Programming Scalable Cloud Services with AEON[*]

Bo Sang
Purdue University
bsang@purdue.edu

Gustavo Petri
IRIF – Université Paris Diderot
gpetri@univ-paris-diderot.fr

Masoud Saeida Ardekani[†]
Purdue University
msaeidaa@purdue.edu

Srivatsan Ravi
Purdue University
srivatsanravi@purdue.edu

Patrick Eugster
Purdue University,
TU Darmstadt
p@cs.purdue.edu

ABSTRACT

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.

This paper addresses this problem by introducing AEON: a framework that provides the following properties: (i) *Programmability*: programmers only need to reason about sequential semantics when reasoning about concurrency resulting from multi-actor events; (ii) *Scalability*: AEON runtime protocol guarantees *serializable* and *starvation-free* execution of multi-actor events, while maximizing parallel execution; (iii) *Elasticity*: AEON supports fine-grained *elasticity* enabling the programmer to transparently migrate individual actors without violating the consistency or entailing significant performance overheads.

Our empirical results show that it is possible to combine the best of all the above three worlds without compromising on the application performance.

CCS Concepts

•Computing methodologies → Distributed programming languages;

[*]Supported by NSF grants # 1117065, # 1421910, and # 1618923, European Research Council grant # FP7-617805 "LiVeSoft – Lightweight Verification of Software" and German Research Foundation under grant # SFB-1053 "MAKI – Multi-mechanism Adaptation for Future Internet".

[†]Now at Samsung Research America.

Middleware'16, December 12-16, 2016, Trento, Italy
© 2016 ACM. ISBN 978-1-4503-4300-8/16/12…$15.00
DOI: http://dx.doi.org/10.1145/2988336.2988352

Keywords

Cloud elasticity, actor system, strong consistency, scalability

1. INTRODUCTION

Providing cloud-based distributed solutions, and adequately leveraging the various capabilities provided by cloud providers is pivotal to many modern low-latency cloud-based applications and services. However, many of these applications (and services) still follow the de facto Internet architecture consisting of stateless front and middle tiers, equipped with a stateful storage tier at the back-end. Since most services must use this storage back-end, the scalability of the system as a whole is limited by the latency and throughput of the storage. To overcome this limitation, it is common practice to add a caching mechanism. While a caching middle tier might be effective in enhancing scalability, it comes at the cost of relaxing the concurrency control provided by the storage back-end. Moreover, this solution fails to exploit the inherent data locality of the application, since cache requests need to be shipped to other processes, potentially residing on a different virtual machine.

An alternative to the above architecture which has the potential to overcome these problems is to build a stateful middle tier using modern programming models based on actors. Actors encapsulate state and communicate with each other by sending *events*. In the actor model, consistency is guaranteed if each event only accesses a single actor, thus eliminating potential data races and deadlocks. Yet, this level of abstraction provided by many existing solutions (e.g., Erlang, Akka) is not appropriate for cloud-based programming since it is nontrivial to provide consistency for events spanning across multiple actors. Typically, the developer using these models still needs to deal with distributed systems and cloud programming issues such as asynchrony, failures and deadlock, to mention but a few.

Recent industrial and academic efforts have proposed actor-based frameworks (e.g., Orleans [8, 5] and EventWave [9]) for building and deploying cloud-based services. For example, Microsoft's Orleans is being used to implement many services, including Skype and the Halo game services [2].

All of these frameworks attempt to ensure a subset of the following properties: (i) Programmability: the simplicity of the framework is paramount to reduce the learning effort, increase developers' productivity, and guarantee the platform adoption. This aspect can be achieved by providing to the programmer the illusion of sequential semantics, hence ignoring the consistency challenges that may arise when the service runs in the cloud. (ii) Scalability: to effectively cope with unpredictable workloads, the framework – and in particular its runtime system – must be able to function at different scales; (iii) Elasticity: to achieve an economical solution, the framework must be able to automatically scale both in and out by adding and releasing resources to adapt to the workload at hand. Moreover, such workload adaptation should not violate application invariants or completely stall the computation.

This paper introduces AEON: a distributed framework that addresses the three concerns above as follows:

(i) To achieve *programmability*, AEON enables reasoning about multi-actor events with sequential semantics in mind. Specifically, AEON applications are modeled as a *partially-ordered set* of dynamically interacting *contexts* that, akin to actors, represent units of data encapsulation. Our protocol ensures that all the events are executed in an atomic and strongly consistent manner (à la strict-serializability in transactional systems). In other words, AEON provides to the programmer the illusion of a server answering to asynchronous requests one at a time in a sequential manner.

(ii) Partial-ordering of contexts in AEON induces an *ownership network* to organize contexts, whereby access to a context is only granted to the contexts that directly *own* it. This partial ordering results in a directed acyclic graph (DAG) of contexts that is the key for AEON to implement an efficient deadlock-free and *starvation-free* synchronization protocol. This protocol maximizes parallel execution of client request events, and is therefore highly scalable. This is in stark contrast to the synchronization employed in Orleans [8, 5], which does not provide strict serializability, or EventWave [9] which severely limits scalability by employing a global synchronization bottleneck.

(iii) As foundation for *elasticity*, AEONs runtime system allows for transparently migrating contexts across different servers of the system without affecting the semantics of the application, and thus dynamically adjusts the number of utilized virtual machines to the actual workload. Specifically, contexts can be automatically distributed across a data center without exposing the actual location of contexts in the network (i.e., it enforces *location transparency* [21]).

We have implemented a highly available and fault-tolerant prototype of AEON in C++. Our empirical results show that it is possible to combine the best of all three worlds: programmability, scalability and elasticity without compromising on the application performance.

Concretely we make the following contributions: **(1)** After detailing challenges in developing elastic software in existing state-of-the-art paradigms such as EventWave [9] and Microsoft's Orleans [8], we present a novel programming model for building elastic cloud applications in a seamless and effortless fashion (§ 3). **(2)** The runtime of AEON implements a novel protocol for executing events in a strict serializable and highly scalable manner (§ 4) **(3)** AEON's runtime supports *customizable* automatic elasticity through the novel notion of an *elasticity manager* (§ 5). **(4)** We report an extensive evaluation, where we compare AEON against EventWave and Orleans on Amazon EC2 through a game application and the standard TPC-C benchmarks for transactional systems (§ 6). Related work and final remarks are the subjects of § 7 and § 8 respectively.

The AEON code along with some extended details, including the operational semantics, are available on the AEON website: https://www.cs.purdue.edu/homes/bsang/aeon/

2. OVERVIEW

In this section, we first identify the challenges with *programming* support for *scalable* cloud services and applications and summarize the drawbacks of existing solutions to the problem. We then provide an overview of AEON, and illustrate how it addresses these challenges.

2.1 Existing Work and Drawbacks

There exist some efforts towards frameworks that help implement scalable elastic applications while reducing programming effort. *EventWave* [9] and *Orleans* [8] are two important works in this space.

Orleans. Orleans is an open-source framework, developed by Microsoft, based on the actor model. It introduces the concept of *grains.* Akin to actors, grains are single-threaded. There are two types of grains: stateful and stateless. Although Orleans was initially described to support transactions [8], the current open-source version does not provide transactional guarantees. However, for many cloud applications, transaction(al) execution is required for correctness since the manual implementation of distributed transactions always requires considerable effort. Moreover, it's easy to run into *deadlocks* in Orleans with (a cycle of) synchronous method calls because general grains are single-threaded and do not allow reentrance. Finally, re-distribution of grains is supported in Orleans, but the migration process provides no guarantees that the application semantics will be unaffected [27].

EventWave. EventWave is the nearest programming model to AEON in which applications are modeled as a *tree* of contexts. EventWave guarantees strict-serializability by totally ordering all requests at the (single) root context, assigning an unique id to each request and executing events in order of their ids. Consequently, EventWave provides only *minimal progress* [18]. This clearly limits scalability and overall performance, as adding more servers provides only limited benefits due to the bottleneck at the tree root. Moreover, EventWave only provides a simple API for the programmer to manually migrate contexts to specific servers by halting all executions during migration. This severely hampers elasticity and introduces a nontrivial performance degradation. EventWave also provides limited programmability since it organizes contexts strictly as a tree and does not support modification of tree edges. This prevents programmers from implementing classic distributed data structures such as B-trees and list-sets. § 7 covers the drawbacks of other (perhaps less) related programming models for the cloud.

2.2 AEON Overview

Consider a massively multiplayer online (MMO) game, where players can circulate through an arena containing different buildings and rooms, each containing different objects. The players can interact with other players and objects in the same room. Such a MMO game has to process thousands of concurrent requests in an *asynchronous* environment, thus emphasizing the need for an efficient protocol to synchronize client requests. When there are too many online players and existing physical servers become contended, new servers must be allocated and some players must be migrated to those. Such players will still be interacting with other players and objects, and so the game service must handle the migrations both quickly and correctly.

Atomic Events and Ownership Network (AEON) is a general programming framework designed precisely to solve these problems. AEON allows the programmer to write applications assuming a sequential semantics. The AEON runtime system efficiently utilizes the distributed computing resources and supports seamless resource migration without sacrificing the application's correctness, thus relieving the application programmer of dealing with intricate concurrency issues.

Programmability. In Figure 1 we outline a simplified AEON implementation of our game. AEON takes an Object-Oriented (OO) approach to implement the server-side logic – the structure of the program follows a standard OO programming approach if we substitute the `contextclass` keyword by `class`, except for a few keywords that we will explain shortly in § 3. Defining object structures as `contextclass`es instead of regular classes means that their instances will be automatically distributed, and relocated under workload pressure by the AEON runtime system as needed. Notice that the programmer does not need to implement any additional logic for the application to adapt to workloads.

For instance, suppose a client wants to put 50 gold coins into `treasure` from `gold_mine`. To this end, she issues a call of the form `event player1.get_gold(50)`. The only difference between an event call and a normal remote method call is the `event` call decoration, which indicates to the runtime system that the call must be executed as an event. This annotation on the call site (as opposed to the method declaration) permits the reuse of methods, e.g., `get_gold`, both as events for client calls and as conventional synchronous methods in the case of another context calling it.

While asynchronous calls and events have been proposed before, the AEON programming model relieves the programmer from reasoning about race conditions, or tediously implementing synchronization mechanisms. AEON guarantees strict serializability. Therefore, events change the state of multiple contexts (i.e., instances of `contextclass`es) even residing on different machines, while maintaining the appearance of executing atomically and sequentially. In our example, an event call to `updateTimeOfDay` in a `Building` context updates the time in all of the rooms before executing any subsequent event.

Scalability. In the interest of maximizing *scalability*, the programmer would like to execute requests from different users in parallel. However, it is not always the case that requests from different users operate on disjoint data. In the case where two or more requests operate on the same data, an efficient arbitration mechanism must be put in place to avoid strict serializability violations. Importantly, this mechanism should also avoid the possibility of deadlocks.

AEON employs a flavor of ownership types (akin to [7, 16] proposed for concurrent programming) to facilitate parallel yet atomic executions of distributed events: contexts form a directed acyclic graph (precisely, a *join semi-lattice* as detailed in § 3) structure indicative of their state sharing. Two events can run in parallel as long as they do not access shared portions of state. In AEON, a simple static analysis guarantees that the context graph derived from the context-accessibility (i.e. ownership hierarchy) between different contexts is *acyclic*. In the example, we can see that a

```
contextclass Building {
  void updateTimeOfDay () { // change time of day in
       parallel
    for (Room* room in children[Room])
       async room->updateTimeOfDay();
  }
  readonly int countPlayers() {    // read-only method
    for (Room* room in children[Room])
       count =+ room->nr_players();
    return count;
  }
  ...
}

contextclass Room {
  readonly int nr_players()        // read-only method
    { return children[Player].size(); }
  readonly int nr_items( )
    { return children[Item].size(); }
  void updateTimeOfDay() { ... }
  ...
}

contextclass Player {
  int playerId;
  Item* gold_mine;
  Item* treasure;
  bool get_gold(int amt) {
    if( gold_mine->get(amt) )
      treasure->put(playerId, amt);
    ...
  }
  ...
}
```

Listing 1: Simplified game example. Fields of contextclasses are not shown. Red keywords represent new AEON constructs.

`Player` can own any number of `Items`, but not vice-versa.

Assuming two contexts of type `Player` sharing a common child of type `Item`, to guarantee the atomic execution of an event targeting one of the `Player` contexts, AEON delays the execution of events targeting the other `Player` until the former event is terminated. Otherwise, the shared `Item` context could be the source of data races, invalidating the serializable execution of both events. However, if two events are sent to `Players` in different `Rooms`, they can be executed in parallel without violating strict serializability of the system since they have no shared children. This enables a high degree of parallelization since a majority of events sent by different clients do not intersect.

Elasticity. To build a scalable distributed application that caters to dynamic workloads, the programmer would have to implement logic to: (i) migrate both data and computation between servers in case of a change in the workload; (ii) resolve which server has which pieces of data at any given time (which is non-trivial given that data might migrate); (iii) guarantee that ongoing requests are not disrupted by migrations. Writing even simple applications which meet the desired scalability criteria would require expert programmers in distributed systems, and even in that case it would remain an error-prone, time-consuming, and expensive endeavor. To avoid that such concerns related to distribution outweigh the concerns related to the actual program logic, AEON employs efficient migration protocols together with an *elasticity manager* that enables the programmer to spec-

	EventWave [9]	Orleans [8]	AEON
Data encapsulation	Contexts	Grains	Contexts
Programmability restraint	Context tree	Unordered grains	Context DAG
Event consistency across actors	Strict serializability	No guarantees	Strict serializability
Event progress	Minimal(due to sequential bottleneck)	Possibility of deadlocks	Starvation-freedom [19]
Automatic elasticity	No	Yes [27]	Yes

Figure 1: Summary of distributed programming models for building cloud-based stateful applications

Variables $x, y \in \mathcal{V}ar$	Expressions $e \in \mathcal{E}xp$
Method Names $m \in \mathcal{M}$	Field Names $f \in \mathcal{F}$
Class Names $cls \in \mathcal{C}ls$	Contextclass Names $\mathtt{C} \in \mathcal{C}tx$

Program Def.	$p \in \mathcal{P}$	$::=$	$\overrightarrow{cxd}\ \overrightarrow{clsd}\ \mathtt{main}(\ldots)\{\ s\ \}$
Contextclass Def.	$cxd \in \mathcal{C}tx\mathcal{D}$	$::=$	$\mathtt{contextclass}\ \mathtt{C}\ \{\ \underline{\overrightarrow{fd}}\ \underline{\overrightarrow{md}}\ \}$
Class Def.	$clsd \in \mathcal{C}ls\mathcal{D}$	$::=$	$\mathtt{class}\ cls\ \{\ \overrightarrow{fd}\ \overrightarrow{md}\ \}$
Type	$\tau \in \mathcal{T}$	$::=$	$\underline{\mathtt{C}} \mid cls \mid \mathtt{int} \mid \mathtt{float} \mid \tau[\,] \mid \ldots$
Field Def.	$fd \in \mathcal{FD}$	$::=$	$\tau\ f$
Method Def.	$md \in \mathcal{MD}$	$::=$	$\mathtt{ro}^{?}\ \tau\ m(\overrightarrow{\underline{\tau}\ x})\ \{\ s\ \}$
Decorated Call	$dc \in \mathcal{DC}all$	$::=$	$\mathtt{event}\ x.g(\vec{x}) \mid \mathtt{async}\ x.g(\vec{x})$
Statements	$s \in \mathcal{S}$	$::=$	$dc \mid \ldots$

Figure 2: Syntax of AEON (excerpt). Underlined types are only allowed in the declarations of context fields and methods, not in class declarations.

ify how contexts scale in/out. In our game example, the elasticity manager can easily move `Room` and `Player` contexts to different servers of the system when their current virtual machines become overloaded. For example, a player that starts a computation-intensive task might be migrated to a single virtual machine for the duration of the task. Figure 1 summarizes the properties provided by AEON with respect to Orleans and EventWave.

3. PROGRAMMING MODEL

In this section we describe the principal programming abstractions offered by AEON. Let us start by presenting a simplified abstract syntax of AEON in Figure 2. Notice first that AEON provides class declarations, as well as methods and fields like most mainstream OO programming languages. In addition, the language provides syntax for the declaration of *contextclasses*.

Classes and contextclasses. An AEON program comprises a series of contextclass declarations, a series of class declarations, and a `main` function which starts the execution of the AEON program. A context (an instance of a contextclass) is a *stateful point of service* that receives and processes requests either (i) in the form of *events* from clients, or (ii) in the form of *remote method calls* from other contexts. At a high level, a context can be considered as a container object or composite object that can be relocated between hosts. Contexts encapsulate local state (in the form of fields) and functionality (in the form of exported methods or events). In particular, AEON contexts hide internal data representations, which can only be read or affected through their methods.

Another aspect that distinguishes contextclass declarations from the standard class declarations is that types appearing in contextclass field and method declarations can also contain context-type expressions, underlined as $\underline{\tau}$ in Figure 2. By inspecting the rule for types, we can see that contextclass names can thus be used as types only in contextclass level code, but not in normal classes. Thus, we vastly simplify the management of references (for example for *garbage collection*, in that passing an object by value does not implicitly create new references to contexts), and enable a simple static analysis to check that ownership respects a DAG structure as we shall describe shortly. Note that this restriction may be relaxed in future revisions of AEON.

Context ownership network. In a nutshell, AEON contexts are guarded by an ownership mechanism loosely inspired by the ones proposed in [3, 7]. The concept of ownership allows AEON to establish a partial order among contexts (when considered transitively), and thus guarantees deadlock freedom when executing events.

We say that a context `C` is "directly-owned" by another context `C'` if any of the fields of `C'` contains a reference to `C` (we shall sometimes call the inverse relation of directly-owned "parent-child"). The ownership relation described above takes into account the transitive closure of the directly-owned relation. To the right of Figure 3, we depict a possible runtime ownership DAG for the application described in Figure 1. Here a `Castle` context of type `Building` owns two `Room` contexts: the `Kings Room`, and an `Armory`. In turn, each of the `Rooms` owns the respective `Players` currently in them, and a number of accessible `Items`. `Players` can also own `Items`. In addition, some contexts like `Treasure` can be owned by multiple contexts, `Player1`, `Player2`, and the `Kings Room`. Moreover, several contexts can own the same context, leading to a form of multi-ownership, which allows the sharing of state, a prevalent characteristic of object-oriented programming.

The ownership network enables the safe parallel execution of events provided that they do not access shared state. When multiple concurrent events can potentially access the same state, AEON serializes the events by exploiting the ownership network. The DAG structure of the ownership network guarantees that for any two contexts that might have a common descendant context, there exists an ancestor context that transitively owns both (we have a join-semi-lattice).[1] In particular, for any set of contexts that have a common set of descendants, we are interested in the *least common ancestor* dominating them. Formally: for context `C` in an ownership network `G`, assuming that $\mathsf{desc}(\mathtt{G},\mathtt{C})$ represents the set of its descendant contexts, let $\mathsf{share}(\mathtt{G},\mathtt{C})$ be the set defined as follows:

$$\begin{aligned}\mathsf{share}(\mathtt{G},\mathtt{C}) = &\left\{\mathtt{C}' \mid \mathsf{desc}(\mathtt{G},\mathtt{C}) \cap \mathsf{children}(\mathtt{G},\mathtt{C}') \neq \emptyset\right\} \cup \\ &\big\{\mathtt{C}' \mid \mathsf{desc}(\mathtt{G},\mathtt{C}') \cap \mathsf{desc}(\mathtt{G},\mathtt{C}) \neq \emptyset\ \& \\ &\quad \mathtt{C}' \notin \mathsf{desc}(\mathtt{G},\mathtt{C})\ \&\ \mathtt{C} \notin \mathsf{desc}(\mathtt{G},\mathtt{C}')\big\}\end{aligned}$$

Then, we find in $\mathsf{share}(\mathtt{G},\mathtt{C})$ all contexts which share a descendant context and are otherwise incomparable with `C` through the directly-owned relation (encoded through `desc`), and all

[1]Unnamed contexts are automatically added in the case of multiple maxima which share common descendants.

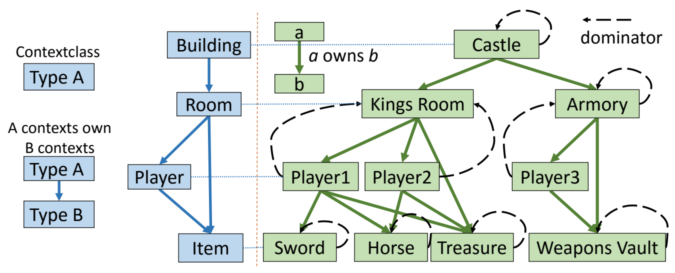


Figure 3: Game static and dynamic context structure.

Algorithm 1 AEON data structures

```
1:  Event:
2:      eid                              ▷ unique event id
3:      dom                              ▷ dominator context
4:      target                           ▷ context the event lands
5:      accessMode                       ▷ indicate readonly or not

6:  Context:
7:      cid                              ▷ unique id of the context
8:      toActivateQueue                  ▷ queue for incoming events
9:      toExecuteQueue                   ▷ queue for executing events
10:     activatedSet         ▷ set of events currently using the context
```

the contexts which might be an owner of C and moreover share a common child with C.

In order to calculate the context dominating all contexts that potentially share something with C, denoted dom(G, C) and dubbed C's "dominator", we can compute the *least upper bound* (lub) of the contexts share(G, C) ∪ {C} in the lattice G.

$$\mathsf{dom}(G, C) \;=\; \mathsf{lub}\big(G, \mathsf{share}(G, C) \cup \{C\}\big)$$

For example, consider Figure 2 which illustrates the ownership network G for the game example and indicates dominators for each context: dom(G, Player1) is Kings room and dom(G, Sword) is Sword.

Methods and events. Events represent asynchronous client requests to the AEON application, and therefore define its external API. To simplify the syntactic categories of AEON, and avoid code duplication, events are simply method *calls* decorated by the event keyword targeted at a context. The same convention applies to asynchronous method calls which are decorated with the keyword async.

The execution of events is distributed and can span multiple contexts, but from the programmers' perspective, the execution of events appears atomic. The execution of an event conceptually begins at a target context: the context providing the method being called. An event executing in a certain context C can issue method calls to any contexts that C owns, and in this way can modify the state of any context transitively reachable in the ownership DAG from C.

In addition to method calls, events are able to dispatch new events within themselves. An event that is dispatched within another event will receive the same treatment as any other client's event, and will execute after its creator event finishes its execution. This is in contrast to synchronous and asynchronous method calls whose execution is entirely contained within the current event execution.

As shown in Figure 2, there is an optional ro method modifier (ro is a shorthand for the more verbose readonly used throughout). This allows the declaration of methods that are readonly, which enables the execution of multiple readonly requests in a single context concurrently. A simple check guarantees that readonly methods can only use other readonly methods, and that they cannot modify the state of a context. In the next section, we will explain in more details how methods and events are executed.

Type-based enforcement of DAG ownership. As stated before, an important invariant to achieve a deadlock-free strictly serializable semantics for AEON is that the ownership network be acyclic (at least with respect to contexts that directly export events, i.e., the entry points for clients to access the application).

In particular, since the directly-owned relation is related to referential reachability in the context-graph, we require that the graph of contextclasses reachable for a context that exports events be acyclic. An example of a hierarchy is shown in the left hand side of Figure 3, where the hierarchy represents essentially which contextclasses are contained in a certain contextclass.

To enforce this property, we put in place a simple analysis that collects for each contextclass method declaration, an over-approximation of the types of contexts that it could access. Since our language is in Administrative Normal Form [14], this can be done by a single pass over the declarations of contextclasses. Whenever a contextclass C_0 declares an event that can use a contextclass C_1, we require that the contextclass C_0 appears always at a higher level in the ownership network than C_1 and we denote this constraint as $C_1 \leq C_0$. The analysis succeeds if the collected constraints are acyclic except for the obvious reflexive cases (i.e., $C \leq C$), and rejects the program otherwise. This exception, made for reflexivity of the relation, allows for the construction of inductive data structures like linked-lists, or trees, at the slight expense of runtime checks upon modifications of context ownership structure. We note that the context ownership structure is modified when the object graph is explicitly modified.

4. EXECUTION PROTOCOL

In this section, we describe our novel synchronization protocol employed by AEON that arbitrates between two concurrent events to ensure *strict serializability*: the execution of an application's events built atop AEON appears like a *sequential* execution of the application that respects the *temporal ordering* of events. In other words, any AEON execution is indistinguishable from a valid sequential execution of the application' events. To synchronize among events that execute in contexts that have shared descendants, AEON employs the dominator context as a sequencer. Intuitively, when an event is launched in a context C of an ownership network G, the dominator context of C (i.e. Dom(G, C)) is conceptually *locked*. An event locking a context has – conceptually – exclusive access to all the descendants of that context. Since we lock the dominator context, we have the guarantee that no other event that shares descendants with C starts its execution until the termination of the current event. These properties are ensured by AEON's implementation.

Protocol overview. Algorithm 2 provides high-level pseudo-

Algorithm 2 Event execution at context C

```
to execute Event $E$:                                  ▷ accept incoming event
    $G \leftarrow$ getOwnershipNetwork()              ▷ return context graph
    $dom \leftarrow G$.getDom($E.target$)            ▷ get context dominator
    send (ACT, $E$) to $dom$                           ▷ send E to its dom

upon receive (ACT, Event $E$) from Context $C'$:
    $toActivateQueue$.enqueue($E$)

task dispatchEvent:                                    ▷ dispatch next event
    while $\exists E \in toActivateQueue$ do
        $E \leftarrow toActivateQueue$.dequeue()
        $G \leftarrow$ getOwnershipNetwork()
        if $\big((\nexists E' \in activatedSet : E'.accessMode = \mathsf{EX})$ &
            $(E.accessMode = \mathsf{RO})\big)$ then
            $activatedSet \leftarrow activatedSet \cup \{E\}$    ▷ activate E
        else
            wait until $activatedSet = \emptyset$
            $activatedSet \leftarrow \{E\}$                     ▷ activate E
        send (EXEC, $E$) to $E.target$                 ▷ send E to execute

upon receive (EXEC, Event $E$) from Context $C'$:
    $toExecuteQueue$.enqueue($E$)

task scheduleNext:                            ▷ scheduling next executing event
    while $\exists E \in toExecuteQueue$ do
        $E \leftarrow toExecuteQueue$.dequeue()
        if ($E \notin activatedSet$) then
            activatePath($E$)            ▷ activate path from target to C
        execute($E$)              ▷ execute event after path is activated

procedure activatePath(Event $E$):
    $G \leftarrow$ getOwnershipNetwork()              ▷ return context graph
    $P \leftarrow$ findPath($G, E.target, C$)          ▷ find a path
    for all ($C' \in p$) do              ▷ activate contexts in the path
        send (ACT, $E$) to $C'$
        wait until $E$ is activated at $C'$
```

code of the AEON synchronization protocol and Algorithm 1 describes the data structures used in Algorithm 2. The execution of an event consists of method calls on the target context, or method calls on contexts that the target context owns. To execute, an event must take the *lock* on the target context. Each context has a set called *activatedSet*, which records events that currently lock that context.[2] When an event tries to obtain the lock over the dominator of its target context, it will be placed into the dominator's *toActivateQueue*. When the event tries to lock a context other than the dominator, it must lock all events in a path from the dominator to itself in a top-down fashion. Finally, when a context method is called, the call is placed in the *toExecuteQueue* of the context based on the same ordering determined by the dominator.

Task dispatchEvent dequeues an event from *toActivateQueue*, and waits until the event obtains the lock, that is, it is added to the *activatedSet*. Note that multiple read-only events can hold the lock to a context at the same time. Once an event takes the lock, it is added to the *toExecuteQueue* for execution. Task scheduleNext is responsible for dequeuing an event from *toExecuteQueue*, and execute it.

When a method finishes its execution in a context, control returns to the caller context, but it is not immediately removed from the context *activatedSet*; the removal happens only when the event has terminated in all contexts.

The execution model for method calls is by default *synchronous*, similar to Java RMI. However, in certain situations, for example when notifying every children of a certain context of a change, it is both unnecessary and inefficient to wait for the completion of a method call before issuing the following one, especially as these calls may be remote. The `async` method call decorator in AEON thus indicates that the execution of method call is *asynchronous*. This is the case of the calls to `updateTimeOfDay` for the `Room` contexts in the method of the same name in the declaration of `Building` of Listing 1.

Evidently, in the case of multiple asynchronous methods that update the state of common children contexts, this behavior can lead to non-deterministism. This is analogous to data races which are considered a programming error, and have no semantics. In AEON this is also considered an error, albeit having a well-defined coarse-grained interleaving semantics (at the level of context accesses). In future work we will consider ruling out programs prone to this kind of error at compile-time.

We remark AEON employs a mechanism similar to read-write locks exploiting the `readonly` annotations (cf. Figure 2). Unlike update events, which completely lock the target context, read-only events conceptually use a *read-lock* (Line 11), so multiple read-only events can execute in parallel in the same context as detailed in Algorithm 2.

Informally, it is straightforward to see why the AEON protocol is strictly serializable. Specifically, let $\mathcal{A}$ be any application built using AEON and π be any execution of $\mathcal{A}$. There exists a sequential execution π' of $\mathcal{A}$ equivalent to π such that for any two events E_1, E_2 invoked in π, $E_1 \rightarrow_\pi E_2$ implies $E_1 \rightarrow_{\pi'} E_2$, where $E_1 \rightarrow_\pi E_2$ denotes the *temporal ordering* between events E_1 and E_2 in an execution π. Indeed, let `G` be any ownership network of an application $\mathcal{A}$ and E_1, E_2 be any two application events participating in an execution of the AEON protocol. Since `G` is a semi-lattice (cf. § 3), there is a deterministic monotonic ordering for E_1 (and resp. E_2) for conceptually locking the contexts accessed that begins with dom(`G`,`C`) (and resp. dom(`G`,`C`$'$), where `C` (and resp. `C`$'$) is the context on which E_1 (and resp. E_2) lands initially. Finally, locks on the contexts accessed during an event are *released* in the reverse order on which they are locked, thus ensuring *starvation-freedom*.

Illustration of event synchronization in AEON. We now illustrate the execution of the AEON synchronization protocol with our game example. Consider the *single-owner* case where a context `C` is its own dominator, and an event is enqueued for execution at `C`. `Castle` and `Armory` in Figure 3 are examples of such single-owner contexts. In this scenario, to ensure that no two events modify context `C` or its descendants at the same time, events hold exclusive access of the context hierarchy starting at `C` during their execution, which is guaranteed by enqueuing all incoming events in the context's execution queue. Events execute once they reach the head of the queue.

However, if there is *context sharing* where the dominator of context `C` is a different context `C`', and the event is forwarded to `C`', it is sequenced at `C`', and starts its execution according to the sequence order. In Figure 3 all `Player` contexts are sharing contexts. The presence of sharing contexts introduces potential for deadlock. Consider for example the

[2] Multiple readonly events can lock the same context.

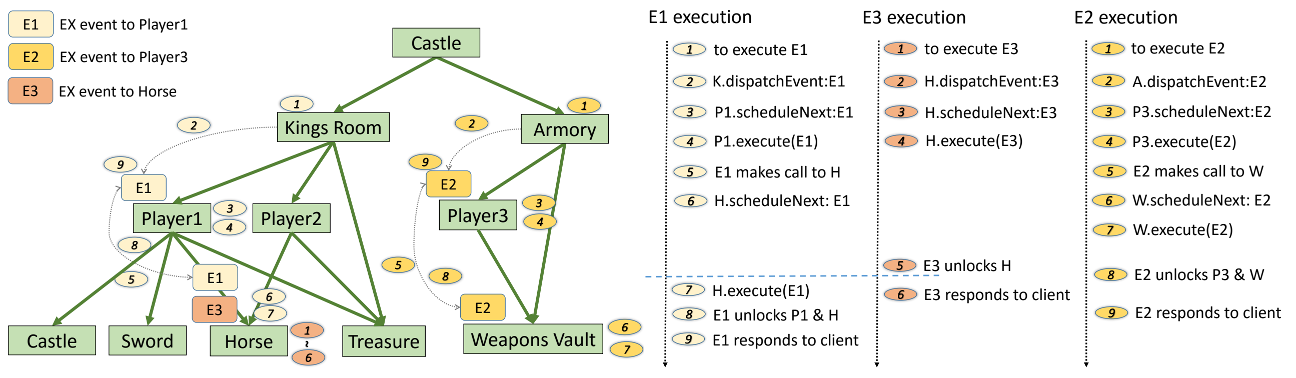


Figure 4: Event execution

network in Figure 3, and assume that `Player1` wants to steal the money from the shared `Treasure`, and then run away using the `Horse`. At the same time, `Player2` wants to use the `Horse` to collect some debts, and then deposit the money in the `Treasure`. The schema delineated above can lead to a deadlock, where none of the players is able to execute their events. To avoid deadlocks, when an event with different target and dominator contexts is dispatched, AEON's runtime delivers that event to the dominator context: the event is serialized at the dominator context before being sent to its target context for execution. Therefore, in Figure 3, events targeting `Player1` and `Player2` need to be serialized in `Kings Room`, whereas events on `Player3` are serialized in `Armory` (i.e., their dominator contexts). Observe that events targeting other contexts can safely execute in their target contexts.

We observe that for most cases, contexts have different dominators. For contexts that do not share sub-contexts with others, their dominators are themselves. Thus events to those contexts will be ordered independently. Generally, if two events are not ordered by the same dominator, they can execute in parallel.

Figure 4 shows a timeline of the execution of three events: E_1 targeting context `Player1` (abbreviated P1 in the timeline); E_2 targeting `Player3` (P3); and E_3 targeting `Horse` (H). The numbers in the timeline correspond to the numbered labels in the ownership graph in the left of the figure.

We can firstly observe that since the dominators of E_1 and E_2 are the `Kings Room` and the `Armory` respectively, and these contexts have no common descendants, they can execute completely independently and in parallel. The dominator context of event E_3 is the `Horse`, which is also its target. According to the rules outlined above, this event is immediately added to the *toActivateQueue* of `Horse` and subsequently activated. Importantly, event E_1 also requires to access `Horse` in the timeline. Therefore, when the execution of E_1 reaches the context `Horse`, the activatePath procedure will temporarily stall since in `Horse` E_3 is currently activated. Hence, E_1 has to wait for the completion of E_3 and the deactivation of E_3 in `Horse` before resuming its execution. In this way, the resulting serialization has the execution of E_3 before that of E_1, where the latter event sees the effect of the former one in the context `Horse`.

5. ELASTICITY

In this section, we explain AEON's elasticity manager called *e*Manager. The *e*Manager provides the following capabilities: (i) maintaining the global *context mapping* and *ownership network*, and (ii) managing context 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.

5.1 Context Mapping

Since contexts can dynamically migrate across hosts, and in order to deliver an event to the appropriate context, AEON first needs to find the host currently holding the corresponding context. To this end, every client and host caches the most recent context mapping that they have queried, and periodically refreshes their context mappings by querying the *e*Manager. In practice, and in order to have a highly scalable and available system, clients and other hosts do not directly query the *e*Manager. Instead, the *e*Manager stores the latest context mappings along with the ownership network in a (configurable) cloud storage. Therefore, to locate a context 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 *e*Manager.

5.2 Elasticity Policy

AEON gives the programmer the ability to define when and where contexts should be migrated. To this end, AEON employs an approach similar to Tuba [35, 34]. Every server periodically sends its resource utilization data (i.e., CPU, memory and IO) to the *e*Manager. AEON provides a simple API to define when the *e*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 *e*Manager triggers a migration. (ii) Server contention: under this policy, a programmer defines the total number of acceptable contexts per server. Hence, once a server reaches its maximum, the *e*Manager triggers a migration.

Once a migration is triggered, AEON computes a list of possible servers that can receive the contexts concerned. The default algorithm tries to move contexts from overloaded hosts to underloaded ones, but programmers can implement their own algorithms for choosing hosts and contexts. In addition, AEON allows programmers to define constraints on any attribute of the system similar to Tuba [35]. For instance, a constraint can disallow certain context migrations, or disallow a migration to a new host if total cost reaches some threshold.

Migration protocol. Once a migration is triggered, the *e*Manager will follow the following *atomic steps* to migrate a context C from host s_1 to a new host s_2.

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

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

III Once the *e*Manager receives the ack, and after δ seconds, it updates its context mapping by assigning C to s_2. Thus, from this point on, the *e*Manager returns s_2 as the location of context 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 $\mathsf{migrate}_c$ in C's execution queue. This event serves as a notification for context C that it must migrate. When $\mathsf{migrate}_c$ 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 *e*Manager that the migration is finished, and starts executing the enqueued events for context C.

Correctness under context migration. Observe that context 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 *e*Manager updates the context map. During this period, s_1 does not accept events targeting context C, and *e*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 context mappings. If s_1 later receives an event for C from a host with stale context map, s_1 will forward those events to s_2 directly and notify source host to update its context map. In § 6, we will evaluate the performance implication of halting the execution of events on a migrating context.

5.3 Fault tolerance

Similar to Orleans [8], AEON provides users with a special *snapshot* API that allows programmers to take consistent snapshots of a given context along with all its children. To this end, upon receiving a snapshot request for a context, the runtime of AEON dispatches a particular event called snapshot to that context. Consequently, this event takes consistent snapshots of that context and its children by getting contexts 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 a context. In case the overridden method returns null for a context, the runtime system will ignore that context during the checkpointing phase.

As we mentioned earlier, in practice the *e*Manager is implemented as a stateless service that is responsible for updating context mapping and the DAG structure that are stored in a cloud storage system. The *e*Manager also leverages the cloud storage system for persisting the steps of ongoing migrations. Therefore, if during the course of a migration, the *e*Manager crashes, a newly elected *e*Manager can read the state of an going migration, and tries to finish it. Details on how individual server and the *e*Manager failures are treated without violating the consistency can be found on the AEON webpage.

6. EVALUATION

We implemented AEON on top of Mace [22], a C++ language extension that provides a unified framework for network communication and event handling. The implementation of AEON consists in roughly 10,000 lines of core code and 110 new classes on top of Mace. In the remainder of this section, we first compare scalability and performance of AEON with the two most closely related frameworks: EventWave [9] and Orleans [8]. We then study AEON's elasticity capabilities, and conclude the section by evaluating AEON's migration protocol and its effect on the overall throughput of the system.

6.1 Scalability and performance

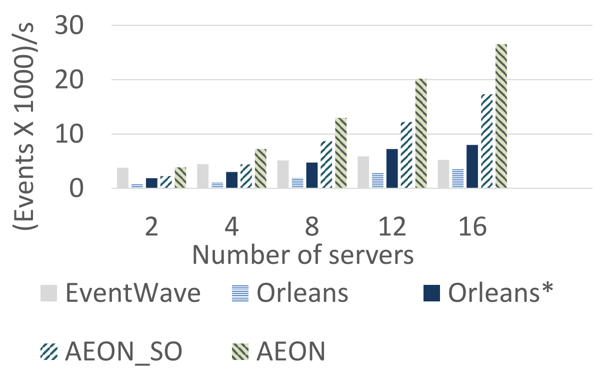
In order to compare scalability and performance of AEON with EventWave and Orleans, we focus on the following two conventional metrics: (i) scaling out: how a system scales out as we increase the number of servers; and (ii) performance: how throughput changes with respect to latency as we increase the number of clients. To evaluate the above metrics, we implemented the TPC-C benchmark [1] and game application in all three systems.

To better study the effect of multiple ownership, the above two applications were implemented with and without multiple ownership. Throughout this section, we refer to the implementation with multiple ownership as AEON, and refer to the one without multiple ownership as AEON_{so} (for **S**ingle **O**wnership). Therefore, the programming effort for implementing the above applications is identical for AEON_{so} and EventWave.

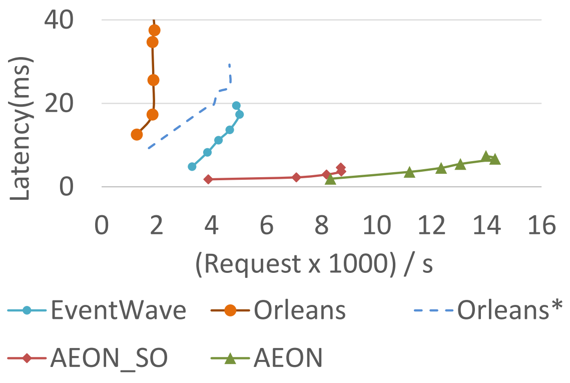
We run AEON, AEON_{so} and EventWave on m3.large Linux VMs on EC2. For Orleans and Orleans*, we used m3.large Windows 2012 VMs on EC2.

6.1.1 *Game application*

Both EventWave and Orleans were previously evaluated using a game application similar to the example of § 2. Therefore, we picked the very same game application described in EventWave [9]. Since EventWave does not support multiple ownership, the implementation does not allow `Players` to access `Items` directly. They could only access `Items` via `Room`. Since Orleans doesn't support transactional execution across multiple grains, we implemented two variants of game application in Orleans: (i) A version that ensures strict serializability. This version ensures `Players` access the shared `Items` atomically by means of locks. The `Players` simply lock the whole `Room` when they access their `Items`. This version is called *Orleans* in this section. (ii) Since one may argue that the above implementation is not the best possible algorithm for implementing the game in Orleans, we also implemented a non-strict serializability



(a) Game Scaling out



(b) Game Performance

Figure 5: Game application scalability and performance

variant of the game called *Orleans**, in which `Players` just access their shared `Items` directly, and without synchronizing with other `Players` that have the same `Items`. This may result in incorrect executions potentially breaking application invariants. We note that this implementation is only used as a best-case scenario for the performance of Orleans, and it should otherwise be considered erroneous.

Scale out. Figure 5a shows scalability of different systems for the game application. In this experiment, we make each server hold one `Room` with fixed number of `Items`. So if there are more `Players` in one `Room`, `Items` will be shared by more `Players`.

As shown in Figure 5a, EventWave reaches maximum throughput with 12 servers since it needs to order all events in the root node. Observe that AEON$_{so}$ (resp. AEON) outperforms EventWave by 3x (resp. 5x) when the number of servers reaches 16. Since both AEON$_{so}$ and EventWave ensures strict serializability, and have identical tree structures, the 3x performance gain is not related to multiple ownership. Instead, the fact that in AEON events are not ordered at the root context along with async method calls lead to the observed substantial performance boost.

Interestingly, both AEON and AEON$_{so}$ outperform Orleans* as well. This is because: 1) AEON is implemented in C++ and Orleans uses C#. Hence, we expect AEON's implementation to have less overhead. 2) with the help of the ownership DAG, the runtime of AEON can optimize contexts placement, which will put `Rooms`, `Players` and `Items` in the `Room` on the same server. Orleans does not have similar rules, which may result in more message passing among servers. 3) due to the single-threaded nature of Orleans' grains, shared `Items` have to process requests from `Players` one by one. Though requests could be executed in parallel in `Players`, throughput is limited by the fixed number of `Items` within one Room.

Because of the parallelism provided by multiple ownership, we observe that AEON's performance is 50% more than AEON$_{so}$ when the number of servers reaches 16. More precisely, since AEON$_{so}$ does not have multiple ownership, in order to access `Items` belonging to a given `Players`, `Room` context needs to be locked. However, multiple ownership allows both `Players` and `Room` contexts to access `Items` thus leading to parallel execution of more events within one `Room`.

Performance evaluation. Figure 5b plots throughput and latency of the game application when the number of servers is fixed to 8. Similar to Figure 5a, AEON outperforms all other systems. As we explained above, optimized AEON exploits allows for more parallelism and reduces the overhead in communication.

6.1.2 TPC-C benchmark

The TPC-C benchmark is an on-line transaction processing benchmark. TPC-C is a good candidate for comparing AEON with its rivals since it has multiple transaction types with different execution structures. Observe that transactions in TPC-C are similar to events in AEON and EventWave. All of our TPC-C implementations are made fault tolerant through checkpointing. We note that we used TPC-C solely to stress-test AEON, and evaluate its performance under high contention. In reality, specifically engineered elastic distributed databases may be a better fit for serving TPC-C style applications.

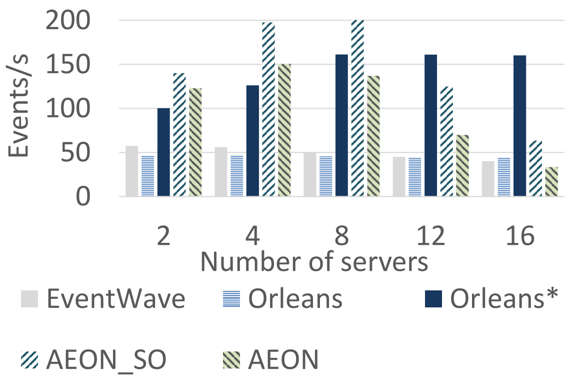
The TPC-C benchmark implementation in AEON uses the following context declarations:

```
contextclass WareHouse {set<Stock> s; set<District> d;}
contextclass Stock { … }
contextclass District {set<Customer> c; set<Order> o;}
contextclass Customer {History h; set<Order> os;}
contextclass Order {set<NewOrder> n; set<OrderLine> l;}
contextclass NewOrder { … }
contextclass OrderLine { … }
```

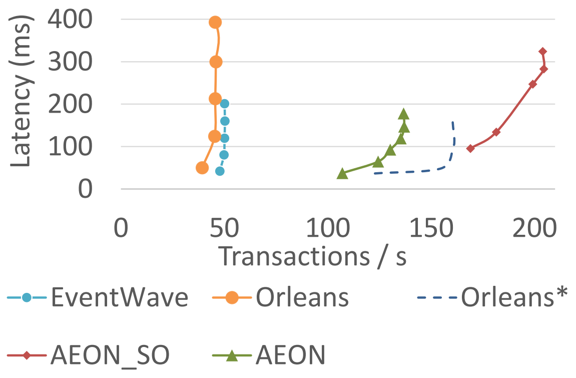
Since the number of items is fixed (i.e., $100K$) in the TPC-C benchmark, and does not need elasticity, warehouse and items form a single context.

Observe that an `Order` context has two owners: `District` and `Customer`. In our AEON$_{so}$ and EventWave implementations, and since they should follow a single ownership structure, the `District` context does not own the `Order` context. In other words, the `Order` context is solely owned by the `Customer` context.

A typical approach for evaluating the scalability of a system using TPC-C is to partition TPC-C by warehouse, and put each warehouse on a single server [10, 37]. But, as pointed out by Mu et al. [26], this approach does not stress the scalability and performance of distributed transactions (i.e., events in our programming model) because less than %15



(a) TPC-C Scaling out



(b) TPC-C Performance

Figure 6: TPC-C scalability and performance

of transactions are distributed. Therefore, we also partition TPC-C by district similar to Rococo [26].

Similar to the game application, we also implemented two variants of TPC-C in Orleans: (i) A version that ensures strict serializability, which we shall name Orleans throughout this section. This version is implemented by exploiting the fact that stateful Orleans grains are single-threaded, and we orchestrate grains in a tree-like structure à la EventWave. (ii) We also implemented a non-strict serializability variant called Orleans*, in which the strict serializability is not guaranteed to be maintained. We note that this implementation is only used as a best-case scenario for the performance of Orleans, and it should otherwise be considered erroneous since it fails to ensure all the invariants of the TPC-C benchmark.

Scale out. Figure 6a plots scalability of different systems for the TPC-C benchmark. In this experiment, we placed one `District` (along with its corresponding `Customers`, `Orders`, etc.) in each server. While neither EventWave nor Orleans can scale as the number of servers increases, we observe that AEON scales up to 4 servers and AEON$_{so}$ scales up to 8 servers. At this point, the `Warehouse` context becomes saturated, thus AEON and AEON$_{so}$ cannot scale beyond 4 and 8 servers.

More specifically, AEON and AEON$_{\mathrm{so}}$ are able to outperform EventWave and Orleans due to (i) its use of the ownership network to order events, and (ii) `async` method calls inside events. As an event (i.e., a TPC-C transaction) finishes its execution in a parent context, it can continue its execution in a child context by using `async` method calls to the child context. For instance, once a payment transaction finishes its execution in a `Warehouse` context, it calls a method in a `District` context asynchronously, and releases the `Warehouse` context. This allows another event to enter the `Warehouse` for execution.

Figure 6a also shows that both Orleans* and AEON$_{\mathrm{so}}$ perform better than AEON when the number of servers reaches 16. This is because in TPC-C, multiple ownership does not help to increase the parallelism. Each `District` context owns several hundreds of `Customer` contexts and each `Customer` context owns several `Order` contexts. With multiple ownership, all these `Order` contexts are shared by both `District` context and `Customer` contexts. Consequently, method calls from `Customer` contexts to `Order` contexts have to be synchronized by the `District` context, which is the dominator of `Customer` contexts. This leads to the `District` context becoming saturated fast. But, in the single ownership case, the dominators for `Customer` contexts are themselves. Therefore, the `District` context does not become the bottleneck.

Performance evaluation. Figure 6b shows TPC-C performance boundaries with 8 servers. As expected, the throughput of EventWave and Orleans reach maximum with few clients (i.e., 4-8 clients) and then their latencies skyrocket immediately. This is due to both implementations failing to handle high contention at the `Warehouse` context properly. As shown in both Figure 6a and Figure 6b, Orleans* outperforms AEON with 8 servers since AEON has to pay extra overhead for strict serializability: events will be synchronized by `District` context.

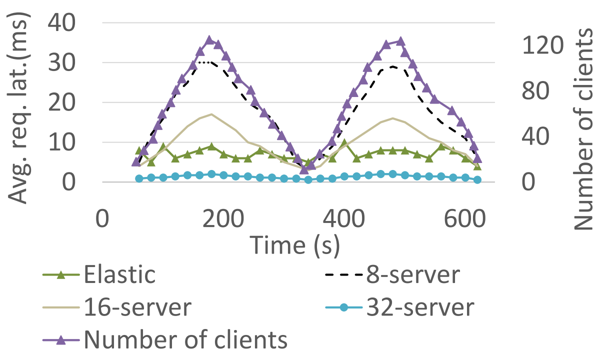
6.2 Elasticity

As it was explained in § 5, AEON has several built-in elasticity policies. In this section, we solely report our results on evaluating elasticity capabilities of AEON using the Service Level Agreement (SLA) metric as the elasticity policy of the game application.

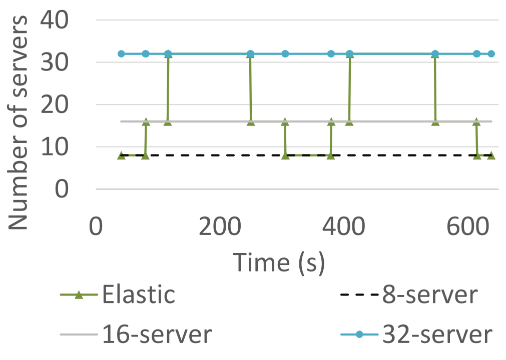
For this experiment, we set the SLA for clients requests to 10ms. Therefore, AEON automatically scales out if it takes more than 10ms to handle a client request. We placed our clients on 8 m1.large EC2 instances. Similar to Tuba [35], we varied the number of clients on each client machine from 1 to 16 according to the normal distribution. Therefore, at its peak time, there were 128 active clients in the system. The game application was deployed on a cluster of m1.small EC2 instances. To better understand the elasticity capabilities of AEON, we also run the game application with fixed numbers of servers (i.e., 8, 16 and 32 servers).

Figure 7a shows the average request latency that clients observed, and Figure 7b plots the variation of the number of servers during the experiments. During peak times, both 8-server and 16-server setups were unable to maintain the latency below 10ms. However, elastic and 32-server setups successfully met their SLAs. Due to migration cost, and fewer servers, clients in the elastic setup observed a slightly higher request latency.

Table 1 shows the percentage of client requests violating



(a) Elastic v.s static



(b) Server number

Figure 7: Elastic game application

Setup	% of requests > 10ms	Avg. servers
8-server	72.6%	8
16-server	44.2%	16
22-server	20.0%	22
32-server	0.0%	32
Elastic	0.0%	21.4

Table 1: Performance and cost

the SLA and the average number of servers used in each setup. While both 32-server and elastic setups managed to meet the SLA, 32-server setup did so with 47% more resources. Lastly, observe that a (non-elastic) 22-server setup was unable to satisfy the SLA while the elastic setup fulfilled the SLA with 21.4 servers (on average).

6.3 Migration

In this section, we first study the effect of migration on the overall throughput of the system. We then evaluate the throughput of *e*Manager when performing migration.

Overall throughput. We now show the effect of migration for different cases in our game application. In a first experiment, which we omit for lack of space, we migrated one context with different sizes up to 100MB. Clearly, as a context's size increases, the time it takes to transfer from one server to another increases, but the overall throughput remains stable.

In the second experiment, we migrated different numbers

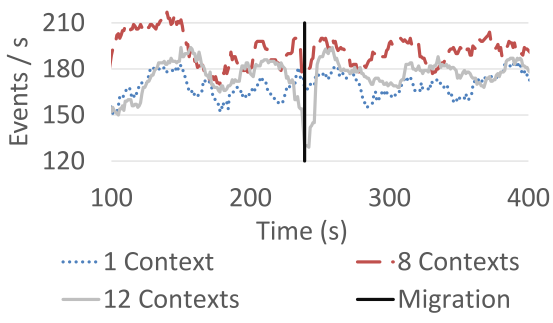


Figure 8: Migrating different number of contexts.

of contexts. Our experiments were deployed on EC2 with 20 m1.small instances. We create 20 `Room` contexts, one for each host. We also fixed the size of each context to 1MB, and migrated contexts in order to determine the accumulated effect of multiple migrations at the same time – expected for high workloads.

Figure 8 shows the overall throughput variation of the system when migrating different numbers of `Room` contexts. The mild degradation observed, especially for the case of 12 simultaneous migrations is due to the fact that when a context is being migrated, requests to it are delayed for the duration of the migration. In this case more than 50 percent of the contexts are being moved, which should obviously impact the performance of the system for a short period of time as shown in the figure.

***e*Manager throughput.** Finally, we evaluated the maximum throughput of the migration algorithm introduced in § 5 using a micro benchmark. To this end, the *e*Manager moves contexts from one AWS instance to another. Figure 9 shows the *e*Manager throughput with different context sizes. With m1.large instances, the *e*Manager is able to move around 90 small contexts (i.e., 1KB in size) or 40 large contexts (i.e., 1MB in size) every second. These numbers are dropped to 60/25 with m1.medium instances, and 40/20 with m1.small instances.

We expect that the number of contexts to be much less than the number of objects for an application. In other words, one context plays the role of a container for several objects as long as these objects do not require an independent elasticity policy. For example, consider the game application. Within a room, there can be several objects like lights and chairs. These objects can all be included in the `Room` context. However, in case light object has some non-trivial CPU or memory usage, it should be treated as a separate context.

7. RELATED WORK

Distributed programming models. The actor model [29, 8, 5] is a popular paradigm that can be used to develop concurrent applications. Actors encapsulate state and execute code that can be distributed across multiple servers. Actors communicate with each other via message passing, and there is at most one thread executing in an actor at all times. This eliminates the complexities involved in guaranteeing data race and deadlock freedom. In that sense, actors are similar to contexts in our model. However, it is important to note that atomicity in actor systems is only given

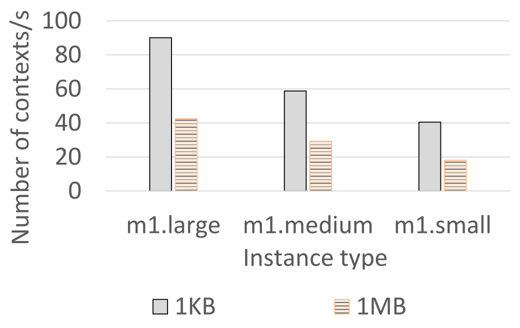


Figure 9: Max migration throughput on eManager

with respect to single actors, whilst an event in AEON can atomically modify several contexts.

Orleans [8] and EventWave [9], described in § 6 provide concepts similar to AEON's contexts and events. The originality of AEON however resides in the *ownership network*, which allows us to guarantee strict serializability, unlike any of these two works, and deadlock freedom unlike Orleans, while still allowing sharing of state, and providing opportunities for automatic parallelization and scale adaptation. EventWave also induces single ownership and limits scalability by invariably synchronizing at a single root node.

Distributed transactional memory (DTM) [33] is a programming paradigm based on Transactional memory (TM) [36] that allows the programmer to build strictly serializable distributed applications with sequential semantics in mind, just as in AEON. However, to the best of our knowledge, there is no efficient DTM implementation that ensures strict serializability and provides implicit elasticity.

Transactors [13] have been proposed as a means to build distributed applications that provide strict serializability for events spanning multiple actors. However, transactors also do not provide support for building applications whose individual actors are distributed across the cloud. Moreover, there is no support for migrating actors without affecting consistency which is an important contribution from this work. We remark though that AEON may be thought of as an extension of transactors to the distributed cloud with support for automatic elasticity.

MapReduce [12] is a functional programming paradigm for the cloud that allows parallelizing computation via two sequential phases: *map* and *reduce*, to build applications involving huge data sets. However, writing a generic stateful application whose operations are non-commutative requires extensive synchronization among threads of computation, which is nontrivial to get right in the MapReduce paradigm [32], let alone supporting automatic elasticity.

AEON also shares similarities with models tailored to multi-core execution environments like Bamboo [39]. Bamboo provides a data-oriented approach to concurrency, where the programmer implements tasks, and the runtime system exploits dynamic and static information to parallelize data-independent tasks. Bamboo uses locks to implement a transactional mechanism for data-dependent tasks. Unlike AEON, Bamboo optimizes concurrency for multiple cores; distribution, migration, and scale adaptation are not considered.

In SCOOP [28], objects are considered individual units of computation and data. *Separate* calls – marked by the programmer – can be executed asynchronously from the main thread of execution. This is similar to the `async` calls of AEON. Similarly, separate calls can only be issued on arguments of a method, which is SCOOP 's way of controlling what AEON achieves through multiple ownership and events. SCOOP is not concerned with distribution or scale adaptation addressed by AEON.

Distributed programming languages. Emerald [20] is an OO distributed programming language, providing locality functionalities to allow programmers to relocate objects across the available servers. Unlike AEON, Emerald does not guarantee atomicity, and synchronization is left to the programmer. Moreover, Emerald was not designed for the cloud, where the existing resources might be unknown or dynamically allocated. Therefore, Emerald provides no elasticity. Identical arguments apply to programming languages like Erlang and Akka that contrast them from AEON and render them insufficient for building complex distributed applications with minimal programming effort.

Transactional key-value stores. Elastic databases (e.g., ElasTras [11], Megastore [4]) are similar to AEON: they partition and distribute data among a set of servers and provide consistency in the face of concurrent accesses. Unlike AEON, these do not provide a self-contained programming environment for writing generic elastic cloud applications.

Pilot job frameworks. A pilot job framework offers dynamic computational resources to a set of tasks [6, 15, 24, 31]. Applications running on such a framework can be split into a set of isolated tasks organized either as a "bag of tasks" [6, 15, 24] or as a DAG workflow [31]. In the former case, tasks can execute in any order, while in the latter case, they should execute in a particular order defined by a DAG. These tasks are similar to the events of AEON, but unlike AEON where events can communicate with each other via contexts, tasks cannot communicate with each other.

Computation offloading. Offloading improves application performance by partitioning it among servers either at compilation or runtime [17, 23, 25, 30, 38]. Clearly, partitioning at compilation fails to provide elasticity. Dynamic partitioning, on the other hand, either targets single-threaded applications, or requires an explicit addition of parallelism in contrast to AEON.

8. CONCLUDING REMARKS

We have presented the design and implementation of the AEON language. AEON provides a sequential programming environment for the cloud based on the standard paradigm of object-orientation. We provide a description of the semantics of AEON, and show that this semantics exploits parallelism while providing strict serializability as well as deadlock and starvation 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. In future work we wish to lift some of the restrictions imposed on the usage of context references in classes and define a fine-grained elasticity policy language to allow the programmer control over the locality of contexts and usage of resources.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions that helped improving this work.

9. REFERENCES

[1] TPC-C. http://www.tpc.org/tpcc/default.asp.

[2] Who is using Orleans. http://dotnet.github.io/orleans/Who-Is-Using-Orleans.

[3] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *Euro. Conf. on Object-Oriented Pging. (ECOOP)*, pages 32–59, 1997.

[4] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-m. L, Y. Li, A. Lloyd, and V. Yushprakh. Megastore : Providing scalable , highly available storage for interactive services. In *Biennial Conf. on Innovative DataSystems Research (CIDR)*, pages 223–234, 2011.

[5] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed Virtual Actors for Programmability and Scalability. Technical Report MSR-TR-2014-41, Microsoft, March 2014.

[6] B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters. In *Annual Linux Showcase & Conference (ALS)*, 2000.

[7] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Conf. on Object-Oriented Prog. Sys., Lang. and Applications (OOPSLA)*, pages 211–230, 2002.

[8] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *Symp. on Cloud Computing (SoCC)*, pages 16:1–16:14, 2011.

[9] W. Chuang, B. Sang, S. Yoo, R. Gu, M. Kulkarni, and C. E. Killian. EventWave: Programming Model and Runtime Support for Tightly-coupled Elastic Cloud Applications. In *Symp. on Cloud Computing (SoCC)*, pages 21:1–21:16, 2013.

[10] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *Usenix Annual Tech. Conf. (ATC)*, pages 223–235, June 2012.

[11] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self managing transactional database for the cloud. *Trans. on Database Sys.*, 38:5:1–5:45, 2013.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[13] J. Field and C. A. Varela. Transactors: A programming model for maintaining globally consistent distributed state in unreliable environments. In *Symp. on Principles of Prog. Lang. (POPL)*, pages 195–208, 2005.

[14] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 237–247, 1993.

[15] J. Frey, T. Tannenbaum, M. Livny, I. T. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, 2002.

[16] D. Gordon and J. Noble. Dynamic ownership in a dynamic language. In *Symp. on Dynamic Languages (DLS)*, pages 41–52, 2007.

[17] X. Gu, A. Messer, I. Greenberg, D. S. Milojicic, and K. Nahrstedt. Adaptive Offloading for Pervasive Computing. *IEEE Pervasive Computing*, 3(3):66–73, 2004.

[18] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory,Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

[19] M. Herlihy and N. Shavit. On the nature of progress. In *Int. Conf. on Principles of Dist. Sys. (OPODIS)*, pages 313–328, 2011.

[20] E. Jul, H. M. Levy, N. C. Hutchinson, and A. P. Black. Fine-grained mobility in the emerald system (extended abstract). In *Symp. on Op. Sys. Principles (SOSP)*, pages 105–106, 1987.

[21] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the jvm platform. In *Int. Conf. on Principles and Practice of Programming in Java (PPPJ)*, pages 11–20, Aug. 2009.

[22] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language support for building distributed systems. In *Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 179–188, 2007.

[23] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Sys. (CASE)*, pages 238–246, 2001.

[24] A. Luckow, L. Lacinski, and S. Jha. SAGA bigjob: An extensible and interoperable pilot-job abstraction for distributed applications and systems. In *Int. Conf. on Cluster, Cloud and Grid Computing (CCGrid)*, pages 135–144, 2010.

[25] P. McGachey, A. L. Hosking, and J. E. B. Moss. Class Transformations for Transparent Distribution of Java Applications. *Journal of Object Technology*, 10:9: 1–35, 2011.

[26] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 479–494, 2014.

[27] A. Newell, G. Kliot, I. Menache, A. Gopalan, S. Akiyama, and M. Silberstein. Optimizing distributed actor systems for dynamic interactive services. In *Euro. Conf. on Comp. Sys. (EuroSys)*, pages 38:1–38:15, 2016.

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

[29] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *Annual Int. Symp. on Computer Architecture (ISCA)*, pages 224–235, 1993.

[30] S. Ou, K. Yang, and A. Liotta. An Adaptive Multi-Constraint Partitioning Algorithm for Offloading in Pervasive Systems. In *Int. Conf. on Pervasive computing and communications (PerComm)*, pages 116–125, 2006.

[31] I. Raicu, Y. Zhao, C. Dumitrescu, I. T. Foster, and M. Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In *Conf. on Supercomputing (SC)*, pages 43:1–43:12, 2007.

[32] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Int. Symp. on High Performance Computer Architecture (HPCA)*, pages 13–24, 2007.

[33] P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo. Cloud-tm: Harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev.*, 44(2):1–6, Apr. 2010.

[34] M. Saeida Ardekani. *Ensuring Consistency in Partially Replicated Data Stores*. Ph.d., UPMC, Paris, France, Sept. 2014.

[35] M. Saeida Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 367–381, Oct. 2014.

[36] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[37] A. Thomson, T. Diamond, P. Shao, and D. J. Abadi. Calvin : Fast distributed transactions for partitioned database systems. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 1–12, 2012.

[38] L. Wang and M. Franz. Automatic Partitioning of Object-Oriented Programs for Resource-Constrained Mobile Devices with Multiple Distribution Objectives. In *Int. Conf. on Parallel and Dist. Sys. (ICPADS)*, pages 369–376, 2008.

[39] J. Zhou and B. Demsky. Bamboo: a data-centric, object-oriented approach to many-core software. In *Conf. on Prog. Lang. Design and Implementation (PLDI)*, pages 388–399, 2010.