Programming Scalable Cloud Services with XYZ

Abstract
Designing distributed applications that efficiently utilize the infrastructure provided by modern cloud computing platforms is hard. Ideally, a cloud programming model along with its corresponding runtime environment must allow applications to scale to different workloads, without simply giving up on consistency or other ends. In this paper, we propose XYZ, a distributed programming model and its implementation that enables the programmer to build truly scalable cloud applications while reasoning in a mainly sequential manner.

XYZ augments object-oriented programming with a notion of contexts that represent units of data encapsulation. Contexts support synchronous and asynchronous invocations, as well as events which can involve multiple invocations that are executed atomically.

We present a formal operational semantics of the XYZ language and prove it to be linearizable and deadlock-free. We evaluate its performance notably in terms of scalability on three benchmarks – a Piazza-like system, an elastic version of memcached and the TPC-C benchmark. As our results demonstrate, applications built using XYZ significantly outperform solutions built with current cloud programming models.

1. Introduction
A fundamental characteristic of Internet-facing applications such as gaming or messaging is that the workloads to be handled cannot be known ahead of time. Moreover, such workloads may vary over time, typically as different numbers of clients connect to a corresponding service. The system developer tasked with conceiving such an application may thus decide to leverage a cloud infrastructure to avoid building own infrastructure which is tailored to an estimated worst-case workload and likely overprovisioned in the general case. Indeed, cloud providers such as Amazon attract customers by the prospect of only having to pay for the amount of resources needed, and when they are needed.

However, devising applications which can scale out to accommodate increasing workloads (and scaling back in) is non-trivial. As pointed out in a recent article targeting practitioners [29], (i) scalability, (ii) re-thinking software architecture, and (iii) designing for a dynamic infrastructure are among the main challenges for programmers when moving to the cloud. The authors state that (ii) is also largely a consequence of the need to scale, and (iii) inversely affects software architecture as one has to ensure that consistency is not violated when applications scale out. In summary it appears that the architectures of cloud applications are strongly subjected to two constraints:

Scalability: To be able to efficiently handle different workloads (e.g., different numbers of client requests), cloud applications need to be able to operate at different scales. This is the prerequisite for the ability to adapt an application’s scale at runtime.

Consistency: No matter at which scale an application is operating, data constituting the application’s state has to be kept consistent in the face of concurrent client requests and sharing among potentially distributed application components.

Unfortunately it is well-known that these requirements may conflict in any concurrent system. In a distributed environment such as the cloud, the problem is exacerbated as most applications (their data) at some point will not fit onto single hosts anymore, and remote accesses across hosts add complexity and overhead to traditional synchronization protocols (e.g., consensus [10]).

Existing application development frameworks for dynamic scaling (“elasticity”) are ill-suited for many stateful cloud-based applications in that they (a) are tailored to very specific applications and thus promote extremely limited programming models, or (b) constitute or regroup only individual services or building blocks. As an example of (a), MapReduce [13] or Dryad [35] achieve scalability for big data analytics but would be non-trivial to use as basis for other applications as they do not mediate between modifications of analyzed data. As an example of (b), Amazon’s ElastiCache[1] provides an elastic implementation of memcached, but not every distributed cloud application can make efficient use of a cache. Frameworks regrouping such services (e.g., AppScale[2]) do not provide support in combining them.

This paper presents a novel programming model called XYZ that relieves the programmer from recon-

Figure 1: Scalable Internet application.

ciling scalability and consistency in a distributed cloud environment. As customary for Internet-based applications, XYZ applications follow a client/server model. Server side logic running in the cloud is implemented following a programming model based on the familiar object-oriented (OO) programming paradigm and responds to events issued by clients. Figure 1 illustrates two possible states of an XYZ application — differing in scale — showing the interactions between clients and the (distributed) server. XYZ leverages a parallel distributed execution model tailored to achieve scalability without compromising on consistency. More precisely, the OO model is augmented by a notion of context (circles in Figure 1) to capture units of application partitioning. By enforcing strict data encapsulation at the level of contexts (no sharing of references) in a way similar to Actors [18], contexts located on one host can easily be relocated to other hosts in case the original host becomes overloaded. To allow programmers to reason at a level of consistency similar to sequential programming, XYZ introduces a notion of events. In a simplified manner these can be viewed as an abstraction for client requests. They can involve multiple contexts yet are executed in an atomic manner. That is, events are executed in a provably linearizable [17] and deadlock-free manner. XYZ achieves a sweet spot between consistency and scalability by leveraging a simple and relaxed notion of ownership [11] amounting to organizing contexts along a distributed acyclic graph (DAG), and by employing specially designed novel synchronization protocols for such DAGs.

Paper contributions. Concretely, this paper makes the following principal contributions:

1. We introduce the programming model of XYZ, a high-level language for cloud applications which extends the popular OO programming model and achieves scalability in that cloud-based server applications developed in XYZ can support large and varying numbers of concurrent clients.

2. We provide a formal semantics for the core of XYZ. We use this semantics to formally prove that XYZ programs are free of deadlocks and are endowed with linearizability semantics; these are two of the core tenants to make XYZ appealing for programmers. Moreover, this semantics serves both as a formal specification of the XYZ language, and as a formal framework for the analysis and verification of XYZ programs.

3. We present an implementation of XYZ as an extension to C++, and compare it to existing frameworks for cloud programming. We provide an exhaustive evaluation of XYZ, empirically demonstrating the benefits advocated by our design.

Roadmap. The remainder of the paper is structured as follows. Section 2 illustrates the XYZ programming model via a Piazza-like application while Section 3 provides more details on the programming abstractions of XYZ. Section 4 concerns the formal semantics of the XYZ language and proofs of linearizability and deadlock-freedom. In Section 5, we discuss implementation issues. Section 6 provides detailed experimental results that demonstrate the scalability of applications built using XYZ. Section 7 covers the related work and Section 8 concludes the paper.

Optional Appendix A presents the detailed proofs for deadlock freedom and linearizability in XYZ. Optional Appendix B shows additional evaluation results, comparing an implementation of memcache in XYZ to Amazon’s ElastiCache.

2. Motivating Example and Overview

Let us introduce the main components of XYZ through a simple example. We consider an application a-la Piazza [31], an online service that allows instructors and students to organize a class by allowing students to register in classes, interact with each other and the instructor through a forum-like environment, etc.

Scenario. Consider that we are interested in providing the following functionality. We have a course where students have multiple assignments that they have to turn in on predefined deadlines. The students are allowed a total of \( n \) days of delays throughout the semester. To that end, they are initially given \( n \) tokens, each of which they can trade for one day of delay. Moreover, to cover for eventualities such as a cancelled lab or class, an additional token can be given to each student concerned. The addition of tokens can only be triggered by a teaching assistant (TA), or an instructor of the class. Moreover, we must guarantee that each of the TAs and instructors are immediately notified of the addition of tokens (otherwise they might repeat the operation).

Since students can decide asynchronously at any point to use a token, the programmer must guarantee that upon a token-increment event, the tokens of each student are updated atomically (to avoid the risk of a student obtaining extra tokens through a race con-
contexts are roughly analogous to distributed classes in frameworks like RMI [30] or CORBA [15]. XYZ context instances are automatically migrated by the XYZ runtime system. More differences affecting the use of context instances to facilitate such migration will be explained shortly in Section 3.

Events and calls. Like other environments to write Internet-facing applications (e.g., [7, 9]), XYZ follows an event-driven model, where clients of the system interact with the server application by issuing events to the latter (cf. Figure 1). Contexts (lines 7 and 15) declare which of their methods can be directly called by clients as events. Importantly, the execution of an individual event is guaranteed to be atomic – more precisely, linearizable with respect to other events. The declaration of which methods can be used as an event is achieved through the events keyword in the declaration of contexts.

Notice for instance at line 15 that while use_token is declared as an event, add_tokens is not, since the latter can only be called as part of an event executing in the course context (initiated by a TA or an instructor). As it has been foreshadowed, context methods can call other context methods. The argument passing convention is that context instances are passed by reference, while objects are passed by value, i.e., deep copying.

In particular, other than event calls, which are generally reserved for “external” calls from clients to the main application, context method calls can be either synchronous – the implicit assumption – or asynchronous, indicated with the async keyword at the call site. Evidently, a synchronous call blocks the execution of the event in the current context until a result is obtained from the target context upon return. Line 11 shows an example of a synchronous context call, which needs to wait for a boolean response indicating if the notification was successfully processed or not (code elided for brevity). Conversely asynchronous calls do not wait for a result, and allow execution to proceed immediately, hence allowing for parallel execution of methods called. An example of asynchronous call is given at line 8. In this case, the execution of the add_tokens method in one student needs not wait for all prior calls to add_tokens in other students. We require that asynchronous calls be only made on methods without return values, but this restriction could be easily lifted by introducing futures [34], which we do not do at the moment.

3. XYZ Programming Model

In this section we describe the principal programming abstractions offered by XYZ. Let us start by presenting a simplified abstract syntax of XYZ in Figure 2. Notice first that XYZ provides class declarations, as well as methods and fields like most mainstream OO pro-

```
context Course events <snow_day, ...> {
  vector<Person> students; // students
  vector<Person> instructors; // instructor
  ...
  void snow_day() {
    // update student tokens in parallel
    for(int i=0 upto students.size())
      async students[i]->add_tokens(1);
    // notify all instructors atomically
    for(int i=0 upto instructors.size())
      instructors[i]->notify("Snow-Day");
  ...
}
context Person events <use_token, ...> {
  Role role; // staff or student
  ...
  class Role {... }
  context University {... {
    vector<Department> departments; ...
  }
  context Department {... {
    vector<Course> Courses; ...
  }
  ... ...
```

Listing 1: XYZ code snippet

dition). Moreover, the token increment event must be made visible immediately – or rather atomically – to all TAs and instructors in the course.

Listing 1 outlines a possible implementation of this scenario in XYZ, which is implemented as an extension of C++. We remark that keywords appearing in red color are new to XYZ, and we use the syntactic form upto to iterate through a numerical range. We will elaborate on the example by providing intuitions about the different constructs, which will be further explained in Section 3.

Contexts. The declarations of contexts at lines 1 and 15 can be thought of as class-like foundries for distributed objects with the following characteristics: (i) They can contain data in the form of fields. As customary such fields can be of class types, as is the case of line 16 declaring a field role, an instance of the class Role declared at line 28. In this case, the class instance role is guaranteed to reside in the same server as the context instance of Person that refers to it. (ii) Alternatively, they can contain references to context instances. For instance at line 2 the field students contains a vector holding references to Person context instances, which possibly reside on different hosts.
Variables $x,y \in Var$

Expressions $e \in Exp$

Method Names $m \in M$

Field Names $f \in F$

Class Names $cls \in Cls$

Context Names $ctxn \inCtx$

Program Def. $p \in P$ $::=$ $\overline{cls} \overline{sd} main\{\ldots\}$\textcolor{red}{\{s\}}

Context Def. $ctxd \inCtxD$ $::=$ $context ctxn \{ fd md \}$

Class Def. $clsd \inCtxD$ $::=$ $class cls \{ \overline{fd} \overline{md} \}$

Type $\tau \in T$ $::=$ $ctxn \mid cls \mid int \mid float \mid \tau \mid \ldots$

Field Def. $fd \in FD$ $::=$ $\tau f$

Method Def. $md \in MD$ $::=$ $\overline{ro} \tau m(\overline{\tau x}) \{ s \}$

Decorated Call $dc \in DC\textcolor{red}{\text{all}}$ $::=$ $event x.g(\overline{x}) \mid async x.g(\overline{x})$

Statements $s \in S$ $::=$ $dc \mid \ldots$

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

Classes and contexts. An XYZ program comprises a series of context declarations, a series of class declarations, and a main function which starts the execution of the XYZ program. A context (instance) 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 context instances. At a high level, a context instance can be considered as a container object or composite object that can be relocated between hosts.

Context instances encapsulate local state (in the form of fields) and functionality (in the form of exported methods or events). In particular, XYZ context instances hide internal data representations, which can only be read or affected through their methods.

Another aspect that distinguishes context declarations from the standard class declarations is that types appearing in context field and method declarations can also contain type expressions which appear underlined $\tau$ in Figure 2. Inspecting the rule for types we can see that context names can thus be used as types only in context 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 context instances), 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 XYZ.

Context ownership network. As we mentioned before, event execution in XYZ is linearizable (see subsection 4.3 for a formal proof), which is achieved by the runtime system through a locking discipline. Before detailing events in more detail, we focus on the orchestration of context instances. In a nutshell, XYZ contexts are guarded by an ownership mechanism loosely inspired by the ones proposed in [2, 6]. Importantly, the locking discipline must avoid the possibility of deadlocks since events might operate on several context instances.

The principal abstraction that allows XYZ to guaran-
tee deadlock freedom is a notion of ownership, estab-
lishing a partial order between context instances (when considered transitively). In general we say that a context instance $ctx_0$ is “directly-owned” by another context instance $ctx_1$ if any of the fields of $ctx_1$ contains a reference to $ctx_0$ (this is trivially extended to collections). The ownership relation described above takes into account the transitive closure of the directly-owned relation. An example of this relation is given in the right hand side of Figure 3, where a context instance “Yale” of type University (cf. line 29 of Listing 1) contains references to context instances “CS” and “ECE” of type Department (cf. line 32), etc. It is important to observe in this figure that all the ownership arrows descend. This is enforced in XYZ through a type-based mechanism that shall be described shortly.

Note that several context instances can own another same context instance, leading to a form of multi-
ownership, which allows the sharing of state, a prevalent characteristic of object-oriented programming.

The ownership network enables parallel execution of events provided that they do not access shared state. When multiple concurrent events can potentially access the same state, XYZ serializes the events by exploiting the ownership network.

The DAG structure of the ownership network guar-
antees that for any two context instances that might have a common descendant context instance, there exists an ancestor context instance that transitively owns both (we have a join-semi-lattice). In particular, for any set of context instances that have a common set of descendants, we are interested in the least common ancestor dominating them. Formally: for context instance $ctx$ in an ownership network $\Omega$, assuming that desc$(\Omega, ctx)$ represents the set of its descendant context instances, $\Omega$ is a join-semi-lattice if for any $ctx_1, ctx_2 \in desc(\Omega, ctx)$ we have $\Omega \cap ctx_1 \geq ctx_2$ and $\Omega \cap ctx_2 \geq ctx_1$.

Figure 3: Piazza static and dynamic context structure.
instances, let \( \text{share}(\Omega, cx) \) be the set defined as follows:

\[
\text{share}(\Omega, cx) = \{cx' \mid \text{desc}(\Omega, cx) \cap \text{children}(\Omega, cx') \neq \emptyset \} \cup \{cx' \mid \text{desc}(\Omega, cx'') \cap \text{desc}(\Omega, cx) \neq \emptyset \land cx' \notin \text{desc}(\Omega, cx) \land cx \notin \text{desc}(\Omega, cx') \}
\]

Then, we find in \( \text{share}(\Omega, cx) \) all context instances which share a descendant context and instance otherwise incomparable with \( cx \) through the directly-owned relation (encoded through desc), and all the context instances which might be an owner of \( cx \) and moreover share a common child with \( cx \).

Then, to calculate the context instance dominating all context instances that potentially share something with \( cx \), denoted \( \text{dom}(\Omega, cx) \) and dubbed \( cx \)'s “dominator” we can compute the least upper bound (lub) of the nodes in \( \text{share}(\Omega, cx) \cup \{cx\} \) in the lattice \( \Omega \).

\[
\text{dom}(\Omega, cx) = \text{lub}(\Omega, \text{share}(\Omega, cx) \cup \{cx\})
\]

Methods and events. Events represent asynchronous client requests to the XYZ application, and therefore define its external API. To simplify the syntactic categories of XYZ, and avoid code duplication, events are simply method calls decorated by the event keyword targeted at a context instance. 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 context instances, but from the programmers’ perspective, the execution of events is atomic. The execution of an event conceptually begins at the target context instance, the context instance providing the method being called. An event executing in a certain context instance \( cx \) can issue method calls to any context instances that \( cx \) owns, and in this way can affect any context instance transitively reachable in the ownership DAG from \( cx \). An event dispatched within another event will receive the same treatment as any other client’s events, and be scheduled at the end of the target context instance’s queue (detailed in Section 4). This is in contrast to synchronous and asynchronous method calls whose execution is entirely contained within – it is part of – the current event execution.

As it can be seen in Figure 2 there is an optional ro method modifier. This allows the declaration of methods that are readonly, which enables the execution of multiple readonly requests in single context concurrently (see Section 5). A simple check guarantees that readonly methods can only use other readonly methods, and that they cannot update the heap.

Type-based enforcement of DAG ownership. As stated before, an important invariant to achieve a deadlock-free linearizable semantics for XYZ is that the ownership network be acyclic (at least with respect to context instances 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 context types 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 context types are contained in a certain context type.

To enforce this property we put in place a simple analysis that collects for each context method declaration, an over-approximation of the types of contexts that it could access. This is easy to do with a recursive procedure which, since the language is in Administrative Normal Form (ANF) [14], requires a single pass over the declarations of contexts. Then, whenever a context type \( cxn_0 \) declares an event that can use a context type \( cxn_1 \), we require that the context type \( cxn_0 \) appears always at a higher level in the ownership network than \( cxn_1 \) and we denote this constraint as \( cxn_1 \leq cxn_0 \). The analysis succeeds if the collected constraints are acyclic except for the obvious reflexive cases (i.e., \( cxn \leq cxn \)), and rejects the program otherwise. The 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 modification of context ownership structure (code is generated).

4. Semantics

In this section we present a formal semantics of a subset of the XYZ language, which we shall call XYZcore. The semantics of XYZcore will be described in two stages. The former considers the executions that happen within a context instance, without communication with other context instances, and the latter, considers the synchronization of multiple context instances in the whole system.

4.1 Intra-context semantics

The first layer is a big-step semantics of a single context instance in isolation. Of course, this semantics can only make progress as long as the currently executing event does not need to communicate with other context instances. In essence, this semantics represents the atomic evolution of an event within the boundaries of actions requiring communication. To formalize this semantics we introduce the following syntactic categories:

\[
\begin{align*}
o & \in O : F \rightarrow Val & \text{Objects} \\
v & \in Val \supset O & \text{Values} \\
\sigma & \in \Sigma : \text{Var} \rightarrow Val & \text{Store} \\
le & \in \mathcal{L}_c : \text{Var} \rightarrow Val & \text{Local Env.} \\
ge & \in \mathcal{G}_c : \text{Var} \rightarrow Val & \text{Global Env.} \\
am & \in \{ro, ex\} & \text{Access Mode} \\
cxn & \in C_{ID} & \text{Context ID}
\end{align*}
\]

where we denote a list with square brackets.
Then, a configuration of this semantics is simply a tuple of the form

\[(\sigma, ge, le, s, am)\]

Moreover, we add labels which are used by the following semantic level to synchronize among multiple context instances. Labels are sampled from the grammar:

\[
\ell \in \text{Labels} ::= \tau \mid \text{ret}_{\text{ctx}_{id}}(v) \mid \text{synch}_{\text{ctx}_{id}}(m, \vec{v}, am) \mid \text{asynch}_{\text{ctx}_{id}}(m, \vec{v}, am) \mid \text{event}_{\text{ctx}_{id}}(m, \vec{v})
\]

where the \(\tau\) label is silent, and shall generally be omitted. The label \(\text{synch}_{\text{ctx}_{id}}(m, \vec{v}, am)\) indicates that the transition issues a synchronous call to context instance \(\text{ctx}_{id}\) for method \(m\) and with arguments \(\vec{v}\). Moreover, the access mode is \(am\). The meaning of the other labels is self-explanatory. Then, the judgment of this semantics is of the form:

\[(\sigma, ge, le, s, am) \overset{\ell}{\rightarrow} (\sigma', ge', le', s', am)\]

The most important semantic rules of this level are shown in Figure 4.

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 \(\tau\). 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 filed assignments, and it requires that the access mode be \(ex\), thus preventing readonly calls from modifying the heap. We assume a given semantic function \([[]]\) to evaluate expressions. It is an error to write in a readonly call, and the semantics simply defines no transitions for programs attempting to do. Indeed, in XYZ a simple static analysis checks that readonly methods never write.

The rules for SYNCH and ASYNCH CONTEXT CALL are similar except for the event that they emit, and the place-holder command that they inject in the resulting configuration. In fact, we consider syntax for commands extended as follows

\[s \in S ::= \ldots \mid \text{waiting} \; \text{ctx} \mid \text{emit}\]

where \text{waiting} \(\text{ctx}\) and emit are runtime commands (they cannot appear in the source program), used to indicate that a synchronization with another context instance is necessary. Similarly, the fact that these commands emit an event represents that this semantic layer will be stopped from making progress until the label has been consumed by the rules of the inter-context semantics which we will consider next. Since the semantics of Figure 4 does not provide rules for these runtime commands, the execution will only be able to continue when they are removed in the subsequent semantic layer.

Finally, CONTEXT RETURN emits a return event on a return statement only if the current local environment \((le)\) contains a single call frame. This rule clears the local environment, where we denote by \(\epsilon\) the empty list. The other rules are standard for sequential programming languages, and are therefore ignored.

### 4.2 Inter-context semantics

The second layer of our semantics considers the composition of all context instances in the system. We will use the following category to indicate the type of requests made to a context instance:

\[d \in \text{Dec} ::= \text{synch} | \text{asynch} | \text{event} \; \text{Decorators}\]

Next, since there could be multiple readonly events executing their methods in a single context instance, we need a pool of activations representing different threads executing readonly methods within a context instance. Since each thread has their own local variable environment and code, we add this information into the activations. Moreover, since at any point there could be at most one exclusive access activation in a context instance, we use the activation set of a context instance as lock. The definition of activations is as follows

\[\text{atv} \in \text{Atv} ::= (e_{id}, am, d, le, s) | (e_{id}, am, \bot)\]

representing in the former case a running activation, and in the latter, an activation that has terminated or is currently inactive in this context instance, but whose lock cannot yet be safely removed.

The next ingredient we need to consider is the (request) queue that each context instance uses to buffer calls made to it. This queue will contain requests sampled from the following syntax where we assume that the metavariable \(e_{id} \in E_{1D}\) represents an event ID:

\[\text{req} \in \text{Req} ::= (e_{id}, m, \vec{v}, d, am) | \text{lub} \; (\text{event} \; \text{ctx}_{id})\]

The former request represents a call from event \(e_{id}\) to method \(m\) with decorator \(d\), with arguments \(\vec{v}\) and access mode \(am\). The latter denotes that an event executing in context instance \(\text{ctx}_{id}\) needs to lock the current context instance (which is the lub of \(\text{ctx}_{id}\)). This latter request serves only to lock the current context instance with the event, and the response to this request is simply a notification to the event \(e_{id}\) in \(\text{ctx}_{id}\) that the lock has been acquired, and execution can begin. Then, a request queue is simply a list of requests, and it will be ranged over with the metavariable \(q : [\text{Req}]\).

We are finally in a position to define context instances as considered by this semantics.

\[\text{ctx} \in \text{Ctx} ::= \{id : C_{1D}, q : [\text{Req}], \sigma : \Sigma, ge : \mathcal{G}_{e}, A : 2^{\text{Atv}}\}\]

In this definition we use a record-like syntax, to name each of the components of a context instance. Then,

---

\(^5\) For readability we omit the method environment, which is also a static element of the intra-context configuration.

\(^6\) Recall that we denote a list with brackets.
Sequence
\[ (\sigma, \text{ge}, \text{le}, c_1, \text{am}) \rightarrow (\sigma', \text{ge}', \text{le}', \text{skip}, \text{am}) \]
\[ (\sigma, \text{ge}, \text{le}, c_1; c_2, \text{am}) \rightarrow (\sigma', \text{ge}', \text{le}', c_2, \text{am}) \]

Sequence Label
\[ (\sigma, \text{ge}, \text{le}, c_1, \text{am}) \overset{\ell}{\rightarrow} (\sigma', \text{ge}', \text{le}', c_1', \text{am}) \]
\[ (\sigma, \text{ge}, \text{le}, c_1; c_2, \text{am}) \overset{\ell}{\rightarrow} (\sigma', \text{ge}', \text{le}', c_1; c_2, \text{am}) \]

Field Update
\[ [\ell](\sigma, \text{ge}, \text{le}) = o \quad [e](\sigma, \text{ge}, \text{le}) = v \quad \sigma' = \sigma[\text{o.f} \leftarrow v] \]
\[ (\sigma, \text{ge}, \text{le}, x.f := e, \text{ex}) \rightarrow (\sigma', \text{ge}, \text{le}, \text{skip}, \text{ex}) \]

Synch Context Call
\[ [\ell]\{v\}(\sigma, \text{ge}, \text{le}) = v \quad m \text{ access mode is am} \]
\[ (\sigma, \text{ge}, \text{le}, y := c, \text{am}) \xrightarrow{\text{synch}} (\sigma, \text{ge}, \text{le}, y := \text{waiting} c; \text{am}) \]

Asynch Context Call
\[ [\ell]\{v\}(\sigma, \text{ge}, \text{le}) = v \quad m \text{ access mode is am} \]
\[ (\sigma, \text{ge}, \text{le}, \text{asynch} c, \text{am}) \xrightarrow{\text{async}} (\sigma, \text{ge}, \text{le}, \text{emit}; \text{am}) \]

Context Return
\[ [e] = 1 \]
\[ (\sigma, \text{ge}, \text{le}, \text{return} v; c, \text{am}) \xrightarrow{\text{ret}} (\sigma, \text{ge}, c, \text{am}) \]

Figure 4: Intra-context big-step semantics

\( cx.q \) represents the queue of context instance \( cx \). Moreover, we will use the following notation for in-place update of the \( \sigma \) component of the \( cx \):

\[ cx[\sigma \leftarrow \sigma'] = x = \begin{cases} \sigma' & \text{if } x = \sigma \\ cx.x & \text{otherwise} \end{cases} \]

a similar convention applies to all other named components of the context instance record.

Finally, a configuration of the whole system is a pair \((E, \Omega)\) comprising a set of currently running events \( E \subseteq 2^{E_i.D} \) and a set of context instances \( \Omega \subseteq 2^{C_i.x} \) representing the ownership network.

The inter-context semantics, presented in Figure 5 is given by judgments of the form

\[ (E, \Omega) \rightarrow (E', \Omega') \]

Composition rules. Let us now consider the rules of Figure 5. These rules are concerned with the calling and returns of events and context calls. They ignore how an event becomes activated in a context instance, which will be discussed in Figure 6.

The rule Lift Intra simply propagates the behavior of the intra-context semantics of Figure 3 by arranging the elements of a context instance \( cx \) and its activations as a configuration of the semantics of Figure 4. Note that this is the only rule to produce an arrow of the form \((\ell, e_id)\) which is used in the premises of the other rules. Moreover, notice that the label is simply propagated, with the added information of the event emitting it.

The rule Synch Call considers the case where the intra-context semantics emits a synchronous call label. This rule involves context instances \( cx_0 \) the source of the call, and \( cx_1 \), its target – notice that we check that \( cx_1 \) is a child of \( cx_0 \). As explained before, the only effect of this step is to add the a request to the tail of \( cx_1 \)’s queue.

The rule Synch Return in turn considers the case where a synchronous call is terminated in \( cx_1 \). In this case we check that indeed an activation \( a \) in \( cx_0 \) is waiting for the response of a synchronous call to \( cx_0 \) (it is an invariant that for each event, at most one activation is waiting for a synchronous call from a single context instance). In that case, we substitute the waiting \( cx_0 \) command for the actual value produced by the call, therefore allowing the semantics of Figure 4 to proceed. Notice that the activation is not removed from context instance \( cx_0 \) upon the return, but rather replaced with a \( \bot \) marker. This is because this activation acts as a lock for the context instance, and it will only be released upon the completion of the whole event \( e_id \).

The rule for Asynch Call is similar to the one for Synch Call, with the exception that the emit placeholder is immediately removed from the command of the issuing context instance when the event is placed in the target’s queue. This is because the caller context instances needs not wait for the termination of the callee, and so it’s intra-context semantics can proceed. Consequently, the rule for Asynch Return is also simpler than its Synch counterpart, and in particular it involves a single context instance \( cx \).

The next three rules consider the case of asynchronous event calls issued to an event. The rule Event Call Unshared considers the case where the target context instance of the event, \( cx \) is its own dominator context instance (cf. the definition of Section 3). In this case, no other context instances need to be contacted for locks, and therefore all the rule does is to pick a fresh event identifier, \( e_id \notin E \), and add the event in the context instance’s queue. On the other hand, if the dominator of \( cx \) is another context instance \( cx_{\ell} \), the new event is added in the queue of \( cx_{\ell} \) as a marker indicating that the context instance \( cx_{\ell} \) has to be locked before the event starts executing. Notice in rule Event Call Shared that the target context instance \( cx \) is unmodified by the transition.

The final rule in this figure is Event Return & Commit. This rule simply removes the event ID \( e_id \) from the event set, and it removes any activation appearing in any context instance for the event \( e_id \). This
Lift Intra
\[(cx, σ, cx, ge, a, le, c, a, am) \mapsto (σ', ge', le', c', a, am)\]
\[cx.A = A \cup \{a\} \quad cx' = cx'[σ' ← σ'][ge' ← ge'][a ← A \cup \{a[le ← le'][c ← c']}]\]

Synch Call
\[cx_1 \in cx_0, c_0h \quad (\text{synch}_{cx_1}(m, e, am), e_{id}) \quad cx_0' = cx_1[q ← q'(e_{id}, m, v, \text{sync}, am)] \quad (E, Ω \cup \{cx_0, cx_1\}) \mapsto (E, Ω \cup \{cx_0', cx_1\})\]

Synch Return
\[cx_1 \leftrightarrow cx_1'. \quad \text{asynch}_{cx_1}(m, e, am) \quad cx_0 = cx_0'[c ← q'(e_{id}, m, v, \text{async}, am)] \quad (E, Ω \cup \{cx_0, cx_1\}) \mapsto (E, Ω \cup \{cx_0', cx_1\})\]

ASynch Call
\[cx_1 \in cx_0, c_0h \quad (\text{asynch}_{cx_1}(m, e, am), e_{id}) \quad cx_0' = cx_1[q ← q'(e_{id}, m, v, \text{async}, am)] \quad (E, Ω \cup \{cx_0, cx_1\}) \mapsto (E, Ω \cup \{cx_0', cx_1\})\]

Event Call Unshared
\[e_{id} \notin E \quad \text{m has access mode am in cx} \quad cx = \text{ lub}(Ω \cup \{cx\}, cx) \quad cx' = cx'[q ← q'(e_{id}, m, v, \text{event}, am)] \quad (E, Ω \cup \{cx\}) \mapsto (E, Ω \cup \{cx', cx\})\]

Event Call Shared
\[cx_{id} \notin E \quad \text{m has access mode am in cx} \quad cx_{id}' = cx_{id}[q ← q'(e_{id}, m, v, \text{event}, am)] \quad (E, Ω \cup \{cx_{id}\}) \mapsto (E, Ω \cup \{cx_{id}', cx\})\]

Event Return & Commit
\[cx \leftrightarrow cx' \quad \text{asynch}_{cx}(m, e, am) \quad cx. A = A \cup \{a\} \quad a.e_{id} = e_{id} \quad a.am = \text{async} \quad cx'' = cx'[A ← A \cup \{e_{id}, a, am, \perp\}] \quad (E, Ω \cup \{cx\}) \mapsto (E, Ω \cup \{cx''\})\]

Figure 5: Global Rules (except activations)

is encoded with the notation \(Ω \downarrow e_{id}\) as follows.
\[Ω \downarrow e_{id} = \begin{cases} \emptyset & \text{if } Ω = \emptyset \\ Ω' \downarrow e_{id} \cup \{cx_0[A ← \{A/(e_{id}, \ldots)\}]\} & \text{if } Ω = Ω' \downarrow \{cx_0\} \end{cases}\]

Notice that since we use activations as a lock, this achieves the effect of removing the lock (either read-only or exclusive) from all context instances that have been visited by the event \(e_{id}\).³

Activation rules. Let us now concentrate on the activation rules show in Figure 6, which dictate how requests are removed from a context instance’s queue, and scheduled for execution.

The rules Exclusive Access and Read-only Access consult the first request in the requests queue. In the former case, of the event as \(cx\) access mode, the rule requires that the current activation set of the context instance \(cx.A\) be empty, meaning that there are no other events currently holding the context instance. If the check is successful an activation is added for this call, which will enable the lift INTRA rule to start the execution. Evidently, the head of the queue is removed. The latter case simply checks that no \(cx\) activation is currently present in the activation set.

The LUB Lock and Schedule ... rules perform a similar check in the lub context instance, and upon successfully activating the event, they simply forward the request to the target context instance (by adding it in its queue). Notice that the activations added are of the form \((e_{id}, am, \perp)\) since the event does not execute in this context instance, but the activation is only used as a lock.

Finally, the CALL PROMOTION rule considers the case where a call is made to a context instance that was already visited by the event (i.e. \(e_{id}\) was already activated in the context instance). In this case, the call can be executed regardless of other events that might have arrived to the context instance⁴ and is therefore upgraded to the front of the requests queue.

Definition 4.1. We take the semantics of an XYZ\(_\text{core}\) program to be given as the transitive closure of the \(\to\) relation (denoted \(\to^*\)) between configurations, starting with a given initial configuration \((\emptyset, Ω_1)\) containing no events.

4.3 Meta-properties

In this section we develop the main claims about the high-level semantics of XYZ\(_\text{core}\).

Deadlock freedom. Let us begin by formally defining how deadlocks could manifest in XYZ\(_\text{core}\).

³Evidently, the implementation of this rule requires communication between the dominator and all context instances affected by \(e_{id}\).

⁴Notice that this could only happen if the context instance \(cx\) is its own dominator, since otherwise these events would not be queued in this context instance, but in its dominator instead.
**Definition 4.2** (Deadlock State). We say a state \((E, \Omega)\) contains a deadlock, if \(\Omega\) can be decomposed as:
\[
\Omega = \Omega' \cup \{cx_0, cx_1, \ldots, cx_n\}
\]
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 \(cx_i.A\), and (ii) \(e_i\) occurs in \(cx_{i+1}.A\), considering addition modulo \(n\).

A representation of a deadlock state is shown below, where the context instances involved in the deadlock are marked with the primed names \((cx_0', cx_1', cx_2')\). Notice that in the queues of each of these context instances there is a request of an event that is currently holding that in the queues of each of these context instances.

**Theorem 4.3.** The semantics of XYZ\textsubscript{core} guarantees that no deadlock state can be reached\(^{11}\).

**Linearizability.** Let us now show that the execution of XYZ\textsubscript{core} events is linearizable. We begin by stating some simple lemmas that are necessary for the proof.

**Linear semantics of XYZ.** We consider an alternative semantics of XYZ, given also by the rules of Figure 4, Figure 5 and Figure 6, where we add the additional constraint that there is at most one event activated in the context instance set \(\Omega\). This essentially encodes a semantics of XYZ where each event executes in complete isolation. Let us call this the linear semantics of XYZ and denote it with the special arrow \(\rightarrow\) and denote context instance sets generated by this semantics as \(\hat{\Omega}\).

The proof of linearizability shows that the linear semantics of XYZ\textsubscript{core} \((\rightarrow\)\) can simulate the semantics of XYZ\textsubscript{core} \((\rightarrow\)\). To that end we consider a simulation relation between configurations of the two different semantics. We denote by \(\text{Act}(\hat{\Omega})\) the set of events that are activated in any context instance of \(\hat{\Omega}\).

**Definition 4.4** (Simulation Relation). We define the following simulation relation between a configuration of the linear semantics of XYZ\textsubscript{core}, say \((E, \hat{\Omega})\), with a configuration of the normal XYZ\textsubscript{core}, say \((E, \Omega)\), which we denote \((E, \hat{\Omega}) \rightarrow (E, \Omega)\) iff the following hold:
(i) \(\text{Act}(\hat{\Omega}) = \emptyset\), and
(ii) \((E, \hat{\Omega}) \xrightarrow{\hat{\Omega}} (E, \Omega)\).

Our simulation proof amounts to showing that the diagram below commutes.

\[
\begin{array}{ccc}
(E, \Omega) & \xrightarrow{\hat{\Omega}} & (E', \hat{\Omega}') \\
\downarrow \mathcal{R} & & \downarrow \mathcal{R} \\
(E, \Omega) & \xrightarrow{\mathcal{R}} & (E', \Omega')
\end{array}
\]

That is to say, each time we start with a pair of similar configurations, whenever the normal semantics of XYZ\textsubscript{core} (the low-level semantics) makes a step, the resulting configuration is still related to the linear one, or there exist a number of steps in the linear semantics that can simulate exactly the same behavior.\(^{12}\) 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.

**Theorem 4.5** (Simulation). The relation \(\mathcal{R}\) defined above is a weak-simulation from the normal (low-level) semantics of XYZ\textsubscript{core} to the linear semantics of XYZ\textsubscript{core}.

The proof of this theorem is supported by a commutativity argument between transitions corresponding to different events (see Theorem A.10).

Evidently, the simulation induces a linearizability proof, where events are linearized at their commit point.

\(^{11}\) Proofs are provided in Appendix A for reviewing.

\(^{12}\) Technically, a weak-simulation.
Corollary 4.6 (Linearizability). Since the linear semantics of XYZcore is linear, and it does not reorder non-overlapping events, Theorem 4.3 implies that the semantics of XYZcore enforces linearizability of events.

4.4 Relaxing the direct-owner restriction

Notice that calls could be issued from a context instance to any descendent context instance, provided that locks are acquired in a top-down hand-in-hand fashion (this is required to guarantee deadlock freedom). But then, allowing the runtime system to perform the locking automatically on a path from the caller to the destination context instance 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 context instance network without accessing all intermediate descendants. To model that behavior we use an asynchronous “runtime” locking rule, which simply takes the lock of a context instance when the runtime best believes that it will be necessary following the above mentioned top-down hand-in-hand policy.

Auto Lock

\[
\begin{align*}
\text{cx}_1 \in \text{cx}_{0,cch} & \quad (e_{id}, am, \ldots) \in \text{cx}_{0,A} \\
\text{am} = \text{ro} & \Rightarrow \text{ex} \quad \text{not in} \quad \text{cx}_1,A \\
\text{am} = \text{ex} & \Rightarrow \text{cx}_1,A = \emptyset \\
\text{cx}_1' = \text{cx}_1[a \leftarrow (e_{id}, am, \perp)] & \\
& \quad (E, \Omega \cup \{\text{cx}_0, \text{cx}_1\}) \rightarrow (E, \Omega \cup \{\text{cx}_0, \text{cx}_1'\})
\end{align*}
\]

This rule allows us to relax the constraint \(\text{cx}_1 \in \text{cx}_{0,cch}\) in the CALL rules for \(\text{cx}_1 \in \text{desc}(\Omega \cup \{\text{cx}_0, \text{cx}_1\}, \text{cx}_0)\), and hence, allows any context instance to call any of its descendant context instances.

5. Implementation

We implemented XYZ on top of Mace [20], a C++ language extension that provides a unified framework for network communication and event handling. The implementation of XYZ consists in roughly 10,000 lines of core code and 110 new classes on top of Mace. Due to space constraints we focus on briefly discussing (i) how scalability is increased through parallelism and scale adjustment and (ii) fault tolerance.

Parallelism. The ownership network of context instances already implies that whenever two events target context instances that do not share descendants, they can execute in parallel and independently as shown in Section 4 (cf. Theorem A.11).

Moreover, the runtime of XYZ puts in place an optimization over the semantics presented in Section 4 for the case where two events target context instances that share only a subset of descendant context instances.

In fact, it even allows bottom-up calls provided that the required context instances are already locked, and therefore not subject to further deadlocks.

Here, the serialization achieved by locking the dominator in subsection 4.2 does not prevent the event arriving later (failing to acquire the dominator lock) from starting to execute on context instances that are unshared. This optimization is clearly safe since the latter event will not attempt to acquire the context instances that are shared by the two targets, therefore preventing the possibility of the former event being delayed by it, and more importantly, avoiding the possibility of deadlock. This implies that if an event does not manage to acquire the lock of the dominator, but does not attempt to access any context instances that are descendants shared with the target of the event currently holding the common dominator, the event can complete.

Automatic scale adaption. XYZ runtime implements a scale manager (smanager) to manage global information and operation. XYZ’s smanager delivers two capabilities: (i) maintaining the global context map (a data structure mapping context instances to servers) and ownership network, and (ii) automatic scale-adjustment by managing the context instances creation and migration. Fault-tolerance of the smanager is achieved through state machine replication [21].

The smanager follows the following protocol to migrate a context instance \(\text{cx}\) from a host \(s_1\) to a new host \(s_2\): (1) The smanager sends a prepare message to \(s_2\), notifying that requests for context instance \(\text{cx}\) might start arriving. Then, \(s_2\) responds by creating a queue for context instance \(\text{cx}\) and acks the smanager. (2) Upon receiving the ack, the smanager informs \(s_1\) to stop receiving events targeting \(\text{cx}\) and waits for \(s_1\)’s ack. (3) Once the smanager receives the ack, it updates its context mapping, assigning \(C\) to \(s_2\). It then sends a special event called \(\text{migrate}(\text{cx}, s_2)\) to \(s_1\) indicating that \(\text{cx}\) has to be migrated to \(s_2\). (4) Upon receiving \(\text{migrate}(\text{cx}, s_2)\), \(s_1\) enqueues an event \(\text{migrate}_{\text{cx}, s_1}\) in \(\text{cx}, q\). This event serves as a notification for context instance \(\text{cx}\) that it must migrate. When \(\text{migrate}_{\text{cx}, s_1}\) reaches the head of \(\text{cx}, q\), \(s_1\) spawns a thread to move \(\text{cx}\) to \(s_2\). (5) Upon completion of the migration, \(s_2\) notifies the smanager that the migration is finished, and starts executing the enqueued events for context instance \(\text{cx}\). Observe that this simple protocol trivially ensures correctness during migrations while still not stalling execution across all context instances. As we show in Appendix B, the performance penalty induced by our migration protocol is limited.

Fault tolerance. Fault tolerance of the smanager is achieved through state machine replication [21] for high availability. XYZ provides programmers with a snapshot API to handle faults in application components. In short, it allows programmers to take consistent snap-
shots of a given context instance and all its children. To this end, and upon receiving a snapshot request for a context instance, the XYZ runtime system dispatches a particular event called snapshot event to that context instance. Snapshots of context instances and their children are achieved by writing them into a persistent storage. To improve 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 instance, the runtime system will ignore it during the checkpointing phase.

6. Evaluation

In this section, we compare scalability and performance of our XYZ implementation with the two most closely related programming models: EventWave [9] and Orleans [7]. We also study the impact of multiple ownership on XYZ’s performance. Subsequently, we compare XYZ potential for dynamic scale adaptation to Amazon’s ElastiCache (due to space constraints, we delegate the elasticity results to Appendix B). All our measurements are done in Amazon EC2. Before we present our results we first briefly review EventWave and Orleans:

Orleans. Orleans is an open-source framework, developed by Microsoft, based on the actor model. It has a 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, the current open-source version does not provide transactional guarantees. But for many cloud applications, transactional execution is required for consistency, and the correct manual implementation of distributed transaction execution always requires considerable effort from programmers. In fact it’s easy to run into deadlocks in Orleans with a cycle of synchronous method calls, grains do not allow reentrance.

EventWave. EventWave provides abstractions analogous to contexts. EventWave guarantees linearizability by totally ordering all requests at the (single) root context instance, thus severely limiting scalability and overall performance. Moreover, EventWave halts all executions during migration of components, which severely hampers elasticity. It also provides only limited expressivity compared to XYZ since it organizes context instances strictly as a tree and does not support modification of tree edges.

6.1 Scalability and performance

In order to compare scalability and performance of XYZ 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 [33] in all three systems.

The TPC-C benchmark is an on-line transaction processing benchmark. TPC-C is a good candidate for comparing XYZ with its rivals since it has multiple transaction types with different execution structures. Observe that transactions in TPC-C are similar to events in XYZ 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 XYZ; in reality, specifically engineered elastic distributed databases may be a better fit for serving TPC-C style applications. In our experimental setup, we run XYZ and EventWave servers on m3.medium servers, and TPC-C clients on m3.large servers of Amazon EC2. The TPC-C benchmark implementation in XYZ uses the following context declarations:

```java
context Warehouse { set<Stock> s; set<District> d; } context Stock { ... } context District { set<Customer> c; set<Order> o; } context Customer { History h; set<Order> os; } context Order { set<Order> o; set<OrderLine> 1; } context NewOrder { ... } context OrderLine { ... }
```

Since the number of items is fixed (i.e., 100K), warehouse and items form a single context instance. Also observe that an Order context has two owners: District and Customer.

A typical approach for evaluating scalability of a system using TPC-C is to partition TPC-C by warehouse, and put each warehouse on a single server [11, 32]. But, as pointed out by Mu et al. [24], this approach does not stress the scalability and performance of distributed transactions (i.e., events in our programming model) because less than 15% of transactions are distributed. Therefore, we also partition TPC-C by district similar to Rococo [24].

We implemented two variants of TPC-C in Orleans: (i) A version that ensures linearizability (atomic execution of transactions). It is called Orleans throughout this section, and is implemented by exploiting the fact that stateful Orleans grains are single-threaded, and we orchestrate grains in a tree-like structure a la EventWave. (ii) We also implemented a non-linearizable variant called Orleans*, in which the application’s invariants are 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 as it fails to ensure consistency.

Scale out. Figure 7a plots scalability of different systems for the TPC-C benchmark. In this experiment, we put one district (along with its corresponding customers, orders, etc.) per server. While neither EventWave nor Orleans can scale as the number of servers...
increases, we observe that XYZ scales up to 8 servers. At this point, the warehouse context instance becomes saturated, thus XYZ delivers its maximum throughput.

Scalability of XYZ is driven by using the ownership network (as explained in § 6.1) along with async method calls inside it. Therefore, as an event (i.e., a TPC-C transaction) finishes its execution in a parent context instance, it can continue its execution in a child context instance and releases the parent context instance by calling an async method in the child context instance. For instance, once a payment transaction finishes its execution in a warehouse context instance, it calls a method in a district context instance asynchronously, and releases the warehouse context instance. This allows another event to enter the warehouse for execution. Consequently, the scalability of the benchmark is bounded to the speed of the warehouse context instance, and how fast it can handle events. It is important to note that the linear scalability of Orleans* is only possible due to lack of enforcement of the TPC-C invariants and thus by sacrificing consistency.

**Performance evaluation.** Figure 7b shows TPC-C performance boundaries of the studied systems 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 the fact that both implementations fail to handle high contention at the warehouse context instance properly. As shown in both Figure 7a and Figure 7b, XYZ outperforms Orleans* with 8 servers. To better understand this phenomenon, and why with 8 servers Orleans* still cannot outperform XYZ, we run an experiment with a single client to completely eliminate any contention. We noticed that on average the client in Orleans* observes 35ms to execute a TPC-C transaction while XYZ’s client on average observes 6ms latency. Along with C# implementation of Orleans, we attribute this latency gap to co-location of contexts in XYZ (respectively grains in Orleans): while XYZ allows programmer to define rules for co-locating certain contexts on one machine, Orleans does not provide any means to this end. Therefore, two grains that may communicate with each other very often can be placed on two separate machines.

**6.2 Multiple ownership**

In this section, we study the effect of multiple ownership, which significantly increases the expressivity of XYZ, on the system performance. To this end, we implemented the following versions of Piazza in XYZ: (i) AION-Nshare: AION with a tree structure where there is no sharing; (ii) AION-Cshare: all Course context instances within a Department will share one Student context instance (i.e., a student registers for all courses in a department); (iii) AION-Share: all Course context instances in the University share one Student context instance. In addition, we also implemented Piazza in EventWave which only supports a tree ownership structure. We omitted Orleans because the goal of this experiment is to focus on the cost of (multiple) ownership.

We initialized Piazza with only one university. Each Department context instance owns 4 Course context instances while every Course context instance has 10 Student context instances. We place Piazza server on m1.small servers. Each such server has only one Department context instance with its Course and Student context instances. Clients are deployed on m1.medium servers with at most 10 clients per server. Finally, 50% of events generated by the clients are directed to Course context instances and 50% to Student context instances.

**Scale out.** Figure 7c shows how Piazza and EventWave scale out. AION-Nshare and AION-Cshare scale out almost linearly with the number of servers. However, compare to AION-Nshare, throughput of AION-Cshare drops by 20% when the number of servers reaches 16. This is due to the fact that events targeting Course context instances in AION-Cshare need to be ordered at Department context instances. As expected, EventWave and AION-Share cannot scale as the number of servers increases since EventWave orders all events by its root, and all events targeted at Course context instances in AION-Share will be ordered at a single University context instance.

**Performance evaluation.** Figure 7d shows performance of Piazza when the number of servers is 8. Our
experiment shows poor performance of AEON-Ashare and EventWave due to their event ordering at a single root node. As soon as their ordering nodes become saturated, their latencies increase sharply. Also observe that the performance of AEON-Nshare still exceeds AEON-Cshare since the former has no multiple ownership. Yet, we observe that the performance degradation is small.

To conclude, and as we presented in this section, clearly multiple ownership has some impact on both performance and scalability of an application running in XYZ. Yet, as long as the application does not require an extreme sharing scenario where one context instance is being shared by all others (i.e., AEON-Ashare in our experiment), the overhead of multiple ownership is low, and significantly increases expressivity.

7. Related work

Formal semantics. The formal semantics of XYZ is based on the techniques of structural operational semantics advanced by [27], and the OO and distributed approaches of [12, 18]. While the techniques employed are based on these works, the similarities end there since the the specifics of XYZ make the formal semantics unique (incorporating the ownership network and concurrent events), and the proofs are specific to the properties provided by XYZ and its semantics.

Programming models. Orleans [7] and EventWave [9], described in Section 6 provide concepts similar to XYZ’s contexts and events. The originality of XYZ however resides in the ownership network, which allows us to guarantee linearizability, 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. Moreover, the specification of both systems is informal at best, whereas we provide a formal semantics and prove the high-level properties ascribed to XYZ.

The literature on programming models for distribution is vast. Here we cover only some of the works that are most related, or inspired XYZ. The actor model [4, 7, 20] 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 with respect to single actors, whilst an event in XYZ can atomically modify several contexts.

XYZ shares similarities with models tailored to multicore execution environments like Bamboo [30]. 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 XYZ, Bamboo optimizes concurrency for multiple cores; distribution, migration, and scale adaptation are not considered.

In SCOOP [25], 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 XYZ. Similarly, separate calls can only be issued on arguments of a method, which is SCOOP’s way of controlling what XYZ achieves through multiple ownership and events. SCOOP is not concerned with distribution or scale adaptation addressed by XYZ.

Emerald [19] is an OO distributed programming language, providing locality functionalities to allow programmers to relocate objects across the available servers. Unlike XYZ, Emerald does not guarantee atomicity, and synchronization is left to the programmer.

Elastic databases (e.g., ElasTras [12], Megastore [9]) are similar to XYZ: they partition and distribute data among a set of nodes and provide consistency in the face of concurrent accesses. Unlike XYZ, these do not provide a rich, self-contained programming environment for writing generic elastic cloud applications.

A pilot job framework offers dynamic computational resources to a set of tasks [5, 22, 28]. Applications running on such a framework can be split into a set of isolated tasks organized either as a “bag of tasks” [5, 22] or as a DAG workflow [28]. These tasks are similar to the events of XYZ, but unlike events in XYZ, tasks cannot communicate with each other.

8. Concluding Remarks

We have presented the design and implementation of the XYZ language. XYZ provides a sequential programming environment for the cloud based on the standard paradigm of object-orientation. We provide a formal semantics of XYZ source level programs, and show that this semantics exploits parallelism while providing linearizability and deadlock-freedom. We have experimentally shown that the XYZ 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 will consider lifting 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.
References


A. Proofs

In this appendix we provide the full proofs for the claims made in [Section 4]. Let us first present some basic remarks about the semantic rules of xyzcore presented in [Section 4].

Remark A.1. If an event e_{id} is executing within a context cx, then e_{id} ∈ cx.A. That is, only activated events can execute within a context.

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 context (i.e. do not make use of the \( \rightarrow \) transitions of Figure 4). Similarly, activation rules do not “execute” within the context – that is, they don’t modify any of the components of the context other than the activation sets.

Remark A.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 A.3. Consider a configuration \((E, \Omega)\), an event e_{id} ∈ E, and a context cx_0 such that e_{id} ∈ cx_0.A. Moreover, assume that \((E, \Omega \cup \{cx_0\}) \rightarrow (E', \Omega' \cup \{cx_0'\})\) where e_{id} ∈ E'. Then we have that e_{id} ∈ cx_0.A. In other words, once an event is activated in a context, it remains so, until completely eliminated from the event set E'.

Proof. This property is a simple consequence of Theorem A.1. Since the only rule that can de-activate an event from a contexts eliminates the event from the activations of all contexts at once.

Remark A.4. For any reachable configuration \((E, \Omega)\) such that a context cx occurs in \(\Omega\) we have that either \(|cx.A| \leq 1\), or every activation in is ro.

Proof. This is immediate from the antecedents of the activation rules (Figure 6). In particular, notice that EXCLUSIVE access rules require that the initial activation set be empty, and READONLY rules require that no exclusive access event is in the current activation set (ex \(\notin cx.A\)).

Deadlock Freedom. Let us refresh the definition of deadlock provided in Theorem 4.2.

Definition A.5 (Deadlock State). We say a state \((E, \Omega)\) contains a deadlock, if \(\Omega\) can be decomposed as:

\[\Omega = \Omega' \cup \{cx_0, cx_1, \ldots, cx_n\}\]

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 \( cx_i.A \), and (ii) \( e_i \) occurs in \( cx_{i+1}.q \), considering addition modulo \( n \).

We repeat the figure of Section 4 representing a deadlock state. Notice that in the queues of each of these contexts there is a request of an event that is currently holding the previous context in the chain (shown with the incoming arrows marked \( req(e_i) \)), therefore closing the cycle.

**Lemma A.6.** For each event \( e_{id} \), and a state \((E, \Omega)\) such that there exists a context \( cx \in \Omega \) with \( e_{id} \in cx.A \), whenever \( \text{dom}(\Omega, target(e_{id})) = cx \) we have that \( e_{id} \in cx.A \).

In words, whenever an event \( e_{id} \) is executing in any context, it is also activated in the dominator context corresponding to the events target context according to \( \Omega \).

**Proof.** We notice first that the only rules that allow events to be added to the activation of a context are the rules of Figure 6.

Moreover, notice that for each of these rules, an item already existed in the queue of the context with the necessary event \( id \). These are the premises of the form \( cx_0.q = (e_{id}, \ldots) \). It is therefore sufficient to show that whenever a request is added to a queue, then either (i) it effectively is the dominator context, or (ii), the event is either enqueued in a context \( cx \) (i.e. \( e_{id} \in cx.q \)), or it is currently activated in it (i.e. \( e_{id} \in cx.A \)), then either \( cx \) is the dominator of \( e_{id} \)'s target \( (cx = \text{dom}(\text{target}(e_{id}), \Omega)) \), or a parent of \( cx \) contains the event \( e_{id} \) in its activations. Then, we have that all the contexts involved in the deadlock (as per Theorem 4.2) have a unique dominator in \( \Omega \). Formally, \( \text{lub}(\Omega, \{cx \mid cx \in \text{deadlock of } \Omega\}) \in \text{desc}(\text{dom}(\text{target}(e_{id}), \Omega)) \).

**Proof.** We proceed by induction on the number of contexts involved in the deadlock of state \((E, \Omega)\). In particular, we relax the condition of there being a cycle, to requiring only that there is a path in between any two contexts (as opposed to a cycle). We consider that there is a bidirectional edge between \( cx_0 \in \Omega \) and \( cx_1 \in \Omega \), if there exists an event \( e_{id} \in E \) such that \( e_{id} \in cx_0.A \) and \( e_{id} \in cx_1.q \), or vice versa. Then, we are interested in the length of the path formed with these bi-directional edges.

In the basic case, with only two contexts \( cx_0 \) and \( cx_1 \), we have that since in at least one of the two contexts one event is activated, and the other is enqueued, then both contexts share a common ancestor, which is guaranteed by Theorem A.7. This concludes the case by considering the dominator of this common ancestor in \( \Omega \).

**Corollary A.7.** For any reachable configuration \((E, \Omega)\), context \( cx_{i} \in \Omega \), and event \( e_{id} \in E \) such that \( e_{id} \in cx_{0}.A \) we have that \( e_{id} \in \text{dom}(\text{target}(e_{id}), \Omega).A \), and there exists a path of contexts \( \text{target}(e_{id}) \cdot cx_1 \cdot \ldots \cdot cx_n \cdot cx_0 \) in \( \Omega \) — considered as a graph — from \( \text{target}(e_{id}) \) to \( cx_0 \) such that for each \( i \in [1, n] \) we have \( e_{id} \in cx_i.A \).

**Proof.** This is a direct consequence of (i) \( \text{Theorem A.6} \) the fact that the \( \Omega \) is a directed-acyclic graph, and (ii) the fact that events are only ever deactivated (i.e. removed from contexts activation sets) by the rule EVENT RETURN & COMMIT which removes the event once and for all form the whole context set \( \Omega \).

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 A.8.** Let us consider a deadlock state \((E, \Omega)\), such that \( \Omega \) is a directed-acyclic-graph (DAG) as prescribed by the XYZ-core discipline. Moreover, let us assume that if an event \( e_{id} \) is either enqueued in a context \( cx \) (i.e. \( e_{id} \in cx.q \)), or it is currently activated in it (i.e. \( e_{id} \in cx.A \)), then either \( cx \) is the dominator of \( e_{id} \)'s target \( (cx = \text{dom}(\text{target}(e_{id}), \Omega)) \), or a parent of \( cx \) contains the event \( e_{id} \) in its activations. Then, we have that all the contexts involved in the deadlock (as per Theorem 4.2) have a unique dominator in \( \Omega \). Formally, \( \text{lub}(\Omega, \{cx \mid cx \in \text{deadlock of } \Omega\}) \in \text{desc}(\text{dom}(\text{target}(e_{id}), \Omega)) \).

**Proof.**
Proof. We consider a proof by contradiction. Let us assume that there is a first state \((E, \Omega)\) in a run of the semantics of Figure 5 containing a deadlock as per Theorem 4.2. Then, combining Theorem A.8 and Theorem A.6 we obtain a contradiction, since both events (one of which requires exclusive access) must be activated in the common dominator.

Linearizability. In this case we prove that the semantics of XYZ_core is linearizable. Let us start by stating some simple properties of the semantics.

Lemma A.10 (Exclusive Access). Given a trace \(\bar{\omega} = (E_0, \Omega_0) \rightarrow (E_0, \Omega_0) \cdots \rightarrow (E_n, \Omega_n)\), assume that two events \(e_0\) and \(e_1\) access at least one coinciding context, say \(cx\), in \(\bar{\omega}\). We have that for any configuration \((E, \Omega)\) appearing in \(\bar{\omega}\):

1. \(\text{dom}(\Omega, \text{target}(e_0)) \in \text{desc}(\text{dom}(\Omega, \text{target}(e_1)))\), or
2. \(\text{dom}(\Omega, \text{target}(e_1)) \in \text{desc}(\text{dom}(\Omega, \text{target}(e_0)))\), and

\(e_0 \notin cx.A\) or \(e_1 \notin cx.A\).

Proof. The claim 1 is a direct consequence of \(\Omega\) forming a DAG. [Theorem A.7 and Theorem A.6] and the fact that both events operate on a common context \(cx\). The claim 2 is a direct consequence of claim 1 in combination with Theorem A.7 and Theorem A.6.

The following lemma was presented in Section 4. Here we restate it and provide its proof. Similarly, we repeat Figure 8 to ease of presentation.

Lemma A.11 (Commutativity). Consider two consecutive transitions where we assume two contexts \(cx_0, cx_1 \in \Omega\), and two events \(e_0, e_1 \in E:\)

\[(E, \Omega_0) \xrightarrow{\text{cx}_0, e_0} (E, \Omega') \xrightarrow{\text{cx}_1, e_1} (E, \Omega_1)\]

Here we denote with the arrow \(\xrightarrow{\text{cx}}\) the fact that the transition involved the event identifier \(e\) and it was performed in the context \(cx\). We have that if either \(cx_0 \neq cx_1\) or \(e_0.am = e_1.am = ro\), there exists an intermediary context set \(\Omega'\) such that the following transitions are also valid:

\[(E, \Omega_0) \xrightarrow{\text{cx}_1, e_1} (E, \Omega'') \xrightarrow{\text{cx}_0, e_0} (E, \Omega_1)\]

Figure 8 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 [Theorem A.10] we have that if any of the events is \(ex\), then \(cx_0 \neq cx_1\). Moreover, we have that \(\text{LUB}(\Omega, cx_0.A) = \{e_0\}\) and \(\text{LUB}(\Omega, cx_1.A) = \{e_1\}\). Hence, by the definition of LUB we have that \(\text{desc}(cx_0) \cap \text{desc}(cx_1) = \emptyset\). It is not hard to see then, that the two events operate on disjoint portions of the graph, and therefore the transitions commute immediately.

Corollary A.12. Consider a sequence of transitions, where we use \(\bar{\omega}_{id}\) denotes any of the events in,

\(\alpha = (E, \Omega_0) \xrightarrow{e_0} (E, \Omega_1) \xrightarrow{e_1} \cdots \xrightarrow{e_n} (E, \Omega_n)\)

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

\(\alpha' = (E, \Omega_0) \xrightarrow{\delta(e_0)} (E, \Omega_1') \xrightarrow{\delta(e_1)} \cdots \xrightarrow{\delta(e_{n-1})} (E, \Omega_n)\)

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

As we did in subsection 4.3 we consider an alternative semantics of XYZ_core which allows at most one event at a time (see subsection 4.3). As before, we call this the linear semantics of XYZ and denote it with the special arrow “\(\rightarrow\)” and denote context sets generated by this semantics as \(\bar{\Omega}\).

The proof of linearizability shows that the linear semantics of XYZ_core can simulate the semantics of XYZ_core. To that end we consider as simulation relation between configurations of the two different semantics. We denote by \(\text{Act}(\Omega)\) the set of events that are activated in any context of \(\Omega\).

We consider as simulation relation between configurations of the two different semantics.
Definition A.13 (Simulation Relation). We define the following simulation relation between a configuration of the linear semantics of XYZ, \((E, \hat{\Omega})\), with a configuration of the low-level configurations of XYZ, \((E, \Omega)\), denoted by
\[(E, \hat{\Omega}) \mathcal{R} (E, \Omega)\]
iff the following conditions are met:
(i) \(\text{Act}(\hat{\Omega}) = \emptyset\), and
(ii) \((E, \hat{\Omega}) \overset{*}{\rightarrow} (E, \Omega')\)

We then prove that the semantics of linear XYZ simulates the semantics of low-level XYZ 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.

\[
\begin{array}{c}
(E, \hat{\Omega}) \quad \cdots \cdots \quad (E', \hat{\Omega}') \\
\mathcal{R} \quad \cdots \cdots \quad \mathcal{R} \\
(E, \Omega) \quad \cdots \cdots \quad (E', \Omega')
\end{array}
\]

Figure 9: XYZ simulation diagram

Theorem A.14 (Simulation). The relation \(\mathcal{R}\) defined above is a weak-simulation from the low-level semantics of XYZ to the linear semantics of XYZ, as shown in Figure 9.

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

The proof proceeds by case analysis on the transition taken by the configuration \((E, \Omega)\) of standard XYZ. 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 \((E, \Omega)\) can reach \((E, \Omega)\), 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 linear 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 linear semantics (i.e. the configuration \((E, \hat{\Omega})\)) must make actual transitions.

In this case, we can directly apply Theorem A.12 to obtain the conclusion, since we have from A.3 and A.2 that the committing event was the only one to touch these contexts (i.e. all other concurrent events are disjoint), and the event continued to hold these contexts until this commit step.

Evidently, the simulation above induces a linearizability proof, where events are linearized at their commit point.

Corollary A.15 (Linearizability). Since the linear semantics of XYZ\(_{core}\) is linear, and it does not reorder non-overlapping events, Theorem 4.3 implies that the semantics of XYZ\(_{core}\) enforces linearizability of events.

Parallelism

Definition A.16 (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 \(\vec{\omega} = \omega_0 \cdot (E, \Omega_0) \xrightarrow{cx_0} (E, \Omega_1) \cdot \omega_1 \cdot (E, \Omega_2) \xrightarrow{cx_1} (E, \Omega_3) \cdot \omega_3\) we have that \(cx_0 \neq cx_1\).

Theorem A.17. [Independence \(\Rightarrow\) Parallelism] Consider a trace \(\vec{\omega}\) and two independent events \(e_0, e_1 \in E\). We have that whenever
\[
\vec{\omega} = \omega_0 \cdot (E, \Omega_0) \xrightarrow{cx_0} (E, \Omega_1) \cdot \omega_1 \cdot (E, \Omega_2) \xrightarrow{cx_1} (E, \Omega_3) \cdot \omega_3
\]
there exists an equivalent trace
\[
\vec{\omega} = \omega_0 \cdot (E, \Omega_0) \xrightarrow{cx_1} (E, \Omega_1') \cdot \omega_1 \cdot (E, \Omega_2') \xrightarrow{cx_0} (E, \Omega_3') \cdot \omega_3
\]
where the order of the transitions of \(e_0\) and \(e_1\) is reversed.

Proof. This is an immediate consequence of Theorem A.11.

In a nutshell this theorem establish 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).

B. Scale Adjustment Experiments

Memcached is a popular key-value memory caching system for web applications [23]. Generally, the cache miss rate is determined by both the capacity of the memcached servers and the workload. For a fixed number of servers, the memory capacity is clearly bounded. When the workload exceeds the memory capacity, the cache miss rate will increase and more requests will be directed to permanent storage, downgrading the overall performance. So the ideal solution is to add and
remove memory capacity according to the workload, which could be done by an elastic memcached.

**XYZ Memcached and Memcached.** We have implemented an elastic memcached using XYZ. Unlike the standard memcached, our implementation is similar to a standard write-back cache mechanism, where the last used key-value is stored into the persistent memory to optimize the memory of the system. That is, clients do not know if there is a cache miss.

We set up our experiment on Amazon EC2 m1.medium instances with 128 clients over 16 instances. The size of a key-value pair is 1KB on average. The client will repeatedly pick a random key and send a read request to the memcached system. We vary the number of active clients over time. The XYZ memcached has 16 tables to store key-value pairs. Therefore, table contexts can be migrated into up to 16 servers according to the number of active clients.

For comparison, we also implemented standard memcached, in which all servers are independent from each other, and clients decide which server holds the pair. The workload setup for the standard memcached (the inelastic version) is the same as elastic memcached.

We run the experiment using XYZ, standard memcached with 5 servers, and standard memcached with 17 servers (one server is used as permanent storage). Figure 10a plots the average latency for each of the systems. The number of active clients (muave lines) is increased and then decreased, in two rounds. During the first round, the latency of the three versions has a similar shape since all of them require some warm-up time to retrieve the pairs from the persistent memory. In the second round, we can observe a more notorious performance gap. For standard memcached with 5 servers, 64 clients clearly exceed its capacity and a high cache miss rate results in increased latency. Standard memcached with 17 servers offers enough memory to handle the workload, so it can keep the average latency at a very low level. On the other hand, the capacity (i.e., the number of servers) varies based on immediate workload for XYZ memcached. Note that there are some short spikes for XYZ in Figure 10a. These are caused by the migration of contexts, as explained earlier.

Figure 10b shows the server utilization of XYZ memcached. It starts with 5 servers, increases first to 9 and then to 17, before decreasing inversely; then the whole process repeats. While the latency of XYZ memcached is higher than that of memcached with 17 servers, the monetary cost of the latter is fixed for 17 servers for the duration of the experiment and thus higher, although the 17 servers are only required for two short periods of time. On the other hand, through elasticity, we successfully obtained good performance with acceptable cost with XYZ. This is shown in Figure 10c, which shows the cost of running XYZ memcached and standard memcached with 5 and 17 servers. Since EC2 charges users hourly, and because we could not run our experiments for hours, we reproduced that argument on a per second rate. Figure 10c shows that the cost of memcached with 17 servers is about $0.35 for 50 seconds while XYZ memcached is charged only $0.2 for the same duration. Observe that when XYZ elastic Memcached scales out to 17 servers (Figure 10b), its latency is still higher than 17-server Memcached (Figure 10a). This is because the newly added servers need some time to populate their cache lines.

**Amazon ElastiCache.** We also compared XYZ memcached to the elastic version of memcached implemented by Amazon, called ElastiCache. We have the following experiment setup: 1. XYZ memcached: we run XYZ memcached on 8 Amazon EC2 m1.xLarge instances. 2. ElastiCache: we create a memcached cluster with 8 Amazon E2 cache.m3.xLarge instances. We have run the experiment with 80 clients and measured the request latency. We also ensured that no cache miss happens during our executions since we wanted to solely measure the latency of memcached, and not the data store attached to it.

We observed that on average, request latency in XYZ memcached is two times higher than Amazon ElastiCache: 2.08 ms in XYZ memcached compared to 1ms in Amazon ElastiCache. We note that cache.m3.xLarge instances are specifically tailored for Amazon Elasti-
Cache, and we believe that they play a crucial role in delivering such low latency in Amazon ElastiCache. On the flipside, while adding a m1.xLarge instance to scale XYZ memcached takes around 1min, it takes around 6 to 10 mins to add a cache.m3.xLarge instance for Amazon ElastiCache, which in applications with strong variations (in terms of frequency and extent) in workloads will likely lead to increased latencies when the system needs to scale out, and to increased costs when it needs to scale back in. The latter observation is particularly relevant as cache.m3.xLarge instances tailored for Amazon ElastiCache are ten times more expensive than m1.xLarge instances ($0.364/h vs. $0.032/h).