on a replicator. We assume a singleton Replicator R on each node. A snapshot S is a tuple composed of a version vector vv_S and a dataset $data_S$. The GSS is a snapshot that is known to be available on all nodes at a given point in time. $LLSS$ stores a set of local snapshots that are committed. When the protocol updates GSS, snapshots from $LLSS$ are merged into GSS using CRDT logic. An ongoing transaction T is stored in *ongoing* at index T. R stores its neighbor n's version vector in kvv at index n. When kvv

updates, the protocol recompute GSS. $lastVV$ stores the latest Version Vector seen by an actor.

R	Local replicator actor
T	Transaction
q_T	Queue containing messages for transaction T
S	Snapshot
vv_S	Version vector of S
$data_S$	Dataset of S
GSS	Globally Stable Snapshot
$LLSS$	Set of Locally Latest Stable Snapshots
$ongoing[T]$	Ongoing transaction is stored at index T
$kvv[n]$	Known Version Vector for neighbor is stored at index n
m	Message sent between a pair of actors
$from_m$	Sender actor of m
vv_m	Version Vector of m
$lastVV$	Last seen Version Vector
B	Buffer for delayed messages
$+=$	CRDT merge operation

Table 1. Notation used in the protocol description.

4.3 Execution on an actor

Algorithm 1 shows the pseudo-code of the protocol for executing transaction T and the reception of message m on a causal actor. Algorithm 1 is responsible for message delivery (and delay) and transaction operations (begin, read, update, commit, abort). A message is delayed if the local shared memory is not up-to-date.

A transaction begins by sending a synchronous $StartTransaction$ message to the *local* replicator R, which contains a transaction id; we use a locally-generated UUID, as it is unique and does not require coordination. R responds with an initialized transaction snapshot, which contains the latest locally available snapshot, which is stored in $LLSS$. $LLSS$ contains the latest *local* committed snapshots that are not yet merged into the GSS. If $LLSS$ is empty, we use vv_{GSS}. Finally, if GSS is empty, we use an empty version vector.

Read and update operations send a $ReadObject$ or $UpdateObject$ message to R respectively. R returns the object's value in the transaction's snapshot.

A message sent in a transaction is stored in a buffer q_T until T commits or aborts (Alg. 1, line 2). On commit, the actor sends a $Commit$ message containing the transaction id and q_T to R. On abort, q_T is emptied, and no messages are sent.

When it receives a message (Alg. 1, line 28), the actor checks if m is causally deliverable. A message m is causally deliverable if: (1) $vv_m \leq lastVV$; (2) $vv_m[from_m] < lastVV[from_m]$; and (3) $\forall d \in (vv_m - vv_m[from_m]), d == lastVV[d]$. (Alg. 1, line 4 and 10). If m is not deliverable, it is appended to buffer B. After the delivery of m, the protocol checks B for any other deliver-

able messages. $lastVV$ is updated by being merged with the received message's version vector.

Algorithm 1 Execution of Actor a

```
1: function SEND_MSG(m, to)
2:   append m to q_T[to]
3: end function
4: function CHECK_DEPENDENCIES(m)
5:   deps ← (vv_m − from_m)
6:   for all d ∈ deps do
7:     return lastVV[d] == d
8:   end for
9: end function
10: function IS_DELIVERABLE(m)
11:   if vv_m ≤ lastVV &
12: vv_m[from_m] < lastVV[from_m] &
13: CHECK_DEPENDENCIES(m) then
14:     lastVV+ = vv_m
15:     return true
16:   else
17:     return false
18:   end if
19: end function
20: function DELIVER_CAUSAL_MESSAGES
21:   for all m ∈ B do
22:     if IS_DELIVERABLE(m) then
23:       deliver m
24:       remove m from B
25:     end if
26:   end for
27: end function
28: function ON_MESSAGE(m)
29:   if IS_DELIVERABLE(m) then
30:     deliver m
31:     DELIVER_CAUSAL_MESSAGES
32:   else
33:     B ← m
34:   end if
35: end function
```

4.4 Execution on Replication actor

Algorithm 2 shows the pseudo-code of the protocol for executing transaction T on R.

When R receives a *StartTransaction* for T and $T \notin ongoing$, the protocol initializes the transaction context by appending the latest snapshot in $LLSS$ to $ongoing[T]$. R replies with a message containing the latest vv_{LLSS}, which represents the latest locally available snapshot.

When R receives *ReadObject*, the protocol materializes the requested data. The protocol requires that $T \in ongoing$. Value v for key k is: (1) materialized from GSS, v is initially set to $data_{GSS}$; (2) all values $\leq vv_T \in LLSS$ are merged into v, using the underlying CRDT merge operation. (3) finally, $data_{ongoing[T]}$ is merged into v. (See Alg. 2, line 7). An update for key k and value v updates $data_{ongoing[T]}$ for k with v. If T aborts, $data_{ongoing[T]}$ is emptied and updates are ignored.

When R receives *Commit* (Alg. 2, line 16), commit version vector cvv_T is initially set to the latest vv_{LLSS}. If $ongoing[T]$ contains update operations or q_T is not empty, cvv_T in incremented. The protocol then updates $kvv[self]$ with cvv_T to maintain an updated version vector for the current node. Then, to terminate the commit and make the new snapshot visible to other actors, $data_T$ moves from $ongoing$ into $LLSS$ at cvv_T. Finally, the resulting snapshot is broadcast to all nodes.

On reception of a snapshot broadcast update message (Alg. 2, line 29), R checks if vv_S is concurrent with a snapshot contained in $LLSS$. This may be the

case, as local transactions can commit without coordination with other nodes. If vv_S is concurrent, we merge vv_S and $data_S$ with $vv_{LLSS[vv_S]}$ and $data_{LLSS[vv_S]}$ respectively. Then, we update $LLSS$ with the resulting snapshot. If vv_S is not concurrent, we update $LLSS$ with S. Finally, $kvv[from]$ is set to vv_S before updating GSS.

The replicator leverages the GSS mechanism that ensures progress by periodically broadcasting the latest local version vector to neighboring nodes [3]. This mechanism is useful to prune $LLSS$ by merging snapshots into GSS for all $data_{LLSS} \leq vv_{GSS}$. Note that high frequency updates may result in a high network and compute overhead, while low frequency updates may result in longer buffering and slow visibility of remote committed snapshots.

Algorithm 2 Protocol executed on Replicator R

```
1: function ON_PREPARE(T)
2:   if ongoing[T] does not exist then
3:     ongoing[T] ← latest LLSS
4:     return latest vv_LLSS
5:   end if
6: end function
7: function ON_READ_OBJECT(T, key)
8:   value = data_GSS for key
9:   value+ = data_LLSS for key
10:  value+ = data_ongoing[T] for key
11:  return value
12: end function
13: function ON_UPD_OBJECT(T, k, v)
14:   put v in ongoing[T] at k
15: end function
16: function ON_COMMIT(T, vv_T)
17:   commitVv ← latest vv_LLSS
18:   if upd or msg ∈ ongoing[T], incr
      commitVv[self]
19:   kvv[self] ← commitVv
20:   LLSS[commitVv] ← ongoing[T]
21:   remove T from ongoing
22:   TRIGGER_BCAST(LLSS[commitVv])
23: end function
24: function TRIGGER_BCAST(S)
25:   for all n ∈ allNodes do
26:     send SnapshotUpdate(S) to n
27:   end for
28: end function
29: function ON_SNAP_UPD(from, S)
30:   if vv_S is concurrent then
31:     vv_S+ = vv_S, vv_LLSS
32:     data_S+ = data_S, data_LLSS
33:     update LLSS with vv_S and data_S
34:   else
35:     update LLSS with vv_S and data_S
36:   end if
37:   update kvv[from] with vv_S
38:   UPDATE_GSS
39: end function
40: function UPDATE_GSS
41:   for i = 1, 2, ..., size(kvv) do
42:     vv_GSS ← min(kvv[i])
43:   end for
44:   data_GSS = data from GSS
45:   data_GSS+ = data_LLSS from vv_LLSS un-
      til vv_GSS
46:   GSS ← (vv_GSS, data_GSS)
47:   remove merged data from LLSS
48: end function
```

5 Implementation

We implement our unified memory model on top of the Akka actor framework.[5] Akka is open source and enables actors to share data using eventual consistency guarantees. An actor accesses data in the shared store through a replicator actor that provides a key-value API and that handles data replication. Each node spawns a singleton instance per node of a replicator actor. The replicator actor spreads object updates to its neighbors via direct replication and gossip-based dissemination.

[5] https://akka.io/

In Akka's key-value API, a key is a unique identifier of a CRDT data value. Our solution consists in applying TTCC next to the existing Akka key-value store by including additional protocols for an actor (Alg. 1) and replicator actor (Alg. 2), and additional metadata to guarantee transitive causal delivery of messages and shared objects.

5.1 Causal shared memory

We add support for transactions by encapsulating an actor's data in a causally consistent snapshot. An actor sends a message and manipulates shared objects within a transaction. A transaction begins by querying the local replicator for the latest available snapshot from $LLSS$. Get and update operations affect only the transaction's snapshot. A get operation for a given key k, materializes the value associated to k by reading from GSS, $LLSS$ and the ongoing transaction snapshot.

On commit, we compute a commit version vector and append the transaction's snapshot to $LLSS$ (Section 4.4). Then, the gossip-based replication mechanism is triggered, which asynchronously broadcasts the newly committed snapshot to other replicators.

5.2 Causal message delivery

A message sent in a transaction is associated with the transaction's snapshot and is causally sent to the recipient actor when the transaction commits. To ensure atomicity, messages remain in a private buffer until the transaction commits. If the transaction aborts, we delete the buffer.

On commit, we send the buffered messages, with an additional version vector that represents the transaction snapshot, to the destination actor. On reception of a message, the piggy-packed version vector is compared to the local replicator's version vector. Actors inherit the *CausalActor* class. This base class is responsible for delaying delivery of messages until the context is causally consistent.

6 Evaluation

Our experimental evaluation address the following questions: What is the overhead of *unified* causal consistency for messages and shared memory? How does TTCC scale on multiple nodes?

6.1 Experimental protocol

We implement four protocols in a transactional key-value store (KVS) that supports messages. Protocol 1, which is our baseline that does not guarantee causal order. Protocol 2, our reference protocol that uses a single version vector (Section 4). Protocol 3, which adds a matrix to track causality with messages. Protocol 4, that ensures causality for messages and shared memory *independently*. In this

protocol, inconsistencies between messages and shared memory may happen if a message is delivered before shared memory is up-to-date. We implement all four protocols using the Akka actor framework.

To conduct performance benchmarks, we modify YCSB [8] (version 0.17.0) to include messages and transactions [6] that we call YCSB+MT.

We provide a custom YCSB+MT workload that is similar to the original YCSB *workload* where read and write operations are run in equal proportion. Our transactional Workloads A and B executes read, update and message operations in the following proportions: 90%/5%/5% for Workload A and 5%/90%/5% for Workload B. We compare protocols 2 and 3 against protocol 4 to evaluate the overhead of mutual causal consistency.

We run the performance experiments on multiple nodes, each equipped with two Intel Xeon E5-2690v3 clocked at 2.60 GHz with 192 GB of memory.

We deploy up to ten instances (i.e., replicas) of our key-value service. Each KVS instance has its corresponding YCSB+MT client that we configure (16 threads each), to reach maximum throughput (ops/s) on each KVS. We measure the overall throughput and latency while increasing the number of nodes.

The size of a version vector is proportional to the number of nodes. In our experiment, we scale up to ten nodes. In Protocol 3, the size of the matrix is equal to the number of actor pairs, which in our experiment scales up to the number of concurrent YCSB+MT threads (16 threads).

6.2 Results

We measure the overhead of protocol 2 and protocol 3 by comparing them with protocol 4 (non-unified). Our results show that protocol 4 performs better in all workloads. We explain this by the lesser number of constraints that the protocol enforces (i.e., rules for interaction. Rules (10) and (11)).

Protocol 3 (extra matrix), performs the worst and does not scale past 4 nodes. We explain this by the cost of maintaining an extra matrix, which is both costly in transferred data and computation. For this reason, we exclude protocol 3 in the following result interpretation.

For read operations, protocol 2 (single unified version vector) performs with an overhead of up to 1.55× compared to protocol 4. We explain this by the required delay caused by our protocol to maintain *mutual* causal consistency. Furthermore, data is re-materialized for all requested values. Caching materialized data would greatly benefit read performance.

Write operations show a similar trend to read operations. Protocol 2 performs with an overhead of up to 1.14× compared to protocol 4. We explain this by the use of isolated snapshots, which enables concurrent writes without synchronization.

Our results show that message delivery also shows a similar trend compared to protocol 4 but is more dependent on write operations. Workload B (90%

[6] GitHub link: https://github.com/benoitmartin88/YCSB

writes) shows a significant increase in message response time compared to workload A, where there are less write operations. This increase in message delay is explained by the addition of required causal dependencies due to more write operations. Protocol 2 performs with an overhead of up to 2.43× compared to protocol 4.

Our experiments show that TTCC performs, in all workload conditions, similarly than a non-unified causally consistent implementation. More importantly, the overhead of maintaining *mutual* causal consistency scales up to ten nodes while providing a reasonable response time.

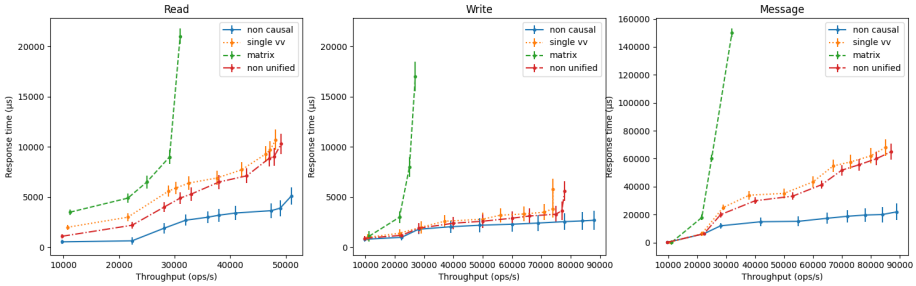

Fig. 3. Transactional workload A (90R/5W/5M)

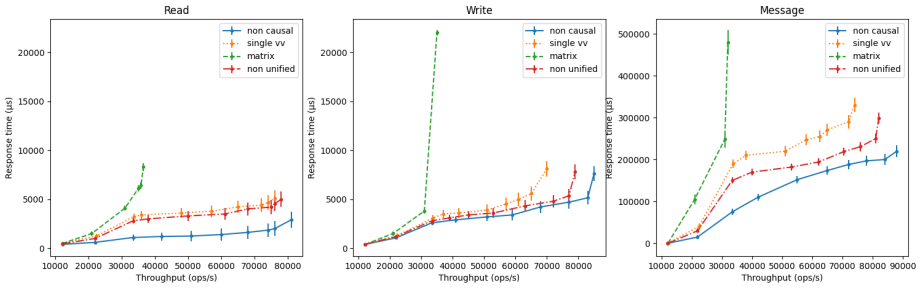

Fig. 4. Transactional workload B (5R/90W/5M)

7 Conclusion

In this paper we describe a transactional, causally consistent, unified model for message passing and shared memory, which supports asynchrony and isolated execution. TTCC is compatible with actor-based frameworks and provides an intuitive memory model that ensures that multiple pieces of information remain *mutually* consistent, whether sent using messages or shared in a distributed memory.

We presented our protocols and actor-based reference implementation. Our evaluation shows an overhead in response time of 1.55×, 1.14× and 2.43× for read, write and messages respectively, compared to two independent causally consistent memory models.

Bibliography

[1] Netflix & AWS lambda case study, https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/

[2] Agha, G.A.: ACTORS: A model of concurrent computation in distributed systems (1985), https://dspace.mit.edu/handle/1721.1/6952, accepted: 2004-10-20T20:10:20Z

[3] Akkoorath, D.D., Tomsic, A.Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. pp. 405–414. Nara, Japan (Jun 2016). https://doi.org/10.1109/ICDCS.2016.98, http://doi.ieeecomputersociety.org/10.1109/ICDCS.2016.98

[4] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. SIGMOD Rec. **24**(2), 1–10 (May 1995). https://doi.org/10.1145/568271.223785, http://doi.acm.org/10.1145/568271.223785

[5] Bernstein, P.A., Goodman, N.: Multiversion concurrency control - theory and algorithms. ACM Transactions on Database Systems **8**(4), 465–483 (1983). https://doi.org/10.1145/319996.319998

[6] Burckhardt, S.: Principles of Eventual Consistency, Foundations and Trends in Programming Languages, vol. 1. Now Publishers (Oct 2014). https://doi.org/10.1561/2500000011, http://research.microsoft.com/pubs/230852/final-printversion-10-5-14.pdf

[7] Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility p. 14 pages (2015). https://doi.org/10.4230/LIPICS.CONCUR.2015.58

[8] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154. SoCC '10, ACM (2010). https://doi.org/10.1145/1807128.1807152, https://doi.org/10.1145/1807128.1807152

[9] De Koster, J., Van Cutsem, T., De Meuter, W.: 43 years of actors: a taxonomy of actor models and their key properties. In: Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control. pp. 31–40. AGERE 2016, ACM (2016). https://doi.org/10.1145/3001886.3001890, https://doi.org/10.1145/3001886.3001890

[10] Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence. pp. 235–245. IJCAI'73, Morgan Kaufmann Publishers Inc. (1973), event-place: Stanford, USA

[11] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563

[12] Mogk, R., Baumgärtner, L., Salvaneschi, G., Freisleben, B., Mezini, M.: Fault-tolerant distributed reactive programming. In: 32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands. pp. 1:1–1:26 (2018)

[13] Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011 - 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS). Lecture Notes in Computer Science, vol. 6976, pp. 386–400. Springer (2011), https://hal.inria.fr/hal-00932836

[14] Toumlilt, I., Sutra, P., Shapiro, M.: Highly-available and consistent group collaboration at the edge with Colony. pp. 336–351. ACM/IFIP, Québec, Canada (online) (Dec 2021). https://doi.org/10.1145/3464298.3493405, https://doi.org/10.1145/3464298.3493405

[15] Viotti, P., Vukolić, M.: Consistency in non-transactional distributed storage systems (2016), http://arxiv.org/abs/1512.00168

[16] Warden, J.: Large step function data – dealing with eventual consistency in s3 – software, fitness, and gaming, https://jessewarden.com/2020/09/large-step-function-data-dealing-with-eventual-consistency-in-s3.html