PROGRAMMING SUPPORT FOR SCALABLE, SERIALIZABLE AND ELASTIC CLOUD APPLICATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Bo Sang

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Patrick Eugster, Co-chair

Department of Computer Science

Dr. Xiangyu Zhang, Co-chair

Department of Computer Science

Dr. Sonia Fahmy

Department of Computer Science

Dr. Daniel G Aliaga

Department of Computer Science

Approved by:

Dr. Chris Clifton

Graduate Program Chair

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my Ph.D. advisor, Professor Patrick Eugster. Under his guidance, I have learned plenty of research skills and how to become a professional researcher. More importantly, he has always provided strong support to me whenever I was frustrated by setbacks and struggled with difficulties in research.

I also would like to express my thanks to Professor Xiangyu Zhang, Professor Sonia Fahmy, Professor Daniel G Aliaga, and Professor Tiark Rompf, for serving in my final exam committee and my prelim exam committee. Their suggestions and feedback on my research work were highly valuable and pertinent.

I was very fortunate to work with many brilliant people. We have published our results in several top conferences. Here, I would like to especially extend my thanks to Gustavo Petri, Masoud Saeida Ardekani, Srivatsan Ravi and Pierre-Louis Roman. I was also fortunate to have had great lab mates who provided me invaluable feedback: Danushka Menikkumbura, Akash Agarwal, Bara Abusalah, James Lembke, Savvas Savvides.

In addition, my gratitude goes to all of my dear friends at Purdue. I am always motivated by their precious encouragement as well as their enthusiasm in/out research. I will keep their names deeply in my mind: Hui Lu, Lianjie Cao, Shandian Zhe, Mingjie Tang, He Zhu, Weihang Wang, Cong Xu, and Zhongshu Gu.

Finally, I dedicate this dissertation to my beloved family members: my parents, Boqing Sang and Yongmei Cai; and my sisters, Cui Wang, Jie Pan, Xinrui Cai and Jie Cai. They are always supportive of all the decisions I make and do their best to help me accomplish my goals. Without their support, I do not think I could have completed my Ph.D.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Sang, Bo Ph.D., Purdue University, August 2020. Programming Support for Scalable, Serializable and Elastic Cloud Applications. Major Professor: Patrick Eugster and Xiangyu Zhang.

Elasticity is an essential feature for cloud applications to handle varying and unpredictable workloads in a cost-effective way on cloud platforms. However, implementing a stateful elastic application is hard, as programmers have to: (1) reason about concurrent execution in the applications (serializability); (2) guarantee the application can process more requests with larger scale (scalability); and (3) provide elasticity management to improve performance and resource efficiency for applications (efficient elasticity management). Unfortunately, addressing all those concerns requires deep understanding and rich experience in distributed systems and cloud computing.

In this dissertation, we provide programming support to help programmers implement their stateful elastic cloud applications in a simpler manner. Specifically, we present AEON, an actor-based programming language, and PLASMA, an elastic programming framework. On the one hand, AEON provides programmers with scalability and serialzability, executing actor-based programs in a serialized manner while still retaining a high degree of parallelism. Meanwhile, AEON can adjust programs' scale via fine-grained live actor migration. On the other hand, PLASMA includes (1) an elastic programming language as a second "level" of programming (complementing the main application programming language) for describing elasticity behaviors, and (2) a novel semantics-aware elasticity management runtime that tracks program execution and acts upon application features as suggested by elasticity behaviors. With these, PLASMA can provide efficient elasticity management to cloud applications.

1 INTRODUCTION

1.1 Thesis Statement

Elasticity is essential to "pay-as-you-go" cloud platforms, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. It is common that workloads of cloud applications changes over time and are unpredictable. In response to variations, an elastic cloud application can scale in and out automatically thus to acquire available resources on demand. With elasticity, cloud users can achieve cost savings by consuming cloud resources as needed, meanwhile maintaining performance and service quality of their cloud applications.

However, it is challenging to develop stateful elastic applications from scratch, as an elastic application needs to have the following key features:

Serializability. Cloud applications usually need to process thousands of (or even more) requests at the same time. It is important to reason about the concurrent execution of those requests. Especially, many requests have conflicting accesses to the same set of the application states. In order to avoid race condition and guarantee correct execution, applications need to process requests with potential conflicts in a sequential manner.

Scalability. To effectively cope with unpredictable workloads, applications must be able to function at different scales. Limited scalability eliminates the benefits with more computing resources (i.e., virtual machines). However, it is especially challenging to achieve scalability and guarantee progress (i.e., deadlock freedom) when adding serializability to applications.

Live state migration. Unlike stateless applications, stateful applications can not adjust scale by replicating/eliminating state on servers in many cases. Instead, they need to migrate state from one server to another server without hampering the performance of applications.

Efficient elasticity. For many applications, especially stateful applications, elasticity management involves more than duplicating state/components of applications on new servers. The state/components in an application may serve different functions and receive unbalanced workloads. The efficient elasticity management must consider the function and current workload of each state/component, the interaction between two state/components, and so on.

There are already some solutions which can help programmers implement elastic cloud applications. AWS Lambda functions [1] and Azure durable functions [2] allow programmers to implement their applications in a set of functions. Those functions are executing independently in response to certain events and can scale in/out (i.e., adjusting the number of running functions) according to the number of incoming events. Programmers can also implement elastic applications in some actor-based programming languages with elasticity support (e.g., Orleans [3] and EventWave [4]). Both language runtimes support automatic scale adjustment for applications.

However, most of those solutions (e.g., AWS Lambda) are designed for stateless applications. Though users can add external storage (e.g., AWS MangoDB) to make them work with stateful applications, this kind of approach usually introduces non-trivial latency on data read/write from/to the external storage. Actor-based programming languages (e.g., Orleans and EventWave) do support stateful applications. However, they fail to achieve scalability with serializability support. Meanwhile, those languages' runtime only provide quite limited elasticity management.

Here we are trying to provide programming support, which is based on the actor model [5,6], to help programmers implement stateful elastic cloud applications. During the last decade, the actor 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. The non-sharing between actors and asynchronous messaging also make actor migration implementation simple, which is the basis of elasticity.

Thesis statement. This dissertation tries to respond to the requirements from programmers who are implementing stateful elastic cloud applications. With a novel actor-based programming language, programmers can implement cloud applications with serializability, scalability and live actor migration support in a simple way. Furthermore, a novel elastic programming framework can help programmers conduct efficient elasticity management to their cloud applications.

1.2 Contributions

The contributions of this dissertation are as follows:

- We design a novel cloud programming language AEON and provide a dynamic semantics and a type and effect system for this programming model. Based on the dynamic semantics and type system, we show how to leverage the DAG-based structure to serialize event execution and ensure a DAG-based structure. We also give the proof sketches of deadlock freedom and serializability for execution of programs in our model.

- We implement a novel synchronization protocol and live actor migration algorithm in the runtime system of AEON. The synchronization protocol serializes the execution of events in a scalable manner, while the live actor migration algorithm can minimize migration overhead on the applications.

- We design and implement an elastic programming framework PLASMA for implementing expressive elastic cloud applications. It adds an elasticity programming language as a second level of programming based on the AEON actor-based programming language. Programmers can customize expected elasticity management for their applications with this PLASMA elasticity programming language. Then PLASMA runtime will conduct efficient elasticity management automatically based on programmers' input.

1.3 Dissertation Organization

This dissertation is organized as follows: the chapter 2 presents the related work of this dissertation. The chapter 3 gives the dynamic semantics and type system of AEON with proofs of deadlock freedom and strict serializability for execution of programs in our model. The chapter 4 discusses the implementation of our programming model and show the performance of AEON with variety applications. The chapter 5 presents the design and implementation of PLASMA and show the benefit of PLASMA with multiple applications. Finally, we have drawn the conclusion in the chapter 6. [1]

[1]The source code of AEON is published on: https://aeonproject.github.io/plasma/webpages/; and PLASMA is published on: https://aeonproject.github.io/aeon/aeon_webpages/. The latest version of this thesis can also be found at: https://aeonproject.github.io/papers/webpages/

2 RELATED WORK

We summarize related work in two dimensions: (1) actor-based programming models and specification, as well as serializability in distributed systems; (2) elasticity management.

2.1 Programming Language and Serializability in Distributed Systems

Actor specification and verification. Synchronizers [7,8] 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 [9] 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., [10–14]). P [13] for instance provides guarantees which are verified via model checking. A more recent work [14] 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 abovementioned works consider performance in a physically distributed deployment.

Actor(-like) languages, runtime enforcement. EventWave [4] is a distributed language based on actor-like *contexts*, targeting scalable cloud settings like Orleans (cf. subsection 4.4.1). EventWave induces a strict single ownership graph (tree) among actors/contexts where all requests are serialized at a single root node, voiding

scalability potential. Moreover, the topology is static, i.e., programs can not change actor variable bindings. Last but not least, EventWave has not been evaluated on or applied to different applications, or compared to other systems.

Several authors have also leveraged topology restrictions to improve non-distributed concurrent programs. E.g., Golan-Gueta et al. [15] 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. Similarly, performance of the Transactor model [16], which uses explicit primitives for checkpointing and rollback to reason about composition under failures, has not been investigated.

Ownership and race conditions. Ownership and related properties have been used to avoid race conditions in shared memory. E.g., Odersky and Haller [17], Clebsch et al. [18], and Elias et al. [19] use capabilities to avoid race conditions in concurrent executions. Odersky and Haller [17] propose a type system for Scala, with annotations to denote capabilities of variables. Clebsch et al. [18] use capabilities to deny certain operations on variables. (More recently, Orca aligned these capabilities with memory reclamation [20].) Those type systems only allow concurrent access to variables with uniqueness capability to avoid race conditions. Elias et al. [19] 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 distributed environments.

With Directors [21], actors are hierarchically grouped 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 [22] 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 (all threads must keep locks in order) compared to AEON. Moreover, implementing locks in a distributed scale is challenging.

Distributed transactions. Distributed transactional memory (DTM) [23] allows programmers to build distributed applications with serializability based on the abstraction of transactional memory (TM) [24, 25]. We were unable to find any DTM openly available for a performance comparison. Several seminal works use TM to implement some form of transactions/strong consistency for actors, e.g., Chocola [26], Domains [27], however focusing on single processes. Using transactional stores for consistency-sensitive shared state is an alternative, explored via Infinispan [28] and HyperDex Warp [29], showing favorable results for our approach.

2.2 Elasticity Management

Stateless/state-agnostic elasticity. Infrastructure-level elasticity services for the cloud are typically provided through auto-scaling [30–33]. The number of VMs etc. can be automatically adjusted based on predefined policies and metrics. To use such infrastructure-level elasticity, cloud applications must follow a *very specific programming model*. E.g., service-oriented programming [34, 35] allows each service of a web application to be scaled in/out via sharding or partitioning [36, 37]. Machine learning techniques have been used for elastic provisioning in such models based on workload change prediction [38, 39]; these however remain coarse-grained.

Serverless computing is a recent trend in elastic cloud programming [1, 40–43]. It allows developers to decompose cloud applications into state*less* functions, deployed and scaled elastically. E.g., AWS Lambda [1] allows users to upload their function code to the cloud; management and capacity planning is done automatically. PLASMA extends the scope of serverless computing to *stateful* applications.

Elasticity for stateful applications. Programming stateful applications in extant serverless computing requires leveraging a storage tier (e.g., database, cross-application cache, network file/object store) to store state across functions [44–49]. Yet relying on such a storage tier fails to exploit inherent data locality (of accessing functions) and thus limits effectiveness of application elasticity. Scaling the storage tier automatically is notoriously hard [3, 4, 50, 51].

Many approaches provide elasticity management for *specific stateful applications* [52–56]. E.g., ElastMan [52] includes an elasticity manager for key-value storage in multi-tier web applications. E-Store [53] and Mizan [56], introduced earlier, realize elastic partitioning for distributed OLTP databases and graph vertex migration for map-reduce based graph processing systems [57–59] respectively.
Ray [60] is a reinforcement learning framework that supports stateful computation with actors. The elasticity management of these solutions is system/application-specific and *fail to provide general solutions.*

Both Locus [61] and Pocket [62] provide a more efficient storage solution for serverless computing. Locus combines cheap but slow storage with fast but expensive storage to achieve good performance and cost-efficiency at the same time. Pocket is an elastic distributed data store for serverless computing that provides similar performance as ElastiCache Redis [63] at a lower cost. While Locus and Pocket tackle the performance of serverless computing from the storage side, PLASMA focuses on elasticity management by, for example, colocating two actors (or functions) that frequently interact.

Azure durable functions [2] allow programmers to implement stateful functions for their serverless applications. Programmers however can not customize their application's elasticity management like they can with PLASMA.

Elastic actor programming languages. The actor programming model [64] allows building applications with scalable (a prequisite for elasticity) relations between entities, well-studied in the context of cloud applications, e.g., in EventWave [4], Or-

leans [50] and AEON [51]. Though these languages support live actor migration, they do not provide automated elasticity management. Previously [65] we sketched the case for elasticity programming, however only providing coarse-grained monolithic constructs instead of fine-grained elasticity conditions and behaviors, and without detailed and evaluated runtime techniques for elasticity action execution.

3 AEON: SCALABLE AND SERIALIZABLE NETWORKED MULTI-ACTOR PROGRAMMING LANGUAGE

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.

Roadmap. The section 3.1 introduces the actor model and distributed programming. The section 3.2 overviews and motivates our programming model. The section 3.3 presents our programming model in more detail. The section 3.4 presents formal semantics and meta-theory for a core subset of AEON. The section 3.5 presents formal proof of properties of AEON. section 3.6 draws a summary for this chapter. [1]

3.1 Background: Actor Model and Distributed Programming

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

[1]The original formal semantics in this chapter was designed by Gustavo Petri and Patrick Eugster with help by myself. In particular Gustavo Petri significantly contributed to the proof sketches for the properties of the AEON programming language. Pierre-Louis Roman contributed the important motivation application.

in the cloud [66] 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 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 (i.e., individual components executing independently) as well as (b) collaboration (i.e., multiple components working together) and achieve a good performance compromise between the two.

Actors to the rescue. During the last decade, the *actor* [6,67] 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 [68] and Scala Actors [69] for Scala/Java, Asynchronous Agents Library [70] and C++ Actor Framework [71] for C++, and Akka.NET [72] for C# and F#. Several actor-based languages such as Orleans [73] 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 [7, 8] allow (constraints on) message compositions to be specified ab-

stractly, independently from actors themselves. Several seminal works propose some form of transactional actors (e.g., [16, 26, 27]).

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 [74], 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* [75] 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 speculative execution or on undoing effects of messages. 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 thus captures many useful application scenarios that require strong consistency, while still supporting a high degree of parallelism and thus scalability. Further, for cases where the DAG structure is overconstraining, actors can issue calls outside it which are decoupled from their parent events.

3.2 A Primer

We introduce the main features of AEON through the example of a mult-player game app that allows players (acting as clients) to manipulate their avatars in an environment and interact with it.

3.2.1 Scenario

This game app shares basic ideas with many simulation games. Listing 3.1 sketches a possible implementation in AEON. We remark that AEON is an extension of C++, and 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) (9) 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 (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 deadlocks or inconsistencies such as multiple acquisitions of the same item. Besides, in a distributed

```
1  actorclass Steakhouse {
2    vector<Cook> cooks;
3    vector<Grill> grills;
4    vector<Customer> customers;
5    ...
6  }
7
8  // Players are cooks
9  actorclass Cook {
10   vector<Grill> grills;
11   vector<Customer> customers;
12   Customer cu;
13   int ckid;
14   ...
15   void cook(int orderSize, int cuid)
16   {
17     cu = customers[cuid];
18     for (int i = 0 upto orderSize) {
19       Steak s = new Steak();
20       for (int j = i upto
             grills.size()) {
21         if (!grills[j].isUsed()) {
22           // 3 options: sync, async,
                 event
23           grills[j].put(s, ckid);
24           // async grills[j].put(s,
                 ckid);
25           // event grills[j].put(s,
                 ckid);
26           break;
27         }
28       }
29   }
30   void grillTimerRings(int gid) {
31     async cu.get(grills[gid].take());
32   }
33 }
34
35 actorclass Grill {
36   yield vector<Cook> cooks;
37   yield Cook cook;
38   Steak steak;
39   int gid;
40   ...
41   bool isUsed() { return cook !=
         NULL; }
42   void put(Steak s, int ckid)
43     { steak = s; cook = cooks[ckid];
           ... }
44   Steak take()
45     { ...; cook = NULL; return
           steak; }
46   void timedOut()
47     { event
           cook.grillTimerRings(gid); }
48 }
49 actorclass Customer {
50   yield vector<Cook> cooks;
51   int cuid;
52   ...
53   void giveOrder(int ckid, int nb)
54     { event cooks[ckid].cook(nb,
           cuid); }
55   void get(Steak s) { ... }
56 }
57 class Steak  { ... }
```

Listing 3.1: AEON code snippet for a steakhouse simulation multiplayer game where players are cooks.

environment, player actors may interact with item actors on remote servers. Unlike concurrent programming [76] on a single server, it is more difficult to implement isolation efficiently in an asynchronous distributed environment.

Consider two cooks in a steakhouse who are responsible for cooking steaks for customers. The order of one table is sent to one cook and this cook 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 have received an order for 6 steaks. Without proper synchronization, each cook may occupy 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.

3.2.2 Actors

AEON's actorclasses, declared for instance at 1 and 9, can be thought of as class-like foundries for distributed objects — actors — that can contain data in the form of fields:

1. Such fields can be of object (class) types, as is the case of 38 declaring a field steak, an instance of class Steak declared at 57. Objects are passed by value so role is guaranteed to reside in the same address space as the actor instance of Player that refers to it.

2. Unqualified fields of actor types can contain references to other actors, which at runtime may be remote. E.g., at 3, field grills contains a vector holding references to Grill actors.

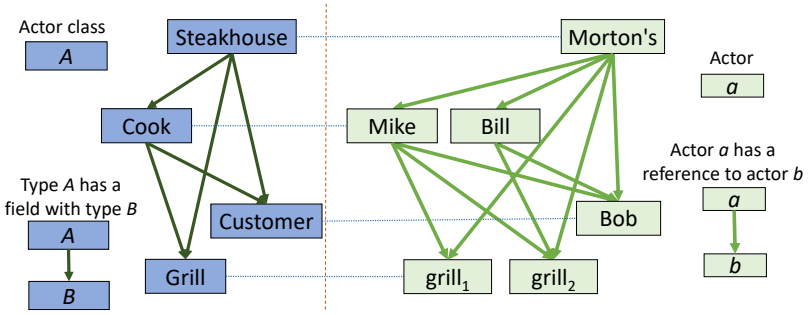


Figure 3.1.: Actor type and actor ownership reference structures in the game app.

3. An actor type field can be declared as a special yield field (37). This is used for internal events which are executed in isolation from the caller, explained further shortly in subsection 3.2.3.

"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). Figure 3.1 depicts the reference structure for both actor classes (left part of Figure 3.1) and actors (right part of Figure 3.1) according to our scenario and Listing 3.1. As we will detail shortly, this graph is essential for efficient synchronization in AEON.

3.2.3 Events

Like other languages designed for implementing Internet-facing applications (e.g., [4, 51,73]), 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(), 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 atomic – more precisely, linearizable with respect to other events. As detailed shortly in section 3.3, and based on the reference DAG structure in Figure 3.1, 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 how the Grill :: put (23) method can be called as part of the Cook::cook 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 *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 *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 which executes after the caller event finished. Details are given in section 3.3.

Other than as events, 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. 23 shows an example of a synchronous actor method call, which needs to wait for the steak to be put on the grill. Conversely asynchronous calls do not wait for a result, and allow execution to proceed immediately, hence allowing for parallelism. An example of asynchronous call is given at 24: the cook puts multiple steaks on grills in parallel.

3.3 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 Figure 5.3, and will be detailed in the following paragraphs. For simplicity we omit from

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

Program Definition	$p \in \mathcal{P}$	::=	$\overrightarrow{aclsd}\ \overrightarrow{clsd}$ main(...){ t}
Actorclass Definition	$aclsd \in \mathcal{AC}lsD$	::=	actorclass $acls$ { $\overrightarrow{fd}\ \overrightarrow{md}$ }
Class Definition	$clsd \in \mathcal{C}lsD$	::=	class cls { $\overrightarrow{fd}\ \overrightarrow{md}$ }
Type	$T \in \mathcal{T}$	::=	$\underline{acls} \mid cls \mid T[] \mid$ 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 $\mid \ldots$

Figure 3.2.: Abstract syntax of AEON$_{mini}$. Underlined terms can refer to actor types.

the syntax all the sequential aspects of C++ which are inconsequential with respect to the ideas presented in AEON.

3.3.1 Execution Model Overview

As mentioned program execution is triggered by the reception 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 and deadlock freedom with respect to all events, even when these access multiple actors. The access to multiple actors is achieved by making sync(hronous) or async(hronous) method calls to actors within an event.

3.3.2 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 and functionality in the form of exported methods, as defined by their classes. Actor method invocations give rise to messages, and internal representations of actors can only be read and/or affected through their methods. Actors are implemented by the means of objects, meaning that their internal states are captured by objects. Unlike an object class, an actor class can also contain actor type expressions, captured by underlining $\underline{T}$ in Figure 5.3. Actors can refer to each other by references. While objects are always passed by value between actors, actors are always passed by reference.

3.3.3 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 discussed at the end of 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 ensures that such conflicting events are serialized, while non-conflicting ones proceed in parallel.

Asserting absence of cycles. 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 do not induce referencing). 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; programmers then have to cater for corresponding exceptions, which are omitted from the formal language and left aside in the following for simplicity.

From descendants to dominators. To ensure deadlock-freedom, AEON guarantees that whenever two calls have the potential of affecting same actors, an order will be established to access these 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.*

Definition 3 (Common descendants)**.** *We say* a_0 *and* a_1 *share* common descendants *iff* $\mathsf{desc}(\Omega, a_0) \cap \mathsf{desc}(\Omega, a_1) \neq \emptyset$*.*

Figure 3.3.: Conditions (i) and (ii) resp. of share definition. Solid edge indicates a child, dashed edge a descendant.

Definition 4 (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:*

$$\begin{aligned}\mathsf{share}(\Omega, a) \;=\; & \Big\{a' \mid \mathsf{children}(\Omega, a') \cap \mathsf{desc}(\Omega, a) \neq \emptyset\Big\} \cup \\ & \Big\{a' \mid \mathsf{desc}(\Omega, a') \cap \mathsf{desc}(\Omega, a) \neq \emptyset \;\wedge a' \notin \mathsf{desc}(\Omega, a) \;\wedge\; a \notin \mathsf{desc}(\Omega, a')\Big\}\end{aligned}$$

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.

In condition (i), as shown in Figure 3.3, actor a' might be an ancestor of a, but still has a direct reference to one of a's descendants. Condition (ii) is needed to avoid considering all ancestors of a in their share set. 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 5 (Dominator)**.** *The* dominator *of actor set* $\mathcal{A}'$ *in* Ω*,* $\mathsf{dom}(\Omega, \mathcal{A}')$*, is defined as*

$$\mathsf{dom}(\Omega, \mathcal{A}') \;=\; \mathsf{lub}\Big(\Omega, \bigcup_{a \in \mathcal{A}'} \mathsf{share}(\Omega, a) \cup \mathcal{A}'\Big).$$

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.

Yield fields. 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 extrude the type-level DAG. In fact all fields of clients are treated as yield fields. As we elaborate on shortly in subsection 3.3.4, yield fields however only support event calls, such as to retain AEON's properties.

3.3.4 Methods and Events

Actors interact via messages induced by invocations to methods on actors. 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. Figure 5.3):

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 asynchronous. Here the caller continues straight after the call rather than waiting until the call completes.

3. Method calls tagged as events are either (i) external(ly issued) asynchronous requests from clients to a (server) application or (ii) internal(ly issued) asynchronous requests from other actors. Both types of events are executed in a serialized manner. Events in (i) define the external (client) API of an AEON application.

Nested events and yield fields. nested event invocations (3.ii), are not a part of the current event. Instead, these will be executed *after* completion of the current event. Just as with asynchronous calls (2), return values of (nested) events can not be accessed. Calls on yield fields *must* be tagged as events or the compiler raises an error.

Results of events are passed to clients via *callbacks* to *client actor methods*. 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 these can execute in parallel on a same actor. The AEON compiler conducts a simple static analysis to ensure that method declarations tagged as ro do not perform assignments. However, readonly methods can also be called asynchronously, 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 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 methods can modify fields of type (2) introduced in subsection 3.2.2 – non-yield actor type fields. As these fields define ownership between actors, methods performing such assignments change the $\rightarrow_o$ relation in the actor DAG $\Omega = (\mathcal{A}, \rightarrow_o)$:

Definition 6 (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.

Table 3.1 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

Table 3.1.: Summary of AEON method call semantics for all types of calls $dc^?x.m(...)$ of methods ro$^?$ T $m(...)$ on fields yield$^?$ with T' an actor type. "External" indicates calls from clients and "internal" all other calls. "Callback" indicates that the caller can not 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

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 only support 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 this event method is called on. However, an event will always be sent to the dominator of the set of actors including target actor and actors passed as arguments to the event. This dominator is determined by the AEON runtime when the event is issued, following 5.

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 24 in Listing 3.1): 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.

3.4 Semantics

We present a formal semantics of a subset of the AEON language, dubbed AEON$_{core}$. AEON$_{core}$ is essentially AEON$_{mini}$ without yield references as well as internal events, but two kinds of terms used only internally. We also focus on event execution in the server-side application. As with AEON$_{mini}$ we concentrate only on the aspects of the language that are new and relevant to guaranteeing the properties enforced by AEON. Importantly, we do consider method calls and assignments, which are important for AEON. We concentrate on aspects relating to actor communication, ownership manipulation, and the overall distributed execution model, and ignore aspects of traditional sequential computation of the C++ language.

Note that we still refer to actor ownership DAG as Ω in the formal semantics. Ω includes dominator information here. Also, we denote it to be the set of currently executing events.

3.4.1 Overview

In Figure 3.4 we present a slightly modified version of the syntax presented in Figure 5.3, where we have removed the yield fields (which are of no consequence to the semantics of the system, since calls through them behave like client requests), and we have added in a box instructions that are only part of the runtime semantics of the language (i.e., they cannot appear in the source code of AEON$_{core}$ programs) and are used as placeholders for synchronization actions.

Since a single actor can execute independently *and* collaborate with other actors, the semantics of AEON$_{core}$ will also be described in two stages or layers:

1. In the first layer, which is named *intra-actor* semantics, we describe how an event executes in a single actor without interacting with other actors.

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

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

Figure 3.4.: Abstract syntax of AEON$_{core}$. Underlined terms can refer to actor types. Boxed terms are only used internally and not by programmers directly.

2. The second layer, called the *inter-actor* semantics, presents the semantics for the composition of all actors in the system. By capturing the *synchronization of multiple actors*, this semantics establishes the global execution of sets of actors.

To aid the reader we provide in Table 3.2 a summary of the different semantic domains, relations, and judgments used in the following.

3.4.2 Intra-actor Semantics

The first layer is a *big-step semantics of single actors*, that is, actors executing in isolated manner. Of course, this semantics can only make progress as long as the currently executing event does not need to communicate with other actors. In essence, this semantics represents the atomic evolution of an event within the boundaries of actions requiring communication.

Table 3.2.: Overview of elements of AEONsemantics.

Notation	Meaning	Section
t	Term under evaluation	3.4.2
ℓ	Label issued for actions requiring actors synchronization	3.4.2
st	Single actor state	3.4.2
am	Access mode of an event (readonly ro or exclusive ex)	3.4.2
$[\![x]\!]$, $[\![\vec{v}]\!]$	Semantics of variables x and sequences of values $\vec{v}$	3.4.2
(st, t, am)	Actor configuration in the intra-actor semantics	3.4.2
$(st, t, am) \overset{\ell}{\hookrightarrow} (st', t', am')$	Label ℓ emitting transition in the intra-actor semantics	3.4.2
a_{id}	Actor identifier ($a.a_{id}$ for given actor a)	3.4.3
ch	Multiset of actor identifiers representing the children ($a.ch$)	3.4.3
q	Queue of requests to be executed ($a.q$)	3.4.3
A	Set of activations representing events currently executing ($a.A$)	3.4.3
(a_{id}, ch, q, st, A)	Actor configuration in inter-actor semantics	3.4.3
e_{id}	Event identifier	3.4.3
$a \xrightarrow{(\ell, e_{id})} a'$	Lifting of the intra-actor semantics to activations in actors	3.4.3
E	Events currently executing in the system	3.4.3
Ω	Encoding of ownership graph	3.4.3
(E, Ω)	Configuration of the whole system (inter-actor semantics)	3.4.3
$\Omega \downarrow e_{id}$	Elimination of event e_{id} from the actor graph Ω	3.4.3
$(E, \Omega) \xrightarrow{(\ell, e_{id})} (E', \Omega')$	Reduction of the inter-actor semantics, where the transition generates label ℓ when executing event e_{id}	3.4.3

To formalize this semantics, we introduce the following syntactic categories:

x	$\in$	$\mathcal{V}ar$	Variables	o	$\in$	$O : \mathcal{F} \rightharpoonup \mathcal{V}al$	Objects
v	$\in$	$\mathcal{V}al \supseteq O \cup \mathcal{V}ar$	Values	a_{id}	$\in$	A_{ID}	Actor ID
st	$\in$	$\Sigma : \mathcal{V}ar \rightharpoonup \mathcal{V}al$	Store	am	$\in$	$\{\mathsf{ro}, \mathsf{ex}\}$	Access Mode

While most of these categories are self-explanatory, we remark that the category of values $\mathcal{V}al$ includes objects and variables $\mathcal{V}ar$. This is just for technical convenience in presenting the semantics. In the runtime of AEON variables are bound in an environment, and are not values themselves. We assume from now on that the language is provided in Administrative Normal Form (ANF) – meaning that only variables

and constants can appear as arguments in calls (for instance method calls follow the syntax $t.m(\vec{v})$). The ANF transformation is standard and it preserves the semantics of the source-level language; we use it only to simplify the semantic rules. It is of no consequence to the actual implementation of the language.

The access mode am represents whether the running event has write permissions over the state and therefore requires exclusive access (ex), or can only read the state (ro). A configuration of this semantics is then simply a tuple of the following form:[2]

$$(st, t, am)$$

st represents the actor's state including the heap, global variables, and a call stack which we abstract away for brevity; t represents the current term under evaluation, and am represents the access mode of the event executing t.

Moreover, we add *labels* used by the following semantic layer *to synchronize* among multiple actors. Labels are drawn from the following grammar:

$$\begin{array}{rcl} \ell \in \mathcal{L}abels ::= & \tau \mid \mathsf{l\text{-}ret}_{a_{id}}(v) & \mid \mathsf{l\text{-}ot}_{a_{id}}(a_n, a_k) \\ & \mid \mathsf{l\text{-}sync}_{a_{id}}(m, \vec{v}, am) \mid \mathsf{l\text{-}async}_{a_{id}}(m, \vec{v}, am) & \mid \mathsf{l\text{-}event}_{a_{id}}(m, \vec{v}) \end{array}$$

The τ label is "silent", meaning there is no consequence for synchronization, and shall generally be omitted.[3] The label $\mathsf{l\text{-}sync}_{a_{id}}(m, \vec{v}, am)$ indicates that the transition issues a synchronous call to actor a_{id} for method m with argument $\vec{v}$ and access mode am. The label $\mathsf{l\text{-}ot}_{a_{id}}(a_n, a_k)$ represents a change in the ownership graph whereby ownership of actor a_n is acquired, and ownership of a_k is relinquished by actor a_{id}. The meaning of the other labels is self-explanatory.

The most important rules of this layer of semantics are shown in Figure 3.5, where the judgment

$$(st, t, am) \overset{\ell}{\hookrightarrow} (st', t', am)$$

[2]For readability we omit the method environment, which is also a static element of the intra-actor configuration, mapping method names to statically bound method implementations.
[3]We use the τ notation for silent labels as customary in process algebra literature.

$$\textsc{Sequence}\quad \frac{(st,t_1,am) \hookrightarrow (st',\mathsf{skip},am)}{(st,t_1;t_2,am) \hookrightarrow (st',t_2,am)}$$

$$\textsc{Sync Actor Call}\quad \frac{[\![x]\!](st) = a_i \qquad [\![\vec{v}]\!](st) = \vec{v'} \qquad m \text{ has access mode } am \text{ in } a_i}{(st, y := x.m(\vec{v}), am) \xhookrightarrow{\mathsf{l\text{-}sync}_{a_i}(m,\vec{v'},am)} (st, y := \text{wait } a_i, am)}$$

$$\textsc{Sequence Label}\quad \frac{(st,t_1,am) \overset{\ell}{\hookrightarrow} (st',t_1',am)}{(st,t_1;t_2,am) \overset{\ell}{\hookrightarrow} (st',t_1';t_2,am)}$$

$$\textsc{Async Actor Call}\quad \frac{[\![x]\!](st) = a_i \qquad [\![\vec{v}]\!](st) = \vec{v'} \qquad m \text{ has access mode } am \text{ in } a_i}{(st, \text{async } x.m(\vec{v}), am) \xhookrightarrow{\mathsf{l\text{-}async}_{a_i}(m,\vec{v'},am)} (st, \text{emit}, am)}$$

$$\textsc{Field Update}\quad \frac{[\![x]\!](st) = o \quad [\![v]\!](st) = v' \quad st' = st[o.f \leftarrow v']}{(st, x.f := v, \mathsf{ex}) \hookrightarrow (st', \mathsf{skip}, \mathsf{ex})}$$

$$\textsc{Actor Return}\quad \frac{[\![v]\!](st) = v' \quad st\text{'s call stack is of size 1}}{(st, \mathsf{return}\ v; t, am) \xhookrightarrow{\mathsf{l\text{-}ret}(v')} (st, \mathsf{skip}, am)}$$

$$\textsc{Ownership Assignment}\quad \frac{[\![v]\!](st) = a_n \qquad [\![\mathsf{this}.f]\!](st) = a_k \qquad st' = st[\mathsf{this}.f \leftarrow a_n]}{(st, \mathsf{this}.f := v, am) \xhookrightarrow{\mathsf{l\text{-}ot}(a_n,a_k)} (st', \text{emit}, am)}$$

Figure 3.5.: Intra-actor big-step semantics (excerpt).

represents a label-emitting transition from the configuration on the left towards the configuration of the right. This big-step operational semantics is driven by the structure of the term t. For brevity, we ignore rules for standard statements such as conditionals and loops (cf. syntax), which are simply inherited from the use of C++ as the supporting language for AEON.

As it can be seen from the Sequence rule, this semantics encodes a big-step evaluation provided that the corresponding command does not emit a label other than τ. The rule Sequence Label enforces that the execution is stopped at the point where the semantics makes a labeled transition. The rule Field Update models field assignments, and it requires that the access mode be ex, thus preventing readonly calls from modifying the heap. We assume a given semantic function "$[\![\ ]\!]$" which, when given a variable returns its value in the current state, and when given a value returns it identically – the fact that the language is in ANF greatly simplifies the definition of this rule. We assume the obvious extension of this rule to sequences of variables and values $[\![\vec{v}]\!]$.

The rules for SYNC and ASYNC ACTOR CALL are similar except that they emit labels, and they inject in the term of the actor a *place-holder term* to prevent execution until synchronization with other actors occurs. Terms wait a_i and emit are the respective *runtime terms* (they cannot appear in a source program) used to indicate that a synchronization with another actor is necessary. These terms, for which no intra-actor semantic rules are given, have the effect of stalling the execution of the actor. The emitted labels serve to synchronize actors in the inter-actor semantics, as shown shortly in subsection 3.4.3. Upon synchronization, the runtime terms are removed allowing the resumption of the intra-actor semantics.

The ACTOR RETURN rule emits a return label on a return statement only if the state of the actor contains a single call frame, meaning that the method was called from another actor or directly from the client. In such case, the return of the method has to be sent to the calling actor or client.

The final rule of Figure 3.5 allows the assignment of an actor to the field of another actor. Due to the ownership discipline of AEON, the fact that the current actor holds a new reference to an actor a_n means that it potentially becomes the parent of a_n if it was not already. Similarly, since a reference to the actor a_k held by the current actor prior to the assignment is overwritten by the assignment, the current actor could lose ownership over a_k. To accommodate for these ownership DAG changes in the inter-actor semantics, the old actor (a_k) held by the field f, and new actor (a_n) assigned to f are propagated in the label for the OT rule of the inter-actor semantics.

We recall that the missing C++ constructs have standard sequential C++ semantics, which is beyond the scope of this paper, and are of no interest to the AEON$_{core}$ extension nor the properties that AEON$_{core}$ enforces. We therefore ignore these constructs throughout the paper.

Decorators			Activations			Requests		
$d \in \mathcal{D}ec$	::=	sync	$\nu \in \mathcal{A}tv$	::=	a-event(e_{id}, t, am)	$req \in \mathcal{R}eq$	::=	r-event$(e_{id}, am, m, \vec{v})$
	|	async		|	a-sync(e_{id}, t, am)		|	r-sync$(e_{id}, am, m, \vec{v})$
	|	event		|	a-async(e_{id}, t, am)		|	r-async$(e_{id}, am, m, \vec{v})$
				|	a-end(e_{id}, am)		|	r-dom(req, a_{id})

Figure 3.6.: Semantic ingredients used in the inter-actor semantics.

3.4.3 Inter-actor Semantics

The second layer of our semantics considers the composition of all actors in the system.

Decorators, activations, queues, and configurations

We will use the semantic category of Decorators shown in Figure 3.6 to indicate the *kinds* of requests made to an actor. These correspond to the ways in which actor methods can be called.

Next, since there could be multiple readonly events executing in a single actor, we need a pool of *activations* representing different threads executing readonly methods within an actor. Since at any point there could be at most one *exclusive* access activation in an actor, we use the *activation set of an actor as lock*. The definition of Activations, where we assume that the metavariable $e_{id} \in E_{ID}$ represents a unique event identifier, is shown in Figure 3.6. The first three cases represent a running activation of a top-level event, an async call, and a sync call respectively; the fourth case represents an activation that has terminated or is currently inactive in the actor, but whose event is kept active to preserve the "lock" which cannot yet be safely released.[4] We will use the metavariable A to represent sets of activations throughout the paper.

The next ingredient we need to consider is the *request queue* that each actor uses to buffer calls received. This queue will contain requests sampled from the grammar

[4]We remark here that activations should also include the local environment of running actors – the call stack. We have abstracted it in the interest of simplicity.

of REQUESTS shown in Figure 3.6. A request $\mathsf{r\text{-}event}(e_{id}, am, m, \vec{v})$ represents a call for a new event e_{id} starting its execution with method m with arguments $\vec{v}$ and access mode am. Requests of the forms $\mathsf{r\text{-}sync}()$ and $\mathsf{r\text{-}async}()$ represent requests to execute synchronous and asynchronous methods for event e_{id} respectively, and the last kind of request, $\mathsf{r\text{-}dom}(req, a_{id})$ denotes that an event attempting to execute in actor a_{id} needs to lock the current actor (which happens to be the lub of a_{id}). This request serves only to lock the current actor for event e_{id}, which enables the event to start execution in its target actor. Finally, a request queue is simply a list of requests, and it will be ranged over with the metavariable $q : [\mathcal{R}eq]$, where the bracket notation represents a list containing elements of type $\mathcal{R}eq$. We can now define actor configurations as considered in this layer. An actor configuration is a tuple of the following form

$$(a_{id}, ch, q, st, A)$$

where a_{id} is an actor identifier drawn from the set A_{ID}, ch is a multiset of actor identifiers representing the children of the actor; [5] the component q is a queue of requests, that is, it has type $[\mathcal{R}eq]$, st is an actor state, and finally the component A is a set of activations representing events currently being executed by the actor. To simplify our notations in what follows we will use the metavariables introduced in this syntax as projections of a record-like definition of an actor. Then, we can reinterpret the definition of configurations as above as a record with the following definition:

$$a \in \mathcal{A}Cls ::= \left[a_{id} : A_{ID},\ ch : A_{ID} \rightharpoonup \mathbb{N},\ q : [\mathcal{R}eq],\ st : \Sigma,\ A : 2^{\mathcal{A}tv}\right]$$

Using this record-like syntax, we will use the metavariables defined for each of the components as a projection of such element in a named actor. We will use a dot to denote such projections. Then, given an actor a the projection $a.st$ denotes the state of actor a, and $a.q$ represents the requests queue of actor a.[6] Moreover, we will use

[5] We use a multiset since an actor can be referenced multiple times, therefore we need to keep track of the multiple "owning" references, similar to reference counting garbage collectors.

[6] By abuse of notation we will use the same convention for activations later.

the notation $a[st \leftarrow st']$ for in-place update of the st field of the actor record a with the new state st' defined as follows

$$a[st \leftarrow st'].x \mathrel{\hat{=}} \mathsf{if}\ x = st\ \mathsf{then}\ st'\ \mathsf{else}\ a.x$$

Notice that in this notation the left occurrence of st represents a member name of the record, whereas the right hand side notation st' is a meta-variable representing a new state. A similar convention applies to all other named components of the actor record.

Finally, a configuration of the whole system is a pair (E, Ω) comprising a set of currently running events $E \in 2^{E_{ID}}$ and a set of actors $\Omega \in 2^{\mathcal{A}Cls}$ representing the ownership graph. It is important to notice here that the definition of of the actor graph Ω in this appendix is more precise than the one introduced in subsection 3.3.3. The vertices and dominators of this representation are implicit throught the children projection of the included actors in the set.

Composition rules

The inter-actor semantics (see Figure 3.7) is given by judgments of the form:

$$(E, \Omega) \rightarrow (E', \Omega')$$

These rules are concerned with the calling and returns of actor method calls. They ignore how an event becomes *activated* in an actor, which will be detailed in Figure 3.9 and discussed in section 3.4.3.

While the semantics of Figure 3.5 considers configurations of single actors as a triple comprising the actor's state, statement and activation mode, in the semantics of this section a single actor could have multiple activations, each of them containing a configuration like above. We therefore need to lift the rules of Figure 3.5 to operate on actors as opposed to intra-actor configurations. To that end, the rule Lift Intra

Lift Intra

$$\frac{a.A = \{\nu\} \cup A \qquad (a.st, \nu.t, \nu.am) \overset{\ell}{\hookrightarrow} (st', t', \nu.am) \qquad a' = a[st \leftarrow st']\Big[A \leftarrow \{\nu[t \leftarrow t']\} \cup A\Big]}{a \xrightarrow{(\ell, \nu.e_{id})} a'}$$

Sync Call

$$\frac{a_1 \in a_0.c_{ch} \qquad a_0 \xrightarrow{(\mathsf{l\text{-}sync}_{a_1}(m,\vec{v},am),e_{id})} a_0' \qquad a_1' = a_1\Big[q \leftarrow q \cdot \mathsf{r\text{-}sync}(e_{id}, m, \vec{v}, am)\Big]}{(E, \Omega \cup \{a_0, a_1\}) \to (E, \Omega \cup \{a_0', a_1'\})}$$

Async Call

$$\frac{a_1 \in a_0.c_{ch} \qquad a_0 \xrightarrow{(\mathsf{l\text{-}async}_{a_1}(m,\vec{v},am),e_{id})} a_0' \qquad a_0'' = a_0'\Big[t \leftarrow a_0'.t[\text{emit}/\text{skip}]\Big] \qquad a_1' = a_1\Big[q \leftarrow q \cdot \mathsf{r\text{-}async}(e_{id}, m, \vec{v}, am)\Big]}{(E, \Omega \cup \{a_0, a_1\}) \to (E, \Omega \cup \{a_0'', a_1'\})}$$

Sync Return

$$\frac{a_1 \xrightarrow{(\mathsf{ret}(v),e_{id})} a_1' \qquad a_1.A = A_1 \cup \{\mathsf{sync}(e_{id}, am, \_, \_)\} \qquad a_0.A = A_0 \cup \{\nu\} \qquad \nu.e_{id} = e_{id} \qquad \nu.t = (y := \text{wait } a_1; t') \qquad a_0' = a_0\Big[A \leftarrow A_0 \cup \Big\{\nu\Big[t \leftarrow y := v; t'\Big]\Big\}\Big] \qquad a_1'' = a_1'[A \leftarrow A_1 \cup \{\mathsf{a\text{-}end}(e_{id}, am)\}]}{(E, \Omega \cup \{a_0, a_1\}) \to (E, \Omega \cup \{a_0', a_1''\})}$$

Async Return

$$\frac{a \xrightarrow{(\mathsf{ret}(v),e_{id})} a' \qquad a.A = A \cup \{\mathsf{a\text{-}async}(e_{id}, am, \_, \_)\} \qquad a'' = a'[A \leftarrow A \cup \{\mathsf{a\text{-}end}(e_{id}, am)\}]}{(E, \Omega \cup \{a\}) \to (E, \Omega \cup \{a''\})}$$

Figure 3.7.: Global rules part 1 (except activations).

Event Call UnShared

$$\frac{e_{id} \notin E \qquad m \text{ has access mode } am \qquad a = \mathsf{lub}(\Omega \cup \{a\}, a) \qquad a' = a\Big[q \leftarrow q \cdot \mathsf{r\text{-}event}(e_{id}, m, \vec{v}, am)\Big]}{(E, \Omega \cup \{a\}) \rightarrow (E \cup \{e_{id}\}, \Omega \cup \{a'\})}$$

Event Return & Commit

$$\frac{a \xrightarrow{(\mathsf{ret}(v), e_{id})} a' \qquad a.A = A \cup \{\mathsf{a\text{-}event}(e_{id}, \_, \_, \_)\}}{(E \cup \{e_{id}\}, \Omega \cup \{a\}) \rightarrow (E, (\Omega \cup \{a'\}) \downarrow e_{id})}$$

Event Call Shared

$$\frac{e_{id} \notin E \qquad m \text{ has access mode } am \qquad a_\ell = \mathsf{lub}(\Omega \cup \{a, a_\ell\}, a) \qquad a_\ell \neq a \qquad a'_\ell = a_\ell\Big[q \leftarrow q \cdot \mathsf{r\text{-}dom}(\mathsf{r\text{-}event}(e_{id}, m, \vec{v}, am), a)\Big]}{(E, \Omega \cup \{a, a_\ell\}) \rightarrow (E \cup \{e_{id}, \}, \Omega \cup \{a, a'_\ell\})}$$

Ownership Transfer

$$\frac{a_k \in a.c_{ch} \qquad a \xrightarrow{(\mathsf{l\text{-}ot}(a_n, a_k), e_{id})} a' \qquad a'' = a'\Big[t \leftarrow a'.t[\text{emit}/\text{skip}]\Big]\Big[c_{ch} \leftarrow (a'.c_{ch}[a_k \leftarrow a.c_{ch}(a_k) - 1, a_n \leftarrow a.c_{ch}(a_n) + 1])\Big] \qquad \mathsf{type}(a) = \mathsf{type}(a_n) \Rightarrow \mathsf{acyclic}(\Omega \cup \{a'', a_n, a_k\})}{(E, \Omega \cup \{a, a_n, a_k\}) \rightarrow (E, \Omega \cup \{a'', a_n, a_k\}))}$$

Figure 3.8.: Global rules part 2 (except activations).

propagates the behavior of the intra-actor semantics by combining the state of the actor a ($a.st$) with one of its activation terms ($\nu.t$) and accessing mode ($\nu.am$), and substituting the appropriate components in the resulting actor configuration. Note also that this is the only rule to produce an arrow of the form $\xrightarrow{(\ell,e_{id})}_{\tau}$ which is used in the premises of the other rules of this figure and denotes the lifting of the intra-actor semantics to activations in actors. The label ℓ is simply propagated, with the added information e_{id} of the concrete event emitting it. Just like for the record representation of actors, the notation $\nu[t \leftarrow t']$ represents the in-place substitution of the term component of the activation ν with the new term t'. This is the only rule to use such a substitution on activations.

The rule Sync Call considers the case where the intra-actor semantics emits a synchronous call label. This rule involves actor a_0 – the source of the call, and a_1 – its target. Notice that we check that a_1 is a child of a_0.[7] As explained before, the only effect of this step is to add the request to the tail of a_1's queue.

The rule Sync Return considers the case where a synchronous call terminates in a_1. In this case we check that indeed an activation ν in the set of activations $a_0.A$ is waiting for the response of a synchronous call to a_0 (it is an invariant that for each event, at most one activation is waiting for a synchronous call from a single actor). In that case, we substitute the wait a_0 term for the actual value produced by the call, therefore allowing the semantics of Figure 3.5 to proceed. Notice that the activation is not removed from actor a_0 upon the return, but rather replaced with an $\mathsf{a\text{-}end}(e_{id}, am)$ activation. This is because this activation acts as a lock for the actor, and it will only be released upon the completion of the whole event e_{id}. Moreover, further calls to the same actor by event e_{id} might re-activate the event, meaning that ownership of the actor cannot be renounced.

The rule Async Call is similar to the one for Sync Call, with the exception that the emit place-holder is immediately removed from the term of the issuing actor when the event is placed in the target's queue. This is because the caller actor needs

[7] This check will be relaxed at the end of the section.

not wait for the termination of the callee, and so its intra-actor semantics can proceed. Consequently, the rule Async Return is also simpler than its Sync counterpart, and in particular it involves a single actor a.

The next three rules consider the case of asynchronous event calls issued to an actor. The rule Event Call Unshared considers the case where the target actor a of the event is its own dominator (cf. the definition of section 3.3). In this case, no other actor needs to be locked, and therefore the rule simply picks a *fresh* event identifier $e_{id} \notin E$, and adds the event in the actor's queue. On the other hand, if the dominator of a is another actor a_ℓ, a new event is added in the queue of a_ℓ as a marker indicating that the actor a_ℓ has to be locked before the event starts executing. This is precisely the purpose of the request $\mathsf{r\text{-}dom}(\mathsf{r\text{-}event}(e_{id}, m, \vec{v}, am), a)$. Notice that in rule Event Call Shared the target actor a remains unmodified by the transition.

The rule Event Return & Commit simply removes the event ID e_{id} from the event set, and it removes any activation appearing in any actor for the event e_{id}. This is encoded with the notation $\Omega{\downarrow}e_{id}$ defined as follows:

$$\Omega{\downarrow}e_{id} = \begin{cases} \emptyset & \text{if } \Omega = \emptyset \\ \Omega'{\downarrow}e_{id} \cup \{a[A \leftarrow (A/\{\mathsf{a\text{-}end}(e_{id}, \_)\}]\} & \text{if } \Omega = \Omega' \cup \{a\} \end{cases}$$

Notice that since we use activations as a lock, this achieves the effect of removing the lock (either readonly or exclusive) from all actors that have been visited by the event e_{id}.[8]

The rule of Figure 3.8 is Ownership Transfer, and it considers the case where an actor is assigned to a field of another actor. This has the effect of modifying the actor graph, and hence, potentially the ownership graph. This rule involves three actors: the parent actor a, the old actor a_k stored in the field that is modified, and the new actor a_n assigned to the field. To accommodate for the possible ownership change, we need to modify the counters keeping track of the children actors owned by a (that is, the $a.c_{ch}$ multiset). In this case, we simply decrement the counter for a_k and increment the one for a_n. If a_k is completely removed from the $a.c_{ch}$ multiset

[8]This rule implies synchronization with all actors used by e_{id}.

(i.e. $a''.c_{ch}(a_k) = 0$) we say that a is no longer a parent of a_k, and symmetrically, if $a''.c_{ch}(a_n) = 1$ we say that a becomes a parent to a_n. An important final remark is that in the special case where the field of the actor a being updated has the same type as the actor a itself (therefore all of a, a_k and a_n have the same type), a runtime check has to be performed to guarantee that no cycles are added to the ownership relation. This check is marked in green in the rule. (A simple static analysis guarantees that no such cycles will happen at runtime in the case of distinct types, cf. section 3.4.4.)

Another important effect of the Ownership Transfer rule is that these transfers could modify the dominator relation, since actors which shared no descendants before, might share descendants after the update. This requires dominators of the sub-graph of the event (performing the update) to be recalculated, and potentially that the event queues of actors whose dominators change be forwarded and merged into the new dominators.

This dominator recalculation procedure is abstracted in the rules by the primitive $\mathsf{adjustDom}(\Omega)$ which can be summarized as doing:

$$\Omega' \leftarrow \mathsf{recalculateLUBs}(\Omega); \forall\ a \in \mathsf{diff}(\Omega, \Omega')\ a.\mathsf{modifyLUB}(a, \Omega')$$

In short, it takes as input the modified ownership graph Ω (generally a downwards closed subgraph of the whole actor graph of the application) and it firstly calculates locally the new dominator actors for Ω. This recalculation is computed by the procedure recalculateLUBs, and its result is stored in Ω'. Subsequently for each actor whose dominator changed as a consequence of the ownership transfer, or that becomes the dominator of another actor after the transfer, we need to perform one of the following operations:

(A) If the actor's dominator changes we send the information of its new dominator actor.

(B) If the actor becomes the dominator of a new actor, it must receive the requests queue of the old dominator of that actor and append it to its request queue.

The algorithm above represents by $\mathsf{diff}(\Omega, \Omega')$ the set of actors that require any of these operations to be performed. The algorithm then calls the synchronous method modifyLUB (reserved to the runtime system) on each of these actors. All the necessary information to update each actor is embedded in the new ownership graph Ω' passed as argument. Note that the invariant of the semantics that each event be enqueued in the actor that dominates the target of the event is preserved, even with non-empty queues of actors whose dominator changes.

Activation rules Let us now concentrate on the activation rules illustrated in Figure 3.9. These rules dictate how requests are removed from an actor's queue, and scheduled for execution.

ACTIVATE

$$\frac{\begin{array}{rcl} & r = \mathsf{r\text{-}async}(e_{id}, am, m, \vec{v}) & \Rightarrow \quad \nu = \mathsf{a\text{-}async}(e_{id}, a.m(\vec{v}), am) \\ a.q = r \cdot q' & r = \mathsf{r\text{-}sync}(e_{id}, am, m, \vec{v}) & \Rightarrow \quad \nu = \mathsf{a\text{-}sync}(e_{id}, a.m(\vec{v}), am) \\ & r = \mathsf{r\text{-}event}(e_{id}, am, m, \vec{v}) & \Rightarrow \quad \nu = \mathsf{a\text{-}event}(e_{id}, a.m(\vec{v}), am) \\ am = \mathsf{ex} & \Rightarrow & a.A \subseteq \{\mathsf{a\text{-}end}(e_{id}, \mathsf{ex})\} \ \wedge \ A' = \{\nu\} \\ am = \mathsf{ro} & \Rightarrow & \mathsf{ex} \notin a.A \ \wedge \ A' = a.A \cup \{\nu\} \end{array}}{(E, \Omega \cup \{a\}) \rightarrow (E, \Omega \cup \{a[q \leftarrow q'][A \leftarrow A']\})}$$

LUB LOCK AND SCHEDULE

$$\frac{a_0.q = \mathsf{r\text{-}dom}(\mathsf{r\text{-}event}(e_{id}, am, m, \vec{v}), a_i) \cdot q' \qquad \begin{array}{rcl} am = \mathsf{ex} & \Rightarrow & a_0.A = \emptyset \\ am = \mathsf{ro} & \Rightarrow & \mathsf{ex} \notin a_0.A \end{array} \qquad a_0' = a_0[q \leftarrow q'][A \leftarrow \{\mathsf{a\text{-}end}(e_{id}, am)\}] \qquad a_i' = a_i[q \leftarrow a_i.q \cdot \mathsf{r\text{-}event}(e_{id}, am, m, \vec{v})]}{(E, \Omega \cup \{a_0, a_i\}) \rightarrow (E, \Omega \cup \{a_0', a_i'\})}$$

CALL PROMOTION

$$\frac{a.q = q_0 \cdot r \cdot q_1 \qquad r \in \{\mathsf{r\text{-}sync}(e_{id}, am, \_, \_), \mathsf{r\text{-}async}(e_{id}, am, \_, \_)\} \qquad e_{id} \text{ does not occur } q_0 \qquad \mathsf{a\text{-}end}(e_{id}, am) \ \in a.A}{(E, \Omega \cup \{a\}) \rightarrow (E, \Omega \cup \{a[q \leftarrow r \cdot q_0 \cdot q_1]\})}$$

Figure 3.9.: Activation rules.

The rule ACTIVATE removes the first request from the requests queue of an an actor a and it generates an activation ν according to the type of request. Notice that in the generated activation the statement takes the form of a method call to the method requested, and that the activation mode and event identifier are simply propagated. Also notice that depending on the access mode am of the request, different conditions

are checked to start the execution. If am is ex the rule requires $a.A$ to be either empty or contain the singleton activation a-end(e_{id}, am), and the resulting activation set A' contains the singleton generated activation ν. On the other hand, if am is ro we verify that there are no exclusive (ex) activations in $a.A$ and add a new activation to the old activation set.

The LUB Lock and Schedule rule performs a similar check in the lub actor, and upon successfully activating the event, it forwards the request to the target actor (by adding it to its queue). Notice that the only effect is to add an activation of the form a-end(e_{id}, am) since the event does not execute in this actor, but the activation is only used as a lock.

Finally, the Call Promotion rule considers the case where a call is made to an actor that was already visited by the event (i.e., e_{id} was already activated in the actor). In this case, the call can be executed regardless of other events that might have arrived at the actor,[9] and is therefore upgraded to the front of the requests queue.

Tying it all together We conclude with a brief summary of the operations involved in executing an event in AEON$_{core}$. The entry point for client requests of the semantics are the Event Call rules. To give the semantics of an AEON$_{core}$ program, we consider that clients can *non-deterministically issue calls* to events (these clients are external to the system and thus not modeled) triggering one of the Event Call rules according to the case of the non-deterministically chosen event. As per the rules, the events get firstly enqueued in the queue of either the target or the dominator actor. Subsequently, one of the activation rules of Figure 3.9 *activates* the event allowing its execution through an interleaving of the global rules of Figure 3.7, which in turn utilize the intra-actor rules of Figure 3.5. Thus the semantics of a AEON$_{core}$ program is the mutually recursive composition of the rules of Figure 3.5, Figure 3.7 and Figure 3.9 with a non-deterministic client issuing events to the system.

[9]Notice that this could only happen if the actor a is its own dominator, since otherwise these events would not be queued in this actor, but in its dominator instead.

Definition 1. *We take the semantics of an AEON$_{core}$ program to be given as the transitive closure of the $\rightarrow$ relation (denoted $\rightarrow^*$) between configurations, starting with a given initial configuration $(\emptyset, \Omega_I)$ containing no events.*

3.4.4 Refinements

This section discusses our type-and-effect system for ensuring the DAG-based ownership structure in AEON programs, and a relaxation to the direct-owner restriction and ownership transfer.

Checking ownership. We use a simple type-and-effect system to check that the ownership graph is indeed a DAG. The main rules are presented in Figure 3.10. Again, we ignore all the constructs that are of no relevance to the property enforced by the type system which is implemented as part of the AEONcompiler. Type-and-effect judgments for statements are of the form

$$\Gamma \vdash t : T, C$$

where, Γ is a standard type environment, mapping variables to their types, t is a term, T is the (standard) type of the term, and finally C is a *set of actor class names*, representing an over-approximation of the *types* of actors manipulated by t. As in C (or Java), signatures of methods are assumed to be given. Then we can simply obtain the type definition of a method by consulting its type as in $\mathsf{type}(cls, m) = T \xrightarrow{C} T'$ where the actor type set C is *inferred* by the type system.

For method definitions the function type is simply labeled with the effects:

$$\Gamma \vdash T'\ m(T)\{t\} : T \xrightarrow{C} T'$$

The type judgment for actor definitions simply collects a series of inequality constraints that require any actor type potentially used by the actor being defined as

$$\frac{\Gamma \vdash t_0 \ : acls,\ C_0 \qquad \mathsf{type}(acls, m) = T \xrightarrow{C} T' \qquad \Gamma \vdash t_1 \ : \ T,\ C_1}{\Gamma \vdash t_0.m(t_1) \ : \ T',\ C_0 \cup C \cup C_1}$$

$$\frac{\Gamma \vdash t_0 \ : \ T_0,\ C_0 \qquad \Gamma \vdash t_1 \ : \ T_1,\ C_1}{\Gamma \vdash t_0; t_1 \ : \ T_1,\ C_0 \cup C_1} \quad \cdots \quad \frac{\Gamma, x : T \vdash \ t \ : \ T',\ C}{\Gamma \vdash T'\ m(T\ x)\{t\} \ : T \xrightarrow{C} T'}$$

$$\frac{\mathcal{C} = \{acls < acls' \mid acls' \text{ occurs in } \mathcal{F}^* \text{ or the effects in } \mathcal{M}^* \text{ and } acls \neq acls'\}}{\text{actorclass } acls\ \{\ \mathcal{F}^*\ \mathcal{M}^*\ \} : \mathcal{C}}$$

$$\frac{\mathcal{C}t = \{\mathcal{C} \mid a \in \mathcal{A}Cls^*\ \&\ a : \mathcal{C}\} \qquad \mathsf{topsort}(\mathcal{C}t) \neq \emptyset}{\mathcal{A}Cls^*\ \mathcal{C}ls^*\ \mathtt{main}(\mathsf{args})\{\ t^*\} : \checkmark}$$

Figure 3.10.: Type-and-effect system to check the DAG structure of the ownership graph (excerpt).

being lower or equal to the one being defined. Finally, for whole programs the type-and-effect system simply checks that the ownership graph is acyclic by calculating a topological sort of the constraints gathered in the declarations of actors. In the case that a topological sort exists we conclude that the type-use hierarchy of the program is acyclic denoted as ✓ in Figure 3.10.

Finally, notice that the constraints added in actor class definitions ignore adding constraints in the case where the two actor types are the same. This allows recursive data types as discussed.

Relaxing the direct-owner restriction Notice that calls could be issued from an actor to any descendant actor, provided that locks are acquired in a top-down hand-in-hand fashion, that is, acquiring the bottom lock before releasing the top one. This is required to ensure deadlock-freedom. Then, allowing the runtime system to perform the locking automatically on a path from the caller to the destination actor allows us to remove the parent-to-child restriction, and in particular, since locks are acquired only once for each event, subsequent calls can directly traverse the actor network without accessing all intermediate descendants. To model that behavior we

use an asynchronous "runtime" locking rule, which simply takes the lock of an actor when the runtime indicates so by following the above mentioned top-down hand-in-hand policy.

AUTO LOCK

$$\frac{a_1 \in a_0.c_{ch} \qquad \mathsf{a\text{-}\_\_\_}(e_{id},\ldots,am) \in a_0.A \qquad \begin{array}{rcl} am = \mathsf{ro} & \Rightarrow & \mathsf{ex} \text{ not in } a_1.A \\ am = \mathsf{ex} & \Rightarrow & a_1.A = \emptyset \end{array} \qquad a_1' = a_1[a \leftarrow \mathsf{a\text{-}end}(e_{id},am)]}{(E,\Omega \cup \{a_0,a_1\}) \rightarrow (E,\Omega \cup \{a_0,a_1'\})}$$

This rule allows us to relax the constraint $a_1 \in a_0.c_{ch}$ in the CALL rules for $a_1 \in \mathsf{desc}(\Omega \cup \{a_0,a_1\},a_0)$, and hence, allows any actor to call any of its descendant actors.[10]

3.5 Properties of AEON$_{core}$

In this section we show more formally how the main properties guaranteed by AEON are established by the semantics of AEON$_{core}$.

Deadlock freedom

Let us begin by formally defining how deadlocks could manifest in AEON$_{core}$.

Definition 1 (Deadlock State)**.** *State* (E,Ω) *contains a deadlock if* Ω *can be decomposed as*

$$\Omega = \Omega' \cup \{a_0,a_1,\ldots,a_n\}$$

where for each $i \in [0,n]$ *we have that there exists an event* $e_i \in E$ *such that: (i)* e_i *occurs in* $a_i.A$*, and (ii)* e_i *occurs in* $a_{i+1}.q$*, considering addition modulo* n*.*

A representation of a deadlock state is shown in Figure 3.11, where the actors involved in the deadlock are marked with the primed names (a_0', a_1', a_2'), and the

[10]This even allows bottom-up calls provided that the required actors are already locked, and thus not subject to deadlocks.

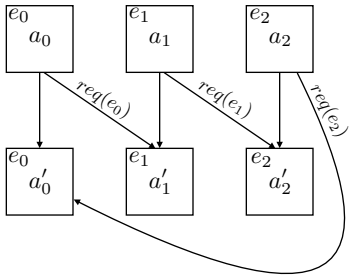


Figure 3.11.: Deadlock state.

arrows represent ownership links with labels representing that there is a request from the source actor to the target actor which is currently enqueued waiting to be activated in the target actor. We also mark in the top-left corner of each actor the name of the event currently holding the lock of the actor. Notice that in the queues of each of these actors there is a request of an event that is currently holding the previous actor in the chain (shown with incoming arrows marked $req(e_i)$), thus closing the cycle.

Theorem 1. *The semantics of AEON$_{core}$ guarantees that no deadlock state can be reached.*

Proof. (Sketch) The proof proceeds by contradiction assuming that a first deadlock state (E,Ω) can be reached. Firstly we observe that whenever an event is executing in an actor, it must be activated in its dominator. This is an invariant preserved by the semantics of AEON$_{core}$. Consider now any of the actors, say a_i, that is issuing a request for event e_i that is enqueued in the queue of an actor a'_{i+1} in the deadlock cycle as shown in the figure above. Also, by the definition of deadlock, there is an event e_{i+1} that is currently holding the actor a'_{i+1}, and its parent a_{i+1}. This, by the definition of share of subsection 3.3.3 implies that all of these actors share a common dominator. Hence, by the above-mentioned invariant, neither event can execute in these actors, since at most one of them is scheduled in the common dominator actor. This contradiction proves the theorem. □

To derive a more formal proof, let us first present some basic remarks about the semantic rules of AEON$_{core}$ presented in section 3.4.

Remark 1. *If an event e_{id} is executing within an actor a, then $e_{id} \in a.A$. That is, only activated events can execute within an actor.*

Proof. It suffices to see that the LIFT INTRA rule requires the event to be in the activation set to execute. All the other global rules do not execute within the actor (i.e. do not make use of the $\hookrightarrow$ transitions of Figure 3.5). Similarly, activation rules do not "execute" within the actor– that is, they don't modify any of the components of the actor other than the activation sets. □

Remark 2. *The only rule that allows removing an event from activations is* EVENT RETURN & COMMIT*, which completely removes the event from the configuration set.*

Proof. Simple observation of the semantic rules.

Corollary 1. *Consider a configuration (E,Ω), an event $e_{id} \in E$, and an actor a_0 such that $e_{id} \in a_0.A$. Moreover, assume that $(E,\Omega \cup \{a_0\}) \xrightarrow{*} (E',\Omega' \cup \{a_0'\})$ where $e_{id} \in E'$. Then we have that $e_{id} \in a_0'.A$. In other words, once an event is activated in an actor, it remains so, until completely eliminated from the event set E'.*

Proof. This property is a simple consequence of 1. Since the only rule that can deactivate an event from an actor eliminates the event from the activations of all actors at once. □

Remark 3. *For any reachable configuration (E,Ω) such that an actor a occurs in Ω we have that either $|a.A| \leq 1$, or every activation in $a.A$ is* ro.

Proof. This is immediate from the antecedents of the activation rules (Figure 3.9). In particular, notice that ACTIVATE and LUB LOCK AND SCHEDULE rules require that the initial activation set be empty in the case of exclusive access (ex) calls, and that no exclusive access event is in the current activation set ($\mathsf{ex} \notin a.A$) in the case of read only (ro) calls. □

Let us refresh the definition of deadlock provided in 1. Recall Figure 3.11 representing a deadlock state. Notice that in the queues of each of these actors there is a request of an event that is currently holding the previous actor in the chain (shown with the incoming arrows marked $req(e_i)$), therefore closing the cycle.

Lemma 1. *For each event e_{id}, and a state (E, Ω) such that there exists an actor $a \in \Omega$ with $e_{id} \in a.A$, whenever $\mathsf{dom}(\Omega, \mathsf{target}(e_{id})) = a_\ell$ we have that $e_{id} \in a_\ell.A$. Note that we write $\mathsf{dom}(\Omega, \mathsf{target}(e_{id}))$ as a slight abuse of notation for simplicity from 5.*

In words, whenever an event e_{id} is executing in any actor, it is also activated in the dominator actor corresponding to the events target actor according to Ω.

Proof. We notice first that the only rules that allow events to be added to the activation of an actor are the rules of Figure 3.9.

Moreover, notice that for each of these rules, an item already existed in the queue of the actor with the necessary event ID. These are the premises of the form $req = \mathsf{r\text{-}\_\_\_}(e_{id}, \dots)$. It is therefore sufficient to show that whenever a request is added to a queue, then either (i) it effectively is the dominator actor, or (ii), the parent actor contains e_{id} in its activations before enqueuing the request. Finally we have to show that no event is removed from an activation prematurely, but this is trivial, since the only rule which removes events from the activations is Event Return & Commit, which removes all the e_{id} atomically from all actors as shown in 1.

Conditions (i) and (ii) above are easy to check by inspection on the rules that can add requests in the queues of an actor. For (i) we have that newly arrived events are always added with a request $\mathsf{r\text{-}dom}(\mathsf{r\text{-}event}(e_{id}, am, m, v), a_i)$ in the LUB actor if it is not directly the target. For (ii) we can check the rules that that add elements to the queues (Synch Call, ASynch Call) require the parent actor to have the event e_{id} in their activations. □

Corollary 2. *For any reachable configuration (E, Ω), actor $a_0 \in \Omega$, and event $e_{id} \in E$ such that $e_{id} \in a_0.A$, we have that $e_{id} \in \mathsf{dom}(\Omega, \mathsf{target}(e_{id})).A$, and there exists a path*

of actors $\mathsf{target}(e_{id}) \cdot a_1 \cdot \ldots \cdot a_n \cdot a_0$ *in* Ω *– considered as a graph – from* $\mathsf{target}(e_{id})$ *to* a_0 *such that for each* $i \in [1,n]$ *we have* $e_{id} \in a_i.A$.

Proof. This is a direct consequence of (i) 1, the fact that Ω is a directed-acyclic graph, and (iii) the fact that events are only ever deactivated (i.e. removed from actors activation sets) by the rule Event Return & Commit which removes the event once and for all form the whole actor set Ω. □

The following lemma states that elements involved in a deadlock cycle must share a common dominator. This is the critical observation to avoid deadlocks.

Lemma 2. *Let us consider a deadlock state* (E,Ω)*, such that* Ω *is a directed-acyclic-graph (DAG) as prescribed by the AEON*$_{core}$ *discipline. Moreover, let us assume that if an event* e_{id} *is either enqueued in an actor* a *(i.e.* $e_{id} \in a.q$*), or it is currently activated in it (i.e.* $e_{id} \in a.A$*), then either* a *is the dominator of* e_{id}*'s target (*$a = \mathsf{dom}(\mathsf{target}(e_{id}),\Omega)$*), or a parent of* a *contains the event* e_{id} *in its activations. Then, we have that all the actors involved in the deadlock (as per 1) have a unique dominator in* Ω*. Formally,* $\mathsf{lub}(\Omega, \{a \mid a \in \textit{deadlock of } \Omega\}) \in \mathsf{desc}(\mathsf{dom}(\mathsf{target}(e_{id}),\Omega))$.

Proof. We proceed by induction on the number of actors involved in the deadlock of state (E,Ω). In particular, we relax the condition of there being a cycle, to requiring only that there is a path in between any two actors (as opposed to a cycle). We consider that there is a bidirectional edge between $a_0 \in \Omega$ and $a_1 \in \Omega$ if there exists an event $e_{id} \in E$ such that $e_{id} \in a_0.A$ and $e_{id} \in a_1.q$, or viceversa. Then, we are interested in the length of the path formed with these bi-directional edges.

In the base case, with only two actors a_0 and a_1, we have that since in at least one of the two actors one event is activated, and the other is enqueued, then both actors share a common ancestor, which is guaranteed by 2. This concludes the case by considering the dominator of this common ancestor in Ω.

In the inductive case we know there is a unique actor that is the dominator of a_0, ..., a_n in Ω. We also know that at least one of these actors has an event activated which is in the queue an additional actor a_{n+1} or viceversa. Then, there are two cases: either

(i) $\mathsf{dom}(a_0,...,a_n,a_{n+1}) = \mathsf{dom}(a_0,...,a_n)$ and we conclude the case, or (ii) otherwise, by the definition of share in section 3.3, we have that the $a_{n+1} \in \mathsf{desc}(\mathsf{dom}(a_0,...,a_n))$. Therefore, by the definition of share, also in section 3.3, the dominator of the actor containing the event that issued the call in a_{n+1} has to also dominate $\mathsf{dom}(a_0,...,a_n)$, which concludes the case. □

We can now prove the deadlock freedom theorem (Theorem 1):

Proof. We consider a proof by contradiction. Let us assume that there is a first state (E,Ω) in a run of the semantics of Figure 3.7 containing a deadlock as per 1. Then, combining 2 and 1 we obtain a contradiction, since both events (one of which requires exclusive access) must be activated in the common dominator. □

Serializability

Let us now show that the execution of AEON$_{core}$ events is *serializable*. We begin by stating some simple lemmas that are necessary for the proof.

We consider an alternative semantics of AEON, given also by the rules of Figure 3.5, Figure 3.7 and Figure 3.9, where we add the additional constraint that there is *at most one* event activated in any configuration (E,Ω), i.e. $\sum_{a\in\Omega} |a.A| \leq 1$. This essentially encodes a semantics of AEON where each event executes in complete

isolation. Let us call this the *serial semantics of AEON* and denote it with the special arrow "$\twoheadrightarrow$". To distinguish configurations generated by this semantics w.r.t. the semantics of section 3.4 we denote actor sets generated by this semantics as $\hat{\Omega}$.

The proof of serializability shows that the serial semantics of AEON$_{core}$ ($\twoheadrightarrow^*$) can simulate the semantics of AEON$_{core}$ ($\rightarrow^*$). To that end we consider a simulation relation between configurations of the two semantics. $\mathcal{A}ct(\Omega)$ denotes the set of events activated in any actor of Ω.

Definition 2 (Simulation Relation)**.** *We define the following simulation relation between a configuration* $(E,\hat{\Omega})$ *of the serial semantics of AEON*$_{core}$ *with a configuration* (E,Ω) *of the normal AEON*$_{core}$*, denoted* $(E,\hat{\Omega})\ \mathcal{R}\ (E,\Omega)$ *iff the following hold:*

$$(i)\ \mathcal{A}ct(\hat{\Omega}) = \emptyset \qquad\qquad (ii)\ (E,\hat{\Omega}) \xrightarrow{*} (E,\Omega)$$

Thus, two configurations, $(E,\hat{\Omega})$ and (E,Ω), are related by our simulation if the former has no active events in any of its actors (no restrictions on the second configuration), and it can reach by executing a number of semantic rules the second configuration. In a nutshell, the configuration $(E,\hat{\Omega})$ represents an alternative trace which could have lead to (E,Ω) where events execute one at a time. This is proved by our simulation proof which amounts to showing commutativity following the depicted diagram. That is to say, each time we start with a pair of similar configurations, whenever the normal semantics of AEON$_{core}$ (the low-level semantics) makes a step, the resulting configuration is still related to the serial one, or there exist a number of steps in the serial semantics that simulate the exact same behavior. In particular, the event sets E in any two similar configurations are identical, meaning that transitions that modify the event set must be matched one-to-one.

Theorem 2 (Simulation)**.** *The relation* $\mathcal{R}$ *defined above is a weak-simulation from the normal (low-level) semantics of AEON*$_{core}$ *to the* serial semantics *of AEON*$_{core}$*.*

Proof. (Sketch) The only interesting case of this proof is the Return & Commit rule, which requires that the serial semantics executes all the transitions of the event making the commit in a single uninterrupted sequence. This proof relies on the fact that all actors that have been visited by this event have been locked since the beginning of the event, therefore no other concurrent event could have modified their state meaning that the serial semantics configuration $\hat{\Omega}$ coincides in this actor with the state in the normal semantics before the execution of the committing event starting at Ω. Since all actor locks are taken in an ordered top-down fashion, and are not released until commit, we have that the state of all these actors in the serial config-

uration coincides with the non-serial configuration before the execution of the event. Moreover, these actors are all free. A simple commutativity argument based on the exclusive access to actors that are locked allows us to conclude that the event can be executed entirely from beginning to commit preserving the exact same behavior. □

The simulation induces a serializability proof, with events serialized at their commit point.

Corollary 3 (Serializability)**.** *Since the serial* semantics of AEON$_{core}$ *is linear and does not reorder non-overlapping events, Theorem 2 implies that the semantics of AEON$_{core}$ enforces serializability of events. This order respects the arrival order of events, thus implying serializability.*

Lemma 3 (Exclusive Access)**.** *Given a trace* $\vec{\omega} = (E_0, \Omega_0) \rightarrow (E_0, \Omega_0) \ldots \rightarrow (E_n, \Omega_n)$*, assume that two events* e_0 *and* e_1 *access at least one coinciding actor, say* a*, in* $\vec{\omega}$*. We have that for any configuration* (E, Ω) *appearing in* $\vec{\omega}$*:*

1 $\mathsf{dom}(\Omega, \mathsf{target}(e_0)) \in \mathsf{desc}(\mathsf{dom}(\Omega, \mathsf{target}(e_1)))$*, or*
$\mathsf{dom}(\Omega, \mathsf{target}(e_1)) \in \mathsf{desc}(\mathsf{dom}(\Omega, \mathsf{target}(e_0)))$*, and*

2 $e_0 \notin a.A$ *or* $e_1 \notin a.A$*.*

Proof. The claim 1 is a direct consequence of Ω forming a DAG, 2, 1, and the fact that both events operate on a common actor a. The claim 2 is a direct consequence of claim 1 in combination with 2 and 1. □

We state that the proof of Theorem 2 relies on a commutativity argument allowing to reordering the execution of actions of different events which operate in non-sharing actors. Here we provide it definition and proof. Figure 3.12 provides a graphical representation of this commutativity argument.

Lemma 4 (Commutativity)**.** *Consider two consecutive transitions where we assume two actors* $a_0, a_1 \in \Omega$*, and two events* $e_0, e_1 \in E$*:*

$$(E, \Omega_0) \xrightarrow[e_0]{a_0} (E, \Omega') \xrightarrow[e_1]{a_1} (E, \Omega_1)$$

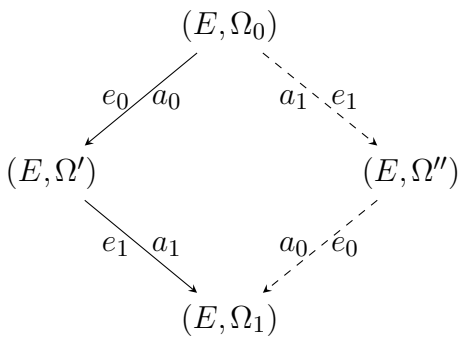


Figure 3.12.: Commutativity of independent events.

Here we denote with the arrow $\xrightarrow[e]{a}$ *the fact that the transition involved the event identifier* e *and it was performed in the actor* a*. We have that if either* $a_0 \neq a_1$ *or* $e_0.am = e_1.am = \mathsf{ro}$*, there exists an intermediary actor set* Ω'' *such that the following transitions are also valid:*

$$(E,\Omega_0) \xrightarrow[e_1]{a_1} (E,\Omega'') \xrightarrow[e_0]{a_0} (E,\Omega_1)$$

Figure 3.12 depicts this lemma, where dashed arrows represent the existential transitions required by the lemma.

Proof. The case where the two events are ro is trivial. By 3 we have that if any of the events is ex, then $a_0 \neq a_1$. Moreover, we have that $\mathsf{lub}(\Omega), a_0.A = \{e_0\}$ and $\mathsf{lub}(\Omega, a_1).A = \{e_1\}$. Hence, by the definition of LUB we have that $\mathsf{desc}(a_0) \cap \mathsf{desc}(a_1) = \emptyset$. It is not hard to see that the two events operate on disjoint portions of the graph, and therefore the transitions commute immediately. □

Corollary 4. *Consider a sequence of transitions, where we use* $\bar{e_{id}}$ *denotes any of the events in,*

$$\alpha \;=\; (E,\Omega_0) \xrightarrow[\bar{e_0}]{} (E,\Omega_1,) \xrightarrow[\bar{e_1}]{} \;\ldots\; \xrightarrow[\bar{e_{n-1}}]{} (E,\Omega_n)$$

where no step except the last one is a commit event. We can conclude that there exists an equivalent trace

$$\alpha' = (E,\Omega_0) \xrightarrow[\delta(\bar{e_0})]{} (E,\Omega'_1) \xrightarrow[\delta(\bar{e_1})]{} \ \ldots \ \xrightarrow[\delta(\bar{e_{n-1}})]{} (E,\Omega_n)$$

where δ is an bijection from $\{\bar{e_0},\ldots,\bar{e_{n-1}}\}$ to itself, such that there exists an $m \in [0,n-1]$ with (i) for all $i < m$ we have $\delta(\bar{e_i}) = \bar{e_{m-1}}$, and (ii) for each $i \geq m$, $\delta(\bar{e_i}) \neq \bar{e_{n-1}}$. Essentially, the bijection δ pushes all transitions of $\bar{e_{n-1}}$ to the front.

Notice that the initial and final configurations are the same.

Proof. This is a trivial consequence of the lemma above, considering that all transitions that need to be reordered correspond to different (concurrent) events.

We consider an alternative semantics of AEON_{core} which allows at most one event at a time. As before, we call this the *serial semantics of AEON* and denote it with the special arrow "$\twoheadrightarrow$" and denote actor sets generated by this semantics as $\hat{\Omega}$.

The proof of serializability shows that the *serial semantics of $AEON_{core}$* can simulate the semantics of AEON_{core}. To that end we consider as simulation relation between configurations of the two different semantics. We denote by $\mathcal{A}ct(\Omega)$ the set of events that are activated in any actor of Ω.

We consider as simulation relation between configurations of the two different semantics.

Definition 3 (Simulation Relation)**.** *We define the following simulation relation between a configuration of the serial semantics of AEON, $(E,\hat{\Omega})$, with a configuration of the low-level configurations of AEON, (E,Ω), denoted by*

$$(E,\hat{\Omega}) \ \mathcal{R} \ (E,\Omega)$$

iff the following conditions are met:

(i) $\mathcal{A}ct(\hat{\Omega}) = \emptyset$*, and*

(ii) $(E,\hat{\Omega}) \xrightarrow{*} (E,\Omega)$

We then prove that the semantics of serial AEON simulates the semantics of low-level AEON by showing that the diagram below commutes. This amounts to proving that each time we start with a pair of similar configurations, whenever the low-level semantics makes a step, there exists a step in the high-level semantics that can simulate exactly the same behavior. Notice in particular, that the event sets E in any two similar configurations are identical, meaning that transitions that modify the event set must be matched one to one.

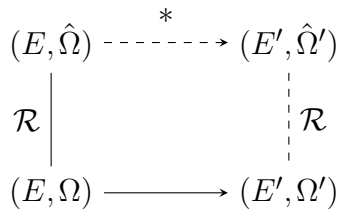


Figure 3.13.: AEON simulation diagram.

Theorem 3 (Simulation)**.** *The relation $\mathcal{R}$ defined above is a weak-simulation from the low-level semantics of AEON to the serial semantics of AEON, as shown in Figure 3.13.*

Proof. We assume a pair of configurations related by the relation $\mathcal{R}$. Let them be $(\hat{E,}\Omega)\ \mathcal{R}\ (E,\Omega)$.

The proof proceeds by case analysis on the transition taken by the configuration (E,Ω) of standard *AEON*. Importantly, we *only need to cater for transitions that modify the event set E*, since all other transitions immediately preserve the relation $\mathcal{R}$, as the second condition of the definition of the relation requires that the configuration $(E,\hat{\Omega})$ can reach (E,Ω), and evidently, similar steps can be taken to preserve the relation. Hence, we consider only transitions that modify the event sets. These are: Event Call UnShared, Event Call Shared and Event Return.

- Event Call UnShared and Event Call Shared. The transitions taken by these rules are trivially matched by the serial semantics, since these events are only added to the tail of the queue (in both configurations).

- Event Return. This is the only important step, since it is here that the serial semantics (i.e. the configuration $(E, \hat{\Omega})$) must make actual transitions. In this case, we can directly apply 4 to obtain the conclusion, since we have from 1 and 2 that the committing event was the only one to touche these actors (i.e. all other concurrent events are disjoint), and the event continued to hold these actors until this commit step.

□

Evidently, *the simulation above induces a serializability proof*, where events are serialized at their commit point.

Parallelism

Here we provide a simple statement showing that independent events can be executed in parallel.

Definition 4 (Independent Events)**.** *Given a trace $\vec{\omega}$ and events $e_0, e_1 \in E$, we say that e_0 and e_1 are independent in $\vec{\omega}$ iff whenever $\vec{\omega}$ can be decomposed as*

$$\vec{\omega} = \vec{\omega_0} \cdot (E, \Omega_0) \xrightarrow[e_0]{a_0} (E, \Omega_1) \cdot \vec{\omega_1} \cdot (E, \Omega_2) \xrightarrow[e_1]{a_1} (E, \Omega_3) \cdot \vec{\omega_3}$$

we have that $a_0 \neq a_1$.

Theorem 4. *[Independence ⇒ Parallelism] Consider a trace $\vec{\omega}$ and two independent events $e_0, e_1 \in E$. We have that whenever*

$$\vec{\omega} = \vec{\omega_0} \cdot (E, \Omega_0) \xrightarrow[e_0]{a_0} (E, \Omega_1) \xrightarrow[e_1]{a_1} (E, \Omega_2) \cdot \vec{\omega_3}$$

there exists an equivalent trace

$$\vec{\omega} = \vec{\omega_0} \cdot (E, \Omega_0) \xrightarrow[e_1]{a_1} (E, \Omega') \xrightarrow[e_0]{a_0} (E, \Omega_2) \cdot \vec{\omega_3}$$

where the order of the transitions of e_0 and e_1 is reversed.

Proof. This is an immediate consequence of 4. □

In a nutshell this theorem establishes that the order of steps of events e_0 and e_1 is inconsequential, and therefore they can be evaluated in any order (i.e., in parallel).

3.6 Summary

This chapter has presented a variant of the actor model specialized for distributed setups where actors commonly communicate across hosts over the network, implemented in our AEON language, which we believe achieves a sweet spot between (a) ensuring serializability guarantees for multi-actor interaction, and (b) enabling a high degree of parallelism in networked distributed systems. Our model provides serializability and deadlock freedom, with largely decentralized synchronization and thus scalability for distributed server-side applications or components following a DAG-based structure of actors.

4 AEON RUNTIME DESIGN AND IMPLEMENTATION

Designing low-latency cloud-based applications that are adaptable to unpredictable workloads and efficiently utilize modern cloud computing platforms is hard. The *actor* model is a popular paradigm that can be used to develop distributed applications: actors encapsulate state and communicate with each other by sending events. Consistency is guaranteed if each event only accesses a single actor, thus eliminating potential data races and deadlocks. However it is nontrivial to provide consistency for concurrent events spanning across multiple actors.

In this chapter, we will give AEON runtime implementation details. The synchronization protocol (section 4.1) implemented in the runtime relies on actor DAG structure to group actors and provide serializability as well as scalability. The elasticity algorithm (section 4.2) enables the programmers to transparently migrate actors without violating the execution of applications or entailing significant performance overhead. We have give runtime and fault tolerance implementation in section 4.3 section 4.4 gives 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 state-of-the-art specialized systems such as HyperDex Warp [29] and Infinispan [28], and to implementations in Orleans [74] on Amazon AWS [77]. AEON outperforms its competitors in most cases, and shows much better scalability. We have draw a summary in section 4.5.[1]

[1]Patrick Eugster and Masoud Saeida Ardekani helped me design the AEON synchronization protocol. Also, Patrick Eugster, Masoud Saeida Ardekani and Srivatsan Ravi made many contributions to the experiments design.

4.1 Multi-Actor Synchronization

We present synchronization in AEON first for a *static ownership* DAG (subsection 4.1.1), followed by the *dynamic ownership* DAG (subsection 4.1.2). In both cases, we first introduce synchronization, then discuss how AEON achieves serializability [75] during execution of concurrent events. We consider the generic case, and leave out optimizations (e.g., for executing multiple readonly events in parallel, cf. section 3.4) as well as relaxations (e.g., allowing actors to access any of their descendants, not only direct children, cf. section 3.4). As we discuss in subsection 4.3.2, synchronization also supports fault-tolerance.

Definition 1 (Static and dynamic ownership)**.** *Let* (E, Ω) *denote a system configuration resulting from an execution of an XYZ 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 2 (Serializability)**.** *Let* E *denote the set of events in a given execution of XYZ. 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.*

4.1.1 Synchronization under Static Ownership

Traditional mechanisms for synchronization like *two-phase locking* enforce serializability on linearly-ordered structures without shape constraints while *two-phase commit* is best suited when *majority voting* is necessary for agreement [78]. 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 locking mechanism* and *multiple*

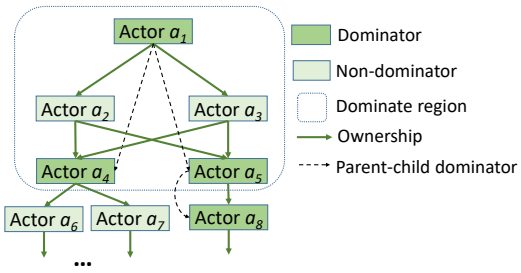


Figure 4.1.: Example of dominate region of actor a_1: actor a_4 is a child dominator of a_1 because a_2 is its parent and a_2 is dominated by a_1. Note how the child dominator a_4's children actors a_6 and a_7 are *not* included in a_1's dominate region.

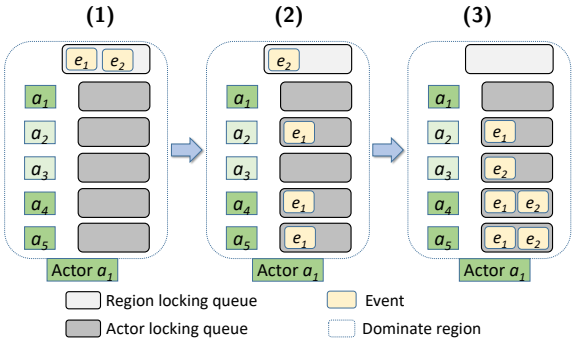


Figure 4.2.: Basic AEON synchronization. e_1 and e_2 are put into region locking queue (1) at first. Then e_1 is dequeued and put into actor locking queues of reachable actors (2). After that, e_2 is dequeued and put into the actor locking queues (3).

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 3 (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. Figure 4.1).*

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. At every dominator "below" a_1, other events may arrive. Thus every dominator serializes events in its dominate region and then "passes them on" to its child dominators similarly to hand-over-hand locking (at dominator level):

(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.

If a_2 is a child dominator the event is forwarded to a_2 and put into its region locking queue. The event is removed from all actor locking queues in the region once

all actors (i) and (ii), including child dominators, finished executing the event (boiling down to a "no-op" for unaffected actors).

Illustration. In Figure 4.2, 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. 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 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 Figure 4.3 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 XYZ 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 Figure 4.2 (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

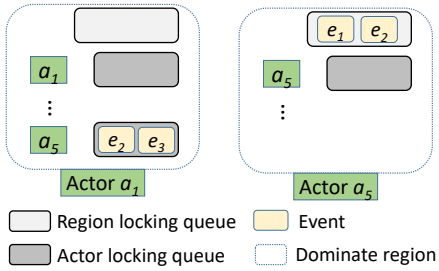


Figure 4.3.: Event execution order in child dominator's region. a_5 is a child dominator of a_1. e_2 is the head of the actor locking queue of a_5 in a_1's region. Thus e_2 will be put into the region locking queue of a_5.

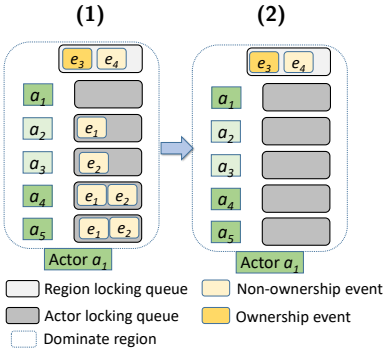


Figure 4.4.: Ownership event e_3 waits until all previous events (e_1 and e_2) finish (1) and starts to execute (2). e_4 can only be put into actor locking queues after e_3 is done and removed from the queue.

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's order in a_2's region locking queue is the same as their order 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 Figure 4.3, 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.

4.1.2 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. 6) 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 Figure 4.4 shows, ownership event e_3 first waits for e_1 and e_2 to finish executing in this dominate region (Figure 4.4 (1)). Eventually e_3 becomes the head of the region locking queue (Figure 4.4 (2)) and can be executed; it remains in the region locking queue until the execution is finished. During the execution of an ownership event,

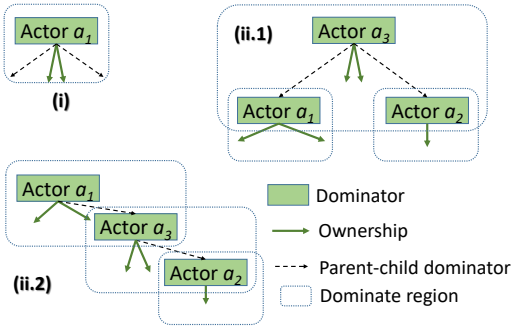


Figure 4.5.: Ownership events (i) modifying ownership between actors in a_1's dominate region; (ii.1) and (ii.2) modifying ownership between actor from a_1's dominate region and another actor from a_2's dominate region.

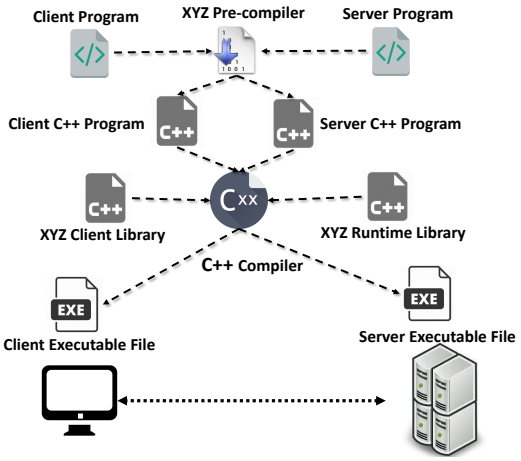


Figure 4.6.: XYZ implementation workflow.

the runtime determines actors whose dominators changed as a consequence, or have new dominate regions, and informs them of the updated DAG. Only after that is the ownership event removed from the region locking queue. If an actor is not a dominator anymore or has new dominate region after DAG updates, it forwards events in its region locking queue to the new respective dominators.

Serializability under dynamic ownership. As argued in subsection 4.1.1, our model guarantees serializability under a static ownership configuration. To claim serializability under dynamic ownership, we thus need to show that ownership events do not affect the safety of event execution. In the following, we show that serializability is maintained under dynamic ownership 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. 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. Figure 4.5 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 will not be affected by concurrent ownership events. As described, an ownership event will lock all actors which may be affected by its ownership operations. At the same time, an ownership event will only start to execute when there are no other events executing in the region. Then 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 can only observe 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 can not 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 [23,79]. 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.

4.2 Elasticity

In this section, we explain AEON's elasticity manager called *s*manager. The *s*manager provides the following capabilities: (i) maintaining the global *actor mapping* and *ownership network*, and (ii) managing actor creation and migration based on elasticity policies. In our experiments, AEON is made fault tolerant using the Zookeeper service. In the remainder of this section, we explain the above two capabilities.

4.2.1 Actor Mapping

Since actors can dynamically migrate across hosts, and in order to deliver an event to the appropriate actor, AEON first needs to find the host currently holding the corresponding actor. To this end, every client and host caches the most recent actor mapping that they have queried, and periodically refreshes their actor mappings by querying the *s*manager. In practice, and in order to have a highly scalable and available system, clients and other hosts do not directly query the *s*manager. Instead, the *s*manager stores the latest actor mappings along with the ownership network in a (configurable) cloud storage. Therefore, to locate an actor for the first time (or in case the local cache has become invalid), a host or a client simply performs a read operation on the cloud storage system to retrieve the latest mapping. In the remainder of this paper, and for the sake of simplicity, we assume that clients and other hosts directly query the *s*manager.

4.2.2 Elasticity Policy

AEON gives the programmer the ability to define when and where actors should be migrated. Every server periodically sends its resource utilization data (i.e., CPU, memory and IO) to the *s*manager. AEON provides a simple API to define when the *s*manager must perform a migration. The following example policies are implemented in AEON by default: (i) Resource utilization: in this policy, a programmer defines a lower and upper bound of a resource utilization along with an activation threshold. Thus, when a resource in a server reaches its upper bound plus a threshold the *s*manager triggers a migration. (ii) Server contention: under this policy, a programmer defines the total number of acceptable actors per server. Hence, once a server reaches its maximum, the *s*manager triggers a migration.

Once a migration is triggered, AEON computes a list of possible servers that can receive the actors concerned. The default algorithm tries to move actors from overloaded hosts to underloaded ones, but programmers can implement their own algorithms for choosing hosts and actors. In addition, AEON allows programmers to define constraints on any attribute of the system For instance, a constraint can disallow certain actor migrations, or disallow a migration to a new host if total cost reaches some threshold.

Migration protocol. Once a migration is triggered, the *s*manager will follow the following *atomic steps* to migrate an actor C from host s_1 to a new host s_2.

I The *s*manager sends a prepare message to s_2, notifying that requests for actor C might start arriving. Then, s_2 responds by creating a queue for actor C and acknowledges the *s*manager.

II Upon receiving the ack, the *s*manager informs s_1 to stop receiving events targeting C and it waits for s_1 ack.

III Once the *s*manager receives the ack, and after δ seconds, it updates its actor mapping by assigning C to s_2. Thus, from this point on, the *s*manager returns

s_2 as the location of actor C. It then sends a special event called migrate(C, s_2) to s_1 indicating that C has to be migrated to s_2.

IV Upon receiving migrate(C, s_2), s_1 enqueues an event migrate$_c$ in C's execution queue. This event serves as a notification for actor C that it must migrate. When migrate$_c$ reaches the head of C's queue, s_1 spawns a thread to move C to s_2.

V Upon completion of the migration, s_2 notifies the *s*manager that the migration is finished, and starts executing the enqueued events for actor C.

Correctness under actor migration. Observe that actor C, at the end of step II when s_1 stops accepting events for C does not take any steps until step III when the *s*manager updates the actor map. During this period, s_1 does not accept events targeting actor C, and *s*manager does not return s_2 as the new host for C.

Once the migration event enters C at s_1 for execution, it will be the only event that is being executed at C.

Following the complete execution, both s_1 and s_2 will have up-to-date actor mappings. If s_1 later receives an event for C from a host with stale actor map, s_1 will forward those events to s_2 directly and notify source host to update its actor map.

Similar to Orleans [73], AEON provides users with a special *snapshot* API that allows programmers to take consistent snapshots of a given actor along with all its children. To this end, upon receiving a snapshot request for an actor, the runtime of AEON dispatches a particular event called snapshot to that actor. Consequently, this event takes consistent snapshots of that actor and its children by getting actors states, and writing them into a (configurable) cloud storage system like Amazon S3. To improve the performance, a programmer is able to override a method returning the state of an actor. In case the overridden method returns null for a actor, the runtime system will ignore that actor during the checkpointing phase.

As we mentioned earlier, in practice the *s*manager is implemented as a stateless service that is responsible for updating actor mapping and the DAG structure that are stored in a cloud storage system. The *s*manager also leverages the cloud storage

system for persisting the steps of ongoing migrations. Therefore, if during the course of a migration, the *s*manager crashes, a newly elected *s*manager can read the state of an going migration, and tries to finish it. Details on how individual server and the *s*manager failures are treated without violating the consistency can be found on the AEON webpage.

4.3 Implementation

We overview our prototype implementation and discuss and fault tolerance in AEON.

4.3.1 Prototype

AEON is implemented as a pre-compiler and two libraries: *AEON runtime library* and *AEON client library*. As Figure 4.6 shows, programmers implement a server program and a client program in AEON. The AEON pre-compiler processes those AEON programs by handling special syntax in AEON (e.g., event) and generating code to handle (potentially remote) method calls on actors. The pre-compiler outputs C++ client and server programs. Then the C++ compiler is invoked to generate server executables from C++ server programs and the AEON runtime library, and client executables from C++ client programs and the AEON client library. The two AEON libraries are implemented in around $\approx$ 40 KLoC in total in C++. The AEON pre-compiler includes $\approx$ 3 KLoC in Python and $\approx$ 4 KLoC in Perl.

4.3.2 Fault Tolerance (FT)

Currently host (or process) crashes are handled by recovery. That is, failed actors 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. Observe how ensuring consistency — a non-trivial undertaking in general graphs [80] — 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.

4.4 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.

4.4.1 Synopsis

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

RQ1 (subsection 4.4.2): How does AEON's synchronization protocol compare to a basic synchronization protocol like two-phase locking?

RQ2 (subsection 4.4.3): How does our model compare to a "regular" actor language (no support for multi-actor programming), achieving consistency by manually synchronizing?

RQ3 (subsection 4.4.4, subsection 4.4.5): How does our model compare to using a storage system for shared state which needs to be kept consistent?

RQ4 (subsection 4.4.5): How does our model compare to actor languages with comparable properties?

RQ5 (subsection 4.4.6, subsection 4.4.7): How well does our model scale and how do the different call types fare?

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 [68] 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 [28] key-value store. Infinispan is one of the most popular systems for caching, being used for instance by the JBoss [81] 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 can and will be executed successfully. At least one of two concurrent conflicting transactions have to abort, yielding an exception which the programmer must handle.

HyperDex Warp: For RQ3 we also use HyperDex Warp [29], 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 [73] distributed programming language which is centered on a notion of actor-like *grains*. Orleans is actively developed by

Table 4.1.: Metadata store LoC.

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

Table 4.2.: Game app LoC.

Implementation	LoC
AEON	564
Infinispan	853
Orleans	434

Table 4.3.: B tree LoC.

Implementation	LoC
AEON original	1370
AEON optimized	1437

Microsoft, and used in a number of projects including the re-engineered Skype and Halo [82] applications. Until recently Orleans did not enforce consistency over multiple grains/actors. Considering the need for consistency in many applications, the version 2.0 [74] 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 chapter 2) 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 [83]) 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.

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

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. Table 4.1, Table 4.2 and Table 4.3 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, Table 4.1 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 technical 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). 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 [84] yet the source code [85] is not FT. Infinispan supports replication for individual key-value pairs, but with no consistency guarantee *across* key/value pairs in transactional mode, so we set its replication degree to 1.

4.4.2 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, with classic mechanisms such as two-phase locking (2PL). We exhibit the efficiency of AEON's synchronization protocol against its closest baseline, a 2PL implemented on AEON's basic C++ runtime.

Assume an application which consists of multiple actors, each on a separate server. Clients issue requests to this application 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 ids, 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. For AEON, as this microbenchmark scenario has no root AEON creates an abstract root (cf. subsection 3.3.3) 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 in combination with 2PL for fault recovery.

We now describe the experimental setup for benchmarking AEON's synchronization protocol with respect to the above described C++ 2PL protocol implementation. We compare the client request latencies of the two implementations on a setup where 1 to 4 clients send their requests to the applications with 1 to 8 actors. In the 1

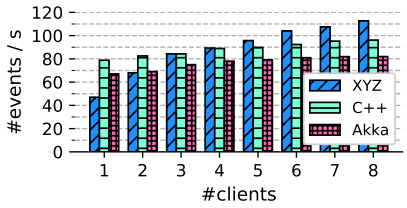


Figure 4.7.: Binary tree throughput in AEON, C++ and Akka. Unlike AEON, the synchronization overhead of a growing number of clients saturates C++ and Akka.

Figure 4.8.: Latency of an app using AEON's protocol vs a two-phase locking in C++ with varying numbers of clients and servers.

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.

Figure 4.8 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; proving the scalability of AEON's synchronization.

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

Most actor programming languages such as Akka [68] 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 can not 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 Figure 4.7 shows, Akka and "C++" outperform AEON with few clients because of the overhead of AEON's serialization protocol. This is due to the fact that the AEONprotocol leverages additional metadata and bookkeeping (with multiple queues) 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 can not 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 4.4.2) or implementing a custom serializable protocol specifically designed for the semantics of the binary tree operations [86]. 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.

4.4.4 RQ3: Metadata Store with HyperDex Warp

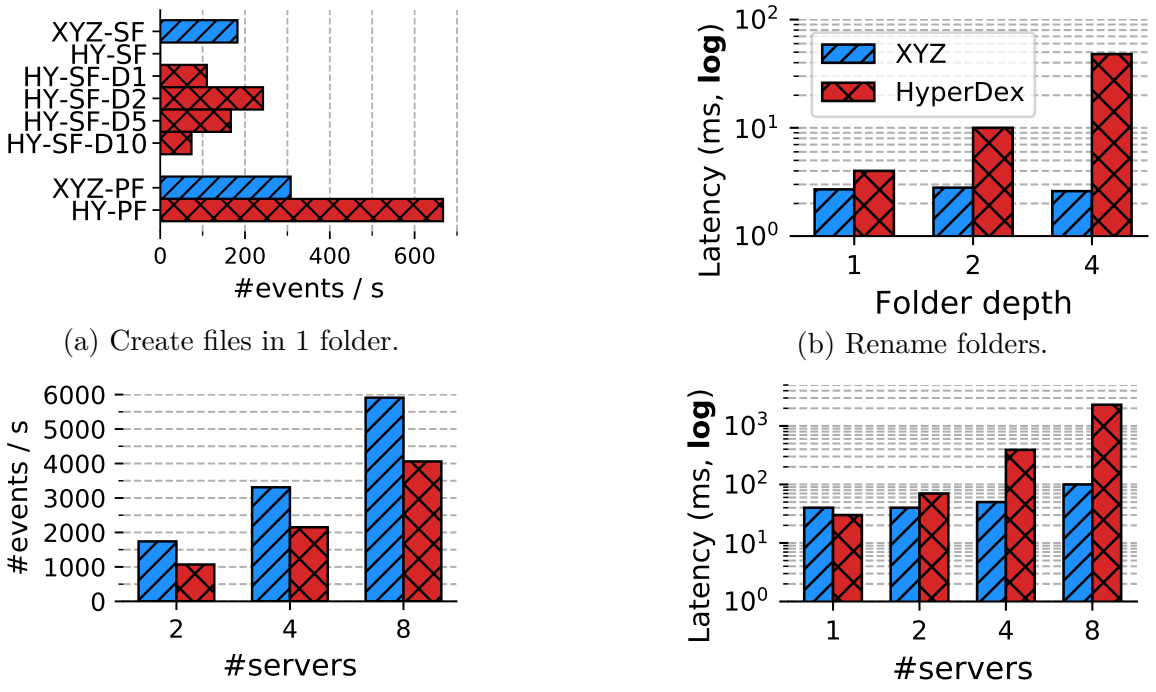
We compare the metadata store of the Warp Transactional Filesystem (WTF) [84] implemented both in AEON and HyperDex Warp. WTF consists of four "tiers": a client library, a metadata store, a replicated coordinator, and storage servers. Only the metadata store requires serializability so clients can update 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 [87], by using "transactions" in the metadata store, thus allowing clients to update the metadata of multiple files (e.g., merge files) together.

Implementation. The original 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 can not 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.

Filesystem operations. 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 cre-



(a) Create files in 1 folder.

(b) Rename folders.

(c) Open private inodes.

(d) MapReduce.

Figure 4.9.: AEON vs HyperDex Warp metadata store.

ate files in the same folder. Figure 4.9a 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 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 can not alleviate conflicts while large delays increase latency. Moreover, selecting the ideal delay is a tedious

task since the chosen one must strive despite various workloads, making it difficult to have an efficient distributed HyperDex Warp.

Creating files in private folders. With the same setup, clients now create new files in their private folders to avoid any conflict. As Figure 4.9a 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. Figure 4.9b 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 since 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. Figure 4.9c 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 section 3.3.3), showing that AEON's performance is also appealing in that case.

MapReduce. We show the performance of AEON and HyperDex Warp metadata stores while running MapReduce [87,88] 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 experiment 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.

Figure 4.9d 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 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's performance, in most cases, is 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.

4.4.5 RQ3 and RQ4: Game App with Infinispan and Orleans

We compare AEON with Infinispan [28] and Orleans [74] on the game app.

Implementation and workloads. Since both Infinispan and Orleans throw exceptions when updates conflict, we introduce random delays of 1-10 ms and retry a

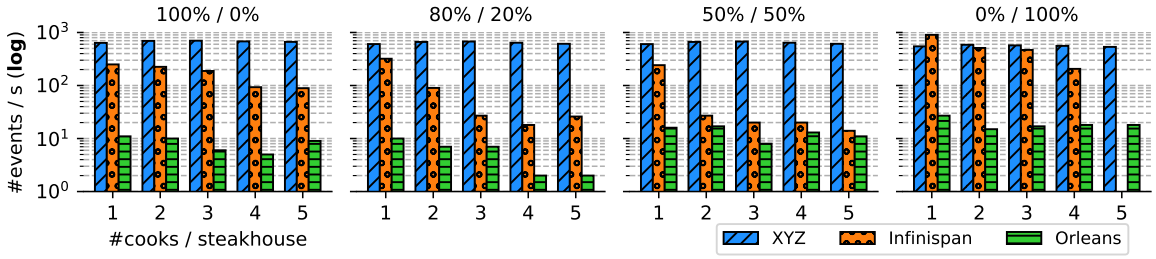


Figure 4.10.: AEON vs Infinispan vs Orleans game app with varying workloads (%*UseGrill* / %*NewGrill* events).

certain number of times (e.g., 3×) when clients are told that their transactions are aborted.

In the game app (Listing 3.1), Cooks can (1) put steaks on Grills belonging to them (15) 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* are events that 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% (a less realistic workload 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 each Steakhouse. Each Cook initially uses 4 random Grills.

Throughput and scalability. We run the AEON, Infinispan, and Orleans game apps each on 8 servers with the four workloads. Figure 4.10 compares the game 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 (transactions) 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 subsection 4.4.3, technical differences alone do not seem to be able to explain the substantial 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.

4.4.6 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. Figure 4.11 depicts the scalability of the AEON game app, with the same *UseGrill*/*NewGrill* workloads as in subsection 4.4.5, when the number of servers increases from 2 to 32. As presented in subsection 4.1.2, 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 section 3.2, 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.

Figure 4.12 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.

4.4.7 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 [89], 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.

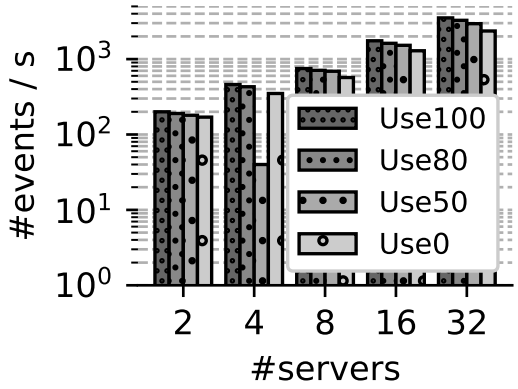


Figure 4.11.: Game scale-out.

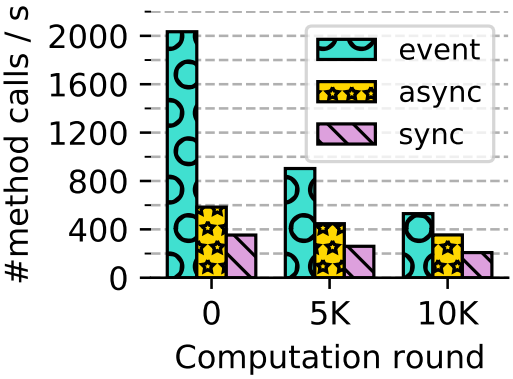


Figure 4.12.: Calls' throughput.

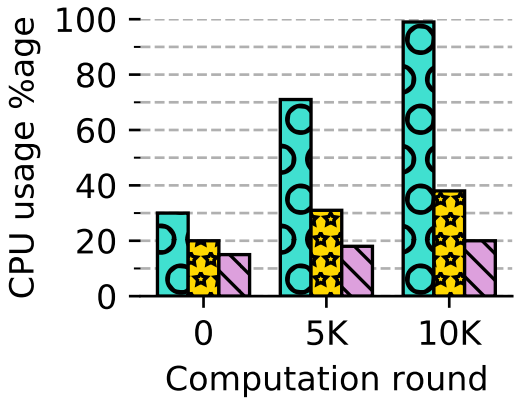


Figure 4.13.: Calls' CPU usage.

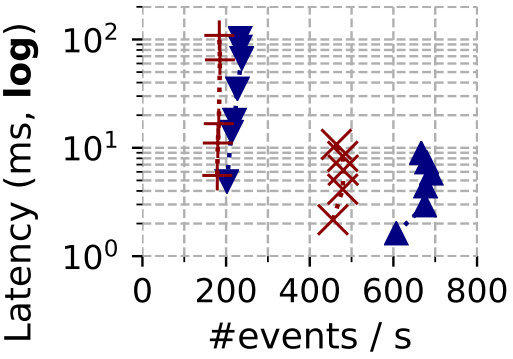


Figure 4.14.: Optimizing B tree.

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. Table 4.3 shows the number of LoC for both 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 [90] 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.

Figure 4.14 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 can not be explained by any inherent differences between read and insert operations (i.e., RAM memory read/write speeds are similar [91]).

4.5 Summary

We have presented the implementation of the AEON language runtime. AEON provides a sequential programming environment for the cloud based on the standard paradigm of object-orientation. We provide a synchronization protocol of AEON, and show that this protocol exploits parallelism while providing strict serializability as well as deadlock freedom. We have experimentally shown that the AEON runtime system scales as the number of client requests increases, and it is able to scale-out/in to provide an economic solution for the cloud.

5 PLASMA: PROGRAMMABLE ELASTICITY FOR STATEFUL CLOUD COMPUTING APPLICATIONS

Developers are always on the lookout for simple solutions to manage their applications on cloud platforms. *Serverless computing platforms (e.g., AWS Lambda) are among the most popular ones, which allow developers to program elastic cloud applications consisting of* stateless *functions that can be automatically scaled in and out, without manual intervention.* Major cloud providers have already been offering automatic elasticity management solutions (e.g., AWS Lambda, Azure durable function) to users. However, many cloud applications are *stateful* — while executing, functions need to share their state with others. Providing elasticity for such stateful functions is much more challenging, as a deployment/elasticity decision for a stateful entity can strongly affect others in ways which are hard to predict without any application knowledge. Existing solutions either only support stateless applications (e.g., AWS Lambda) or only provide limited elasticity management (e.g., Azure durable function) to stateful applications.

PLASMA (Programmable Elasticity for Stateful Cloud Computing Applications) is a programming framework for elastic stateful cloud applications. It includes (1) an elasticity programming language as a second "level" of programming (complementing the main application programming language) for describing elasticity behavior, and (2) a novel semantics-aware elasticity management runtime that tracks program execution and acts upon application features as suggested by elasticity behavior. We have implemented 10+ applications with PLASMA. Extensive evaluation on Amazon AWS shows that PLASMA significantly improves their efficiency, e.g., achieving same performance as a vanilla setup with 25% fewer resources, or improving performance by 40% compared to the default setup.

Roadmap section 5.1 discusses the existing elasticity management. section 5.2 discusses challenges in providing elasticity for stateful applications and overviews PLASMA. section 5.3 and section 5.4 detail PLASMA's EPL and EMR respectively. section 5.5 presents empirical evaluation. section 5.6 concludes.[1]

5.1 Background: Elasticity Management

Elasticity is essential to the "pay-as-you-go" cloud computing model [92], allowing cloud applications to *automatically* scale their demand for cloud resources in/out in adaptation to workload changes. Elasticity maximizes the use of resources and thus reduces infrastructure costs, meanwhile maintaining performance and service quality of cloud applications.

Developers can program elastic cloud applications as a set of functions executing independently in response to specific events (e.g., AWS Lambda and Azure Durable Function). Such functions, usually encapsulated in VMs/containers, can be automatically scaled in/out on corresponding platforms [1, 40–43], freeing developers and administrators from server management.

This kind of solution provides developers with ideal automatic elasticity management. However, existing automatic elasticity management provide better support to applications consisting of stateless functions, such as routines for image processing [93] or handlers of IoT devices [94]. These provide a pure transformation from input to output without external dependencies at execution. When the state of functions is thus limited to *internal* state, automating elasticity is relatively straightforward; it can simply focus on placing a function on a server node with available resources, or adjusting the number of function instantiations.

However, many cloud applications are *stateful*, i.e., functions need to share state with each other. Such scenarios are common across multiple abstraction levels, e.g.,

[1]In this chapter, with the help of Gustavo Petri and Patrick Eugster I designed the semantics for the PLASMA elastic programming language. Patrick Eugster and Srivatsan Ravi also made several key propositions for the evaluation for the PLASMA runtime.

metadata of distributed file systems (one component of an application), data access tier of web applications (an entire tier or layer), microservice applications [95] (multiple loosely coupled components), and massively multi-user online games (an entire application). Those stateful applications can not be executed efficiently in state-of-the-art serverless computing platforms (e.g., AWS Lambda [96]).

Providing elasticity for generic stateful functions – or more generally components or (micro-)services – is very challenging, as an elasticity decision for one stateful component depends on not only that component, but also on others and its interaction with them. Existing programming models and frameworks enabling automated elasticity [41, 96, 97] can not capture such stateful scenarios, thus limiting the scope of the elasticity paradigm. Automating elasticity in such cases is difficult without any knowledge of applications; many non-functional requirements are hard if not impossible to learn just by looking at executions of applications. For instance one may be tempted to straightforwardly rate low-frequency component interactions as secondary to others and thus spread corresponding components across servers. Even if frequency were straightforwardly correlated with "importance", such a placement policy could adversely affect frequent interactions — of such infrequently interacting components — with others. Existing profiling approaches [98, 99] tracking system-level performance (e.g., server usage) can not connect low-level performance data to application semantics and trigger appropriate elasticity decisions.

We present PLASMA (Programmable Elasticity for Stateful Cloud Computing Applications), a novel framework for implementing expressive elastic cloud applications. It extends an existing *actor-based* [64] *application programming language* along two dimensions:

Elasticity programming language (EPL) PLASMA adds a second "level" of programming to the underlying application programming language. That is, while actors support building stateful cloud applications that have horizontal, scalable relations between stateful components [4, 50, 51], PLASMA includes a complementary

elasticity programming language (EPL). The EPL allows users to express desired *elasticity behavior* through simple rules based on *high-level* application semantics exposed to the runtime to help it carry out fine-grained elasticity management. This is realized without violating application invariants induced by the actor programming model. In this paper we use EPL implementations for an actor-based language for building stateful distributed applications, AEON [51], but our concepts are applicable to others like Microsoft's Orleans [3] and Scala [100].

Elasticity management runtime (EMR) To guide PLASMA applications running on cloud platforms in achieving elasticity, PLASMA involves a novel *elasticity management runtime* (EMR) with two main components. (1) The *elasticity profiling runtime* tracks the behavior of actors (e.g., location, resource usage) and their interactions (e.g., message rates), as per the stated EPL elasticity policy. The information is used in making global elasticity decisions (e.g., co-locating highly interactive actors, (de-)provisioning resources) that are acted upon by (2) PLASMA's *elasticity execution runtime* leveraging a two-level architecture to reconcile global optimization accuracy with scalability.

We know of no other programming framework supporting programmable elasticity for *stateful* cloud applications. Since our above-mentioned novel concepts are independent from the exact underlying actor language, we focus on those concepts in this paper. We have implemented various (10+) distributed cloud applications with PLASMA[2] including Metadata Server, E-Store [53], PageRank [101], Halo's Presence Service [102], and a Media (micro-)Service [103]. We evaluated PLASMA using these applications, showing how PLASMA enables fine-grained elasticity with only high-level user input, even outperforming application-specific elasticity solutions like Mizan [104].

[2]`https://aeonproject.github.io/plasma/webpages/`

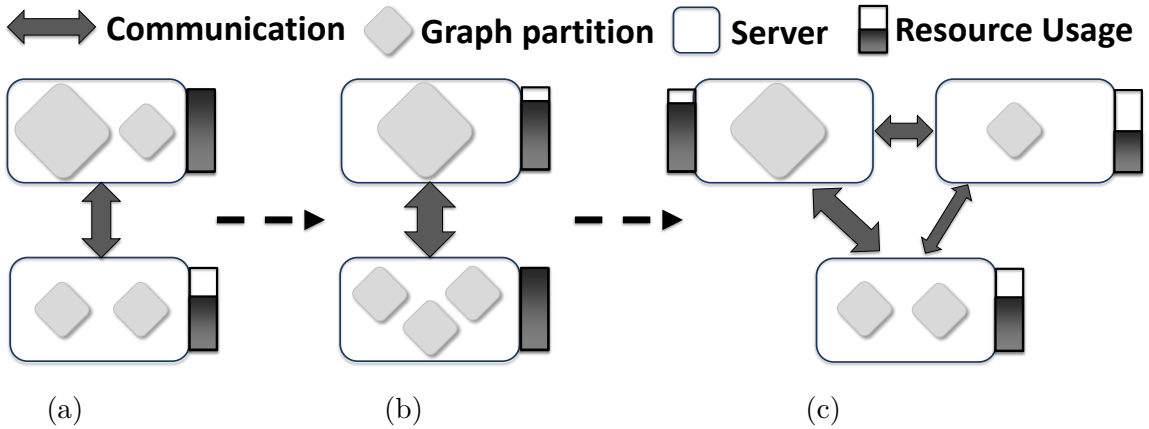


Figure 5.1.: PageRank elasticity management example: **(a)** The initial placement of graph partitions overloads the top server which calls for partition migration. **(b)** Once migration is performed, the bottom server becomes congested. **(c)** With both servers reaching their maximum capacity, PLASMA migrates to a new server to split the load of the bottom one.

5.2 Motivation and Overview

We first motivate our work by a simple example, and further provide an overview of PLASMA in this section.

5.2.1 Elastic PageRank

While elasticity in cloud computing enables an efficient use of resources, no existing framework supports fine-grained elasticity management for *stateful* applications. Consequently, a whole range of essential algorithms and applications are being left behind, forcing their developers to resort to an infrastructure with suboptimal resource consumption, or build custom-tailored elastic solutions (e.g., [53, 55, 105]). PageRank [101] is a fitting and simple example of a popular stateful application. A common approach to speed up PageRank is to partition the graph it runs on into independent subsets, and have subgraphs processed in parallel. However, as partitions need to communicate with one another, deploying an elastic partitioned PageRank on a stateless platform like AWS Lambda [1] requires the use of latency-costly external

storage (e.g., S3 [44], DynamoDB [106]). We have observed 25 ms average latency for DynamoDB write requests and more than 70 s to write graph vertices, edges, and partitions from a small 22 MB graph into a DynamoDB table; hence it is currently impractical to develop stateful applications requiring frequent state load/store (e.g., the distributed PageRank in subsection 5.5.4 needs to update ≈1.2 GB data at each round).

Another approach to obtain an elastic PageRank could be to directly implement the algorithm using an elastic programming language such as Orleans [50], AEON [51] or Akka [68]. Orleans balances workload by equalizing the number of actors on each server and by replicating stateless actors as the workload increases. Orleans also co-locates actors that frequently communicate with one another on the same server to avoid remote communication. AEON also evenly distributes actors across a cluster. Akka allows programmers to define router actors that forward received messages to replicated routee actors in a certain pattern (e.g., round-robin).

However, none of these languages consider server metrics (e.g., CPU usage) for ongoing elasticity management. They can not therefore properly handle applications such as PageRank – balanced graph partitioning being a notoriously difficult task [57–59, 107]. Consider the example given in Figure 5.1 that provides an intuition of elasticity management for PageRank applications. In this example, a graph is split into four partitions. PageRank requires both network (i.e., partitions need to exchange data) and CPU (i.e., processing graph partitions) resources. While exact performance characteristics depend on exact graph partitions, cloud platform, and implementation, simple tests on AWS show that PageRank can be easily CPU-bound (more details in subsection 5.5.4). Assume partitions are originally evenly distributed across two servers, as in Orleans, while trying to minimize remote communication between actors as a secondary objective. But despite an even split and fair initial placement, a partition can eventually require much more computation time (Figure 5.1a) to the point where the CPU consumption upper-bound of the server hosting it is crossed. With PLASMA, a developer can set CPU consumption bounds

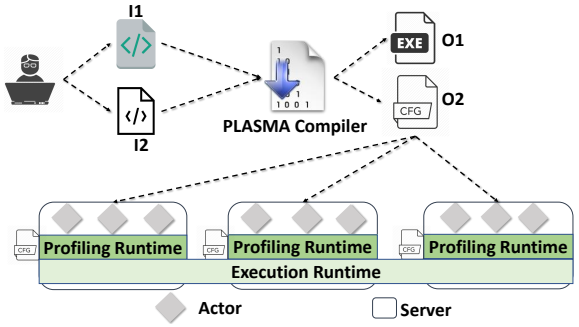


Figure 5.2.: PLASMA toolchain overview.

(e.g., $60\% \leq load \leq 80\%$) to ensure that a server is neither over- nor underloaded. To alleviate the load of a server, PLASMA then migrates actors to another server to respect the aforementioned CPU consumption bounds (Figure 5.1b). While the migration was sufficient at first, the bottom server becomes the congested one as workloads vary, and with no available server to host additional workload, PLASMA has no choice but to spawn a new server and migrate actors to that new server (Figure 5.1c).

PageRank demonstrates the need for application insights (e.g., CPU is more important than network in PageRank) for efficient elasticity management, and for platform metrics (i.e., CPU usage). As we shall show, with custom-fitted elasticity rules, PLASMA can optimize performance of various applications and adjust applications' resources to avoid under- and overprovisioning. We explore several further applications benefiting from elasticity management in subsection 5.3.3.

5.2.2 PLASMA Overview

We outline how to develop elastic stateful applications with PLASMA, which is designed to extend the popular actor model [64, 108, 109]. Several corresponding languages have been already conceived for building scalable stateful elastic cloud applications (e.g., [3, 4, 51, 68]).

Programming with PLASMA Elasticity in PLASMA is programmed *separately* from the application logic. PLASMA assumes only basic concepts for the underlying actor-based programming language (cf. Figure 5.3.I); neither the underlying actor language nor the programs need to be changed to benefit from PLASMA's elasticity model. This allows programmers to easily enable elasticity for existing applications without changes, simply by defining elasticity rules in a separate program following PLASMA's *elasticity programming language* (EPL).

As shown in Figure 5.2, programmers develop the (I1) application program using the actor programming language and the (I2) elasticity program using PLASMA's EPL. PLASMA's compiler takes these two programs as input and generates two output files: (O1) an executable containing both the application's binary code and PLASMA's elasticity management runtime code, and (O2) a file with elasticity actions for this application. To deploy the application programmers only need to launch the executable that takes O2 as input.

Elastic execution with PLASMA As shown in Figure 5.2, when the application is executing on a cloud platform, the PLASMA elasticity management runtime puts the EPL elasticity rules to work based on the actual performance features observed at servers about actors and their interaction. More specifically, PLASMA's *elasticity management runtime* (EMR) involves an *elasticity profiling runtime* (EPR) and an *elasticity execution runtime* (EER). The EPR keeps track of the performance of actors, focusing on those that are affected by elasticity decisions, as per the elasticity rules. Given the declared EPL elasticity rules and the performance information from the EPR, the EER takes decisions for placing actors among available servers and/or adapting the number of servers, hence realizing automated application elasticity. The EMR performs adjustments when creating actors, and every *elasticity (time) period* (set by users). Note that the EMR will not blindly follow rules to conduct elasticity management, but rather will consider the actual runtime situation (e.g., resource limitations, migration overhead) in its decisions.

The EMR does not interfere with the execution of the original language's runtime; the EPR only collects runtime data of actors. In particular PLASMA inherits the fault tolerance mechanism from the original runtime and relies on it to handle failures in the application. Failures in the EMR are handled by a separate mechanism (presented in section 5.4).

5.3 Elasticity Programming Language (EPL)

This section describes how PLASMA's EPL captures elasticity behaviors based on high-level application semantics.

5.3.1 Actor-based Elasticity

Given a distributed actor-based application, elasticity decisions boil down to placing/migrating actors among available servers for adjusting to workload and variations therein.

Execution features Numerous *features* of an actor-based application's execution can be used as cues for its performance, and to drive actor placement. The features supported by PLASMA pertain to three categories:

[F-RA] Resource usage of actors (e.g., CPU).

[F-RS] Resource usage of servers (e.g., network).

[F-IA] Interaction between actors (e.g., message rate).

Elasticity rules Similarly to the above classes of runtime features, we can classify *rules* guiding elasticity decisions based on the above features by the type of *reaction* (*behavior*) they induce on the application execution:

[R-R] *Resource elasticity rules* correspond to resource features, and strive for a better resource usage. Server resources are not directly influenced, but rather affected indirectly by adjusting the placement (and thus resource usage) of actors, and so this category of rules corresponds to both [F-RA] and [F-RS]. These rules provide

programmers with a way to reserve certain amounts of resources for actors or balancing resource usage among servers. For example, programmers can specify upper and lower bounds on CPU resources for servers. If a server's CPU utilization hits the upper bound, a select group of its actors will be migrated to other servers with idle CPU resources.

[R-I] *Interaction elasticity rules* correspond to actor interaction features [F-IA]. These rules allow programmers to expose high-level application semantics to the runtime, allowing it, e.g., to co-locate actors that strongly interact, as per application semantics and actually observed at execution, thus reducing communication latency.

5.3.2 Syntax

Next, we detail the syntax and usage of PLASMA's EPL, realizing the above elasticity programming model. The EPL is declarative, as defining a set of elasticity rules for actors is simpler for developers (e.g., [110]) over explicitly defining an algorithm to follow with an imperative language. We opted for a declarative language over an imperative one as we feel that it is more natural for developers to express "policies" that way (cf. [110]). The EPL assumes only basic features of the underlying actor programming language, as shown in Figure 5.3.I: a program includes a set of actors of different types (), each declaring a set of functions ($func$) — which give rise to messages — and fields (properties – $prop$). $\overline{x}$ denotes several instances of x.

Actor-condition-behavior The EPL (see Figure 5.3.II) is used to describe – *separately* from the main program – a policy *pol* that consists of a set of elasticity rules *rul*. PLASMA's elasticity rules (both [R-R] and [R-I]) are expressed in an *actor-condition-behavior* style. That is, rules usually purport to features regarding certain *actor*s, and when certain *condition*s on those features are met which may adversely affect performance, the runtime is advised to apply elasticity *behavior*s (to certain actors).

I. Actor-based application programming language basics

Program	*prog*	::=	
Actor class		::=	$aname\{\overline{prop}\ \overline{func}\}$
Property	*prop*	::=	*type pname*;
Function	*func*	::=	$type\ fname(\overline{type\ \ldots})\{\ldots\}$

II. Elasticity programming language

Policy	*pol*	::=	$\overline{rul}$	
Rule	*rul*	::=	$cond \Rightarrow \overline{beh};$	
Actor	*actor*	::=	(*var*) | | *var*	
Actor type		::=	*aname* | **any**	
Condition	*cond*	::=	*cond* **or** *cond* | *cond* **and** *cond*	
		|	**true** | *feat.stat comp val*	
		|	*actor* **in** **ref**(*actor.pname*)	[F-IA]
Feature	*feat*	::=	*entity.res*	
		|	*cllr*.**call**(*actor.fname*)	[F-IA]
Entity	*entity*	::=	*actor*	[F-RA]
		|	**server**	[F-RS]
Caller	*cllr*	::=	**client** | *actor*	
Statistic	*stat*	::=	**count** | **size** | **perc**	
Resource	*res*	::=	**cpu** | **mem** | **net**	
Comparison	*comp*	::=	< | > | >= | <=	
Behavior	*beh*	::=	**balance**({},*res*)	[R-R]
		|	**reserve**(*actor*, *res*)	[R-R]
		|	**colocate**(*actor*, *actor*)	[R-I]
		|	**separate**(*actor*, *actor*)	[R-I]
		|	**pin**(*actor*)	[R-I]
Value	$val \in \mathbb{N} \cup \mathbb{R}$			

Figure 5.3.: Basic definitions for actor programming language and abstract syntax of PLASMA's EPL.

Actors As actors of the same type tend to have similar behavior patterns, elasticity rules are expressed a priori for actor *types*, and, as detailed shortly, behaviors are expressed similarly with respect to actors of given types. A subject type of *actor* is specified by the name of the actor type's name *aname* as defined in the main application program. However, a rule can contain several actors of the same type. In order to disambiguate yet keep definitions concise by avoiding verbose variable declarations in front of every rule, we use a form of implicit variable declaration, where the use of an actor type in a rule can declare a variable *var* in an inline fashion. E.g., a condition relating to `Innernode(i)` specifies actor type `Innernode` and introduces variable `i` to refer later to all `Innernode` instances to which the condition applies.

PLASMA also introduces the special type **`any`**, allowing rules to be defined for all actors in an application. Note that PLASMA currently treats actor subtypes in the application program as distinct types from their parent types.

Conditions Conditions are used to specify situations that may affect the performance of applications. Though rules – and thus a priori also conditions – relate to actor types, as alluded to above, conditions end up selecting a subset of actors of such a type. While actors of the same type follow the same execution logic, their actual runtime behaviors will also be affected by workloads and thus differ.

Conditions can be composed (**`and`**, **`or`**) of more elementary conditions. Basic conditions can be trivial (**`true`**) or compare statistics *stat* (highlighted in **orange**) for a runtime feature $feat$ to some value *val* (a lower or upper bound), where statistics can be a number of instances (`count`), a `size` value, or a `perc`entage. Note that not all statistics apply to all features.

Actual features are of three basic categories (the actual features in the syntax are highlighted in **green**).

The first category (i) consists in conditions of the shape *actor* **`in`** **`ref`**`(`*actor*$'$`.`*pname*`)` which essentially select actors of the former type *actor* that are referenced by specific fields *pname* of actors of the second type *actor*$'$.

The second category (*ii*) corresponds to resource features of specific entities (*entity.res*). Two subcategories arise from the two types of entities considered: *actor*s (*ii.a*) or `server`s (*ii.b*). They correspond to [F-RA] and [F-RS] respectively. The resources considered here, in turn, are of three types (**blue**): `cpu`, `mem`ory, and `net`work usage.

The third category (*iii*) corresponds to interaction features [F-IA] just like conditions on referencing (*i*). For specific types of messages *fname* sent from either `client`s or *actor*s of one type to actors of another type (*cllr*`.call(`*actor.fname*`)`) we consider the number of such messages sent per time unit, e.g., 1 min, their size, or the percentage of a particular type of calls received by an actor, out of the total number of this type of calls received by all actors on the same server.

Behaviors Finally, (elasticity) behaviors *beh* (**red**) tell the runtime how to react to specified conditions on given actors. There are five kinds of elementary behaviors (Figure 5.3). The first two give rise to resource elasticity rules [R-R] and the others yield interaction elasticity rules [R-I].

In the former category we have `balance` and `reserve`. `balance({}`, *res*`)` prompts the runtime to balance the workload on each server by migrating actors of types indicated in from overloaded servers to ones with idle resources. *res* refers to the type of critical resource that should be taken into consideration when balancing workloads. Note that `balance` does not allow type variables to be used in its first argument – using as opposed to *actor*. This is because `balance` targets servers instead of actors. Referring to specific actors here, programmers could add conditions on those actors (e.g., `cpu.perc>30`); then the runtime could only migrate those actors to balance the workload. This would eliminate most flexibility for the runtime (e.g., migrating actors with CPU below 30% might alleviate a bottleneck). `reserve(`*actor*,*res*`)` instructs the runtime to keep those *actor*s on *dedicated* servers exclusively, whose resources are sufficient to meet the actors' demands. *res* specifies the type of desired resources on the dedicated servers.

The second behavior category spans `colocate`, `separate` and `pin`. `colocate`$(actor, actor)$ tells the runtime to keep the concerned actors on the same server. Notice that conditions in the rule can also (further) constrain the interaction between the actors. Consider $actor_2$.`call`$(actor_1.fname_1)$.`count` with $fname_1$ a function of $actor_1$. Placing a condition on this term can, for instance, restrict the total number of messages $fname_1$ to $actor_1$ called by $actor_2$. Conversely, behavior `separate`$(actor, actor)$ instructs the runtime to keep the actors of the two types separated whenever resources are available whilst the accompanying conditions are satisfied. This can be used for example when actors of the two types run computationally demanding activities (i.e., instead, `colocate` may obstruct their operations). Lastly, `pin`$(actor)$ indicates that particular actors should not be moved. This prevents migration from hampering highly available services.

5.3.3 Examples

We show the use of PLASMA's EPL via five concrete examples. These applications are evaluated empirically in section 5.5.

Metadata Server The Metadata Server is composed of folders and files, handled by `Folder` actors and `File` actors respectively, all of which can be opened and accessed by remote clients. Some folders are in much higher demand than others, thus receiving a significant portion of the overall number of requests. To avoid congestioning servers, we opt to migrate highly demanded folders to idle servers.

Performing such elasticity managements requires the runtime to have knowledge of application semantics, as is easily achieved with PLASMA. For instance, the aforementioned elasticity behavior can be expressed in PLASMA by defining the following elasticity rule: a `Folder` actor is migrated to an idle server (`reserve`) when (1) the current server's CPU usage exceeds 80% and (2) this folder receives more than 40% client requests among all `Folder` actors on this server. The rule also tells the runtime to place all `File` actors under this `Folder` actor on the same server (`colocate`).

Table 5.1.: Applications implemented with PLASMA. We show elasticity rules and evaluation for the first five applications.

Application	LoC	Elasticity rules (and number of rules)
Metadata Server	253	**1.** Colocate Folder with Files in it (since they are accessed together)
PageRank [101]	465	**1.** Balance CPU workload
E-Store [53]	645	1. Put hot Partitions on idle servers 2. Colocate parent-child Partitions **3.** Balance CPU workload of Partitions
Media Service [103]	756	1. Balance network workload for FrontEnds 2. Provide VideoStream with enough CPU 3. Colocate linked VideoStream and UserInfo 4. Avoid migrating MovieReview 5. Balance CPU workload of ReviewChecker **6.** Colocate linked ReviewEditor and UserReview
Halo Presence Service [102]	314	1. Balance CPU workload of Routers **2.** Colocate Session with Players in it
B+ tree	1457	1. Colocate parent-child inner nodes **2.** Put leaf nodes on separate servers
Piccolo [47]	564	1. Balance CPU workload for Workers **2.** Colocate Worker and Table that Worker reads data from
zExpander [111]	506	**1.** Put leaf nodes on idle servers
Cassandra [112]	221	**1.** Put table replicas on different servers

```
server.cpu.perc > 80 and
client.call(Folder(fo).open).perc > 40 and
File(fi) in ref(fo.files) ⇒
    reserve(fo, cpu); colocate(fo, fi);
```

PageRank As we introduced in subsection 5.2.1, we should balance the PageRank partitions according to CPU usage:

```
server.cpu.perc > 80 or server.cpu.perc < 60 ⇒ balance({Partition}, cpu);
```

E-Store E-Store [53] is an elastic partitioning framework for distributed OLTP DBMSs. Initially, root-level keys are range-partitioned into blocks of fixed size and

co-located with descendant tuples. At runtime, the system monitors the workload on each server and avoids imbalance. When observing a server's resource usage (e.g., CPU) exceeding a high-water mark, the system selects $k\%$ partitions with high activity on the hot server and migrates them to idle servers. If inversely observing a server's resource usage being below a low-water mark, the system also redistributes the data.

It is a typical balancing problem where programmers need to define the conditions to trigger data distribution (i.e., high-water mark and low-water mark), and how to redistribute data (i.e., migrate hot data to idle servers). Furthermore, since data is organized in a tree structure, we can not solely migrate the hot partitions but also need to consider moving their descendants. We express E-Store needs with these 3 rules:

```
server.cpu.perc > 80 and
client.call(Partition(p1).read).perc > 30 ⇒ reserve(p1, cpu);
Partition(p2) in ref(Partition(p1).children) ⇒ colocate(p1, p2);
server.cpu.perc < 50 ⇒ balance({Partition}, cpu);
```

Media Service The Media Service [103] is a more intricate stateful application, it provides two major functions, (1) rent and watch movies and (2) review movies, involving 8 types of interdependent actors in a *cloud microservice*.

Specifically, the `FrontEnd` actors are the entrance of the Media Service and are network-intensive. `VideoStream` actors stream movies to users and are CPU-intensive and latency-sensitive. A `UserInfo` actor contains the information of a user: when a user is watching a movie, the `VideoStream` actor keeps updating this user's watching history to the user's `UserInfo` actor. This yields the following three rules:

```
server.net.perc > 80 or server.net.perc < 60 ⇒ balance({FrontEnd}, net);
server.cpu.perc>50 ⇒ reserve(VideoStream(v),cpu);
VideoStream(v).call(UserInfo(u).track).count > 0 ⇒ pin(v); colocate(v, u);
```

In addition, users can read/write movie reviews via the `ReviewEditor` actors, which frequently interact with the `UserReview` actors by updating reviews on them. `MovieReview` actors, on the other hand, store a large amount of reviews by movie types (e.g., comedy), thus are memory-intensive. Finally, users' reviews will be checked by `ReviewChecker` actors before publication, and hence are CPU-intensive. Such semantics lead to the remaining three elasticity rules:

```
ReviewEditor(r).call(UserReview(u).update).count > 0 ⇒ pin(r);
    colocate(r, u);
true ⇒ pin(MovieReview(m));
server.cpu.perc > 90 or server.cpu.perc < 70 ⇒
balance({ReviewChecker}, cpu);
```

Halo Presence Service The Halo Presence Service [102] is a deployed actor-based system that tracks player liveness in Halo 4. Active game consoles periodically send heartbeat messages to the service. Each of these messages is first decrypted by a randomly selected `Router` actor, before it is forwarded to the related `Session` actor which finally forwards it to the corresponding `Player` actor.

A `Session` actor can only send messages to `Player` actors partaking in the session it manages, while `Player` actors can only belong to one session at a time. This isolation between `Session` actors and between the `Player` actors of different sessions can be leveraged to improve the system communication performance. For instance, remote messaging can be avoided by co-locating at runtime `Player` actors with their corresponding `Session` actor:

```
Player(p) in ref(Session(s).players) ⇒ pin(s); colocate(p, s);
```

`Router` actors require a nontrivial amount of CPU resources to decrypt messages from clients, to the point where it becomes essential to balance the CPU workload by carefully distributing `Router` actors across servers:

```
server.cpu.perc > 80 or server.cpu.perc < 60 ⇒ balance({Router}, cpu);
```

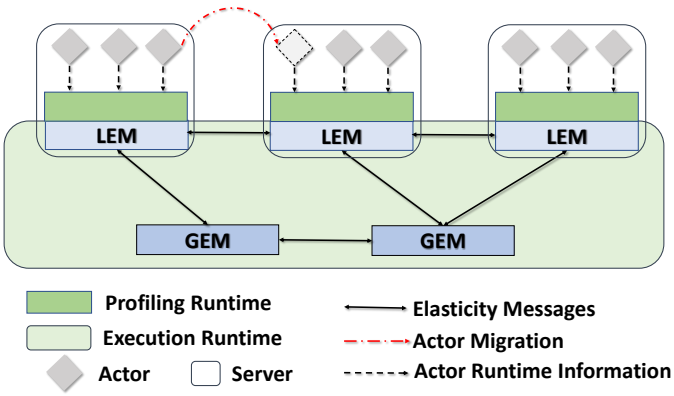


Figure 5.4.: PLASMA's runtime system: GEMs manage application scale; LEMs handle actors of single servers.

5.3.4 Discussion on Language

We made several concessions for simplicity. An important choice was to consider only restricted *scopes*: in all conditions relating to interaction features , we only consider *direct* interaction. E.g., when conditioning the number of calls as in $actor_2$ `call`($actor_1.fname_1$)`.count` we only consider direct calls from an instance of $actor_2$ to $fname_1$ of one of $actor_1$, and not indirect ones, e.g., through a function $fname_3$ of intermediary actors of some type $_3$. Considering such transitive interaction could become quite complex: for given instances of $actor_1$ and $actor_2$, one could focus on calls through a *single* instance of $actor_3$, or any number. While both options (and more) could be expressed by introducing a larger set of variables and making existential vs universal quantification explicit, it leads to a much more complex language. Restricting scopes also simplifies profiling, especially with recursive types. subsection 5.4.3 discusses conflict resolution. Our language is one point in the design space, and extensions such as additional features and behaviors are the subject of ongoing investigations.

5.4 Elasticity Management Runtime (EMR)

PLASMA's elasticity management runtime (EMR) is integrated into the runtime of the underlying actor programming language. The EMR involves an *elasticity profiling runtime* (EPR) and an *elasticity execution runtime* (EER) (cf. Figure 5.4).

5.4.1 Elasticity Profiling Runtime (EPR)

In short, the EPR, which runs on each server, is responsible for collecting performance-related metrics, and feeding them to the EER. Corresponding to the three types of execution features of PLASMA (cf. subsection 5.3.1), the EPR collects performance information on resource usage of actors [F-RA], of the server it runs on [F-RS], and on interaction among actors [F-IA].

For [F-RS], the EPR reads system-level performance metrics from the server directly — we assume servers expose an interface such as `stat` under `/proc` in Linux. For features [F-RA] and [F-IA], the EPR keeps track of the behaviors of actors. As the elasticity rules involve specific actor types, the EPR can focus solely on the actors affected by these rules, thus limiting the overall profiling overhead. In particular, the EPR collects (1) the execution time of each task, (2) the size of actors (i.e., memory footprint), and (3) the count and size of different types of messages and involved actors. Note that the elasticity management service usually runs at every elasticity period (a configurable interval). The EPR only collects performance information since its last run.

5.4.2 Elasticity Execution Runtime (EER)

The EER decides on placement of actors among available servers and on adapting the number of required servers.

Two-level architecture PLASMA's two-level EER design includes *local* and *global elasticity managers* (LEMs and GEMs respectively). This design makes a practical

tradeoff between *scalability* (distributed LEMs with local views and actions) and *accuracy* (centralized GEMs to provide a more complete, global, view of the system). Duties are thus split in that a LEM works only for a single server and is responsible for interaction elasticity rules [R-I] as these can be monitored locally. On the other hand a GEM oversees a group of servers, and is responsible for resource elasticity rules [R-R] among these servers, as these need to place actors among a group of available servers, requiring global performance information.

LEMs When an elasticity management cycle begins, a LEM reads the performance information of the server and actors from its local EPR. The LEM summarizes such information, and reports *imperative* information (e.g., focusing on actors furthest beyond thresholds) to its GEM. The LEM also iterates through its actors and checks related interaction elasticity rules [R-I] (if any). If the conditions in the rules are satisfied, the LEM identifies *executable actions*. Take for example a `colocate` rule specifying actors of two types and $'$, with frequent interaction through some function $fname$, e.g., `.call(`$'$`.`$fname$`).count > 1000`. The LEM goes through each actor and checks the message count between it and remote $'$ actors generating a migration action when a message count is larger than the threshold defined. Notice that the LEM managing the $'$ actor does the same but on a different server. The two LEMs communicate directly, without passing by a GEM, to decide whose actor to migrate to the other's server (by default the one with lower resource usage).

GEMs A GEM manages a subset of servers; each server is managed by one GEM *at a time*. After a certain time period (e.g., elasticity time period), each LEM randomly picks a new GEM. After receiving the profiling information from LEMs of its managed servers, a GEM creates a *global runtime snapshot* for all its managed servers. Referring to this snapshot, the GEM checks the conditions of resource elasticity rules. If any are met, the GEM identifies executable actions (O2 in Figure 5.2) from the rules, and tasks servers.

Table 5.2.: API summary.

(a) GEM & LEM local functions

Function	Functionality
getActRules	Return actor elasticity rules
getActorsRuntime	Return runtime info of all local actors
applyActRules	Return migration actions as per actor runtime info and elasticity rules on LEMs
getResRules	Return resource elasticity rules
collectActorsFResRules	Return runtime info of actors related to resource elasticity rules
getServerRuntime	Return local server runtime info
resolveActions	Resolve conflicted migration actions of LEMs and GEMs. Return final actions
applyResRules	Return migration actions according to actor and server runtime info and resource elasticity rules on GEMs
checkIdleRes	Decide if one server has enough idle resources to accept an actor

(b) Action datatype

Action datatype field	Content
actor	Actor for migration
srcServ	Server currently holding the actor
trgServ	Target server for actor migration

Take for example a `balance` rule. When its condition is met (typically a lower or upper bound on server resources is exceeded), the GEM migrates a select set of actors among its managed servers to balance workload. PLASMA uses a simple heuristic to thus select actors: a GEM only migrates actors from overloaded servers (i.e., with resource usage above upper bounds) to servers with enough idle resources – especially below specified lower bounds. If all of a GEM's managed servers are overloaded (resp. under-utilized), it broadcasts an *adjustment* message to all other GEMs. GEMs reply whether their observations are similar. If the majority of replies received by the requester GEM corroborate its own view, it increases (resp. decreases) the number of servers.

LEM-GEM interaction 1 and 2 outline how LEMs and GEMs coordinate on generating migration actions based on elasticity rules and runtime performance infor-

mation (using APIs summarized in Table 5.2). Each LEM (1) reads actors' runtime information from the profiling runtime and identifies migration actions (line 7) based on actor elasticity rules (line 5). The LEM then checks resource elasticity rules and reports related actors' runtime information as well as its server runtime information to a GEM (line 12).

The GEM (2) starts processing reports from LEMs when it receives enough of those (line 8). It only checks resource elasticity rules and generates corresponding migration actions (line 10) and informs relevant LEMs (line 14). A LEM and a GEM can generate different, potentially conflicting, actions for the same actor. A LEM then resolves such conflicts once it receives migration actions from its GEM (line 14). Finally, the LEM starts migrating actors when the target server agrees to accept them (line 22).

New actor creation When the application creates an actor (of type) on a server, this server's LEM queries the GEM, which managed it during last the elasticity period, where to place the new actor. The GEM checks relevant elasticity rules and decides the initial placement. E.g., the rules require to co-locate actors with references, or identify actors as CPU-intensive. Then the new actor is to be co-located with another actor that has a reference to it, or put on a server with idle CPU resources. If the GEM can not find a valid rule for actors, it randomly picks a server from the ones it manages. With the help of input elasticity rules, PLASMA has a higher chance to place new actors on the right servers from the start, as we will see shortly.

5.4.3 Discussion on Runtime

Conflict resolution As stated above, LEMs and GEMs identify executable actions based on rules. These actions are enqueued at LEMs, and are executed by the actor programming language's runtime via its live actor migration procedure. However, as programmers may define multiple elasticity rules for one actor type, conflicts may arise. PLASMA provides two mechanisms to resolve these. (1) When *compiling*

Algorithm 1 Elasticity execution on LEM *lem*

```
1: Local variables:
2:    existActors                    ▷ local actors and actors to be migrated to this server
3:    gems                           ▷ addresses of GEMs
4: task processElasticity:
5:    actRules ← getActRules()
6:    actorsRT ← getActorsRuntime()
7:    lemActions ← applyActRules(actorsRT, actRules)
8:    resRules ← getResRules()
9:    ractorsRT ← collectActorsFResRules(actors, resRules)
10:   serverRT ← getServerRuntime()   ▷ collect server runtime info
11:   gem ← gem_x | gem_x ∈ gems      ▷ pick random GEM
12:   send (REPORT, ractorsRT, serverRT) to gem   ▷ report to GEM
13:   wait until receive (RREPLY, gemActions) from gem
14:   finalActions ← resolveActions(lemActions, gemActions)
15:   for all (action ∈ finalActions) do
16:      send (QUERY, action) to action.trgServ   ▷ can server accept
17: upon receive (QUERY, action) from lem′:
18:   if checkIdleRes(existActors, action.actor) then
19:      existActors ← existActors ∪ {action.actor}   ▷ take resources
20:      send (QREPLY, action) to lem′
21: upon receive (QREPLY, action):
22:   migrate action.actor to action.trgServ
```

Algorithm 2 Elasticity execution on GEM *gem*

```
1: Local variables:
2:    actorsRTs                      ▷ queue for actor runtime information from LEMs
3:    serversRTs                     ▷ queue for server runtime information from LEMs
4:    actionQs                       ▷ map of addresses to action queues
5: upon receive (REPORT, actorsRT, serverRT) from lem :
6:    actorsRTs ← actorsRTs ⊕ {actorsRT}
7:    serversRTs ← serversRTs ⊕ {serverRT}
8:    if |servers| > K then          ▷ K is given by user
9:       resRules ← getResRules()
10:      actions ← applyResRules(actorsRTs, serversRTs, resRules)
11:      for all (act ∈ actions) do
12:         actionQs[act.srcServ] ← actionQs[act.srcServ] ⊕{act}
13:      for all (addr | ∃ actionQs[addr] do
14:         send (RREPLY, actionQs[addr]) to addr   ▷ ret to LEM
```

elasticity rules, PLASMA's compiler detects conflicting rules for the same actor type, and issues warnings. (2) When applications are *running*, LEMs resolve the remaining conflicts by choosing the migration action for an actor with the highest priority, which can be specified by programmers or determined by assigning priorities to migration actions. E.g., `colocate` can break the resource elasticity actions of `balance` in that multiple LEMs might try to migrate their actors to the same server and overload it. If PLASMA prioritizes `balance` over `colocate`, it will only allow the target server to accept actors if it has enough resources. Existing conflict resolution approaches [112–114] can also be leveraged in PLASMA; they are beyond the scope of this paper.

Fault tolerance As is evident from 1 and 2, no state synchronization is required between LEMs and GEMs or among GEMs. Hence, if a GEM fails while computing the set of migration actions, LEMs can still progress by picking another GEM through the shuffling process described. We run multiple GEMs for scalability and fault tolerance when evaluating PLASMA in section 5.5.

Actor placement stability We opt for a conservative policy to actor migration to minimize the cost associated with actor state "re"-migrations. More aggressive migration policies [115] could be employed, e.g., by pre-profiling actor resource consumption or migrating more actors than strictly needed, but no optimal policy exists.

To avoid frequent actor migrations, an actor can only be migrated if it stayed on the same server for a certain time, which is set to be equal to the elasticity period (cf. section 5.2.2).

5.5 Evaluation

The concepts of PLASMA can be implemented in many actor programming languages.

5.5.1 Synopsis

We evaluate our approach through an implementation in AEON [51] involving 3500 Python LoC added to the AEON compiler, that parses both PLASMA elasticity rules and AEON program to generate an elasticity configuration file (PLASMA compiler in Figure 5.2). We also extend AEON's runtime by 5000 C++ LoC to collect actors' and platform's runtime information (profiling runtime in Figure 5.2), and conduct elasticity management (execution runtime in Figure 5.2). We chose AEON over Orleans [50] and Akka [68], as when starting our prototyping, Orleans' code-base was undergoing frequent significant updates while Akka lacks live actor migration features.

We evaluate PLASMA with several stateful applications on Amazon AWS. The elasticity rules used for each scenario are described in subsection 5.3.3, with their summary in Table 5.1, demonstrating the low effort with which a multi-actor application can be complemented with PLASMA.

We first evaluate the overhead of PLASMA's runtime (subsection 5.5.2). Next we demonstrate how a simple elasticity rule leveraging application-specific knowledge improves a Metadata Server's elasticity management (subsection 5.5.3). We compare the efficiency of PLASMA against the state of the art on an elastic PageRank (subsection 5.5.4). Then we showcase how PLASMA can help developers implement specific elasticity management in E-Store (subsection 5.5.5). We show how PLASMA handles highly dynamic workloads in a Media Service (subsection 5.5.6). Finally, we evaluate how different number of GEMs impact the performance of the Halo Presence Service (subsection 5.5.7).

5.5.2 PLASMA's Runtime Overhead

First we assert that the EPR does not impose high overheads on applications when tracking performance data.

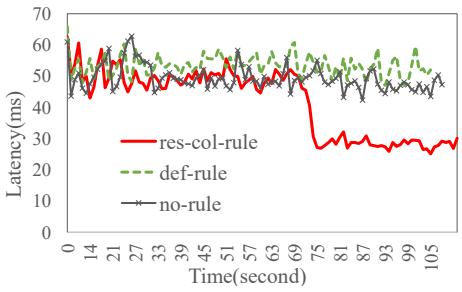
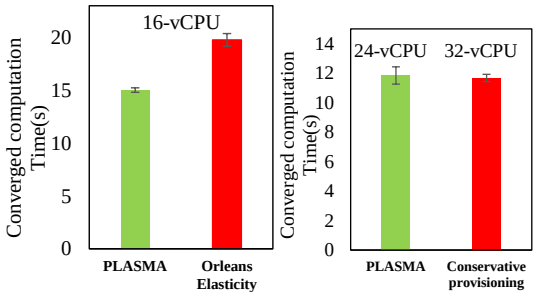


Figure 5.5.: Simple reserve & colocate vs default vs no rule in Metadata Server.



(a) (b)

Figure 5.6.: PageRank PLASMA's vs Orleans' elasticity, **(a)** static & **(b)** dynamic allocation.

The EPR only collects runtime information of actors on the server it is deployed on, the EPR overhead is therefore only affected by the number of actors and messages on a single server, regardless of the number of servers used by this application.

To this end, we use an online chat room microbenchmark where users, represented each by an actor, can exchange messages with others within the same room. The EPR tracks information on all messages (e.g., type, size, number) and the times for actors to process them. The chat room is deployed on a single AWS instance, i.e., actors are stationary, and is tested with different numbers of users. Tab. 5.3 shows the profiling overheads of PLASMA's EPR on the chat room actors by normalizing the execution time of PLASMA with that of a vanilla system without elasticity (e.g., 1.007 means 7‰ overhead). The setup x-instance refers to the number of users x, with $x \in \{8, 16, 32\}$, deployed on either a m1.small instance identified as s or a m1.medium instance identified as m. In all setups, users keep generating messages at high rates to put pressure on the server's CPU. In this overloaded situation we never observe more than 2.3% overhead, showing that message latency is virtually unaffected by profiling.

While the overhead of the EER is highly related to the elasticity rules, we do not observe any noticeable overhead (i.e., over 1%) on any of the applications we evaluate. The EER overhead remains low thanks to: (1) its periodical execution, the EER only

Table 5.3.: PLASMA EPR overhead, normalized.

8-s	16-s	32-s	8-m	16-m	32-m
1.007	1.001	1.023	1.003	1.006	1.005

executes for a couple of seconds per period in our scenarios, and (2) the low rule count (i.e., less than 10) needed to cover applications elasticity requirements.

5.5.3 Metadata Server

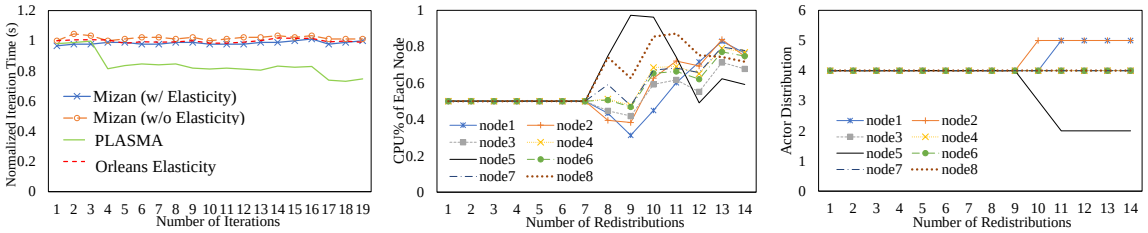
In our first scenario we display the effect of a simple mix of *Resource elasticity rules* and *Interaction elasticity rules* in PLASMA when deployed on a Metadata Server (cf. subsection 5.3.3).

In the experiment, we create 4 folders with 8 files in each. The server is deployed on an AWS m1.small instance, and 16 clients on another m1.medium instance. This setup overloads an m1.small instance, i.e., simulates a service under high demand. Among the 4 `Folder` actors, 1 actor receives 50% of requests from clients, and the other 3 evenly share the remaining 50%. `File` actors in a same folder have the same workloads. We compare three setups: (1) res-col-rule executes the `reserve` and `colocate` elasticity rule defined in subsection 5.3.3; (2) def-rule mimics a default rule, simply migrating actors with heavy workload (i.e., `Folder` actors) to an idle server; (3) no-rule does not conduct any elasticity management. The first two setups require an extra server; they use an elasticity time period of 80 s. We run each setup for ≈100 s to collect enough data before and after elasticity management.

Figure 5.5 shows that the elasticity rule (res-col-rule) reduces latency by 40% compared to both other setups. The second setup (def-rule) however does not display any visible latency benefit compared to the setup without elasticity (no-rule) because accessing a folder implies accessing the files contained in it, even when the `Folder` and `File` actors are on different servers. Therefore all accesses to a `Folder` actor on one server end up being forwarded to `File` actors on another server, provoking an overheard that nullifies the potential migration gains. This demonstrates the importance of application knowledge in elasticity management.

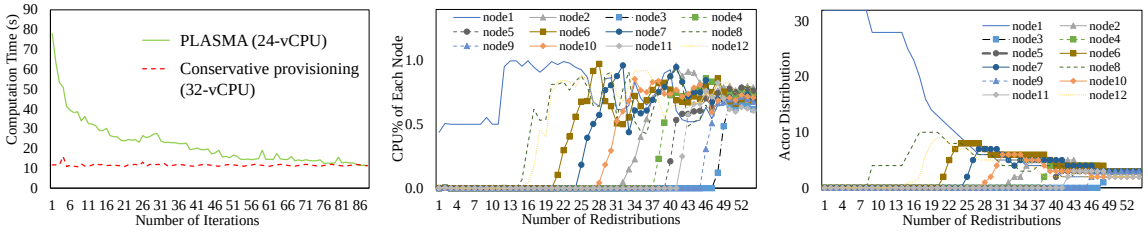
5.5.4 PageRank

In this scenario, we show the efficiency of PLASMA on a distributed actor-based variant of the popular PageRank [101] algorithm. We focus on the basic algorithm without specific optimizations as these do not address workload imbalances.



(a) Computation time of each iteration. (b) CPU% of each server over time. (c) Distribution of worker actors over time.

Figure 5.7.: PageRank dynamic workload balance. PLASMA achieves 24% faster iteration times after initial automated balancing.
(In **(b)** and **(c)**, each server is busy reading data in the early re-distributions.)



(a) Computation time of each iteration. (b) CPU% of each server over time. (c) Distribution of worker actors over time.

Figure 5.8.: PageRank dynamic resource allocation. PLASMA achieves the same application-level performance with 12 servers in comparison with the conservative provisioning case using 16 servers (with one worker per vCPU).

In our implementation, each `Worker` actor iteratively computes on one partition and synchronizes at the end of each iteration with the other workers to exchange computation results. Since all workers synchronize at the same time, the overall execution speed is limited by the slowest worker.

Though many partitioning schemes and systems have been proposed for partitioning graphs [57–59, 107] and thusly balancing workloads across workers, ensuring a fair workload distribution remains a non-trivial task. We use SNAP's LiveJournal online social network [116] as dataset. The graph is split with the popular graph partitioning tool METIS [107] that computes balanced partitions.

Dynamic workload balance We first show how PLASMA balances workloads among a *fixed* set of resources. This experiment uses 8 VMs, each being an AWS

m5.large instance (2 vCPUs and 8 GB memory), for a total of 16 vCPUs that are connected with 10 Gbps links. All runs are congestion-free.

We first compare PLASMA with elasticity management to the limited elasticity management of Orleans consisting in attempting to balance workload by putting the same number of actors on each server. We implement the same elasticity rules in AEON. We use METIS to evenly split the graph in 32 partitions, resulting in 32 actors, and *randomly* assign them across the 8 VMs. Since the number of actors is already balanced across servers, Orleans elasticity management does not take further action. PLASMA uses the same partition assignment, but combines it with a `balance` resource elasticity rule that sets the lower bound to 60% and upper bound to 80% (as for Piccolo in section 5.3.2). Once the PageRank application starts, PLASMA's EMR balances the worker actors among the 8 VMs based on their CPU resource usage, while the worker actors stay in the same VMs with Orleans' elasticity. The experiment is repeated 3 times for each of the 5 different random distributions used. Figure 5.6a shows that PageRank converges 24% faster with our elastic solution PLASMA redistributing actors, compared to Orleans elasticity management.

Figure 5.7b and Figure 5.7c show the detailed behavior in a given experiment run of the CPU consumption for each server and the per-server actor distribution respectively. Once worker actors finish reading data and start iterative computations, the CPU usage of each server starts diverging greatly despite the even partitioning performed by METIS. PLASMA detects load imbalance and moves worker actors from overloaded servers (e.g., server 5) to under-utilized ones (e.g., servers 1 and 2, until the CPU usage of servers falls between the upper and lower bounds (i.e., 60%-80%). As a result, PageRank converges faster since the computation time of each partition is more homogeneous with elastic PLASMA, and no one worker actor is lagging behind for the system-wide synchronization happening at the end of each iteration. This experiment clearly shows that under fixed resources (e.g., CPU), PLASMA can detect workload imbalance and improve resource efficiency by automatic actor re-location.

We further compare PLASMA with Mizan [56], a state-of-the-art graph processing system for dynamically balancing computation across servers via graph vertex migration. We run the open source Mizan [104] with both its default configuration (without elasticity) and its dynamic migration scheme (with elasticity) in the same setting as PLASMA– 8 AWS m5.large VMs and the LiveJournal graph with 32 partitions. Since the absolute iteration time of Mizan is about 4× longer than that of PLASMA, to compare elasticity effectiveness of the two solutions, we normalize each iteration time to the first iteration of the respective case without elasticity. Figure 5.7a shows Mizan with elasticity reduces iteration time by up to 3% compared to the case without elasticity. In contrast, PLASMA with elasticity reduces iteration time by up to 24% compared to the case without elasticity, showing that PLASMA balances load more effectively while being generic.

Dynamic resource allocation Trivially we can reach best PageRank convergence time by over-provisioning resources (conservative provisioning). We thus run 16 AWS m5.large instances for a total of 32 vCPUs, randomly mapping each of the 32 worker actors to their own vCPU. As expected, the convergence time is nearly halved (11.67 s, cf. Figure 5.6b) compared to that of a 16-vCPU setup (19.77 s, cf. Figure 5.6a). But can we achieve the same (or close) performance using fewer resources? To answer this question, we set PLASMA to allocate resources *dynamically* – we re-use the above `balance` rule, but once all of the existing servers are overloaded, PLASMA provisions a new server (cf. subsection 5.4.2). In our experiments (cf. Figure 5.8), we start with one running server for PLASMA and place all 32 worker actors on it. Figure 5.8b clearly shows PLASMA provisioning new servers (via AWS Instance Scheduler [117]) until it reaches a stable state where the CPU usage of every server is within the defined lower and upper bounds. Figure 5.8c shows the details of the worker actor re-distributions as PLASMA provisions new instances. Performance improves each round as PLASMA performs elasticity management gradually (cf. subsection 5.4.3) and inches towards an optimal actor distribution.

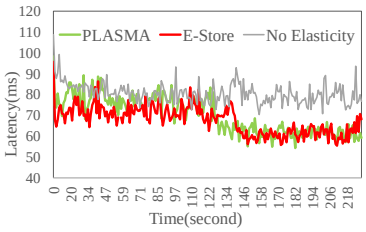
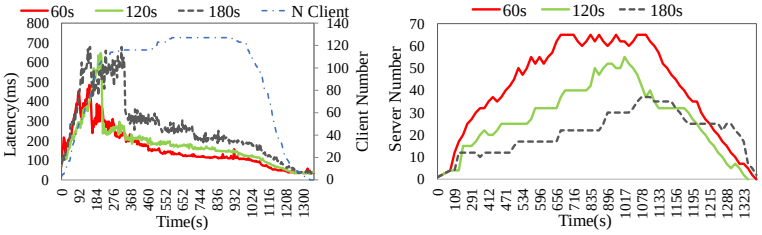


Figure 5.9.: Latency of E-Store application. Similar for E-Store and PLASMA E-Store.



(a) Average latency. (b) Number of servers.

Figure 5.10.: Elasticity management for the Media Service. A small elasticity period lowers the latency and fasten resources allocation/reclaiming.

Eventually, PLASMA performance comes very close to the best performance as shown in Figure 5.8a for one run and in Figure 5.6b for the average across runs. PLASMA achieves nearly identical performance with 12 servers as the over-provisioning case does with 16 servers, saving 25% of resources.

5.5.5 E-Store

Programmers often end up implementing specific elasticity management in their applications without help from specialized tools. We show here that PLASMA can deliver similar performance as in-app implemented elasticity management.

As detailed in subsection 5.3.3, E-Store [53] has to migrate hot key partitions to handle unbalanced workloads. At the same time, root level partitions must also be co-located with their descendants to avoid remote communication. We implemented E-Store in AEON [51] and added 3000 LoC to its runtime to include the specific platform details (e.g., CPU usage, actor placement) used by E-Store for elasticity. We compare the performance of this AEON E-Store with PLASMA E-Store that (only) executes the elasticity rules defined in subsection 5.3.3.

We evenly deploy 40 root level partitions of E-Store on 4 m1.small instances. Each such partition has 4 child partitions. Querying E-Store are 48 clients on another 2 m1.medium instances, generating unbalanced workload on partitions. The first root partition receives 35% of total requests; the second receives 35% of the remaining 65%

of requests; the third receives 35% of the requests remaining after that, aso. Requests arriving at a root partition will continue to access one child partition randomly. We also run a non-elastic version for comparison. During execution, AEON E-Store and PLASMA E-Store both require an extra instance.

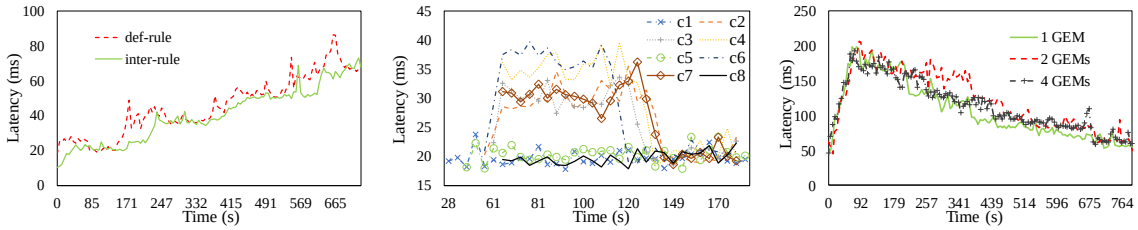
As Figure 5.9 shows, the performance of AEON E-store and PLASMA E-store are close to each other, and they both show obvious performance improvement compared to AEON E-Store without elasticity management (No Elasticity). After detailed analysis, we find elasticity behaviors in both versions are quite similar despite differences in their concrete elasticity management. For example, AEON E-Store migrates the top k% root partitions on overloaded servers to idle servers while PLASMA picks root partitions that receive a certain percentage of requests among all root partitions on the same server. This demonstrates the usefulness of PLASMA for implementing application-specific elasticity behavior.

5.5.6 Media Service

We next show how PLASMA improves the Media Service (subsection 5.3.3) performance with highly dynamic workloads, with a focus on showing the impact of elasticity time periods.

We deploy 128 clients on 8 AWS m1.small instances. In the first 10 minutes, these clients join the service (i.e., start making requests) following a normal distribution ($\mu = 2$ min, $\sigma = 90$ s) and they keep sending requests for 4 more minutes. They all leave the service starting from the 14 minute mark, for a period of 10 minutes, following a normal distribution ($\mu = 19$ min, $\sigma = 90$ s). Half of the clients' requests are "watch movie" and half are "review movie".

On the server side, the Media Service is deployed on multiple m1.small instances, 4 instances initially, and can scale up to 65 instances. One `UserInfo` or `UserReview` actor only serves one client while all other actors serve two clients each (e.g., `FrontEnd`). More actors are created as more clients access the service. As the number of clients



(a) Latency: interaction vs frequency rule. (b) Single client latency with frequency rule. (c) CPU workload balance for router actors.

Figure 5.11.: Elasticity management for Halo Presence Service. **(a)** shows the elasticity rules enable smoother player latency evolution, **(b)** shows the importance of actors colocation, and **(c)** shows the slight impact on latency of number of used GEM(s).

(i.e., workload) evolves, PLASMA's runtime gradually adjusts those actors' placement to ensure low request latency. We start 1 GEM to execute elasticity rules as described in section 5.3.2 and run a scenario per elasticity period of 60 s, 120 s and 180 s.

As shown in Figure 5.10a, a smaller elasticity time period enables a better responsiveness from PLASMA's runtime, with the 60 s elasticity period displaying the best latency results. Figure 5.10b details the number of servers used by the application over time under different elasticity time periods: PLASMA's runtime — with shorter elasticity time period — can allocate and reclaim resources in a faster manner corresponding to the workload dynamics. Yet too frequent elasticity management may incur additional overhead. This clearly shows PLASMA works well with more (e.g., 6) elasticity rules under dynamic workload, with a well-chosen elasticity time period.

5.5.7 Halo Presence Service

In this scenario, we show how to improve performance of the Halo Presence Service [102] with rules for different types of actors, and assess the effect of the number of GEMs used.

Interaction elasticity rule As discussed in subsection 5.3.3, a `Session` actor can only send messages to `Player` actors in it, suggesting that they

be co-located. We could also instead use a rule that co-locates actors that *frequently* interact with one another, but this rule can lead to poor migration choices (e.g., if a router actor happens to send many messages to one session actor for a while). Moreover, *frequent* can have various interpretations. We use this frequency-based rule as our default rule (baseline) for our evaluation.

The experiment setup is composed of 8 `Router` actors and 8 `Session` actors that are deployed on 8 AWS m1.small servers, with one of each actor type deployed per server. To highlight remote messaging latency, router actors do not perform decryption and forward messages directly. We simulate players behind game consoles by starting 32 clients on another 2 AWS m1.medium servers. The 32 clients join the game in 4 rounds (of 180 s each), with 8 clients joining per round, each at a random time during the round. A joining client is assigned to a session and the application creates a corresponding `Player` actor for it. Depending on the rule in place, this new `Player` actor either (1) gets co-located with its session actor when the aforementioned elasticity rule is established, or (2) is first placed on a random server and the default rule attempts to co-locate it with the actors it frequently interacts with. Only one GEM is started and the elasticity period is set to 70 s.

Figure 5.11a depicts the average message latencies resulting from the two rules. With the elasticity rule (inter-rule), clients largely avoid remote messaging from the start of the experiment while the default rule (def-rule) leads to degraded performance for certain timespans (e.g., 0 s–85 s, 171 s–247 s). Only once `Player` actors are co-located with their `Session` actor do message latencies become similar (85 s–171 s, 247 s–332 s). The elasticity rule enables smoother latency evolution for an enhanced user (i.e., Halo player) experience.

Figure 5.11b shows the detailed performance of each client for the first round of a single run under the default rule. Out of the 8 joining clients, c1, c5 and c8 are fortuitously instantiated on the right servers for their `Session` actor, while c2, c3, c4, c6 and c7 experience a latency between 30 ms and 40 ms, which is ≈35% higher than that of well-placed clients. All clients' latency is reduced to 20 ms after re-

distribution (after 70 s of presence). Note that high latency in the first few minutes may limit a player's interest in the game. Attempting to obtain better results by shortening the elasticity period might lead to overzealous actor migration that can worsen performance.

Resource elasticity rule As mentioned in subsection 5.3.3, we need to provide `Router` actors with enough CPU resources. To demonstrate the efficiency of this rule, we define a setup made of 64 `Session` actors and 32 `Router` actors and deploy them on 64 AWS `m1.small` servers. Each `Session` actor is hosted on a separate server whereas `Router` actors are initially evenly distributed across 8 of these servers. We run up to 128 clients (i.e., 128 `Player` actors) on 8 AWS `m1.medium` servers with a varying number of GEMs: 1, 2, and 4. The elasticity time period is set to 80 s.

Figure 5.11c shows a sudden rise in average latency as more and more clients join the game, indicating that the 8 servers with `Router` actors are overloaded. In response, the GEM(s) start to balance the workload as per the resource elasticity rule until each `Router` actor ends on a server with enough resources, allowing latency to stabilize. Additionally, we see that deploying several GEMs, for scalability and fault tolerance, only has a small impact on latency.

5.6 Summary

Existing automatic elasticity solutions (e.g., serverless computing) simplify development and deployment of distributed applications executing in third-party infrastructure by providing simple abstractions such as functions, and dealing with resource provisioning completely automatically in the face of fluctuating workloads. PLASMA introduces the same benefits to state*ful* applications by complementing the actor-based programming model with: (1) a second "level" of programming for delineating *actor-condition-behavior* rules that drive elasticity management; (2) an elasticity-aware runtime that accordingly profiles actors of specified types and applies

corresponding actions. While a core design goal was to keep PLASMA's elasticity programming language simple, we are investigating several extensions.

6 CONCLUSION

In the dissertation, we have presented a variant of the actor model AEON, which is specialized for cloud setups. Actors communicate across hosts over the network and are migrated for scale adjustment in the response of the workload. We believe our AEON language and runtime (a) ensure serializability guarantees for multi-actor interaction, (b) enable a high degree of parallelism in networked distributed systems, and (c) support efficient actor migration. Our model provides serializability and deadlock freedom, with largely decentralized synchronization and thus scalability for server-side cloud applications or components following a DAG-based structure of actors. We have empirically demonstrated the strong potential for scalability and presented the usability of our model through case studies of wide-ranging applications.

Furthermore, based on our AEON actor programming language, we have proposed a novel solution PLASMA, which allows programmers to customize the elasticity management for their cloud applications. Compared to existing solutions (e.g., AWS Lambda), which simplify development and deployment of distributed applications executing in third-party infrastructure by providing simple abstractions such as functions, and dealing with resource provisioning completely automatically in the face of fluctuating workloads, PLASMA introduces the same benefits to state*ful* applications by complementing the actor-based programming model with: (1) a second "level" of programming for delineating *actor-condition-behavior* rules that drive elasticity management; (2) an elasticity-aware runtime that accordingly profiles actors of specified types and applies corresponding actions.

With these programming supports (i.e., AEON and PLASMA), programmers can implement, deploy and manage their stateful elastic cloud applications in a much simpler manner.

REFERENCES

[1] AWS Lambda. `https://aws.amazon.com/lambda/`.

[2] Azure Durable Function. `https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview`.

[3] Philip A Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans : Distributed virtual actors for programmability and scalability. Technical report, Microsoft Research, 2014.

[4] Wei-Chiu Chuang, Bo Sang, Sunghwan Yoo, Rui Gu, Milind Kulkarni, and Charles Edwin Killian. EventWave: Programming Model and Runtime Support for Tightly-coupled Elastic Cloud Applications. In *Proceedings of the 4th ACM Symposium on Cloud Computing, SoCC'13*, pages 21:1–21:16, 2013.

[5] Piotr Nienaltowski, Volkan Arslan, and Bertrand Meyer. Concurrent object-oriented programming on .NET. *IEE Proceedings - Software*, 150(5):308–314, 2003.

[6] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.

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

[8] Peter Dinges and Gul Agha. Scoped synchronization constraints for large scale actor systems. In *Coordination Models and Languages - 14th International Conference, COORDINATION'12*, pages 89–103, 2012.

[9] Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *Proceedings of the 16th IFIP WG 6.1 International Conference on Coordination Models and Languages - Volume 8459*, pages 131–146, 2014.

[10] Carlos H. C. Duarte. Proof-theoretic foundations for the design of actor systems. *Mathematical. Structures in Comp. Sci.*, 9(3):227–252, 1999.

[11] Arnd Poetzsch-Heffter, Ilham W. Kurnia, and Feller Christoph. 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*, pages 289–305, 2011.

[12] Ilham W. Kurnia and Arnd Poetzsch-Heffter. 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*, pages 47–58, 2012.

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

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

[15] Guy Golan-Gueta, Nathan Grasso Bronson, Alex Aiken, G. Ramalingam, Mooly Sagiv, and Eran Yahav. 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*, pages 225–242, 2011.

[16] John Field and Carlos A. Varela. 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*, pages 195–208, 2005.

[17] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-oriented Programming, ECOOP'10*, pages 354–378, 2010.

[18] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!'15*, pages 1–12, 2015.

[19] Elias Castegren and Tobias Wrigstad. Reference capabilities for concurrency control. In *30th European Conference on Object-Oriented Programming, ECOOP'16*, pages 5:1–5:26, 2016.

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

[21] Carlos A. Varela and Gul Agha. A hierarchical model for coordination of concurrent activities. In *Proceedings of the Third International Conference on Coordination Languages and Models, COORDINATION'99*, pages 166–182, 1999.

[22] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, 2003.

[23] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris C. Kirkham, and Ian Watson. Distm: A software transactional memory framework for clusters. In *2008 International Conference on Parallel Processing, ICPP'08*, pages 51–58, 2008.

[24] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[25] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for java. In *22nd European Conference on Object-Oriented Programming, ECOOP'08*, pages 129–154, 2008.

[26] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. 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!@SPLASH'18*, pages 33–43, 2018.

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

[28] Infinispan. Infinispan, January 2020.

[29] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Lightweight multi-key transactions for key-value stores. *CoRR*, abs/1509.07815, 2015.

[30] Autoscaling. `https://aws.amazon.com/autoscaling/`.

[31] Autoscaling Groups of Instances. `https://cloud.google.com/compute/docs/autoscaler/`.

[32] Azure Autoscale. `https://azure.microsoft.com/en-us/features/autoscale/`.

[33] Autoscaling with Heat. `https://docs.openstack.org/senlin/latest/scenarios/autoscaling_heat.html`.

[34] Guy Bieber and Jeff Carpenter. Introduction to service-oriented programming (rev 2.1). 2001.

[35] OASIS. OASIS SOA Reference Model TC. `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm`.

[36] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[37] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[38] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: elastic distributed resource scaling for infrastructure-as-a-service. In *10th International Conference on Autonomic Computing, ICAC'13*, pages 69–82, 2013.

[39] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: predictive elastic resource scaling for cloud systems. In *Proceedings of the 6th International Conference on Network and Service Management, CNSM'10*, pages 9–16, 2010.

[40] Serverless. `https://cloud.google.com/serverless/`.

[41] Serverless Computing. `https://azure.microsoft.com/en-us/overview/serverless-computing/`.

[42] Alex Ellis. Introducing Functions as a Service (OpenFaaS). `https://blog.alexellis.io/introducing-functions-as-a-service/`.

[43] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. *Elastic*, 60:80, 2016.

[44] Amazon S3. `https://aws.amazon.com/s3/`.

[45] Memcached. `http://memcached.org/`.

[46] Redis. `https://redis.io/`.

[47] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. 2010.

[48] Shaohua Wang, Iman Keivanloo, and Ying Zou. How do developers react to restful api evolution? In *International Conference on Service-Oriented Computing*, pages 245–259. Springer, 2014.

[49] Michael Zur Muehlen, Jeffrey V Nickerson, and Keith D Swenson. Developing web services choreography standards–the case of REST vs. SOAP. *Decision Support Systems*, 40(1):9–29, 2005.

[50] Andrew Newell, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein. Optimizing distributed actor systems for dynamic interactive services. pages 38:1–38:15, 2016.

[51] Bo Sang, Gustavo Petri, Masoud Saeida Ardekani, Srivatsan Ravi, and Patrick Th. Eugster. Programming scalable cloud services with AEON. In *Proceedings of the 17th International Middleware Conference*, pages 16:1–16:14, 2016.

[52] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: autonomic elasticity manager for cloud-based key-value stores. In *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13*, pages 115–116, 2013.

[53] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing. *PVLDB*, 8(3):245–256, 2014.

[54] Bailu Ding, Lucja Kot, Alan J. Demers, and Johannes Gehrke. Centiman: elastic, high performance optimistic concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC'15*, pages 262–275, 2015.

[55] Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikolaos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine(s) that could: scaling online social networks. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 375–386, 2010.

[56] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys'16. ACM, 2013.

[57] Apache Incubator Giraph. `http://incubator.apache.org/giraph/`.

[58] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[59] Semih Salihoglu and Jennifer Widom. Gps: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[60] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI'18*, pages 561–577, 2018.

[61] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI'19*, pages 193–206, 2019.

[62] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. *;login:*, 44(1), 2019.

[63] Amazon ElastiCache Redis. `https://aws.amazon.com/elasticache/redis/`.

[64] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. 1987.

[65] Bo Sang, Srivatsan Ravi, Gustavo Petri, Mahsa Najafzadeh, Masoud Saeida Ardekani, and Patrick Eugster. Programmable elasticity for actor-based cloud applications. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, pages 15–21, 2017.

[66] Chandra Krintz. Cloud computing. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[67] Gul Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, 1990.

[68] Akka. `https://akka.io/`.

[69] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, February 2009.

[70] Microsoft. Asynchronous Agents Library, January 2020.

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

[72] Akka.NET, January 2020.

[73] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. pages 16:1–16:14, 2011.

[74] Orleans. Orleans, January 2020.

[75] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26:631–653, 1979.

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

[77] AWS. AWS, January 2020.

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

[79] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. In *Proceedings of the 19th International Conference on Distributed Computing, DISC'05*, pages 324–338, 2005.

[80] Dominik Aumayr, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 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*, pages 157–171, 2019.

[81] Red Hat. JBoss Middleware, January 2020.

[82] Microsoft. Who is Using Orleans?, January 2020.

[83] Shams M. Imam and Vivek Sarkar. 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*, pages 67–80, 2014.

[84] Robert Escriva and Emin Gün Sirer. The design and implementation of the warp transactional filesystem. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI'16*, pages 469–483, 2016.

[85] HyperDex Warp. GyperDex Warp, January 2020.

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

[87] Apache. Hadoop, January 2020.

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

[89] Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. A practical scalable distributed b-tree. *PVLDB*, 1(1):598–609, 2008.

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

[91] Wikipedia. DDR3 SDRAM, January 2020.

[92] Danamma M Bulla and VR Udupi. Cloud Billing Model: A Review. In *International Journal of Computer Science and Information Technologies*, pages 1455–1458, 2014.

[93] Module: Coordinate a Serverless Image Processing Workflow with AWS Step Functions. `https://github.com/aws-samples/aws-serverless-workshops/tree/master/ImageProcessing`.

[94] Cloud IoT Core. `https://cloud.google.com/iot-core/`.

[95] Tomás Cerný, Michael J. Donahoo, and Jiri Pechanec. Disambiguation and comparison of soa, microservices and self-contained systems. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems, RACS'17*, pages 228–235, 2017.

[96] Amazon Lambda Programming Model. `https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-features.html#gettingstarted-features-programmingmodel`.

[97] Cloud Functions Programming Model. `https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference`.

[98] Amazon CloudWatch. `https://aws.amazon.com/cloudwatch/`.

[99] Overview of Azure Monitor . `https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/monitoring-overview-azure-monitor`.

[100] Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[101] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 107–117, 1998.

[102] Caitie McCaffrey. Architecting and launching the halo 4 services. In *USENIX Association*, Santa Clara, CA, 2015.

[103] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'19*, pages 3–18, 2019.

[104] The Graphs Blog. https://thegraphsblog.wordpress.com/presentations/.

[105] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *PVLDB*, 7(12):1035–1046, 2014.

[106] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles, SOSP'07*, pages 205–220, 2007.

[107] METIS Graph Partition Library. http://exoplanet.eu/catalog.php.

[108] Erlang. https://www.erlang.org/.

[109] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[110] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. Fate and destini: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, 2011.

[111] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. zexpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16, pages 14:1–14:15, 2016.

[112] Bakhtiar Khan Kasi and Anita Sarma. Cassandra: Proactive conflict minimization through optimized task scheduling. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 732–741, 2013.

[113] Swarnendu Biswas, Minjia Zhang, Michael D Bond, and Brandon Lucia. Valor: efficient, software-only region conflict exceptions. *ACM SIGPLAN Notices*, 50(10):241–259, 2015.

[114] Mário Luís Guimarães and António Rito Silva. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering*, pages 342–352. IEEE Press, 2012.

[115] Brendan Jennings and Rolf Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, 23(3):567–619, Jul 2015.

[116] SNAP. https://snap.stanford.edu/data/.

[117] AWS Instance Scheduler. https://aws.amazon.com/answers/infrastructure-management/instance-scheduler/.