Scalable and Serializable Networked Multi-actor Programming

BO SANG, Purdue University and Ant Group, USA
PATRICK EUGSTER, USI Lugano, Switzerland, Purdue University, USA, and TU Darmstadt, Germany
GUSTAVO PETRI, ARM Research Cambridge, United Kingdom
SRIVATSAN RAVI, University of Southern California, USA
PIERRE-LOUIS ROMAN, USI Lugano, Switzerland

A major challenge in writing applications that execute across hosts, such as distributed online services, is to reconcile (a) parallelism (i.e., allowing components to execute independently on disjoint tasks), and (b) cooperation (i.e., allowing components to work together on common tasks). A good compromise between the two is vital to scalability, a core concern in distributed networked applications.

The actor model of computation is a widely promoted programming model for distributed applications, as actors can execute in individual threads (parallelism) across different hosts and interact via asynchronous message passing (collaboration). However, this makes it hard for programmers to reason about *combinations* of messages as opposed to individual messages, which is essential in many scenarios.

This paper presents a pragmatic variant of the actor model in which messages can be grouped into units that are executed in a serializable manner, whilst still retaining a high degree of parallelism. In short, our model is based on an orchestration of actors along a directed acyclic graph that supports efficient decentralized synchronization among actors based on their actual interaction. We present the implementation of this model, based on a dynamic DAG-inducing referencing discipline, in the actor-based programming language AEON. We argue serializability and the absence of deadlocks in our model, and demonstrate its scalability and usability through extensive evaluation and case studies of wide-ranging applications.

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**.

Additional Key Words and Phrases: actor, distribution, scalability, serializability

ACM Reference Format:
Bo Sang, Patrick Eugster, Gustavo Petri, Srivatsan Ravi, and Pierre-Louis Roman. 2020. Scalable and Serializable Networked Multi-actor Programming. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 198 (November 2020), 30 pages. https://doi.org/10.1145/3428266

1 INTRODUCTION

Distributed programs are the backbone of most highly demanding online services, including for instance critical business transaction systems and multiplayer games with an ever growing user base. These distributed programs are typically running in the cloud and are split into different components that can process certain (parts of) requests from different clients independently, yet have to collaborate — often over the network — with other components to process others. For

Authors' addresses: Bo Sang, Purdue University and Ant Group, USA, bsang@purdue.edu; Patrick Eugster, USI Lugano, Switzerland , Purdue University, USA , TU Darmstadt, Germany, eugstp@usi.ch; Gustavo Petri, ARM Research Cambridge, United Kingdom, gustavo.petri@arm.com; Srivatsan Ravi, University of Southern California, USA, srivatsr@usc.edu; Pierre-Louis Roman, USI Lugano, Switzerland, romanp@usi.ch.

2475-1421/2020/11-ART198
https://doi.org/10.1145/3428266

example, in an online game, a player can explore an area by herself but can also interact with other players. Thus it is important for programmers of such applications to be able to reason about (a) parallelism (individual components executing independently) as well as (b) collaboration (multiple components working together) and achieve a good performance compromise between the two.

Actors to the rescue. During the last decade, the *actor* [Agha 1990; Hewitt et al. 1973] model has become yet more popular for implementing concurrent applications thanks to its remarkable simplicity for achieving parallelism and collaboration. Parallelism comes naturally as actors a priori do not share state with other actors and execute independently, yielding excellent potential for *scalability* of distributed applications. Collaboration among actors can be implemented via asynchronous message passing, with actors reacting to incoming messages from other actors. This simplicity has motivated actor extensions and libraries for most mainstream programming languages, e.g., Akka [Lightbend 2020] and Scala Actors [Haller and Odersky 2009] for Scala/Java, Asynchronous Agents Library [Microsoft 2020a] and C++ Actor Framework [Charousset et al. 2016] for C++, and Akka.NET [akk 2020] for C# and F#. Several actor-based languages such as Orleans [Bykov et al. 2011] have also been more recently proposed specifically for implementing scalable networked distributed online services.

Beyond single messages. However, in many scenarios, it is useful if not necessary for programmers to reason not only about individual messages, but in terms of *compositions* of such messages for collaboration. A set of actors may be involved in multiple tasks which do not permit interleaved execution between them (i.e., the tasks require strong consistency). The original asynchronous "singleton" messages in the actor model do not guarantee any execution order, and do not support task isolation. Different extensions and variants of actors have thus been proposed. E.g., Synchronizers [Dinges and Agha 2012; Frølund 1996] allow (constraints on) message compositions to be specified abstractly, independently from actors themselves. Several seminal works propose some form of transactional actors (e.g., [Field and Varela 2005; Koster et al. 2015; Swalens et al. 2018]).

A thin line. As underlined by decades of research in distributed data management, care is required when introducing strong consistency guarantees for message compositions, as this may necessitate costly mechanisms whose overhead can hamper the potential for scalability of a programming model – especially across hosts communicating over a network. Synchronization protocols, re-/un-doing of (partial) computations, etc. are easily abstracted in formal models or implemented in single processes, but can introduce significant overheads in practice. Many popular databases thus for instance offer *snapshot isolation* instead of stronger consistency models (e.g., Oracle, PostgreSQL). Rather recently Orleans was augmented with a form of two-phase locking for strong consistency [Orleans 2020], which however introduces high overhead as we show in our evaluation.

Scalable serializability over the network. Adding strong consistency to programming models for *networked* distributed asynchronous environments in a way achieving scalability and guaranteed progress (e.g., deadlock freedom) at the same time without hampering performance is challenging. In this paper, we propose a novel actor-based programming model settling both ends of the challenge. More precisely, we provide a model with *serializability* [Papadimitriou 1979] for multi-actor programming, meaning messages issued as so-called *events* (i.e., requests) execute in an isolated fashion. Our model is deadlock-free yet scalable as it enables decentralized coordination that does not rely on rollbacks or transient inconsistencies. This is particularly important in the context considered herein where communication between actors commonly takes place over the network with latency orders of magnitude higher than in local interaction. The crux behind our model is to streamline execution along an actor communication graph in the form of a directed acyclic graph (DAG). Each actor is under the aegis of a unique *dominator* actor, according to its position in the DAG, assigned in a dynamic manner (i.e., the DAG structure may change at runtime).

The execution of events is partially serialized by those dominators: events executing on actors with the same dominator are serialized there, while events on actors with different dominators execute in parallel. Our model captures many useful application scenarios requiring strong consistency, while still supporting a high degree of parallelism and thus scalability. When the DAG structure is overconstraining, actors can issue calls outside it which are decoupled from their parent events.

Contributions and outlook. More specifically, the contributions of this paper are:

- a pragmatic actor-based programming model for implementing networked distributed online services based on a dynamic DAG-inducing referencing discipline. Our model — implemented in an extension to C++ dubbed AEON — introduces a high-level notion of events representing serialized units of execution across actors, and several practical refinements. We illustrate AEON through a multiplayer game app, demonstrating the need for serializability.
- a synchronization scheme that leverages the DAG-based structure to serialize event execution and enable highly parallel execution in a dynamic manner.
- a characterization of properties — serializability and deadlock freedom — enjoyed by applications implemented in AEON.
- a study of the use of AEON and its different features across a wide variety of applications. We discuss implementation choices for different consistency levels through the game app.
- a performance study of several applications (e.g., metadata storage, B-tree) implemented in AEON, in particular distilling the costs of serializability. We compare AEON to state-of-the-art specialized systems such as HyperDex Warp [Escriva et al. 2015] and Infinispan [Infinispan 2020], and to implementations in Orleans [Orleans 2020] on Amazon AWS [AWS 2020]. AEON outperforms its competitors in most cases, and demonstrates much better scalability.

Roadmap. § 2 overviews and motivates our programming model. § 3 presents our programming model in more detail. § 4 presents synchronization and properties of AEON. § 5 discusses implementation, optimizations, fault tolerance and the applicability of AEON. § 6 evaluates AEON's performance. § 7 discusses related work. § 8 draws final conclusions.

2 A PRIMER

This section presents the high-level design goals underlying AEON [AEON 2020], motivates them through a concrete example, and gives an overview of how AEON addresses them.

2.1 Goals and Bird's-Eye View

AEON's design targets the following high-level goals:

G1: Support scalable networked distributed (actor) programming in a mainstream programming model.

G2: Enforce consistency (serializability) between concurrent *events* involving invocations across multiple actors to avoid races between such events.

G3: Relieve programmers from error-prone manual locking to prevent deadlocks between events.

To achieve G1 we augment the C++ programming language with actors. In § 3 and § 4, we elaborate on how we address G2 and G3 in a scalable manner.

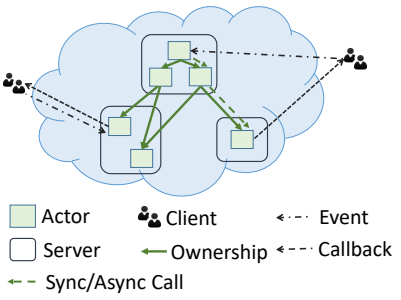


Fig. 1. AEON application overview.

Next we motivate the goals above and provide an overview of the main features of AEON through the example of a multi-player game app that allows players (acting as clients) to manipulate their avatars in an environment and interact with it. Fig. 1 gives an overview of an AEON application. Clients issue events to actors which are distributed across multiple servers. Events in the AEON

```
1  actorclass Steakhouse {
2    vector<Cook> cooks;
3    vector<Grill> grills;
4    vector<Customer> customers;
5    ...
6  }
7  actorclass Cook { // Players are cooks
8    vector<Grill> grills;
9    vector<Customer> customers;
10   Customer cu;
11   int ckid;
12   ...
13   void cook(int orderSize, int cuid) {
14     cu = customers[cuid];
15     for (int i = 0 upto orderSize) {
16       Steak s = new Steak();
17       for (int j = i upto grills.size()) {
18         if (!grills[j].isUsed()) {
19           // 3 options: sync, async, event
20           grills[j].put(s, ckid);
21           // async grills[j].put(s, ckid);
22           // event grills[j].put(s, ckid);
23           break;
24   }}}}
25   void grillTimerRings(int gid)
26     { async cu.get(grills[gid].take()); }
27 }
28 actorclass Grill {
29   yield vector<Cook> cooks;
30   yield Cook cook;
31   Steak steak;
32   int gid;
33   ...
34   bool isUsed() { return cook != NULL; }
35   void put(Steak s, int ckid)
36     { steak = s; cook = cooks[ckid]; ... }
37   Steak take()
38     { ...; cook = NULL; return steak; }
39   void timedOut()
40     { event cook.grillTimerRings(gid); }
41 }
42 actorclass Customer {
43   yield vector<Cook> cooks;
44   int cuid;
45   ...
46   void giveOrder(int ckid, int nb)
47     { event cooks[ckid].cook(nb, cuid); }
48   void get(Steak s) { ... }
49 }
50 class Steak  { ... }
```

Listing 1. AEON code snippet for a steakhouse simulation multiplayer game where players are cooks.

application are executed in a serializable manner. The return values of those events are passed back to clients via callback methods. Programmers can specify high-level placement policies, e.g., on collocation/separation of particular types of actors, or on limits for the number of collocated actors, separately from the main program [Sang et al. 2020]. By default the runtime tries to collocate an actor with its parents. In this paper we focus mostly on the server-side portion of AEON programs.

2.2 Scenario

The game app considered shares basic ideas with many simulation games. Lst. 1 sketches a possible implementation in AEON, an extension of C++. The keywords appearing in green are new to AEON; we use the syntactic form `upto` to iterate through a numerical range.

Users can manipulate their avatars (cooks) (Line 7) in a steakhouse to finish certain cooking tasks. Consider possibly large numbers of cooks, steakhouses, and the other types of actors in this game. Those actors are distributed across multiple servers to guarantee that there are enough resources provided to this game. Steakhouses (Line 1) include different types of items (e.g., grills). Cooks use those items to achieve certain cooking tasks. Different tasks require different sets of items. However the number of items are limited and cooks have to share them in certain cases. Cooks contend on one or more items at the same time. To avoid conflicts, it is important to guarantee that cooks access items in an isolated manner in order to avoid inconsistencies such as multiple acquisitions of the same item (G2) or deadlocks (G3). Besides, in a distributed environment, player actors may interact with item actors on remote servers. Unlike concurrent programming [Lea 2005] on a single server, it is more difficult to implement isolation efficiently in an asynchronous distributed environment (G1) as any invocation might take place over the network (cf. Fig. 1).

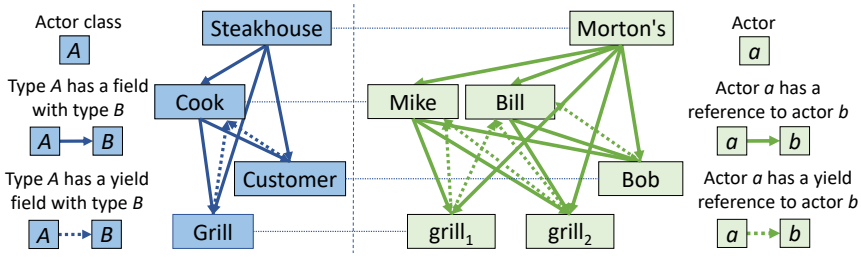


Fig. 2. Actor type and actor ownership reference structures in the game app.

Consider two cooks in a steakhouse who are responsible for cooking steaks for customers. The order of a table is sent to a cook who needs to put the required number of steaks on grills (one steak per grill) at the same time such that all the customers of a table may be served at the same time. Assume there are 10 grills in total and both cooks receive an order for 6 steaks. Because resource availability check (Line 18) and resource locking (Line 20) are performed in separate instructions, without proper synchronization, each cook may occupy only 5 grills and run into deadlock.

As another example, a table has ordered two kinds of steaks and one cook is responsible for one kind. It is possible for both cooks to observe that the grills of the other cook are still empty, then they pick up one kind for cooking randomly. A possible outcome without synchronization could be that the two cooks have prepared the same kind of steak.

2.3 Actors

AEON's `actorclass`es, declared for instance at Line 1 and Line 7, can be thought of as class-like foundries for distributed objects — actors (G1) — that can contain data in the form of fields:

(1) Such fields can be of object (class) types, as is the case of Line 31 declaring a field `steak`, an instance of class `Steak` declared at Line 50. Objects are passed by value so `steak` is guaranteed to reside in the same address space as the actor instance of `Grill` that refers to it (Line 31).
(2) Unqualified fields of actor types can contain references to other actors, which at runtime may be remote. E.g., at Line 3, field `grills` contains a vector holding references to `Grill` actors.
(3) An actor type field can be declared as a special `yield` field (Line 30). This is used for internal events which are executed in isolation from the caller event.

"Regular" actor type references, i.e., references of type (2) above induce an ownership graph. In short, AEON enforces that this graph is a directed acyclic graph (DAG) at the type level (actor classes), thus enforcing the same at the instance level (actors). Fig. 2 depicts the reference structure for both actor classes (left part) and actors (right part) according to our scenario and Lst. 1. As we will detail shortly, this graph is essential for AEON to efficiently execute methods concurrently invoked on actors as part of events in a way avoiding inconsistencies (G2) and deadlocks without manual locking (G3). Yield fields increase expressiveness beyond such a DAG, however with restrictions regarding isolation: an actor invocation through such a field leads to an *internal* event which is serialized separately from any calling event, as explained shortly in § 2.4.

2.4 Events

Like other languages designed for implementing Internet-facing applications (e.g., [Bykov et al. 2011; Chuang et al. 2013]), AEON follows an event-driven model, where clients of the system (e.g., users) interact with the server application by issuing events to the latter through calls, tagged in AEON *at the call site* with the `event` keyword in front of the expression representing the *target actor* of the event. For instance, by calling `event cook.cook()` (Line 47), a customer prompts the `cook` to perform a cooking task, with `cook` the target actor of the event. Importantly, the execution

of an individual event is guaranteed to be isolated — more precisely, serialized — with respect to other events. As detailed shortly in § 3, and based on the reference DAG structure in Fig. 2, the two `Cook` actors (Mike and Bill) of `Steakhouse` will be assigned the same *dominator* actor (Morton's). All events have to access those actors in a synchronized manner by locking the dominator, thus guaranteeing strong consistency and avoiding deadlocks. This means that steak cooking tasks made by different `Cooks` on the same `Grill` will be executed in sequential order.

Note here when to call the `Grill::put` as `sync` or `async` method (Line 20 and Line 21), it will execute as part of the `Cook::cook` event. In this scenario, `Grill::put` should better be called as `async` method. Furthermore, the `Grill::put` can also be called as the `event` method (Line 22), which will generate an independent event. As foreshadowed, actor (class) methods can call other actor methods, with actors being passed by reference, and objects passed by value, i.e., deep copying.

If not called as an *external(ly issued)* event by a client to the main application, an event call can also be made within an event, which we refer to as an *internal(ly issued)* event. E.g., `Grill::timeOut` makes an event call `Cook::grillTimerRings` to inform the cook that the steak is ready. Since every method call is made directly or transitively in the context of some event, this call leads to a "sub-event" – an independent new event executed after the caller event ends. We detail this in § 3.

As part of an event, actor methods can be called either synchronously (default) or asynchronously, indicated with the `async` keyword at the call site. A synchronous call blocks the execution of the event in the current actor until a result is obtained from the target actor upon return. Line 20 shows an example of such a synchronous call, which needs to wait for the steak to be put on the grill. Conversely asynchronous calls allow execution to proceed immediately, increasing parallelism. An example of asynchronous call is given at Line 21: the cook puts multiple steaks on grills in parallel.

3 PROGRAMMING MODEL

This section presents the actor-based programming model of AEON in more depth. The abstract syntax of AEON$_{mini}$— a core sub-language of AEON — is shown in Fig. 3, and will be detailed in the following paragraphs. For simplicity we omit from the syntax all the sequential aspects of C++ which are inconsequential with respect to the ideas presented in AEON.

3.1 Execution Model Overview and Core Principles

At a high level, AEON induces a two-level hierarchy of entities: actors and objects. The former are accessed via reference, while the latter are treated as values. As mentioned program execution is triggered by the receipt of a request from a client in the form of an external event, which is simply the invocation, tagged as `event`, of an actor method. Any "regular" (i.e., non-`event`) nested method invocation is executed as part of that event without interference of other events. Events can also trigger other, internal, events. All such events are delegated until completion of the calling event. This means that nested events do not execute logically as part of their calling events, but are rather decoupled and serialized with respect to them. AEON guarantees serializability (G2) and deadlock freedom (G3) with respect to all events. As foreshadowed in § 2, AEON achieves these goals in a scalable manner through the following design decisions:

D1: Exploit application semantics, i.e., (constraints on) actor application topology induced through references, for efficient synchronization. That is, the execution of events follows a DAG-based arrangement of actors, exploiting this structure for automated, largely localized, lock-based synchronization of actors. (Analyzing existing distributed applications, cf. § 5.4, we observed that DAGs are common.) The DAG can be statically enforced by ownership at the type level.

D2: Induce a tree overlay for the DAG. Intuitively an actor ownership tree is easy to exploit for synchronization, as any actor can only be reached by one path from the root — thus any two events targeting a same actor can be serialized on the first node of any common path traversed

Method Names $m \in \mathcal{M}$ Field Names $f \in \mathcal{F}$ Variables $x, y \in \mathcal{V}ar$
Class Names $cls \in Cls$ Actorclass Names $acls \in \mathcal{A}Cls$

Program Definition	$p \in \mathcal{P}$	::=	$\overrightarrow{aclsd}\ \overrightarrow{clsd}$ `main`$(\ldots)\{\ t\}$
Actorclass Definition	$aclsd \in \mathcal{A}ClsD$	::=	`actorclass` $acls\ \{\ \overrightarrow{fd}\ \overrightarrow{md}\ \}$
Class Definition	$clsd \in ClsD$	::=	`class` $cls\ \{\ \overrightarrow{fd}\ \overrightarrow{md}\ \}$
Type	$T \in \mathcal{T}$	::=	$acls \mid cls \mid T[] \mid$ `int` $\mid$ `float` $\mid \ldots$
Field Definition	$fd \in \mathcal{F}D$	::=	`yield`$^?$ $T\ f$
Method Definition	$md \in \mathcal{M}D$	::=	`ro`$^?$ $T\ m(\overrightarrow{T\ x})\ \{\ t\ \}$
Decorated Call	$dc \in \mathcal{D}Call$	::=	`event` $\mid$ `async`
Term	$t \in \mathcal{T}erms$	::=	$dc^?\ t.m(\vec{t}) \mid t.f \mid x \mid$ `skip` $\mid$ `return` $\mid$ `this` $\mid \ldots$

Fig. 3. Abstract syntax of AEON$_{mini}$. Elements with $^?$ denote options. $\vec{z}$ denotes several instances of z.

on the way there. While a DAG benefits from a topological ordering, it is more complex to exploit it for serializability. We thus introduce a novel concept of *dominator* for serializing events.

D3: Extend expressiveness by allowing AEON applications to use references to actors *bypassing* the DAG shape. However such references must be appropriately tagged (`yield`), and introduce constraints regarding consistency in that calls through them give rise to separate events.

3.2 Actors and Objects

A program p consists of class declarations *clsd*, actor class declarations *aclsd*, and a main expression `main` (inherited from the structure of C++ programs). An actor is a *stateful point of service* that receives and processes requests in the form of messages from clients, directly (as external events) or indirectly (via other actor instances). Actors encapsulate local state in the form of fields fd and functionality in the form of exported methods *md*, as defined by their classes. Actor method invocations $dc^?\ t.m(\vec{t})$ give rise to messages, and internal representations of actors can only be read and/or affected through their methods (`this`.f). Actors are implemented by the means of objects, i.e., their internal states are captured by objects. Both actor classes and object classes can contain actor type expressions, as shown in Fig. 3. However, an object class can only hold actor references in `yield` fields. Actors can refer to each other by references. While actors are always passed by reference, objects are always passed by value between actors. The former takes precedence, i.e., an actor referred to by an object's field will be passed by reference when passing the object. Objects are passed by value even between collocated actors to avoid races.

3.3 References and Ownership

A pragmatic choice in our model is to streamline execution along graphs of a directed acyclic nature induced by actor references within an application (at the exception of `yield` fields ignored for now and discussed later in this section). The key idea behind our model is to determine the set of actors which can be accessed by an event when it is issued. In an AEON application, one event can only access an actor when this event has obtained the actor's reference. Then, the event can access actors referenced by the fields of the actor where the event is executing. Based on the ownership graph, the AEON runtime system can verify whether the sets of accessible actors of two events overlap. Our synchronization model introduced in § 4 ensures that such conflicting events are serialized, while non-conflicting ones can proceed in parallel.

Asserting absence of cycles (D1). Absence of cycles is ensured by applying a conservative static type-based program analysis. The type system keeps track of the types of fields and method

arguments which are of actor types (object types are not considered for this as they only contain yield fields). E.g., an actor of type $acls_0$ cannot have a field of type $acls_1$ if $acls_1$ has a field of type $acls_0$. The analysis is simplified by disallowing inheritance/method overriding or subtyping for actor classes, and by a strict separation of actor and object class hierarchies. Note that a small exception is made for direct recursion by the use of runtime cycle checks which can be made fast through their "local" nature (omitted from the formal language and in the following for simplicity).

From descendants to dominators (D2). To avoid deadlocks, AEON guarantees that whenever two calls have the potential of affecting same actors, an order will be established to access these actors. To that end, AEON leverages the topological order underlying DAGs. As several paths can lead to an actor in such a graph, however, serializing events accessing same actors is more involving than in a tree structure (e.g. [Blessing et al. 2017; Golan-Gueta et al. 2011], cf. § 7). We thus introduce a novel notion of *dominator*. Intuitively, a dominator for a set of actors is the "lowest" actor in the DAG traversed by all paths leading to any of those actors, or actors reachable from them. Thanks to its position in the DAG, a dominator can be responsible for serializing events for this set of actors.

More precisely, ownership of actors in AEON can be described as a DAG $\Omega = (\mathcal{A}, \rightarrow_o)$, with $\mathcal{A}$ representing the set of actors and thus *vertices* in Ω, and the relation $\rightarrow_o$ representing *edges* (ownership relations) between such actors. That is, if actor a_0 has a reference to a_1 in one of its non-yield fields, then we have a *directed edge* $a_0 \rightarrow_o a_1$.

Definition 1 (Parent and child actors). *For $\Omega = (\mathcal{A}, \rightarrow_o)$ and $a_0, a_1 \in \mathcal{A}$, if $a_0 \rightarrow_o a_1$ we say a_0 is a* parent actor *of a_1, and a_1 is a* child actor *of a_0. $\mathsf{children}(\Omega, a)$ is the set of a's child actors.*

Definition 2 (Descendant actors). *For $\Omega = (\mathcal{A}, \rightarrow_o)$ and $a_0, a_1 \in \mathcal{A}$, if $a_0 \rightarrow_o^* a_1$ with $\rightarrow_o^*$ the transitive closure of $\rightarrow_o$, then we say a_0 is an* ancestor (actor) *of a_1, a_1 is a* descendant (actor) *of a_0, and a_1 is* reachable *from a. $\mathsf{desc}(\Omega, a_0)$ presents the set of a's descendant actors.*

An actor must be made a dominator when it does not *share descendants* with any other actor apart from its own ancestors and descendants. We thus first define the notion of sharing descendants.

Definition 3 (Sharing). $\mathsf{share}(\Omega, a)$ *represents the set of actors which* share *descendants with a in $\Omega = (\mathcal{A}, \rightarrow_o)$ and is defined as follows:*

$$\mathsf{share}(\Omega, a) = \{a' \mid \mathsf{children}(\Omega, a') \cap \mathsf{desc}(\Omega, a) \neq \emptyset \ \wedge \ a \in \mathsf{desc}(\Omega, a')\} \cup \quad \text{(i)}$$
$$\{a' \mid \mathsf{desc}(\Omega, a') \cap \mathsf{desc}(\Omega, a) \neq \emptyset \ \wedge a' \notin \mathsf{desc}(\Omega, a) \ \wedge \ a \notin \mathsf{desc}(\Omega, a')\} \quad \text{(ii)}$$

That is, $\forall a' \in \mathsf{share}(\Omega, a)$ we find actor a' satisfies one of following conditions:

(i) a is the descendant of a', and has at least one descendant that is a child of a'.
(ii) a' has shared descendants with a, but is not a descendant of a nor vice versa.

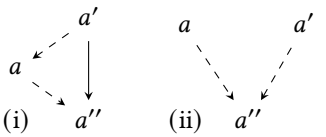


Fig. 4. Conditions (i) and (ii) resp. of share definition. Solid edge indicates a child, dashed edge a descendant.

In condition (i), as shown in Fig. 4, actor a' might be an ancestor of a, but might still have a direct reference to one of a's descendants. Condition (ii) is needed to avoid considering all ancestors of a in their share set.[1]

For a given actor set $\mathcal{A}'$, we calculate the "lowest actor above" all actors sharing descendants with any a in $\mathcal{A}'$, denoted $\mathsf{dom}(\Omega, \mathcal{A}')$ and dubbed $\mathcal{A}'$'s *dominator*, computed as the *least upper bound* (lub) of the nodes in $\bigcup_{a \in \mathcal{A}'} \mathsf{share}(\Omega, a) \cup \mathcal{A}'$ of Ω:

Definition 4 (Dominator). *The* dominator *of actor set $\mathcal{A}'$ in Ω, $\mathsf{dom}(\Omega, \mathcal{A}')$, is defined as* $\mathsf{dom}(\Omega, \mathcal{A}') = \mathsf{lub}\big(\Omega, \bigcup_{a \in \mathcal{A}'} \mathsf{share}(\Omega, a) \cup \mathcal{A}'\big)$.

[1] Note that if in (i) a'' is a descendant (not child) of a', there is a a''' between a' and a'' that can be mapped to a' in (ii).

We focus on actor *sets*, as method calls can have actor references as arguments. Hence a called method, or any method targeted by a nested call from it, can try to access such an argument or any actor accessible from it. The dominator for a single actor a can be straightforwardly derived as $\mathsf{dom}(\Omega, a) = \mathsf{lub}(\Omega, \mathsf{share}(\Omega, a))$. These dominators induce a tree overlay for the DAG. As we detail in the next section, the locking of any actor (set) is managed by its dominator.

In the special case of a DAG with multiple maxima without common ancestor the AEON runtime system adds an abstract *root* actor as the parent of all such maxima. This way, the runtime system can ensure the existence of a dominator for any actor in the program.

Illustration. In Fig. 2 (left), actorclasses `Steakhouse` and `Cook` "share" (at the type level) actorclasses `Grill` and `Customer`, and `Cook` is the "child" of `Steakhouse`. `Cook` actors are likely to be dominated by `Steakhouse` actors. As the right part of Fig. 2 shows, `Cook` actors *Mike* and *Bill*'s dominator is `Steakhouse` actor *Mortons*. *Mortons*'s dominate region includes *Mike*, *Bill*, *Bob*, $grill_1$, and $grill_2$.

Yield fields (D3). As explained above, actor references define the shape of the runtime DAG of an AEON application. To increase expressiveness, AEON allows fields to be declared with an optional `yield` qualifier, which means that an actor referenced by such a field is not a child (or owned) by the current actor. Thus such fields can be of types which bypass the type-level DAG. In fact all fields of clients are treated as yield fields. As we elaborate on shortly in § 3.4, yield fields however only support (new) event calls, such as to retain AEON's properties.

3.4 Actor Methods and Events

Actors interact via messages induced by invocations to methods on actors. (We focus on these methods here.) Method declarations have a return type T, a method name m, a sequence of formal arguments of the form $T\ x$, a body term t, and, optionally, a leading `ro` modifier denoting *readonly* methods. Actor methods can invoke other actor methods in a nested manner as part of their body t.

Call types. More precisely, AEON offers three types of calls — standard synchronous calls and two types of decorated calls denoted by dc in the syntax (cf. Fig. 3):

(1) Standard method calls in AEON are synchronous, denoted by the usual dot notation: $t.m(t)$. Our model ensures that these calls are not subject to deadlocks.
(2) Similarly to the basic actor model, method calls can be `async`hronous: `async` $t.m(t)$. Here the caller continues straight after the call rather than waiting until the call completes.
(3) Method calls tagged as `event`s are (i) external(ly issued) asynchronous requests from clients to a (server) application actor or (ii) internal(ly issued) asynchronous requests from other actors typically but not necessarily through yield fields. Both types of events are executed in a serialized manner. Events in (3.i) define the external (client) API of an AEON application.

Nested events, yield fields. Nested event invocations (3.ii) are not a part of the current event, but are executed *after* completion of the current event. Calls on yield fields *must* be tagged as `event`s or the compiler raises an error. Just as with asynchronous calls (2), return values of events cannot be accessed. Results of events are passed via *callback* events. To that end (client) actors can pass references to themselves as arguments to (external) events. This also allows for multiple results.

Effects. Our model includes readonly methods as they can execute in parallel on a same actor. The AEON compiler conducts a simple static analysis to ensure that methods tagged as `ro` do not perform assignments to actors' fields or calls to non-readonly methods. As for asserting the DAG shape, the analysis is simplified by the absence of actor inheritance and subtyping; the code for a readonly method call is exactly known. However, readonly methods can also be called `async`hronously, or as events, which may seem counter-intuitive since the return values are then not accessible. As alluded to above, AEON supports a programming style where returns of events and asynchronous methods

Table 1. Summary of AEON method call semantics for all types of calls $dc^?\,x.m(...)$ of methods `ro`$^?$ T $m(...)$ on fields `yield`$^?$ T' f with T' an actor type. "External" indicates calls from clients and "internal" all other calls. "Callback" indicates that the caller cannot access any return value; instead the callee can send returns through callback methods; "included" means the method is executed as part of the caller event while "serialized" indicates the method will be executed as separate event.

Caller	Call ($dc^?$)	Field (`yield`$^?$)	Method (`ro`$^?$)	Result	Blocking	Execution
External	`event`	`yield`	Any	Callback	Non-blocking	Serialized
Internal non-`ro`	`event`	Any	Any	Callback	Non-blocking	Serialized
Internal `ro`	`event`	Any	`ro`	Callback	Non-blocking	Serialized
Internal non-`ro`	Sync	Non-`yield`	Any	Return (T)	Blocking	Included
Internal non-`ro`	`async`	Non-`yield`	Any	Callback	Non-blocking	Included
Internal `ro`	Sync	Non-`yield`	`ro`	Return (T)	Blocking	Included
Internal `ro`	`async`	Non-`yield`	`ro`	Callback	Non-blocking	Included

are handled via callbacks. When such callbacks are made to clients, these may have side-effects yet can still be made by readonly methods, which otherwise can only call other readonly methods. The `ro` qualifier thus refers to server-side code. Other (i.e., non-readonly) methods can modify fields of type (2) introduced in § 2.3 — non-`yield` actor type fields. As they define ownership between actors, methods performing such assignments change the $\rightarrow_o$ relation in the actor DAG $\Omega = (\mathcal{A}, \rightarrow_o)$:

Definition 5 (Ownership method and event). *If method m updates non-yield actor type fields, m alters $\rightarrow_o$ of the actor DAG $\Omega = (\mathcal{A}, \rightarrow_o)$, and is thus called an* ownership (altering) method. *If m executes as part of event e, e is called an* ownership (altering) event.

Tab. 1 summarizes the different features of method calls in AEON and their composition. For example, as shown in the 2nd row, an internal non-`ro` caller — a (non-client) actor method which is not declared as readonly — can issue an event through any (yield or non-yield) field to any (readonly or non-readonly) methods; any return values are not accessible, so returns must be handled via explicit callbacks, yet the caller will not be blocked and the call is treated as an independent event. As per the 5th row, in a similar calling context, asynchronous calls are only permitted through non-yield fields (as yield fields can only be used for event calls), and will not block yet, still execute as part of the same ongoing event (included). Clearly the only distinctions made on the caller/calling context are statically determined (internal vs external, readonly vs non-readonly).

Target(s). We refer to the *target* actor of an event (internal or external) as the actor which the event method is called on. However, an event will always be sent to the dominator of the set of actors including target actor and any actors passed as arguments to the event. This dominator is determined by the AEON runtime when the event is issued, following Def. 4.

Note that for simplicity we made it sound so far like all calls on actors were issued directly through actor fields. It is easy to see how all analyses for determining permissible call types can be made also on formal arguments and returns of methods, and in fact any expression (e.g., vector access Line 21 in Lst. 1): `ro` is a characteristic of the called method, the call decorator dc determines whether a return value can be accessed, and the type-based analysis determining whether a field must be tagged as `yield` can be applied to any expression.

3.5 Designing AEON Programs

When designing AEON applications, programmers should keep the DAG structure in mind. That means, knowing which descendants an actor class may share with other actor classes to get a perception of dominators. Understanding at the actor class (*type*) level (i.e., the left side of Fig. 2) suffices mostly. Our experience shows the number of actor classes and their referencing interactions

is limited making this tractable (cf. § 5.4). Some care is required for sharing of actors among actors of the same class, which the type-level DAG does not reveal. While sharing can affect performance — less sharing means more dominators which increases efficiency by allowing more events to execute in parallel — DAG violations are detected and notified by the compiler. Concrete dominator actor *instances* are determined by the runtime based on the current DAG structure. E.g., following the example in Fig. 2, assume the `Steakhouse` actor class has the (unshown) `Street` actor class as a unique parent. Assume that `Cook` and `Customer` actors are not shared among different `Steakhouse` actors. Here, a `Cook` actor's dominator can be a `Steakhouse` actor but can never be a `Street` actor.

The `yield` qualifier is intended for scenarios when programmers do not need serializability including events/calls through such fields, in which case parallelism can be increased by correspondingly annotating them, or when the DAG constraint might get violated.

4 MULTI-ACTOR SYNCHRONIZATION

After providing an overview and basic definitions (§ 4.1), we present synchronization in AEON for a *static ownership* DAG (§ 4.2), followed by a *dynamic ownership* DAG (§ 4.3). In both cases, we introduce synchronization and discuss how AEON achieves serializability [Papadimitriou 1979] of concurrent events' execution. For brevity we omit optimizations (e.g., parallel execution of readonly events) and relaxations (e.g., allowing actors to access their descendants beyond direct children).

4.1 Overview and Basic Definitions

Simply put, AEON leverages D1 and D2 for event serialization via automated locking as follows:

(1) When called by an actor a as part of an event e, an actor a' is first locked for e by a, unless it is already locked for e.
(2) Events traversing the same actor a are serialized at that actor a, and passed in the same order along to any called child actor a' by locking a' in the same order.
(3) When starting a new event by calling an actor a the event is sent to a's dominator, unless the call involves $a_1, ..., a_n$ as arguments: then the dominator is chosen for the set $\{a, a_1, ..., a_n\}$.

Note that the second case of (3) implies that any actor which is passed as argument to a nested call within an event is also dominated by the original dominator of the event start (cf. (2)).

In what follows we will be using these two basic definitions:

Definition 6 (Static and dynamic ownership). *Let (E, Ω) denote a system configuration resulting from an execution of an AEON program, with E the set of issued events and Ω the ownership DAG. We say that (E, Ω) is a* static ownership *configuration if E does not contain any ownership event, otherwise a* dynamic ownership *configuration.*

Definition 7 (Serializability). *Let E denote the set of events in a given execution of AEON. We say that the execution is* serializable *if there exists an* equivalent serial *(i.e., sequential) execution respecting the* temporal *relations among the same set of events E. The temporal condition stipulates that for any two events $\{e_1, e_2\} \in E$, if e_1 precedes e_2 in* real-time *in the concurrent execution, then e_1 precedes e_2 in the equivalent serial execution.*

4.2 Synchronization under Static Ownership

Traditional mechanisms for synchronization like *two-phase locking* (2PL) enforce serializability on linearly-ordered structures without shape constraints while *two-phase commit* is best suited when *majority voting* is necessary for agreement [Bernstein et al. 1987]. Both approaches require non-trivial synchronization which becomes costly across remote parties. Unlike these mechanisms, the synchronization technique developed for AEON exploits the DAG structure to rely on a *two-step*

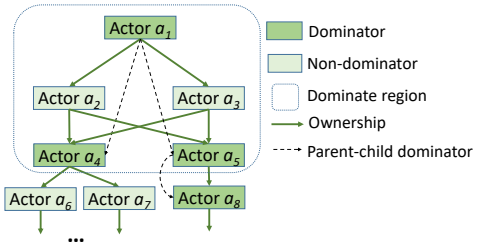


Fig. 5. Example of dominate region of actor a_1: actor a_4 is a child dominator of a_1 because a_2 is its parent and a_2 is dominated by a_1. Note how the child dominator a_4's children actors a_6 and a_7 are *not* included in a_1's dominate region.

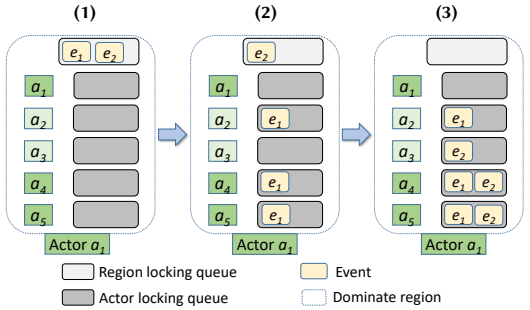


Fig. 6. Basic AEON synchronization. e_1 and e_2 are put into region locking queue (1). e_1 is then dequeued and put into actor locking queues of reachable actors (2). e_2 is then dequeued and put into actor locking queues (3).

locking mechanism and *multiple FIFO queues* implemented at (dominator) actors to enforce *partial ordering across events*. Before we explain the runtime synchronization for multi-actor semantics, we introduce the following crucial terminology:

Definition 8 (Parent-child dominator and dominate region). *For dominators $a_1 \neq a_2$ we say a_2 is the* child dominator *of a_1, or a_1 is the* parent dominator *of a_2, when a_2 is a descendant of a_1, and there is no dominator actor a_3 such that a_2 a descendant of a_3 and a_3 is a descendant of a_1. Given a dominator a_1, its* dominate region *is defined as the set of actors consisting of the (1) actors dominated by a_1 (including a_1) and (2) the child dominators of a_1 (cf. Fig. 5).*

Basic synchronization protocol. First off, AEON uses two types of FIFO queues on every (dominator) actor: (1) a *region locking queue* for the actor's dominate region; (2) an *actor locking queue* for each actor in the region. (1) is used to guarantee that events enter the dominate region in sequential order, while (2) control locks on the corresponding actors.

Intuitively when an event targets a set of actors, it executes from the corresponding dominator a_1 downwards. Following the DAG, the runtime can find paths from a_1 to those actors. At every dominator "below" a_1, other events may arrive. Thus every dominator serializes events in its dominate region and then dispatches them to its child dominators:

(1) When an event arrives at a dominator a_1 it needs to lock that dominate region at first. To that end the event is first placed into the region locking queue of a_1.
(2) When the event becomes the head of that queue, it is removed and put into the actor locking queues of: (i) a_1 and (ii) any actor a_2 reachable from a_1 in this dominate region. Note that actors in (ii) are only "pre-locked" in case the event executes in them.

If a_2 is a child dominator, the event is forwarded to a_2 and put into its region locking queue. An event executes in the involved actors in a dominate region. When the event finishes execution in those actors, it is removed from all actor locking queues (it is enqueued on) on the dominator. The event can propagate downwards and execute in actors beyond this region as per the event logic.

Illustration. In Fig. 6, event e_1 tries to execute on actor a_2 while e_2 is targeting actor a_3. Initially, e_1 and e_2 are placed into a_1's region locking queue (1). e_1 is removed from the region locking queue and put into actor locking queues of a_2, a_4, and a_5 (2) since a_4 and a_5 are reachable from a_2 while a_3 is not, so a_3 is not affected by e_1. e_2 is then removed from the region locking queue and put into actor locking queues of a_3, a_4, and a_5 (3) since a_4 and a_5 are reachable from a_3. Finally when e_1 becomes the head of a_2's actor locking queue, e_1 can execute in a_2, but when e_1 tries to execute in a_4 and becomes the head of a_4's actor locking queue, e_1 will be forwarded to a_4 (i.e., a_4 is a

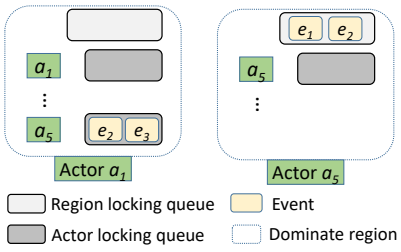


Fig. 7. Event execution order in child dominator's region. a_5 is a child dominator of a_1. e_2 is the head of the actor locking queue of a_5 in a_1's region. Thus e_2 will be put into the region locking queue of a_5.

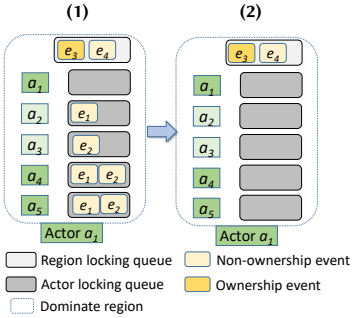


Fig. 8. Ownership event e_3 waits until all previous events (e_1 and e_2) finish (1) and starts to execute (2). e_4 can only be put into actor locking queues after e_3 is done and removed from the queue.

child dominator). Note that in (3) e_1 and e_2 are the heads of actor locking queues of a_2 and a_3 respectively, so they can execute in parallel in a_2 and a_3.

Observe that the actor locking queue of child dominators in their parent dominator's region is used to control the order in which events enter child dominators' regions from parent dominators' regions. Those events still need to obtain exclusive locks on the former regions separately since they can access a dominate region either from parent dominators' regions or as direct targets. As Fig. 7 shows, a_5 is a child dominator of a_1. Events e_2 and e_3 are entering a_5's dominate region from a_1's dominate region, while e_1 lands in a_5's dominate region directly. When e_2 becomes the head of a_5's actor locking queue on a_1, e_2 is forwarded to a_5 and put into its region locking queue. When e_2 finishes execution in all actors in the dominate region of a_1, e_2 is removed from *all* actor locking queues in a_1's region. Note that e_2 may be still executing in other actors in a_5's region but finished executing in a_5, and that these actors in which e_2 is executing are not included in a_1's region.

Serializability in a single dominate region. We argue for serializability by showing that no two events in a given execution of AEON are *interleaved*. Since an event e can only execute in an actor a when e is the head of a's actor locking queue, and e will not be removed until it completes execution within the dominate region, subsequent events are put on hold.

Consider the event execution order in actors of one dominate region. According to the synchronization protocol above, when an event e_1 becomes the head of a region locking queue, it will be immediately put into the actor locking queues of any actor a_1 that e_1 is targeting and actors that can be reached from a_1 in the dominate region. Since events follow the ownership DAG top-to-bottom to access actors, e_1 can only access actors reachable from a_1 there. Assume there exists an actor a_2 in the same dominate region that e_1 can access and a_2 is not reachable from a_1. Then (i) a_1 must be reachable from a_2 or (ii) there is a_3 which both a_1 and a_2 are reachable from. In (i) e_1 will be put into the actor locking queue of a_1 when it tries to lock a_2; in (ii) e_1 will be put into actor locking queues of both a_1 and a_2 when it tries to lock a_3. In Fig. 6 (3), e_2 will execute in a_3 and it is put into actor locking queues of a_4 and a_5 since a_4 and a_5 are reachable from a_3.

Finally, as the region locking queue is a FIFO queue, any two events executing in a dominate region are dequeued in sequential order. As an event is placed into actor locking queues of all accessible actors when it is thus dequeued, two events' order in any actor locking queues will be the same as their order in the region locking queue. Thus any two events in a dominate region will execute in any actor in the same order and will not interleave.

Serializability across dominate regions. By following the DAG an event can only access dominate regions which are reachable from its current dominate region. By showing that two events have

the same execution order in a dominate region and its child dominator's region, we can inductively argue that those two events have the same order in all dominate regions.

Assume two events e_1 and e_2 are accessing dominator a_2 from an actor a_3, which is dominated by a_1. Then a_2 is a child dominator of a_1 and is in the dominate region of a_1. e_1 and e_2 are put into the actor locking queues of a_2 when they are dequeued from the region locking queue of a_1. When e_1 (or e_2) becomes the head of a_2's actor locking queue (on a_1), it is forwarded to a_2 and put into a_2's region locking queue. Clearly, e_1 and e_2 are in the same order in a_2's region locking queue as in a_2's actor locking queue (on a_1), because e_1 will not be removed from a_2's actor locking queue (on a_1) before it finishes execution in a_2. In Fig. 7, a_5 is a child dominator of a_1. e_2 is before e_3 in a_5's actor locking queue on a_1. e_2 will thus be put in a_5's region locking queue (on a_5) before e_3.

Unlocking actors and early unlocking in async methods. A dominator manages the locking and unlocking of all actors in his dominate region. Then all actors' locking queue are placed on the dominator. When an event becomes the head of an actor a locking queue, this event is locking a and can execute on a. When the event finishes its execution on a, a's dominator is informed. The dominator, in turn, tags the event as *unlock* in a's actor locking queue. If the event is not locking any ancestor of a, the event is removed from a's actor locking queue. Once an event is no longer in any actor locking queue on a dominator, we say the event left the dominate region.

Async methods allow early unlocking in dominate regions. Assume an event e makes an `async` call from actor a to a'. e continues execution on a without waiting for a' to be locked. When e finishes execution on a, the dominator is told to unlock a even if e is still executing on a'.

Event commit. When an event e updates an actor a, e does not modify a directly. e instead creates a copy of a and updates that. Once e releases the lock on a, the subsequent event e' creates a copy of a from e's copy. When an event finishes all execution, it needs to commit in all actors it executed on in the reverse accessing order. This guarantees that an event e commits in an actor a after e finished executing and committed in all a's descendants. When e commits on a, a is overwritten by e's copy of a. If e fails midway, all e's copies and following events' copies are discarded.

4.3 Synchronization under Dynamic Ownership

In our model, an event can include several ownership modifications, i.e., addition or removal, of ownership between two actors (field re-assignments incur two modifications).

Synchronization protocol. Ownership events (cf. Def. 5) are checked statically by the AEON compiler. For any ownership modification, the ownership event involves locking both corresponding parent and child actors by locking their dominators' regions. As with non-ownership events, ownership events are put into corresponding region locking queues. However, when an ownership event becomes the head of such a queue, it is not removed and put into actor locking queues (unlike other events). Instead, the event waits until all actor locking queues are drained to finally execute. All following events in the region locking queue are blocked until this ownership event is finished and removed from the queue. Ownership events therefore lock the whole dominate region during their execution. Since no other event can execute in one dominate region while an ownership event is being executed, no other event can observe (or suffer from) the DAG updating process.

As Fig. 8 shows, ownership event e_3 first waits for e_1 and e_2 to finish executing in this dominate region (Fig. 8 (1)). Eventually e_3 is the head of the region locking queue (Fig. 8 (2)) and can execute; it remains in the region locking queue until execution finished. During the execution of an ownership event, the runtime determines actors whose dominators changed as a consequence, or have new dominate regions, and informs them of DAG updates. Only thereafter is the ownership event removed from the region locking queue. If an actor stops being a dominator or has a new dominate region after DAG updates, it forwards events in its region locking queue to the new dominators.

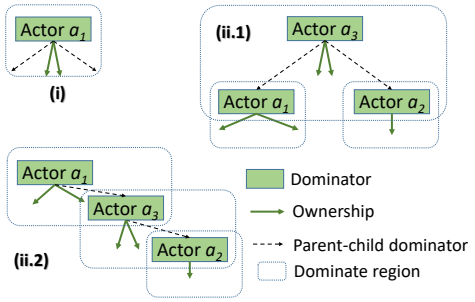


Fig. 9. Ownership events (i) modify ownership between actors in a_1's dominate region; (ii.1) and (ii.2) modify ownership between actor from a_1's dominate region and another actor from a_2's dominate region.

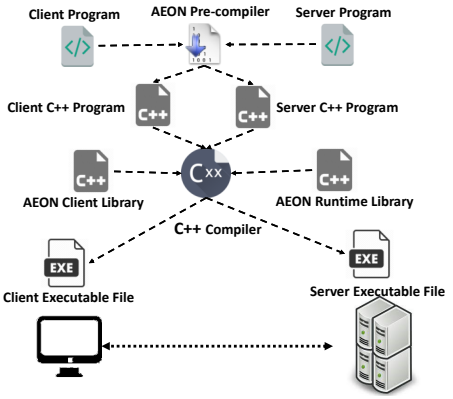


Fig. 10. AEON implementation workflow.

Serializability under dynamic ownership. For serializability under dynamic ownership, we show that ownership events do not affect the safety of event execution. In the following, we show this is the case because *ownership events lock all actors which may be affected by its ownership operations.*

Let us consider the graph induced by the ownership DAG's dominators and parent-child dominator relationships. Clearly the ownership structure is a tree (cf. Fig. 9). Each tree node (dominator in the DAG) serializes events executed in its dominate region. According to the synchronization protocol, ownership events lock the whole dominate regions during their execution. When an ownership event modifies an ownership, it must lock both corresponding parent and child actors by locking their dominators' regions. Fig. 9 depicts the two possible cases: (i) both actors are dominated by the same dominator; (ii) the two actors are dominated by different dominators. For (i), the ownership modification will not affect other dominators (and their dominated actors) since it happens within one tree node (dominator) which is already locked by the event. For (ii), assume the parent and child actors of the ownership are dominated by a_1 and a_2. There are two sub-cases here: (ii.1) there is an actor a_3 with paths to both a_1 and a_2; (ii.2) there is a path from a_1 to a_2. In case (ii.1), potentially affected actors can only be actors dominated by dominators on the path from a_3 to a_1 or from a_3 to a_2, which are are all locked by the event. Similarly, in (ii.2), the event must lock all dominators on the path from a_1 to a_2 and affected actors can only be actors dominated by dominators on this path. Thus all affected actors are locked by this ownership event.

Thus non-ownership events are not affected by concurrent ownership events. As described, an ownership event locks all actors which may be affected by its ownership operations. An ownership event only starts executing when there are no other events executing in the region. For events appearing before the ownership event in the region locking queue, the ownership operations only happen after they leave this region, while events after the ownership event see the updated DAG.

Deadlock freedom (and starvation freedom). As deadlocks only happen when two events access the same set of actors in different orders, they cannot occur in AEON. As explained earlier, any two events will have the same execution order on any common affected actors. Assuming every event finishes in a finite period of time, starvation freedom is guaranteed under static ownership: any event in a region locking queue will eventually become the head and be placed into corresponding actor locking queues. This event will also be the head of actor locking queues and eventually execute in corresponding actors. Starvation is possible under dynamic ownership though if a non-ownership event is continually forwarded to a new dominator following DAG updates via ownership events. However, we remark that such executions are not uncommon for distributed transactional protocols in which a writer interrupts a reader infinitely many times forcing starvation in order to preserve

strong consistency [Herlihy and Sun 2005; Kotselidis et al. 2008]. Note that deadlock freedom is provided even in such a pathological execution as the ownership change terminates although the non-ownership event is starved. We are currently working on a solution mitigating this situation.

5 IMPLEMENTATION AND APPLICATION

We overview AEON's implementation and discuss optimizations, fault tolerance, and application.

5.1 Prototype

AEON is implemented as a pre-compiler and two libraries: (1) *AEON runtime library* and (2) *AEON client library*. As Fig. 10 shows, programmers implement a server program and a client program in AEON. The AEON pre-compiler processes those AEON programs into C++ programs; it handles AEON's special syntax (e.g., `event`) and generates code to handle (potentially remote) method calls on actors. The C++ compiler then generates server executables from C++ server programs and (1), and client executables from C++ client programs and (2). The two AEON libraries are implemented in ≈40 kLoC in C++. The AEON pre-compiler includes ≈3 kLoC in Python and ≈4 kLoC in Perl.

5.2 Optimizations on Remote Communication

In a distributed setting, remote communication between servers is always a major source of runtime overhead. We therefore implemented several optimizations to reduce remote communication.

Since dominators serialize events for actors within their dominate region, frequent communication between actors and their dominator can be expected. As the first optimization, the AEON runtime attempts to collocate on the same server all actors of a dominate region such as to avoid any remote communication between actors and their dominator. As the second optimization, dominators lock actors in batches when possible to reduce the number of locking requests. For instance, when an event requests a lock on a to a's dominator (cf. § 4.2), the dominator, in response, locks at once every unlocked actor that this event is accessing, therefore avoiding individual locking requests to be made by the same event on different actors. As our last optimization, the AEON runtime on a server does not store the whole DAG structure but only the dominate region(s) affecting the actors it is executing. Remote communication can thus be minimized when the DAG structure changes.

5.3 Fault Tolerance (FT)

Actors which failed due to process crashes are resurrected at a new host (or process). To that end AEON takes *consistent snapshots* of actors, and loads the latest consistent state of a failed actor when resurrecting it. Specifically AEON periodically takes global snapshots by sending a snapshot event to the root actor. This event persists the state of all the actors starting from the root actor. The event performs an `async` snapshot method call on all of the root actor's child actors first, then snapshot the root actor's state. The event repeats this process in those child actors. Observe how ensuring consistency — a non-trivial undertaking in general graphs [Aumayr et al. 2019] — is fairly simple in our model as AEON guarantees serializability among all the events, and a snapshot event runs like any other event. Moreover, by means of `async` semantics, a snapshot event can obtain snapshots of the root actor's descendants in parallel, minimizing the impact on system performance. Advanced programmers can configure snapshots and manually trigger and manage them on select parts of the DAG through an API. We assess snapshot overhead in § 6.9.

Events update *copies* of actors' states (cf. § 4.2) to handle event failures, akin to failed/aborted transactions, due to various logical errors (e.g., ownership cycles in recursive data structures, cf. § 3.3) by discarding their updates. If an external event fails, the runtime calls a default error callback method on the corresponding client. Callbacks issued during an event, like any nested event, are only dispatched and executed on clients if and when the event completes server side.

Table 2. Overview of applications re-implemented in AEON, their characteristics, and features used. "Ownership constraint" denotes ownership structure, e.g., actors are organized as DAG or tree. For applications involving multiple clusters, each cluster may have a different constraint; here we pick the strongest constraint.

Application	Type	Ownership constraint	Multiple clusters	Uses `ro`	Uses `async`	Uses int. events	Actor classes	LoC
Game app	App	DAG	No	No	No	No	4	564
Bank account	Finance	Tree	No	No	No	No	4	176
zExpander[a]	Caching	Tree	Yes	Yes	Yes	Yes	3	506
Piazza[b]	Course management	DAG	No	Yes	Yes	No	5	259
BigTable[c]	Store	Tree	No	Yes	Yes	No	3	216
Cassandra[d]	Store	Tree	No	Yes	Yes	No	2	221
Metadata store	Store	Tree	No	Yes	Yes	No	2	552
Silo[e]	Store	DAG	No	Yes	Yes	No	7	1899
Skip list	Data structure	DAG	No	Yes	Yes	No	2	322
LSM[f]	Data structure	Tree	Yes	Yes	Yes	Yes	2	1587
B+ tree	Data structure	Tree	No	Yes	Yes	No	2	1457
B tree	Data structure	Tree	No	Yes	Yes	No	2	1437

[a] [Wu et al. 2016], [b] [Technologies 2016], [c] [Chang et al. 2008], [d] [Lakshman and Malik 2010], [e] [Tu et al. 2013], [f] [O'Neil et al. 1996]

5.4 Application Study

We investigated a dozen applications of different types (e.g., caching, storage, data structures) and re-implemented their core functionalities with AEON. This section briefly reports on experiences gathered. Tab. 2 summarizes characteristics of applications and AEON features they used.

Consistency. Though weak consistency may suffice for many applications (e.g., social networks), strong consistency (i.e., serializability) is necessary in many others to guarantee correct execution, e.g., to transfer money from one bank account to another without loss or duplication, or to insert two items into a B+ tree concurrently. Tab. 2 shows 12 applications that need serializability; 2 of those (Cassandra and game app) use several consistency levels. For all applications needing serializability, we were able to naturally organize actors in a DAG, making AEON an ideal tool to implement them. For example, in a multiplayer game (Lst. 1), users can manipulate their avatars (i.e., cooks) to finish tasks in a steakhouse. Assuming cooks (players) are coordinating to prepare dishes for one table in the steakhouse, and each cook is only responsible to prepare one kind of food. Programmers can implement different variants of this functionality for different consistency requirements. First assume cooks can decide which kind of dishes to cook by themselves, but each cook must use different ingredients in their dishes. Then a cook cannot choose an ingredient that is already used by other cooks. Then serializability is required. Otherwise, it is possible for customers on a table to be served the same dish twice. Programmers must put all cooking method calls (i.e., sync or async) in a single event. If there is no constraint on used ingredients, programmers can simply call event methods on individual `Cook` actors. Here, the cooking is executed independently and can achieve much higher throughput, which we demonstrate shortly in § 6.

Method calls. Reviewing various types of applications has shown the benefits of *readonly events*, especially in data store systems and data structures (e.g., BigTable, Cassandra, B+ tree, skip list) as readonly requests constitute a major portion of requests in such applications. In AEON, multiple readonly events can execute concurrently even within the same actors, which enhances throughput. Most of the applications we investigated (10 out of 12, cf. Tab. 2) use readonly events.

Async method calls do not break serializability yet can enhance parallel execution within an event. Cassandra is a typical example – write requests need to update all `ReplicaTable` actors from

the `MasterTable` actor, and can only return thereafter. Intuitively, async calls make particular sense in AEON given the streamlining for DAG-based ownership: actors often have several children of the same type, and many updates across these children can be parallelized.

Distributed applications often consist of multiple clusters, which achieve different functionalities. These clusters are not necessarily in a hierarchical relationship to each other. E.g., LSMs [O'Neil et al. 1996] include two separate B+ trees. Forcing a hierarchy to connect them can hamper performance (and is not necessary, cf. § 3.3). The solution to process requests accessing multiple clusters (e.g., both trees) is trigger "new", *internal events*. Yield fields can be used to retain references to actors of other clusters, and to issue such events. As Tab. 2 shows, this is exactly what happens in the two applications with multiple clusters. Admittedly, several systems remain rather basic; adding further features (e.g., more elaborate task/job/node trackers in MapReduce) is likely to add more clusters, and thus increase the use of yield fields and internal events.

The next problem is to return results to clients (or internal actors) if their requests are served by multiple clusters, thus involving multiple events. Though *callbacks* may be less simple to use for programmers than direct return values (and we consider adding futures later), they solve the problem of accessing multiple clusters efficiently, and can deal with multiple return values.

Note that programming with only `yield` fields and resulting internal events and callbacks is the same as the original actor model with "singleton" one-way asynchronous messages.

6 EVALUATION

In this section we evaluate the performance of (applications written in) AEON, by comparison to several related state-of-the-art systems and frameworks.

6.1 Synopsis

Research questions. The goal of this evaluation is to answer the following questions:

RQ1 (§ 6.2): How does AEON's synchronization protocol compare to a basic synchronization protocol like two-phase locking?
RQ2 (§ 6.3): How does AEON compare to a "regular" actor language (no support for multi-actor programming), achieving consistency by manually synchronizing?
RQ3 (§ 6.4, § 6.5): How does AEON compare to using a storage system for shared state which needs to be kept consistent?
RQ4 (§ 6.5): How does AEON compare to actor languages with comparable properties?
RQ5 (§ 6.6–§ 6.8): How well does AEON scale, considering different call types and DAG structures?
RQ6 (§ 6.9): What is the overhead of mechanisms for FT used by AEON?

Comparisons. We use the following systems and languages in our experiments:

C++: For RQ1&RQ2 we use the same basic C++ distributed runtime used in AEON to show the efficiency of AEON's synchronization protocol. For brevity we refer to it simply as "C++".
Akka: For RQ2 we use Akka [Lightbend 2020] as it is a popular actor framework and well-suited for distribution like AEON, yet does not provide built-in support for multi-actor synchronization.
Infinispan: For RQ3, we chose two leading "transactional" distributed storage systems for comparison – the first is the (Java-based) Infinispan [Infinispan 2020] key-value store. Infinispan is one of the most popular systems for caching, being used, e.g., by the JBoss [Red Hat 2020] application server. Infinispan allows programmers to execute multiple read/write operations on key-value pairs as one transaction. However, Infinispan follows the traditional execute-validate mechanism and does not guarantee all transactions will be executed successfully. At least one of two conflicting transactions have to abort, yielding an exception the programmer must handle.

Table 3. Metadata store LoC.

Implementation	LoC
AEON basic	552
HyperDex Warp basic	400
AEON MapReduce	390
HyperDex Warp MapReduce	296

Table 4. Game app LoC.

Implementation	LoC
AEON	564
Infinispan	853
Orleans	434

Table 5. B tree LoC.

Implementation	LoC
AEON original	1370
AEON optimized	1437

HyperDex Warp: For RQ3 we also use HyperDex Warp [Escriva et al. 2015], a next-generation NoSQL store. Programmers can create multiple *spaces* (similar to database tables) and put, delete, update objects in them. Each object is referred to by a unique key. Multiple objects (across spaces) can be accessed in a transactional manner. When the runtime detects concurrent transactions on the same objects, it aborts them and throws exceptions to clients.

Orleans: For RQ4, we use the Orleans [Bykov et al. 2011] distributed programming language which is centered on a notion of actor-like *grains*. Orleans is actively developed by Microsoft, and used in a number of projects including the re-engineered Skype and Halo [Microsoft 2020b] applications. Until recently Orleans did not enforce consistency over multiple grains/actors. Considering the need for consistency in many applications, the version 2.0 [Orleans 2020] started supporting cross-actor transactions. We use transactions only where needed.[2]

Note that many actor languages and extensions have been proposed (several are discussed in § 7) with designs overlapping with that of AEON. Languages and models designed and implemented without support for distribution can however not be compared against here. This is not to say that those could not be extended to a distributed scope, but doing so require addressing specific non-trivial design questions. AEON's model and synchronization protocol have been specifically designed to minimize coordination over the network due to increased latency. Similarly, several actor benchmarks (e.g., Savina [Imam and Sarkar 2014]) are geared towards, and implemented for, concurrent single-process setups and it is not clear whether corresponding applications (e.g., thread ring, dining philosophers) are meaningful in the networked setups targeted herein.

Applications and settings. To compare against Infinispan and HyperDex Warp we have implemented systems with the same features in AEON, or more directly applications with the same functionality as applications built on top of these systems. For brevity we may simply refer to AEON (or any of the approaches compared to) for a given application, rather than specifying both application and approach used. Tab. 3, Tab. 4 and Tab. 5 compare the number of LoC used for the more involving applications across the different approaches. The number of LoC must be treated with caution since each approach uses a different language: AEON is extended from C++, HyperDex Warp provides Python APIs, Infinispan is a Java package, and Orleans is based on C#. In addition, the levels of abstraction also differ. For example, Tab. 3 shows that AEON's programs have more LoC compared to those in HyperDex Warp, yet programmers have to declare actor classes and implement actor class methods in AEON, while they only need to create and access tables with built-in API functions in the more specialized HyperDex Warp. It is thus reasonable to expect that the AEON metadata store implementation requires more LoC compared to HyperDex Warp. We believe that the numbers of LoC show that AEON does not impose high burden on the programmer.

The platform differences similarly affect the performance measurements. However, we believe this effect is not as pronounced as one might expect, especially at larger scales where the overhead of remote communication becomes more important compared to purely local processing speeds (explaining why increasingly many distributed systems are implemented in languages like Java).

[2]Orleans' transactional semantics are controlled by tagging variables whose accesses need corresponding synchronization.

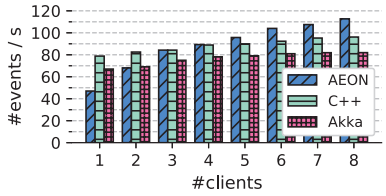


Fig. 11. Binary tree throughput. Unlike AEON, C++ and Akka saturate with growing client count.

Fig. 12. Latency of an app using AEON's protocol vs a two-phase locking in C++ with varying numbers of clients and servers.

In the performance comparison, we assess the performance of each framework via a set of driver applications/workloads on AWS cloud. We investigate scalability of AEON by varying the number of servers, and compare performance of different call types.

Each experiment is run entirely at least 3 times and averaged, with averaging also in runs.

FT We disabled snapshots in AEON when comparing with other systems, as these do not have comparable mechanisms. Orleans does not provide automatic FT in its transactional version. HyperDex Warp claims FT [Escriva and Sirer 2016] yet the source code [HyperDex Warp 2020] includes no FT. Infinispan supports replication for individual key-value pairs, but with no consistency guarantees *across* key/value pairs in transactional mode, so we set its replication degree to 1.

6.2 RQ1: Two-Phase Locking in C++

While AEON has its own DAG-based synchronization protocol to serialize applications, developers can obtain a similar degree of synchronization with other languages, albeit with more efforts, using classic mechanisms such as 2PL. We compare AEON's synchronization protocol against its closest baseline, a 2PL implemented on AEON's basic C++ runtime.

Assume an application consisting of multiple actors, each on a separate server. Clients issue requests consisting in updating two randomly chosen target actors in an isolated manner. In the "C++" implementation, consistency is guaranteed by using 2PL. To avoid deadlocks as in AEON though, all actors are sorted according to their id(entifier)s, and locking happens in increasing order of id; for two chosen actors, any actor with smaller id than either of them has to be locked. Then the client updates their states, and releases all locks. As this microbenchmark scenario has no root, AEON creates an abstract root (cf. § 3.3), placed on the same server as one of the actors. In addition to their respective synchronization protocols, both AEON and C++ implementations include (write-ahead) logging of operations as used typically with 2PL for fault recovery. We compare the client request latencies of the 2PL and AEON implementations on a setup where 1 to 4 clients send their requests to the applications with 1 to 8 actors. In the 1 actor setup, clients only update 1 actor's state. We deploy clients and actors on AWS m1.small instances; each are deployed on their own instance. Fig. 12 shows the two implementations achieve close latencies with 1-2 actors. However, AEON's synchronization protocol clearly outperforms 2PL as the number of actors increases, and even more so as the number of clients increases too.

6.3 RQ2: Manual Synchronization on Binary Trees in Akka and C++

Most actor programming languages such as Akka [Lightbend 2020] focus on providing highly concurrent execution with "simple" asynchronous messaging. AEON is not thought of as a replacement for these languages, but specialized for applications requiring serializability. We demonstrate the benefits of its inherent serializability support over manual synchronization.

Assume multiple clients are issuing requests to a binary tree. Each request randomly picks a path from the root node to a leaf node and accesses all nodes on the path from top to bottom.

Additionally, all those requests must access the tree nodes in the same order. Simple asynchronous messaging cannot guarantee that the arrival order of each request is the same as those requests' send order. One request may leave a parent tree node after another request, and yet arrive at the child node first. A simple, yet correct solution is to serialize the execution of requests by allowing one request access at a time on the whole binary tree.

We implemented a 10-depth binary tree in (1) AEON (155 LoC), (2) Akka Scala (274 LoC) and (3) on AEON's basic C++ runtime (1510 LoC). (3) is used as an intermediate baseline between AEON and Akka to isolate the benefits of C++, which AEON is based on, over the JVM, and those proper to AEON. The binary tree implementation in (2) and (3) serializes all requests via the root node. The tree is deployed on two AWS m1.medium instances. We enforce remote messaging by placing a parent node on a different machine from its children. Clients are on another m1.medium instance.

As Fig. 11 shows, Akka and "C++" outperform AEON with few clients because of the overhead of AEON's serialization protocol. This is due to the metadata and bookkeeping (with multiple queues) leveraged by the AEON protocol for parallelism. However, with increasing numbers of clients, the benefits of this fine-grained synchronization become apparent. AEON provides serializability without sacrificing fine-grained parallelism. The experiment also shows AEON's benefits are due to its synchronization protocol and cannot just be ascribed to C++'s greater runtime efficiency.

We remark that it is possible to potentially improve the Akka (and C++) implementation of the binary tree enforcing the execution order of requests with timestamps or with a distributed locking mechanism (as discussed in § 6.2) or implementing a custom serializable protocol specifically designed for the semantics of the binary tree operations [Ellen et al. 2010]. However, such approaches require a deep understanding of distributed applications and non-trivial implementation efforts; both of which can be avoided by using AEON as it requires no additional code for serializability.

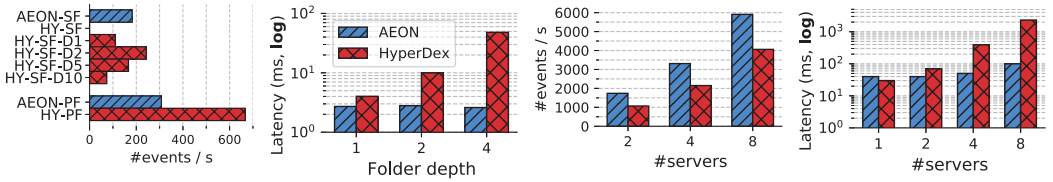
6.4 RQ3: Metadata Store with HyperDex Warp

We compare the metadata store of the Warp Transactional Filesystem (WTF) [Escriva and Sirer 2016] implemented both in AEON and HyperDex Warp. WTF consists of four "tiers": a client library, a metadata store, a replicated coordinator, and storage servers. Only the metadata store requires serializability so clients can update metadata of *multiple* files stored on the metadata store at a time. That is, when a client tries to read, write, or manage a file, it first connects to the metadata store to retrieve file information, and then connects to one of the corresponding storage servers to access the file. WTF supports regrouping of accesses on multiple files, unlike other filesystems (FSs) such as HDFS [Apache 2020], by using "transactions" in the metadata store, thus allowing clients to update the metadata of multiple files (e.g., merge files) together.

Implementation. The metadata store for WTF is implemented using HyperDex Warp. In an FS, each file and folder has a corresponding *inode* containing its metadata. Inodes reflect the hierarchical structure of files and folders. However, HyperDex Warp is similar to a key-value store and cannot inherently support a hierarchical structure. Consequently, HyperDex Warp stores each inode as an object with the path of inode as key, and each folder inode includes the names of its direct child inodes. In contrast AEON can easily support a hierarchical metadata store by simply organizing actors (implementing inodes) accordingly. We compare the performance of the AEON and HyperDex Warp metadata stores via common FS operations. In short, our AEON version outperforms the HyperDex Warp version in most cases. All our experiments use AWS m1.medium instances.

Creating files in the same folder. In this experiment, the metadata store is deployed on one instance while 4 clients are placed on another instance and all create files in the same folder. Fig. 13a shows that AEON (AEON-SF) reaches a throughput of 180 *conflict-free* requests/s.

In contrast, to create a file in a folder in the HyperDex Warp implementation, a client adds the newly created file as a child of that folder. Multiple clients creating files in the same folder all lock



(a) Create files in 1 folder. (b) Rename folders. (c) Open private inodes. (d) MapReduce.

Fig. 13. AEON vs HyperDex Warp metadata store.

the same folder inode resulting in concurrent updates and thus repeating aborts blocking all file creations (HY-SF). To handle concurrent updates in HyperDex Warp, we had to add random delays of 0-1 ms (HY-SF-D1), 0-2 ms (HY-SF-D2), 0-5 ms (HY-SF-D5), and 0-10 ms (HY-SF-D10) periods.

As shown, delays reduce conflicts between concurrent updates and allow progress. We note that the HyperDex Warp implementation can outperform AEON's when we introduce 0-2 ms random delays (HY-SF-D2). Delays however exhibit a trade-off between operation throughput and latency: small delays cannot alleviate conflicts while large delays increase latency. Moreover, selecting the ideal delay is a tedious task since the chosen one must strive despite varying workloads.

Creating files in private folders. With the same setup, clients now create new files in their private folders to avoid any conflict. As Fig. 13a shows, without conflicts, HyperDex Warp's (HY-PF) throughout is about twice as high as AEON's (AEON-PF). In AEON, to create a new inode actor and add it as some actor's child, the runtime does not only need to create the actor, but also has to update the DAG ownership structure, which results in greater latency.

Renaming folders. With the same setup, we now consider folder renaming in the FS. Each client tries to rename their private folder. Fig. 13b shows the results for different directory depths, which indicates the inode level under the client's private folder. Each level includes two child inodes. Observe that AEON has similar performance across depths as it only needs to update the name of one inode, while its competitor has to update paths for all inodes of this folder as it relies on paths of inodes to capture the hierarchy, thus degrading performance and increasing programming effort.

Open private inodes. We now consider the most common operations in an FS: single inode access and update. HyperDex Warp allows clients to retrieve an inode via its path, which makes this operation both simple and fast. AEON also allows clients to access an actor (i.e., inode) via reference directly. In this experiment, we evaluate the scalability and maximum throughput for different numbers of servers. Fig. 13c shows that both AEON and HyperDex Warp scale well for distributed FSs, yet AEON supports higher throughput than HyperDex Warp at any scale, though being more generic. This scenario gauges the performance of AEON when programming in the original actor sense with "singleton messages" (cf. § 3.3), showing AEON's performance is also appealing then.

MapReduce. We show the performance of AEON and HyperDex Warp metadata stores while running MapReduce [Apache 2020; Dean and Ghemawat 2008] jobs over the FS. We do not run the complete MapReduce jobs but only simulate their operations on metadata store: (1) We only implemented the metadata store in AEON as only this component is implemented by HyperDex Warp. No other component requires serializability. (2) Compared to actual data reads/writes and computation in MapReduce, metadata operations have low latency. Performance of store servers and computation are out of scope. We thus focus on manipulation and creation of input, intermediate data, and output files and reading/writing data from/to these. Each server hosts 3 mappers and 1 reducer. There is 1 client submitting 1 job at a time, occupying all mappers and reducers.

Fig. 13d shows the time a MapReduce job takes to finish all operations in metadata store. AEON clearly outperforms HyperDex Warp when multiple servers are used. For a single operation in the

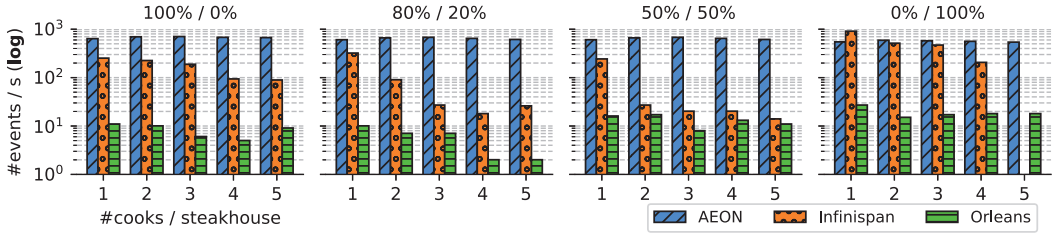


Fig. 14. AEON vs Infinispan vs Orleans game app with varying workloads (*%UseGrill* / *%NewGrill* events).

metadata store (i.e., open a file), HyperDex Warp is only slightly slower than AEON. However, there are, admittedly, other reasons why HyperDex Warp MapReduce is much slower than AEON: (1) HyperDex Warp only provides APIs for Python while AEON is implemented in C++, and Python usually performs worse than C++. (2) HyperDex Warp is a distributed store rather than a complete programming language. Without non-trivial efforts of programmers to optimize important parts of their implementation (e.g., thread pools, socket communication), the performance of a distributed application is affected. AEON comes with built-in optimized versions of these basic features.

6.5 RQ3 and RQ4: Game App with Infinispan and Orleans

We compare AEON vs Infinispan [Infinispan 2020] and Orleans [Orleans 2020] on the game app.

Implementation and workloads. As both Infinispan and Orleans throw exceptions when updates conflict, we introduce random delays of 1-10 ms and retry a certain number of times (e.g., 3×) when clients learn that their transactions are aborted. In the game app (Lst. 1), `Cooks` can (1) put steaks on `Grills` belonging to them (Line 13) generating a *UseGrill* event, and (2) abandon an owned `Grill` to pick a new one (omitted from the code snippet) generating a *NewGrill* event. Note that *NewGrill* events change the ownership DAG structure. We tested four workloads with different ratios of events *UseGrill*/*NewGrill*: (a) 100%/0%; (b) 80%/20%; (c) 50%/50%; (d) 0%/100% (less realistic, used as worst case). We initialize the app with as many `Steakhouses` as there are servers. Each `Steakhouse` owns 10 `Grills`. `Cooks` are equally assigned to `Steakhouses`. Each `Cook` starts with 4 random `Grills`.

Throughput and scalability. We run the AEON, Infinispan, and Orleans game apps each on 8 servers with the four workloads. Fig. 14 compares app throughput across systems. AEON always outperforms Orleans and Infinispan (except in one single-`Cook` case): AEON's throughput is higher and scales much better with an increasing number of `Cooks` per `Steakhouse`. As this number increases, the chance of executing updates on the same set of actors also increases, leading to contention. Those events may be aborted with Infinispan and Orleans, and exceptions thrown to clients. An event may have to be retried several times, increasing latency and degrading throughput.

In our experiments, the performance of Orleans was far behind that of both AEON and Infinispan. We also observed that the performance of transactional execution in Orleans degrades substantially with the same operations compared to non-transactional execution. For instance, with one client the transactional version of Orleans is 20-35× slower than the non-transactional one. While these problems may be mitigated in future versions, they demonstrate the difficulties of providing transactional guarantees with good performance. Similarly to the comparison between AEON and Akka in § 6.3, platform differences alone do not seem to be able to explain the large differences.

Infinispan outperformed AEON at 1 player with workload (d), as *NewGrill* events update AEON's DAG. However, in all other cases, AEON outperformed Infinispan, especially for mixed workloads (b) and (c), where events may conflict with each other on both `Steakhouse` and `Grill` actors. Thus the throughput of Infinispan drops dramatically when more `Cooks` are added to a same `Steakhouse`. Also this difference in trend to AEON is not due to mere technical differences.

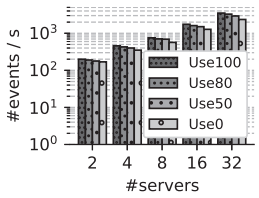


Fig. 15. Game scale-out.

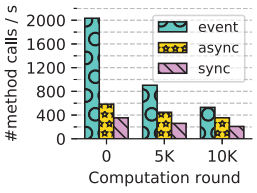


Fig. 16. Calls' throughput.

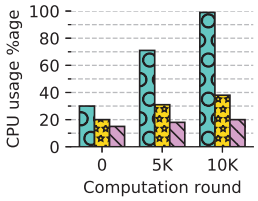


Fig. 17. Calls' CPU usage.

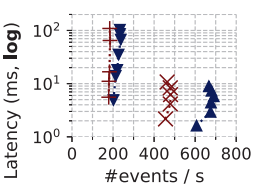


Fig. 18. Optimizing B tree.

6.6 RQ5: Game App Scalability

Scalability. In this experiment, we study how the ownership DAG helps the runtime system execute events in a scalable manner, including dynamic changes of the ownership DAG itself. Fig. 15 depicts the scalability of the AEON game app, with the same *UseGrill*/*NewGrill* workloads as in § 6.5, when the number of servers increases from 2 to 32. As presented in § 4.3, ownership events (*NewGrill*) always lock the dominator (`Steakhouse`), while other events (*UseGrill*) release dominators upon execution. Hence the total throughput decreases as the percent of *NewGrill* requests increases. Since the implementation of the distributed DAG structure guarantees that only related actors are affected in *NewGrill* events, AEON's game app scalability is ensured even with 100% *NewGrill* events.

Call types. In § 5.4, we discussed how to implement a cooking arrangement function in the game app using sync(hronous), async(hronous) and event method calls for different consistency and performance requirements. We measure the performance of each type of method call.

We set the game app with 1 `Cook` and 16 `Grills` deployed on 4 AWS `m1.small` instances, hosting 4 `Grills` each. The `Cook` keeps making `put` method calls on those `Grills`. We simulate `put` requiring a certain amount of computation by adding 0, 5k, and 10k computation rounds in `Grill::put`.

Fig. 16 shows the throughput (i.e., number of method calls executed/s on all `Grills`) for different types of method calls and computation loads. With little computation, async's throughput is around twice that of sync while the throughput of (separate sub-)events is almost 4 times higher than that of async. Thus while async can improve parallel execution within a single event, this improvement is limited by the overhead of synchronization. Serializability is not enforced across (sub-)events, enabling higher throughput. Fig. 17 shows a maxed-out CPU limits this benefit with 10k rounds.

6.7 RQ5: B Tree

Here we substantiate how programmers can benefit from AEON's serializability and readonly events through B trees, popular indexing data structures in storage systems.

In the original B tree, every operation accesses the root node, thereby limiting overall performance. Caching is thus a common optimization to improve performance (of B and B+ trees alike [Aguilera et al. 2008], the difference being only the storage of data also on inner nodes for the former). The cached information of inner nodes allows clients to forward operations to related nodes directly. However, the cache-based B tree implementation requires serializability, or the expected semantics of the B tree can get violated. E.g., inserting a key into a node which is to be merged at the same time could lead to exceeding the maximum number of keys in nodes, or worse, lost data.

Implementation. Thanks to its serializability, programmers can implement an optimized B tree using AEON without extra effort to sidestep the above-mentioned limitation. For comparison, we also implemented the original B tree without caching information of any inner nodes on clients. In addition, both versions can benefit from readonly semantics for read operations. As we have already compared AEON to Orleans, and the latter does not provide readonly semantics, we dive into the benefits of (1) lightweight serializability (enabling efficient optimized B trees without root-node

synchronization of all events) and (2) readonly semantics (acceleration on read-heavy workloads) of AEON. Tab. 5 shows the number of LoC for the original and optimized AEON implementations. The minor difference in LoC is due to the caching of inner nodes in the optimized version.

Scalability. We compare the scalability of the two implementations on a single AWS m1.medium instance. A second m1.medium instance was running from 1 to 8 clients. We follow the YCSB [Cooper et al. 2010] benchmark, and use 2 types of workloads: write-heavy (50% read operations vs 50% insert operations) and read-heavy (95% read vs only 5% insert). Each client starts by issuing 100 read operations as a warm-up. We repeated the experiments 3 times for all 4 combinations of the 2 implementations, OPTimized and ORIGinal, and the 2 YCSB workloads.

Fig. 18 shows mean throughput and mean latency of operations for all setups (legend: ▼ 50%-OPT, ▲ 95%-OPT, + 50%-ORIG, X 95%-ORIG). As expected, the optimized version (1) outperforms the original one for both workloads, scaling to more operations performed with shorter latency, while the original version saturates much faster at its maximum throughput, and (2) shows the benefit of readonly events on the read-heavy workload. The huge differences between the read-heavy and write-heavy measurements cannot be explained by any inherent differences between read and insert operations (i.e., RAM memory read/write speeds are similar [Wikipedia 2020]).

6.8 RQ5: DAG Structure Impact with Piazza

We use Piazza, a university course management system, to assess scalability under different DAG structures. Generally speaking, less sharing among actors increases parallelism in event execution (cf. § 3.5). In Piazza the DAG structure follows the natural hierarchy of a university, i.e., a `University` has multiple `Departments` each responsible for `Courses` which `Students` can register for. In this evaluation, we consider three levels of actor sharing that lead to different DAG structures: (1) One – each student can register for at most one course, i.e., `Courses` do not share `Students`. (2) Dep. – students can register for multiple courses in their department, i.e., every `Course` in a `Department` share the same set of `Students` but `Departments` do not share `Students`. (3) Any – all constraints are lifted so students can register for courses in any department, i.e., every `Department` shares the same set of descendant `Students`.

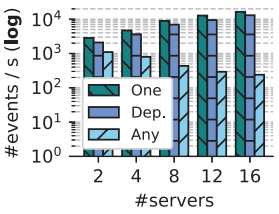


Fig. 19. Piazza scalability with varied DAG structures.

As Fig. 19 shows, scalability is hindered in the Any case since there is only a single dominate region for the entire application, thus eliminating parallel execution. Any is an extreme case used to show the worst-case scenario. If we constrain students to only register for courses in a single department (Dep. case), for instance students enrolled in a master's degree, the application scales well as each `Department` is a dominator for a set of `Students`, thus enabling parallelization. This shows AEON programmers should be mindful of actor sharing at the level of an entire application.

6.9 RQ6: FT Overhead

We use a microbenchmark based on a tree to assess the overhead of snapshots. In the mechanism (cf. § 5.3), DAG depth and the size of actors are two potential factors which may impact the performance of the application when a snapshot is taken. The tree includes 1 root node and 30 tree nodes. We vary the tree depth between 1, 2, and 4. In the first case we have 1 (abstract, cf. § 3.3) root node with 30 child nodes. In the second case, each tree node including the root node has 5 children (5 + 25 = 30 nodes). In the third case, each tree node has 2 children (2 + 4 + 8 + 16). The size of actors at each node is varied between 1kB, 100kB, and 1MB. In Fig. 20, xd-y indicates a tree depth of x and actor size of y. The application is deployed on a single m1.small instance. 30 clients are deployed uniformly on another 3 m1.medium instances. Each client picks a different tree node and keeps sending requests to it.

Fig. 20 shows the latency of clients' requests over time, with the snapshot taken at ≈50s. We do not observe any performance impact when the size of actors is ≤100kB (thus we grayed out the plot lines and omitted measurements with 10kB). However, when the actors' size increases to 1MB, the snapshot results in much higher latency, as serializing large actors requires more time and CPU resources. This evaluation shows that actor size affects snapshot overhead more than DAG depth. Upon recovery, actors are restored in parallel and the DAG structure does not impact recovery performance. Restoring the tree (with 30 nodes) from the snapshot generated in the experiments takes 1.1s, 5.94s, and 49.33s respectively for node sizes of 1kB, 100kB and 1MB. Clearly, the size of the actors determines the time to restore the application.

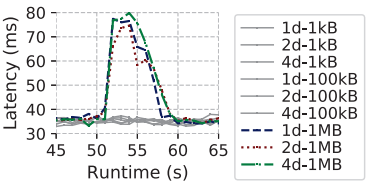


Fig. 20. Snapshot overhead with varied DAG depths and actor sizes.

7 RELATED WORK

We summarize related work on actor-based programming models and specification, as well as serializability in distributed systems (besides works already introduced in § 6.1).

Actor specification and verification. Synchronizers [Dinges and Agha 2012; Frølund 1996] support reasoning about multi-actor interaction through application-specific global constraints on message consumption. Efficiency of implementation has not been investigated. Several works combine behavioral typing with the actor model for guaranteeing coordination safety of actors. Neykova and Yoshida [Neykova and Yoshida 2014] for instance introduce a Python library based on multiparty session types, which allows programmers to specify and verify coordination among actors. Each actor may take multiple roles in different sessions. Several other works apply formal methods to actor languages (e.g., [Desai et al. 2013; Duarte 1999; Kurnia and Poetzsch-Heffter 2012; Poetzsch-Heffter et al. 2011; Summers and Müller 2016]). P [Desai et al. 2013] for instance provides guarantees verified via model checking. A more recent work [Summers and Müller 2016] provides a proof system based on Hoare logic. Its verification phase includes reasoning about individual actors and message passing among them. To guarantee certain invariants two types of permissions are used, *immutable* and *exclusive*, which resemble AEON's readonly and regular access modes respectively. None of the above-mentioned works consider performance in a physically distributed deployment.

Actor(-like) languages, runtime enforcement. EventWave [Chuang et al. 2013] is a distributed language based on actor-like *contexts*, targeting scalable cloud settings like Orleans (cf. § 6.1). EventWave induces a strict single ownership tree among actors/contexts where all requests are serialized at the root node, voiding scalability potential. Moreover, the topology is static, i.e., programs cannot change actor variable bindings. AEON is a follow-up work to EventWave inheriting its notion of context [Sang et al. 2016]. While removing some of the restrictions of EventWave (e.g., relaxing from a tree to a DAG), AEON originally retained most others (e.g., a simple synchronization protocol without dynamic ownership). Just as with EventWave, its programming model and synchronization protocol are only presented in a high-level manner, without clear guarantees, practical considerations, refinements, or discussion of applicability. PLASMA [Sang et al. 2020] adds a second "layer" of programming to languages like AEON or Orleans for system programmers to specify policies guiding the runtime in (re-)distributing actors in the face of workload fluctuations, thus improving placement. Orleans also allows initial placement to be configured [Newell et al. 2016].

Several authors have also leveraged topology restrictions to improve non-distributed concurrent programs. E.g., Golan-Gueta et al. [Golan-Gueta et al. 2011] focus on heap-allocated data structures. It is unclear what performance characteristics would be observed in the networked distributed settings considered herein, which present different bottlenecks than shared memory. Blessing et

al. [Blessing et al. 2017] exploit topology for message delivery in networks, but focus on strict trees instead of DAGs and support causal ordering rather than serializability. The Transactor model [Field and Varela 2005], uses explicit primitives for checkpointing and rollback to reason about composition under failures. Its performance has not been investigated.

Ownership and race conditions. Ownership and related properties have been used to avoid race conditions in shared memory. E.g., Haller and Odersky [Haller and Odersky 2010], Clebsch et al. [Clebsch et al. 2015], and Castegren et al. [Castegren and Wrigstad 2016] use capabilities to avoid race conditions in concurrent executions. Haller and Odersky [Haller and Odersky 2010] propose a type system for Scala, with annotations to denote capabilities of variables. Clebsch et al. [Clebsch et al. 2015] use capabilities to deny certain operations on variables. (More recently, Orca aligned these capabilities with memory reclamation [Clebsch et al. 2017].) Those type systems only allow concurrent access to variables with uniqueness capability to avoid race conditions. Castegren et al. [Castegren and Wrigstad 2016] only allow particular operations on variables with certain capabilities; concurrent accesses to shared resources must be wrapped in explicit locking instructions. These works focus on deadlock freedom and race condition avoidance yet none provides serializability across multi-actor interactions, as AEON, even less in networked settings.

Varela and Agha [Varela and Agha 1999] hierarchically group actors into *casts*, each coordinated by a *director* actor, which itself may belong to another cast. However, unlike in AEON, an actor can only have a single director, and ownership transfer is prohibited. Performance implications are not considered. SafeJava [Boyapati 2003] statically prevents data races and deadlocks by partitioning locks into a fixed number of lock levels manually. Threads take locks according to a specified partial order. SafeJava limits flexibility (static lock partition) and parallel execution (threads must keep locks in order) w.r.t. AEON. Moreover, implementing locks at distributed scale is challenging.

Distributed transactions. Distributed transactional memory (DTM) [Kotselidis et al. 2008] allows programmers to build distributed applications with serializability. We were unable to find any DTM openly available for a performance comparison. Several seminal works use transactions to implement some form of strong consistency for actors, e.g., Chocola [Swalens et al. 2018], Domains [Koster et al. 2015], however focusing on single processes. Using transactional stores for consistency-sensitive shared state is an alternative, explored in § 6 via Infinispan [Infinispan 2020] and HyperDex Warp [Escriva et al. 2015], showing favorable results for our approach.

8 CONCLUSIONS

This paper has presented a variant of the actor model, implemented in our AEON language, specialized for networked distributed setups where actors commonly communicate over the network. Our model (a) enforces serializability and deadlock freedom for multi-actor interaction, while (b) enabling a high degree of parallelism. This sweet spot is achieved by using strongly decentralized synchronization for server-side applications following a DAG-based structure of actors. We have empirically demonstrated the scalability of our model and presented its usability of through case studies of wide-ranging applications. We are investigating ways to bypass synchronization in components using only yield references to extend the application scope of AEON, and several relaxations and extensions, e.g., for (safely) passing objects by reference between collocated actors.

ACKNOWLEDGMENTS

We thank the reviewers for their invaluable feedback. This work was supported by NSF grant #1618923, ERC grant #617805, DFG center #1053, SNSF grant #192121, and AWS Credits in Research.

REFERENCES

2020. Akka.NET. http://getakka.net

AEON. 2020. AEON. https://aeonproject.github.io/aeon/aeon_webpages/

Gul Agha. 1990. Concurrent Object-Oriented Programming. *Commun. ACM* 33, 9 (1990), 125–141.

Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. 2008. A Practical Scalable Distributed B-Tree. *PVLDB* 1, 1 (2008), 598–609.

Apache. 2020. Hadoop. http://hadoop.apache.org/

Dominik Aumayr, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2019. Asynchronous Snapshots of Actor Systems for Latency-sensitive Applications. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR'19*. 157–171.

AWS. 2020. AWS. https://aws.amazon.com/

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Sebastian Blessing, Sylvan Clebsch, and Sophia Drossopoulou. 2017. Tree Topologies for Causal Message Delivery. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!'17*. 1–10.

Chandrasekhar Boyapati. 2003. *SafeJava: A Unified Type System for Safe Programming*. Ph.D. Dissertation. Massachusetts Institute of Technology.

Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *ACM Symposium on Cloud Computing, SOCC'11*. 16.

Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming, ECOOP'16*. 5:1–5:26.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4:1–4:26.

Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2016. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures* 45 (April 2016), 105–131.

Wei-Chiu Chuang, Bo Sang, Sunghwan Yoo, Rui Gu, Milind Kulkarni, and Charles Edwin Killian. 2013. EventWave: Programming Model and Runtime Support for Tightly-Coupled Elastic Cloud Applications. In *ACM Symposium on Cloud Computing, SOCC '13*. 21:1–21:16.

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!'15*. 1–12.

Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. 2017. Orca: GC and Type System Co-design for Actor Languages. *PACMPL* 1, OOPSLA (2017), 72:1–72:28.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC'10*. 143–154.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-driven Programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*. 321–332.

Peter Dinges and Gul Agha. 2012. Scoped Synchronization Constraints for Large Scale Actor Systems. In *Coordination Models and Languages - 14th International Conference, COORDINATION'12*. 89–103.

Carlos H. C. Duarte. 1999. Proof-theoretic Foundations for the Design of Actor Systems. *Mathematical. Structures in Comp. Sci.* 9, 3 (1999), 227–252.

Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-Blocking Binary Search Trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC'10*. 131–140.

Robert Escriva and Emin Gün Sirer. 2016. The Design and Implementation of the Warp Transactional Filesystem. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI'16*. 469–483.

Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2015. Warp: Lightweight Multi-Key Transactions for Key-Value Stores. *CoRR* abs/1509.07815 (2015).

John Field and Carlos A. Varela. 2005. Transactors: A Programming Model for Maintaining Globally Consistent Distributed State in Unreliable Environments. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'05*. 195–208.

Svend Frølund. 1996. *Coordinating Distributed Objects - An Actor-based Approach to Synchronization*. MIT Press.

Guy Golan-Gueta, Nathan Grasso Bronson, Alex Aiken, G. Ramalingam, Mooly Sagiv, and Eran Yahav. 2011. Automatic Fine-grain Locking Using Shape Properties. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'11*. 225–242.

Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying Thread-based and Event-based Programming. *Theor. Comput. Sci.* 410, 2-3 (Feb. 2009), 202–220.

Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *Proceedings of the 24th European Conference on Object-oriented Programming, ECOOP'10*. 354–378.

Maurice Herlihy and Ye Sun. 2005. Distributed Transactional Memory for Metric-space Networks. In *Proceedings of the 19th International Conference on Distributed Computing, DISC'05*. 324–338.

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI'73*. 235–245.

HyperDex Warp. 2020. GyperDex Warp. http://hyperdex.org/

Shams M. Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control, AGERE!'14*. 67–80.

Infinispan. 2020. Infinispan. https://infinispan.org/

Joeri De Koster, Stefan Marr, Theo D'Hondt, and Tom Van Cutsem. 2015. Domains: Safe Sharing Among Actors. *Sci. Comput. Program.* 98 (2015), 140–158.

Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris C. Kirkham, and Ian Watson. 2008. DiSTM: A Software Transactional Memory Framework for Clusters. In *2008 International Conference on Parallel Processing, ICPP'08*. 51–58.

Ilham W. Kurnia and Arnd Poetzsch-Heffter. 2012. A Relational Trace Logic for Simple Hierarchical Actor-based Component Systems. In *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions, AGERE!'12*. 47–58.

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *Operating Systems Review* 44, 2 (2010), 35–40.

Doug Lea. 2005. The java.util.concurrent Synchronizer Framework. *Sci. Comput. Program.* 58, 3 (2005), 293–309.

Lightbend. 2020. Akka. https://akka.io/

Microsoft. 2020a. Asynchronous Agents Library. https://docs.microsoft.com/en-us/cpp/parallel/concrt/asynchronous-agents-library

Microsoft. 2020b. Who is Using Orleans? https://dotnet.github.io/orleans/Community/Who-Is-Using-Orleans.html

Andrew Newell, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein. 2016. Optimizing Distributed Actor Systems for Dynamic Interactive Services. In *Proceedings of the 11th European Conference on Computer Systems, EuroSys'16*.

Rumyana Neykova and Nobuko Yoshida. 2014. Multiparty Session Actors. In *Proceedings of the 16th IFIP WG 6.1 International Conference on Coordination Models and Languages - Volume 8459*. 131–146.

Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (1996), 351–385.

Orleans. 2020. Orleans. https://dotnet.github.io/orleans/

Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26 (1979), 631–653. Issue 4. https://doi.org/10.1145/322154.322158

Arnd Poetzsch-Heffter, Ilham W. Kurnia, and Feller Christoph. 2011. Verification of Actor Systems Needs Specification Techniques for Strong Causality and Hierarchical Reasoning. In *International Conference on Formal Verification of Object-Oriented Software, FoVeOOS'11*. 289–305.

Red Hat. 2020. JBoss Middleware. https://developers.redhat.com/middleware/

Bo Sang, Gustavo Petri, Masoud Saeida Ardekani, Srivatsan Ravi, and Patrick Eugster. 2016. Programming Scalable Cloud Services with AEON. In *Proceedings of the 17th International Middleware Conference, Middleware'16*. 16:1–16:14.

Bo Sang, Pierre-Louis Roman, Patrick Eugster, Hui Lu, Srivatsan Ravi, and Gustavo Petri. 2020. PLASMA: Programmable Elasticity for Stateful Cloud Computing Applications. In *Proceedings of the 15th European Conference on Computer Systems, EuroSys'20*. 42:1–42:15.

Alexander J. Summers and Peter Müller. 2016. Actor Services. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. 699–726.

Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. 2018. Chocola: Integrating Futures, Actors, and Transactions. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!'18*. 33–43.

Piazza Technologies. 2016. Piazza. https://piazza.com

Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13*. 18–32.

Carlos A. Varela and Gul Agha. 1999. A Hierarchical Model for Coordination of Concurrent Activities. In *Proceedings of the Third International Conference on Coordination Languages and Models, COORDINATION'99*. 166–182.

Wikipedia. 2020. DDR3 SDRAM. https://en.wikipedia.org/wiki/DDR3_SDRAM

Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. 2016. zExpander: A Key-Value Cache With Both High Performance and Fewer Misses. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys'16*. 14:1–14:15.