Tanks: Multiple reader, single writer actors

Joeri De Koster Stefan Marr Theo D'Hondt Tom Van Cutsem

Vrije Universiteit Brussel,
Pleinlaan 2,
B-1050 Brussels, Belgium
jdekoste@vub.ac.be stefan.marr@vub.ac.be tjdhondt@vub.ac.be tvcutsem@vub.ac.be

Abstract

In the past, the Actor Model has mainly been explored in a distributed context. However, more and more application developers are also starting to use it to program shared-memory multicore machines because of the safety guarantees it provides. It avoids issues such as deadlocks and race conditions by construction, and thus facilitates concurrent programming. The tradeoff is that the Actor Model sacrifices expressiveness with respect to accessing shared state because actors are fully isolated from each other (a.k.a. "shared-nothing parallelism"). There is a need for more high level synchronization mechanisms that integrate with the actor model without sacrificing the safety and liveness guarantees it provides. This paper introduces a variation on the communicating event-loops actor model called the TANK model. A tank is an actor that can expose part of its state as a shared read-only resource. The model ensures that any other actor will always observe a consistent version of that state, even in the face of concurrent updates of the actor that owns that state.

1. Introduction

The usefulness of the actor model [4] has mainly been explored in a distributed setting, where the actors directly map onto the different distributed nodes in the system. Currently, the actor model is gaining popularity as the single-node concurrency mechanism of choice in modern languages such as Scala [16] and Clojure [23]. A recent study [21] has shown that 56% of the examined Scala programs use actors purely for concurrency in a non-distributed setting. That same study has shown that in 68% of those applications, the programmers mixed actor library constructs with other concurrency mechanisms. When asked for the reason behind this design decision, one of the main motivations programmers brought forward was inadequacies of the actor model, stating that certain protocols are easier to implement using shared-memory than using asynchronous communication mechanisms without shared state. In the case of Scala, developers can fall back on the underlying shared-memory concurrency model. This is, however, not always possible. For example in languages that strictly enforce the no-shared-state rule of the actor model.

In practice, the actor model is either made available via dedicated programming languages (actor languages), or via libraries in existing languages. Actor languages are mostly *pure*, in the sense that they often strictly enforce the isolation of actors: the state of an actor is fully encapsulated, cannot leak, and asynchronous access to it is enforced. Examples of pure actor languages include Erlang [6], E [14], AmbientTalk [22], SALSA [24] and Kilim [20]. The major benefit of pure actor languages is that the developer gets strong safety guarantees: low-level data races are ruled out by design. The drawback is however that it is difficult to express shared mutable state in these languages.

Actor libraries on the other hand are often added to an existing language whose concurrency model is based on shared-memory multithreading. Examples for Java include ActorFoundry [7] and AsyncObjects [2]. Scala, which inherits shared-memory multithreading as its standard concurrency model from Java, features multiple actor frameworks, such as Scala Actors [11] and Akka [1]. These libraries have in common that they do not enforce actor isolation, i. e., they do not guarantee that actors do not share mutable state. The upside is that developers can easily use the underlying shared-memory concurrency model as an "escape hatch", when direct sharing of state is the most natural or most efficient solution. However, once the developer chooses to go this route, the benefits of the high-level actor model are lost, and the developer typically has to resort to manual locking to prevent data races.

Combining this knowledge with the results from the aforementioned study [21], we can conclude that, on the

one hand, the *pure* actor model is a useful programming model for exploiting shared-memory concurrency as it provides the programmer with a number of safety guarantees. On the other hand, in an amount of instances it has proven too restrictive. There is a need for high level synchronization mechanisms that integrate well with the actor model.

This paper aims to remove some of the restrictions of *pure* actor languages in order to make them more useful in the context of shared-memory concurrency. This paper introduces a novel programming model called TANK. The name is a derivative from *vats* in the E programming language. While vats are opaque containers for objects, *tanks* are transparent containers (picture a fish tank). Vats enforce asynchronous communication to observe each other's state while tanks are allowed to do that synchronously. Tanks can directly observe each other's state without having to go through the message passing system. The motivation behind TANK is to allow a programmer to express common programming patterns involving shared state within the event-loop actor model. More specifically it targets applications that are typically implemented using threads and single-writer multiple-reader locks.

2. Communicating event-loops

The concurrency model of TANK is based on the event-loop model of E [14] and AmbientTalk [22] where actors are represented by *vats*. The communicating event-loop actor model marries the actor model with object oriented programming. In this model, each vat has a single thread of execution (the event-loop), an object heap and an event queue. Each object in a vat's object heap is *owned* by that vat and those objects are owned by exactly one vat. Within a vat, references to objects owned by that same vat are called *near references*. References to objects owned by other vats are called *far references* (see figure 1). The type of reference determines the access capabilities of that vat's thread of execution on the referenced object.

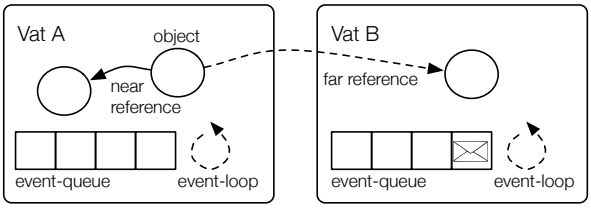

Figure 1. The event-loop actor model

Generally, actors are introduced to one another by exchanging addresses. In the event-loop actor model such an address is always in the form of a far reference to an object. The referenced object then defines how another actor can interface with that actor. The big difference between communicating event-loops and traditional actor languages is that traditional actor languages usually only provide a single entry point or address to an actor. An event-loop actor can define multiple objects and hand out different references to those objects.

Asynchronous communication. Vats are not first class entities and do not send messages to each other directly. Instead, objects *owned*[1] by different vats send asynchronous messages to each other using far references to objects inside another vat. An asynchronous message sent to an object in another vat is enqueued in the event queue of the vat that owns the receiver object. The thread of execution of that vat is an event-loop that perpetually takes one event from its event queue and deliver it to the local receiver object. Hence, events are processed one by one. The processing of a single event is called a *turn*.

Synchronous communication. The event-loop of a vat can only process synchronous messages when the receiver object is owned by the vat that is processing that message. In other words, a vat can only send synchronous messages to near references. Any attempt to synchronously access a far reference is considered to be an erroneous operation. This limitation has a number of benefits and breaking the model by removing this restriction leads to a number of problems, which are explained below.

Atomic turn property

With the macro-step semantics [5], the actor model provides an important property for formal reasoning about program semantics, which also provides additional guarantees to facilitate application development. The macro-step semantics says that in an actor model, the granularity of reasoning is at the level of a turn. This means that the processing of a single turn can be regarded as being processed in a single atomic step. Throughout the rest of this paper we will refer to this property as **the atomic turn property**. The atomic turn property leads to a convenient reduction of the overall state-space that has to be regarded in the process of formal reasoning. Furthermore, this property is directly beneficial to application programmers as well because the amount of processing done within a single turn can be made as large or as small as necessary, which reduces the potential for problematic interactions. This means that the event-loop actor model is free of low-level data races. However, as the actor model only guarantees atomicity within a single turn, high-level race conditions can still occur with bad interleaving of different messages.

Consider the example shown in figure 2. There are three vats created on lines 1, 14 and 19. Evaluating the `vat` expression will create a new vat with an event-loop, an empty event-queue and a single object in its heap for which the interface is defined by the body of the `vat` syntax. Evaluating the `vat` syntax returns a far reference to the object that was created. On line 1, a vat is created that contains a database of accounts and allows users to withdraw

[1] An object is owned by an vat if it is part of that vat's object heap

```
1  let bank = vat {
2    db = ...
3    deposit(id, amount) {
4      db[id] += amount
5    }
6    withdraw(id, amount) {
7      db[id] -= amount
8    }
9    summary() {
10     print(sum(db.values))
11   }
12 }
13
14 let client = vat {
15   bank<-withdraw(my_id, amount)
16   bank<-deposit(other_id, amount)
17 }
18
19 let manager = vat {
20   bank<-summary()
21 }
```

Figure 2. Bank account example

and deposit money to the different bank accounts. There is also a method to query the bank for the total sum of all its account balances. On line 14, a `client` vat is created that wants to transfer money from his account to another account. Lastly, on line 19 there is a `manager` vat that queries the bank for the total sum of all account balances.

The asynchronous messages sent by the `client` on lines 15 and 16 (we use the arrow notation, "`<-`", for asynchronous communication and the dot notation, "`.`", for synchronous communication) can potentially be interleaved with the `summary` message sent on line 20 by the `manager` potentially causing a race condition. This happens when the `summary` event is being processed in between the deposit and withdraw messages. This type of high level race condition is typically avoided in the actor model by coarsening up the amount of operations in a single event.

Consider the example shown in figure 3 where we replaced the `deposit` and `withdraw` methods with a single `transfer` method. The atomic turn property guarantees that the `transfer` event can be considered as an atomic operation. Processing that event can under no circumstances be run concurrently with other events belonging to the same actor such as the `summary` message. Because the `transfer` message does not alter the invariant total sum of all account balances in the bank, any `summary` messages processed before and after the `transfer` message should always observe the same total sum. Generally speaking, in-between turns (a turn is the processing of a single event), the programmer should make sure that the actor is in a consistent state as that state can potentially be observed

```
1  let bank = vat {
2    ...
3    transfer(from_id, to_id, amount) {
4      db[from_id] -= amount
5      db[to_id] += amount
6    }
7    ...
8  }
9
10 let client = vat {
11   bank<-transfer(my_id, other_id, amount)
12 }
13
14 ...
```

Figure 3. Bank account example revised

or altered by other incoming messages such as the `summary` message in our example.

It is important to note that the event-loop actor model does not provide any synchronization mechanism for groups of messages. For this paper we assume that in a correct application, from a programmer's point of view, after the processing of a single turn, an actor can be considered to be in a *consistent* state.

3. The TANK model

The TANK model is a variation on the communicating event-loop actor model where vats are represented by *tanks*. In the same way as with vats, a tank's state is conceptually separated from other tanks. Under no circumstances can any tank modify another tank's state. The main difference between a tank and a vat is that tanks are allowed to employ synchronous communication to read each other's state. In the TANK model, if a tank holds a reference to an object, independent of whether that reference is near or far, it can be accessed synchronously by that tank's event-loop. There are however a number of restrictions put on what is possible when synchronously accessing the different types of references. In the case of a near reference, the same applies as with the traditional vat model: the tank is allowed to synchronously read from and write to a near reference's object state. With far references however, a tank's event-loop is only allowed to perform read operations on the referenced object.

3.1 Synchronous communication in the TANK model

In the TANK model any actor can synchronously invoke methods on far references. The only limitation being those methods cannot modify that object's state.

Figure 4 illustrates the use of synchronous communication on far references in TANK. Note that there is always an implicit "main" tank. The main tank has its own event-loop and object heap and starts with a single event in its event-queue that is responsible for evaluating the program expres-

```
1  let container = tank {
2    value = 0
3    set(v) {
4      value = v
5    }
6    get() {
7      value
8    }
9  }
10
11 let value = container.get()
12 container<-set(value + 1)
```

Figure 4. A container with a getter and setter method

sion. On line 11, the main tank can synchronously invoke the `get` method of the container. This is allowed because the `get` method is a read-only operation that does not mutate the internal state of the `container` tank. Conceptually, the main tank will first take a snapshot of the `container` tank's state before executing the `get` method. For the duration of its current turn, the main tank will always observe the same value from the `container`. Even when that container is being modified concurrently. Attempting to modify the state of another tank using a synchronous operation is considered to be an erroneous operation. For example, synchronously invoking the `set` method would result in an error as that method assigns a new value to the `value` field of our container. Using asynchronous communication to modify the state of another tank is allowed. For example, to increase the value of our `container`, on line 12, we asynchronously send the `set` message. Once the event-loop of the `container` is ready to process the event associated with that message, the `set` method is executed by the event-loop of the `container` tank, which has both read and write access to the `value` field. Processing that message can be done in parallel with any future read operations of the main tank.

```
1  ...
2  let manager = tank {
3   bank.summary()
4  }
```

Figure 5. The manager tank can synchronously invoke the read-only summary method

Similarly, in figure 5 we rewrote the asynchronous message of the manager vat of figure 3 and replaced every occurrence of the keyword "vat" with "tank". The TANK model allows us to change the asynchronous call to `summary` on line 3 by a synchronous call. In this case, TANK will guarantee that the atomic turn property is still valid. Synchronously executing the `summary` method will always return the correct sum of all account balances even though the `summary` method can potentially be executed concurrently with a `transfer` method. Please note that, from the reading tank's perspective, the atomic turn property also only holds for the duration of a turn. For the duration of the processing of a single event of the `manager` tank, the atomic turn property ensures that the manager tank will always observe the same values for all accounts in the bank's database. However, after the manager has completed processing its turn, those values can potentially have been changed. The TANK model only guarantees consistency on the turn boundaries. This leads us to the following important property of TANK:

TANK preserves the atomic turn property. Any tank has synchronous read-only access to the internal state of any other tank. This means that any number of reader methods can be executed in parallel with at most one writer method. Any tank executing reader methods of objects inside another tank will always perceive a consistent snapshot of that tank's internal state. Any state updates of concurrently running events are not visible for the duration of the turn.

The TANK model has a lot of similarities to multiple reader, single writer locking. However, with reader-writer locks only a single writer can exist at any given time but more than one process can write the same data over the course of execution. Contrary to reader-writer locks, at all times, only a single tank can have write access to its own data. Another difference between both synchronization mechanisms is that with reader-writer locks a single writer lock will block all reader processes while in the TANK model all tanks are guaranteed to always make progress, even in the face of concurrent read and write operations. In the original event-loop actor model, the atomic turn property was guaranteed by sequentially processing messages one by one in the event-loop. In the TANK model, tanks are allowed to observe another tank's state in parallel with the owner of that state, which was impossible in the original event-loop model. The difficulty of implementing the TANK model is to ensure that the atomic turn property is still valid while still maintaining parallel execution of the different event-loops (see section 4).

Allowing synchronous communication between actors does not change the state encapsulation properties of communicating event-loops and the TANK model. When accessing a far reference, the referenced object defines what part of the object heap from that actor is accessible. The TANK model does not change this property. It changes only the way already accessible state can be accessed. While in the traditional event-loop actor model, a vat would have to employ asynchronous communication to decompose an object to which it has a far reference, the TANK model allows the different tanks to use synchronous communication for that purpose.

4. An implementation of the TANK model

There are two important properties for the processing of events that have to be upheld by any implementation of the TANK model:

- Any turn is processed in **isolation**. The TANK model ensures that, during the processing of a single event, the tank processing that event will see a consistent snapshot of the whole TANK state-space. Any state updates of concurrently running events are not visible for the duration of the turn.
- Any turn is processed **atomically**. A turn is always processed only once and is always processed to its entirety.

Software Transactional Memory (STM) [19] is a concurrent programming model that provides us with both properties. Other implementation strategies may be used to implement the tank model. However, for this paper we will discuss an implementation strategy that uses transactional memory to ensure atomicity and isolation for the processing of events. An implementation for this instantiation of the TANK model can be found online.[2] The processing of events has a transactional behavior, as such, STM is a good implementation method for the TANK model. However, it is important to note that while events have transactional behavior, the TANK model does not have any STM specific keywords to delineate transactions. Rather than introducing new keywords, the processing of a single event is considered a single transaction. To uphold the properties discussed above, the underlying implementation of tank uses a particular type of STM that has the following properties. To ensure that all events are processed in **isolation**, the entire tank's heap is part of the transactional memory. Typical STMs only apply transactional behavior to a user defined subset of the memory space because there is a significant computational overhead in making every memory location part of the transactional memory. This was a deliberate design decision in TANK to better integrate with actors. To ensure that all events are processed **atomically** and are processed only once the STM should avoid aborting transactions. The following section details the STM used for our implementation of the TANK model.

4.1 The TANK STM: A Multi-Version History STM

Most STMs can be divided into two broad categories: pessimistic and optimistic STMs. A pessimistic STM assumes that all concurrent access to the same data is dangerous and should be prevented. Usually, this involves some locking mechanism to ensure transactional atomicity. The disadvantage of this approach is that because of the locking writers can potentially block reading transactions. Optimistic STMs on the other hand assume that, in most cases, concurrent access to shared state will not conflict. Processes can read from and write to shared state concurrently. If any conflict is detected, the transaction is *aborted* (rolled back) and is retried. Typically, optimistic STMs have difficulties in handling non-idempotent operations (such as I/O) because they cannot simply be rolled back and redone when a transaction fails to commit. TANK supports a number of such non-idempotent operations such as printing something on the screen or sending an asynchronous message. This means that the implementation of TANK cannot support aborting transactions. Not being able to support aborting transactions means we have the following restrictions for our STM:

1. All writers have to have access to the latest version of the memory. Writing to a memory location based on an old version would otherwise cause the transaction to be in an inconsistent state and would have to be aborted.
2. All readers have to see values from a single snapshot of the memory. Otherwise readers could read from memory location that change value during a transaction which would then need to be aborted.

A Multi-Version History STM [17] is an optimistic approach to transactional memory where read-only transactions are guaranteed to successfully commit by keeping multiple versions of the transactional objects. The only transactions that can abort in such a system are conflicting writers. The TANK model allows only a single writer for each object, namely the owner of that object. If there is only one writer, there cannot be any conflicting writers. If all readers are guaranteed to succeed and there are no conflicting writers, all transactions will succeed and we have successfully supported both restrictions.

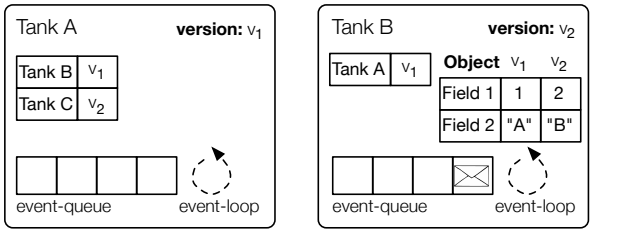

Figure 6. The tank model

Figure 6 illustrates how the multi-version history STM is integrated with tanks. Each tank t stores its own version number v_n. That number represents the latest version that was committed and is considered to be consistent. A tank also keeps track of what versions it is reading from other tanks in an associative data-structure. In our example `Tank A` is reading from `Tank B` on version v_1 and from `Tank C` on version v_2. Note that the latest consistent version of `Tank B`, namely v_2, is not the same as the version `Tank A` is reading. `Tank B` is currently only reading from `Tank A` on version v_1. Because a tank only needs to guarantee consistent reads while in a single turn, every tank only keeps track of the versions it is reading for the duration of that

[2] `http://soft.vub.ac.be/~jdekoste/shacl`

turn. The processing of any following turns can potentially observe a different version of those tanks' state. After the turn has ended, the tracked versions are discarded. For example, if `Tank A` is done processing its current event it will discard the knowledge that it was reading from `Tank B` on version v_1. Any event that is processed afterwards by `Tank A`'s event-loop and also reads from `Tank B` will observe version v_2 of that tank as that is the current version `Tank B` is in. After an event is processed, a tank can commit any modifications it has made. Committing a new consistent version of a tank's object heap is done simply by increasing its own version number.

4.2 An example

Consider the example in figure 5 again. The asynchronous message send `summary` was replaced with a synchronous call to that method. In the original example a `summary` message was sent to the `bank` tank and event-loop of that tank was the one processing that method. However, replacing that asynchronous message with a synchronous call means that the event-loop of the `manager` tank is the one processing that call. Because of that, the processing of the `summary` method can potentially run concurrently with the `transfer` method. The implementation of the TANK model should make sure that the `summary` method always returns the same result. Even in the face of concurrent updates by the `transfer` method.

From the manager tank's perspective: The `manager` tank executes the `summary` method. To do so, it must read the `db` field from the other tank. When it reads the value of that field for the first time, the main tank registers what version from the other tank it is reading. Any consecutive reads of any fields in that tank will be done with respect to that version. When the `manager` tank's turn has ended it discards all registered version numbers and is ready to process incoming messages again. When a message would cause the event-loop of the `manager` tank to process another event a new transaction is started.

From the bank tank's perspective: In figure 3, the `client` tank sends a `transfer` message to the `bank` tank. When the `bank` tank is ready to process this event, the tank will modify the database twice and then return. Updating a field with a new value will result in the creation of a new version for that field. For the duration of that event, the tank processing that event will always observe the latest version of those fields. Assigning to the same field twice does not create a new version for each assignment. Uncommitted versions are simply overwritten with the new version. After the event has been processed to completion the tank will commit the new version for the `db` field by increasing its own version number and proceed with processing the next event in its queue.

4.3 Garbage collecting old versions

One of the main challenges when maintaining multiple versions in software transactional memory is knowing what old version can be garbage collected and when. In the current implementation of the TANK model, old versions are never garbage collected, which wastes memory space. While it is impossible for a STM garbage collection algorithm to be *space optimal*, there exist efficient algorithms for collecting old versions [17]. This paper focuses more on the correctness of the proposed model rather than an efficient implementation. Part of our future work includes extending our implementation with an efficient garbage collector for old versions, for example those that are no longer reachable or read by any tank in the system.

5. Related work

The restrictions of the actor model, especially with respect to accessing shared state, have been recognized by many. The solutions to this problem however vary depending on context:

Passing arguments by reference. There is a body of related work that care about sharing state through efficiently passing the arguments of a message between actors by reference. Some of them abandon the safety guarantees delivered by the actor model for the sake of performance, e. g., Scala [11]. Others use static analysis to determine when it is safe to pass an object by reference, e. g., SOTER [15] and Kilim [20]. And others use a runtime ownership flag to attribute message arguments to the different actors in the system, e. g., Gruber and Boyer [10]. This class of research wants to avoid the cost of deep copying a data-structure when it is passed by reference. While this is useful in the context of ownership transfer, it does not really solve the state-sharing issue in the actor model. While objects can migrate between different actors, there is always only one owner for each object. It is impossible for different actors to read from a shared data structure in parallel.

Mixing actors with other concurrency models. Another body of related work cares about mixing the actor model with other concurrency mechanisms to allow the programmer to express different synchronization patterns. Scala actors [12] do not enforce the strict rules of the actor model. This means that programmers are free to share state and use traditional locking to protect concurrent access to that shared state. Unfortunately, the two concurrency mechanisms in Scala, namely actors and threads, are not always well integrated with one another and programmers are forced to resort to manual locking to prevent data races.

The Clojure [23] programming language provides primitives to use message passing based concurrency (agents) as well as software transactional memory. In Clojure, both concurrency models have some integration with one another. For example, any message sent to an agent from within a

transaction is buffered. If that transaction fails to commit, the buffered messages are discarded and the transaction is restarted.

Another attempt to mix actors with software transactional memory are Akka's transactors [3]. Here actors use a general STM system which allows the interaction with for instance threads using the STM as well. Thus, the actors do not provide the strong isolation properties we would like to preserve. As an interesting design point of these transactors is that they can be used to coordinate the state changes of different actors by utilizing the transaction boundaries as synchronization points. Similar to barriers, all actors that participate in a coordinated transaction need to complete the transaction before any of them can make progress beyond it. In the communicating event-loop model, on which TANK is based, a similar coordination can be achieved by grouping futures in order to coordinate multiple actors.

Coarse-grained synchronization. Finally, there is a body of related work that cares about integrating coarse-grained synchronization mechanisms with other concurrency mechanisms. Some of them with static compiler annotations. For example Demsky and Lam [9] propose a number of compiler annotations for defining static view definitions. A view is a coarse-grained locking mechanism for concurrent Java objects. Similarly, Axum [13] is an actor based language that allows static view-like definitions to synchronize access to shared state. Unfortunately, in both models, accessing shared state without the use of views is not prohibited by the compiler thereby compromising any general assumptions about thread safety. Another approach tried to automate parallelization by evaluating different read-only messages in the message queue of an actor in parallel, namely Parallel Actor Monitors [18] (PAM). While PAM enables parallelism inside a single actor, it does not provide a solution for sharing state between different actors. Finally, in previous work [8] of the authors of this paper, we investigated an extension of the actor model where parallel access to shared state is allowed by introducing the concepts of *domains* and *views*. A domain is an object heap that is not associated with a particular actor. Any actor can synchronously access objects inside that domain after acquiring a shared read- or exclusive write-view on that domain. Acquiring a view is an asynchronous operation that registers a callback event for when that domain becomes available for shared or exclusive access. The big difference between domains and views and the approach in this paper is that views always require the actor to use at least one asynchronous message to acquire the view. In this approach tanks can synchronously access any shared resource as long as they have a reference to it regardless of any other context. An additional benefit of the TANK model is that, contrary to views, reader operations can be executed in parallel with write operations.

6. Conclusion

The actor model is a concurrent programming model that provides a number of safety guarantees with regards to parallel programming such as deadlock and data race freedom. However, in its pure form it has shown too restrictive as programmers mix the actor model with other concurrency models. They opt to do so because of the lack of shared-memory concurrency. This shows that there is a need for more coarse grained shared-memory synchronization mechanisms that are tailored towards the actor model. Introducing shared state while still maintaining the guarantees delivered by the original model. This paper introduces a new model, called the TANK model, that is based on communicating event-loop actors. The benefit of tanks versus traditional actors is that they can expose part of their state as a shared read-only resource for other tanks. The encapsulation rules for objects living inside a tank remain the same in comparison to traditional event-loop actors. A tank has access to any object it holds a reference to. The big difference between both models is in the type of communication tanks can use to access those objects. In the TANK model a tank can use synchronous method invocation on any object it holds a reference to, regardless of whether that object is part of its own heap or of the heap of another tank. The underlying implementation of the TANK model employs transactional memory to ensure that the atomic turn property remains valid.

7. Acknowledgements

Joeri De Koster is supported by a doctoral scholarship granted by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

Tom Van Cutsem is a Postdoctoral Fellow of the Research Foundation, Flanders (FWO)

References

[1] Akka. `http://akka.io/`.

[2] Asyncobjects framework. `http://asyncobjects.sourceforge.net/`.

[3] Akka transactors, 2013. URL `http://tinyurl.com/lt5grdp`.

[4] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[5] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[6] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall PTR, 2 edition, 1996. ISBN 013508301X.

[7] M. Astley. The actor foundry: A java-based actor programming environment. *University of Illinois at Urbana-Champaign: Open Systems Laboratory*, 1998.

[8] J. De Koster, T. Van Cutsem, and T. D'Hondt. Domains: safe sharing among actors. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, AGERE! '12, pages 11–22, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1630-9. .

[9] B. Demsky and P. Lam. Views: Object-inspired concurrency control. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 395–404. ACM, 2010.

[10] O. Gruber and F. Boyer. Ownership-based isolation for concurrent actors on multi-core machines. In G. Castagna, editor, *ECOOP 2013*, volume 7920, pages 281–301. Springer, July 2013. ISBN 978-3-642-39037-1. .

[11] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009. ISSN 0304-3975. .

[12] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3): 202–220, Feb. 2009. ISSN 0304-3975. .

[13] Microsoft Corporation. Axum programming language. `http://tinyurl.com/r5e558`, 2008-09.

[14] M. S. Miller, E. D. Tribble, J. Shapiro, and H. P. Laboratories. Concurrency among strangers: Programming in e as plan coordination. In *In Trustworthy Global Computing, International Symposium, TGC 2005*, pages 195–229. Springer, 2005.

[15] S. Negara, R. K. Karmani, and G. A. Agha. Inferring ownership transfer for efficient message passing. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 81–90, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. .

[16] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008. ISBN 0981531601.

[17] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 16–25, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. .

[18] C. Scholliers, É. Tanter, and W. De Meuter. Parallel actor monitors. In *14th Brazilian Symposium on Programming Languages*, 2010.

[19] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. .

[20] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. *ECOOP 2008–Object-Oriented Programming*, pages 104–128, 2008.

[21] S. Tasharofi, P. Dinges, and R. Johnson. Why do scala developers mix the actor model with other concurrency models? In G. Castagna, editor, *ECOOP 2013 Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 302–326. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39037-1. .

[22] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)*, pages 3–12. IEEE Computer Society, 2007.

[23] L. VanderHart and S. Sierra. *Practical Clojure*. Apress, Berkely, CA, USA, 1st edition, 2010. ISBN 1430272317, 9781430272311.

[24] C. A. Varela and G. A. Agha. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.