EventWave: Programming Model and Runtime Support for Tightly-Coupled Elastic Cloud Applications

Wei-Chiu Chuang, Bo Sang, Sunghwan Yoo, Rui Gu, Milind Kulkarni
Purdue University
{weichiu,bsang,sunghwanyoo,ruigu,milind}@purdue.edu

Charles Killian
Purdue University and Google
ckillian@google.com

Abstract

An attractive approach to leveraging the ability of cloud-computing platforms to provide resources on demand is to build *elastic* applications, which can dynamically scale up or down based on resource requirements. To ease the development of elastic applications, it is useful for programmers to write applications with simple sequential semantics, without considering elasticity, and rely on runtime support to provide that elasticity. While this approach has been useful in restricted domains, such as MapReduce, existing programming models for general distributed applications do not expose enough information about their inherent organization of state and computation to provide such transparent elasticity.

We introduce EVENTWAVE, an event-driven programming model that allows developers to design elastic programs with inelastic semantics while naturally exposing isolated state and computation with programmatic parallelism. In addition, we describe the runtime mechanism which takes the exposed parallelism to provide elasticity. Finally, we evaluate our implementation through microbenchmarks and case studies to demonstrate that EVENTWAVE can provide efficient, scalable, transparent elasticity for applications run in the cloud.

1 Introduction

One of the major promises of cloud computing is the ability to use resources-on-demand. Rather than build out a large-scale infrastructure to support a networked service with fluctuating needs, companies can build elastic components that start running on a small-scale infrastructure (perhaps only a handful of hosts), then scale up as the usage or popularity of the service grows. Elasticity can also allow changing infrastructure size on a smaller time-scale: weekly, daily or even hourly variations in load could trigger the infrastructure size to change with the needs of the service. Further, elasticity could allow variable infrastructure size as a result of the availability of discount resources, or the sudden unexpected needs of the service in response to, e.g., flash crowds.

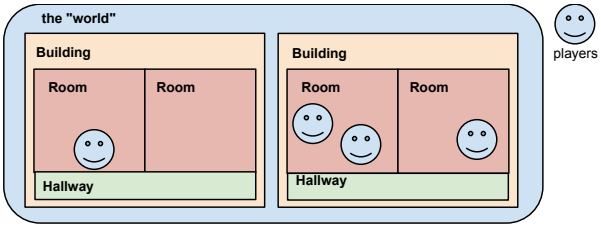


Figure 1: A multiplayer game composed of buildings, rooms, hallways and players

However, elastic infrastructure alone does not suffice to provide these benefits, because programmers must design the applications with elasticity in mind. Consider a simple multi-player game server shown in Figure 1. In this game, players wander in the virtual world. If a building is crowded with more players than the machine can handle, a logical solution to scale up the system is to split the world into buildings, and delegate the requests of different buildings to different nodes. This game can further scale up by splitting the buildings into rooms and hallways, and delegate the requests to more machines. Any such program must explicitly account for elasticity, which means that it must support the arrival or departure of nodes from the system: safely "splitting" and "merging" the world state as the system expands and contracts without losing program state, transparently migrating state, handling partial failures of the now-distributed system, etc. And all of this without affecting gameplay semantics. While all of these tasks can

SoCC'13, October 01–03 2013, Santa Clara, CA, USA

ACM 978-1-4503-2428-1/13/10...$15.00.
http://dx.doi.org/10.1145/2523616.2523617

be programmed manually, implementing the necessary run-time mechanisms and reasoning about the resulting application is an exceptional burden for developers.

To accelerate the development of elastic applications for the cloud, what we need is a programming model where programmers need not be aware of the actual scale of the application, or the run time support that dynamically reconfigures the system to distribute application state across computing resources. To support complex, tightly coupled applications, such as the game server we described, such an elastic programming model should provide several key features: (i) stateful computation, (ii) transparent elasticity, and (iii) simple semantics. Unfortunately, existing programming models for writing elastic applications do not satisfy all of these criteria.

In recent years, there have been many systems that provide elasticity for various distributed applications. Scalable databases can adaptively distribute database state to scale up a database's capabilities [1, 4, 12]. However, these systems only target part of a program's functionality and do not, for example, provide elasticity for a program's computation.

Functional programming models, such as MapReduce [13], Dryad [19], and "bag of tasks" batch schedulers [24, 30], are based on models of computation where computations do not depend on prior system state, and individual tasks are independent of one another. As a result, the runtime system can freely distribute tasks among varying numbers of nodes without involving the programmer. However, in a game server, the game world consists of state, and the "tasks" consist of actions taken by players in interacting with this world and with each other, precluding a functional approach.

Actor-based programming models support simple, event-driven, stateful computation [17]: state is partitioned among isolated actors, each of which can be run on a separate node. The actors receive incoming events that manipulate their state, and actors interact with each other by passing messages that trigger additional events. Unfortunately, traditional actor-based models do not support elasticity, as an actor is conceived of as a monolithic unit that handles incoming events atomically. The scale of an actor-based system is determined by the number of actors.

A recent system, Orleans, supports elasticity in the actor model, by replicating the state of actors, allowing multiple events to be processed simultaneously [8]. However, this state replication leads to complicated semantics for merging changes to actor state, and as a result, Orleans does not provide simple, sequential semantics. While this is acceptable in many applications, it may not be appropriate in certain cases. For example, in a game server, it seems desirable that events triggered by multiple players be resolved in some sequential order, and that every player observe that same order.

We propose EVENTWAVE, a new programming model for tightly-coupled, stateful, distributed applications that provides transparent elasticity while preserving sequential semantics. In EVENTWAVE, an application consists of a set of *logical nodes*. A logical node consists of some system state, and that state is manipulated by events that execute at the node. Logical nodes interact with each other by message passing. As in the actor model (and many other event-driven systems), EVENTWAVE logical nodes provide *atomic event semantics*: events the node receives appear to execute sequentially, and in order.

Our model: EVENTWAVE

EVENTWAVE applications provide elasticity by allowing logical nodes to execute multiple events in parallel, and by allowing a single logical node to be *distributed* over multiple *physical nodes* (e.g., multiple machines in a cluster, or multiple virtual hosts). The fundamental guarantee of the EVENTWAVE model is that a logical node distributed over multiple physical nodes *will maintain atomic event semantics*. In other words, a programmer can reason about the behavior of her program without considering elasticity: if a program is correct when running on a set of logical nodes, its semantics will be preserved even as the program scales up or down by running on various physical nodes. Programmers focus on the *logic*, not the *scale*.

Our model is based on the insight that many applications decompose into a hierarchy of different *contexts*, namely that modular and component design and object-oriented programming have led to program designs where large sections of code are inherently bound to a subset of application state, and that the application state is often further bound to the parameters of the function calls. For example, a context-based design of a game server can be composed of contexts for each room and building in the game world, and contexts for each player. Events that manipulate just a portion of the state will execute in the context that state resides in. For example, when assessing which direction a player may move, the program needs to consider information only about the current room, and players in it, not the global state.

To exploit the elasticity exposed by EVENTWAVE programs, we design a distributed run-time system. The system distributes the contexts of a single logical node across multiple physical nodes (for example, distributing different building contexts to different physical nodes). The group of physical nodes appears to the rest of the system as a single logical node, preserving the semantics of an inelastic application running on a fixed set of resources while providing the performance of an application running on a larger set of resources. The run-

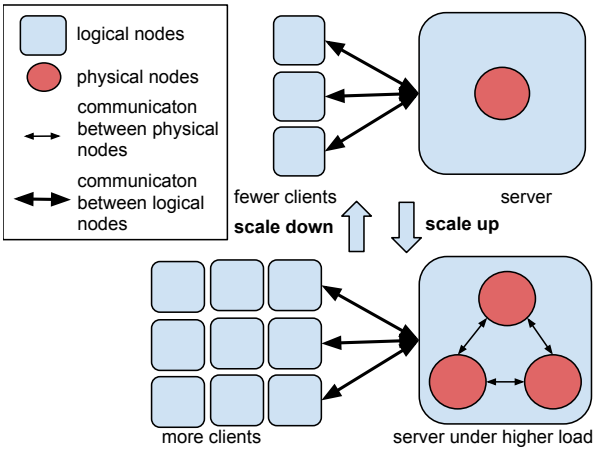


Figure 2: A game server logical node connected by several client nodes. Regardless of scale, clients see only one game server logical node. The actual composition of a logical node at runtime is transparent to the programmers. The programmers can only see at the level of logical nodes.

```
void kidInit(int nKid) {
  Kid[nKid].location = LOCATION_IN_WORLD;
  Kid[nKid].kidDirection = DIRECTION_STATIONARY;
}
```

Figure 3: The code in Mace syntax

time executes the tasks of a single logical node across multiple physical nodes, dispatching events in a particular context to the physical node where that context resides, and can *transparently* and *dynamically* migrate contexts to change the number of physical nodes constituting a logical node, scaling up application resources in response to demands.

Figure 2 illustrates a game server in EVENTWAVE. When running at small scales, all of the state resides on a single physical node. As the number of clients (players) increases, the game state can be dynamically distributed among more physical nodes, increasing the execution resources available to the system. As seen in the figure, the clients still interact with this newly-distributed server as though it were executing on a single node.

We build EVENTWAVE on top of Mace, a toolkit for writing event-driven distributed systems [22], providing context information through annotations. As a result, many applications can support elasticity with straightforward modification, similar to converting from C to C++. As an example, a short code snippet in Mace syntax is listed in Figure 3, and the corresponding code in EVENTWAVE syntax is listed in Figure 4.

After reviewing the key points of event-driven programming, we present the EventWave model, followed by successive levels of detail about our runtime system. We evaluate our runtime system using microbenchmarks and example applications before detailing related work and concluding.

```
[Kid<nKid>] void kidInit(int nKid) {
  location = LOCATION_IN_WORLD;
  kidDirection = DIRECTION_STATIONARY;
}
```

Figure 4: The code with context annotation

2 Event-driven programming

A popular approach to writing distributed applications is the *event-driven* programming model. Conceptually, an application runs on one or more *logical nodes*, which traditionally represent separate physical machines. A logical node's execution consists of processing *events*, which are self-contained units of computation, consisting of a set of method invocations. An event can be triggered by external events such as receipt of messages, or internally by other events.

We adopt the event-driven programming model for EVENTWAVE programs for two reasons. First, event-driven programming is a natural model for writing many distributed programs. Many applications (*e.g.*, the multi-player game server) are designed and described in terms of reacting and responding to messages. The event-driven model is also a good fit for asynchronous distributed applications, such as the multiplayer game, where events are triggered by different external agents but must be handled in a coordinated manner. We also note that traditional task-graph style programs (*e.g.*, Cilk programs) can be reformulated as event-driven programs, with each task represented as a separate event, and "spawns" of new tasks represented as starting a new event. Second, there is already substantial tool support for event-driven programming, including programming languages that make expressing event-based applications easy [22] and development tools such as model checkers to verify event-handling protocols [21]. Building on top of these existing tools, which have been used to implement numerous distributed applications, broadens EVENTWAVE's applicability.

While many event-based programming models use threads for low-level services such as timer scheduling and network I/O, most adhere to an *atomic event model*. In this model, the entire sequence of computational steps necessary to process an event must appear as though it executed atomically and in isolation from other events; the event processing is transactional. Thus, even if an event is triggered while another event is being processed, an application behaves in a *sequentially consistent manner*: the behavior of the application is as if the events were processed one-at-a-time, in the order they were received. This execution model is attractive because it allows programmers to easily reason about the behavior of their applications, even if many events are initiated simultaneously, or events actually run in parallel.

Typical atomic event systems preclude parallelism.

```
auto_types{
  Building{
    map< int , Room > rooms;
    Hallway hallway;
  }
  Room {
    set<int> kidsInRoom;
  }
  Hallway {
    array<array<int> > hallwayMap;
    set<int> kidsInHallway;
  }
  Kid{
    int currentBuilding;
    int currentRoom;
    coordinate coord;
    int kidDirection;
  }
}
state_variables {
  set<int> kidsInWorld;
  map< int , Building > buildings;
  map< int , Kid > kids;
  Building buildings;
  Kid kids;
}
```

Figure 5: State definitions of the example game in Figure 1

Instead, EVENTWAVE dispatches and executes events in parallel provided they are accessing disjoint state (this model is similar to, but richer than, that proposed by Yoo *et al.* based on read/write locking of application state [33]). The challenge, then, is to either detect or discover completely independent events. EVENTWAVE accomplishes this by extending the programming model to capture state isolation, and enhancing the runtime model to enforce event isolation, rather than either predicting (through static analysis) or detecting (through run-time checks) whether the developer correctly isolated events in their application. Section 3.3 discusses EVENTWAVE's approach to parallelism in detail.

3 EVENTWAVE

This section introduces EVENTWAVE, an event-driven, elastic programming model. We introduce the notion of *contexts*, which provide a hierarchical, dynamic partitioning of a node's state and associated operations on those partitioned states. We then describe an *event model*, which constrains how events can interact with contexts. This event model enables a parallel *execution model*, that allows multiple events to run in parallel while preserving atomic event semantics. In Section 4, we describe how this event and execution model can support distributed execution—running the events of a single logical node across multiple physical nodes—a key step in enabling elastic execution.

3.1 Contexts

One key feature of event-driven distributed systems (unlike, *e.g.*, dataflow models) is that logical nodes in the system contain mutable state. For instance, Figure 5 shows the application state definition for the simple game in Figure 1. The state consists of self-contained state variables: players ("kids") running around a game world as well as buildings, rooms and hallways. Some state variables are part of player objects and capture, *e.g.*, the player's information in the game world, such as the player's identifier. Other variables define the world: rooms and hallways reside in a building and several buildings populate the world.

This state can be inspected and modified while responding to an event. Crucially, however, not every event requires accessing all the state in a node; instead, different events may actually access disjoint portions of a node's state. To capture this behavior, state in a EVENTWAVE logical node is organized into *contexts.* A context, as the name suggests, provides the environment in which an event can execute.

As an event executes, it exists in one or more contexts, and the context(s) in which an event executes control which portions of a program's state the event can access. Intuitively, two events that are executing in different contexts are accessing disjoint state, and hence can be executed in parallel (as we elaborate in Section 3.3).

A context in EVENTWAVE consists of a portion of an application's state. For example, in a game server, a "building" in the world might be one context, while a player of the game would have a separate context. Context definitions are analogous to structure definitions: there can be multiple instances of a particular context type (for example, multiple players in a game, or multiple buildings in a game world). To express this, each context has an associated identifier (*e.g.*, a player's user name), which serves to distinguish different instances of the same context type.

Importantly, contexts are *dynamic*. A context is not explicitly instantiated by a programmer, but is instead instantiated when first referenced (by its identifier). For example, as new players enter a game, events are triggered with new player user names, creating the contexts in which those events execute. Intuitively, one can think of a map, for each context type, between context ids and contexts; when a particular context id is referenced, if the context exists in the map, it is used, otherwise a new context is created and added to the map. This approach to instantiation naturally fits the expected behavior of many event-driven programs (*e.g.*, in a distributed hash table, creating new contexts as files are added or as new peers join). Figure 6 shows the state definition in Figure 5 translated into contexts. Each object in the original state is naturally translated into a context. For instance, the Room object becomes the Room context.

A hierarchy of contexts To this point, we have assumed that contexts in EVENTWAVE programs are "flat": there is no relationship between different contexts, and all contexts are independent. In reality pieces

```
set<int> kidsInWorld;
context Building <int buildingID> {
  context Room <int roomID> {
    set<int> kidsInRoom;
  }
  context Hallway {
    array<array<int>> hallwayMap;
    set<int> kidsInHallway;
  }
}
context Kid <int kidID> {
  int currentBuilding;
  int currentRoom;
  coordinate coord;
  int kidDirection;
}
```

Figure 6: Context definition of the game in figure1

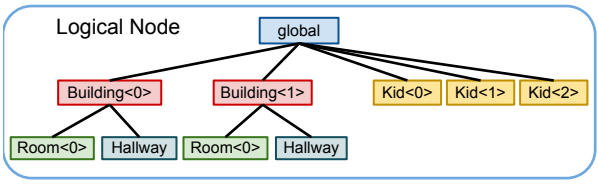


Figure 7: Sample context hierarchy

of state in an application have *hierarchical* relationships: one context can contain other, sub-contexts. All explicitly declared contexts have a *parent* context. By default the parent context is global, but if the context is explicitly declared inside another context, the former's parent is the latter. For example, in Figure 6, the room and hallway contexts have a building context as their parent.

Figure 7 shows the context hierarchy for the example in Figure 6, with two buildings, rooms, hallways and multiple players. To name a specific context in the hierarchy, we use double colon as the connector. For example, the first room in the first building is labeled as Building⟨0⟩::Room⟨0⟩.

EVENTWAVE currently supports applications with one-to-many relation between contexts. Our future work will support applications with more complex interactions between contexts. For example, contexts may form a lattice, which can be used to represent many-to-many relationships: a building has many departments, and a department may be housed in many buildings.

3.2 Event model

As discussed in Section 2, event-driven programs can be thought of as a series of atomic events executing in response to various stimuli. EVENTWAVE supports multiple events executing simultaneously while preserving sequential, atomic event semantics (see Section 3.3). To accomplish this, the EVENTWAVE *event model* uses context information to restrict what events can do.

3.2.1 Context methods

An event in EVENTWAVE executes a series of methods. Each method m is associated with a context, c. We will write such methods as $m[c]$. A method $m[c]$ can *read and write* state associated with c, but cannot access state associated with any other context.

Recall that contexts are identified by a context type *and* by a unique identifier. Methods are not associated with context types, but specific contexts. The binding of methods to specific contexts occurs at run time: the context annotation takes the form C⟨x⟩, where x refers to a method argument. When the method is invoked, the value of this argument is used to bind the method to a particular context. Hence, the signature for a `reportLocation` method for a particular player *p_id* invoked in the Kid⟨*p_id*⟩ context would be:

```
[Kid<p_id>]reportLocation(int p_id)
```

3.2.2 Event execution

When an event starts, it invokes a *single* context method, $m[c]$—all other methods must be invoked from $m[c]$. This first context method is called a *transition* which is also known as the event handler.

To read and write state in context c, the event must "acquire" c. This acquisition occurs implicitly by invoking methods in context c. An event may acquire multiple contexts by executing multiple methods, called *routines*. Acquiring multiple contexts within a single event ensures that a consistent state is seen across the contexts. Figure 8 shows an example code in EVENTWAVE language. A message delivery transition triggers a new event at a Building context which calls routines synchronously to relocate a player from a room to the hallway. If the routines in the figure were implemented as independent events, consistency would not be guaranteed (*e.g.*, the player could be in both the room and the hallway simultaneously).

Crucially, events cannot acquire contexts at will. If two events e_1 and e_2, with e_1 logically earlier than e_2, access c in the opposite order, e_2's execution violates sequential semantics. Furthermore, if an event accesses multiple contexts, its operations across those contexts must appear atomic. While these problems could be addressed by requiring that only one event access the context hierarchy at a time, such a restriction precludes parallelism. Instead, we place several restrictions on events' behaviors that enable a parallel execution model that preserves sequential semantics (see Section 3.3).

If an event accesses multiple contexts, it must acquire them in a particular order. In particular, to access a context c, an event must either start by accessing c, or must have acquired a context higher than c in the context hierarchy. Because the context hierarchy creates a partial order of events, this access rule means that an event starts at a particular point in the hierarchy, and the set of contexts it can write to "grows" down in the hierarchy.

If an event no longer requires *write* access to a context, it can call a special *downgrade* method to release

```
transitions {
  [Building<r.nBuilding>]
    deliver(from, to, RelocateRequest r){
      downgrade();
      moveOut( r.kidID, r.nRoom, r.nBuilding);
      moveToHallWay( r.kidID, r.nBuilding);
    }
}
routines {
  [Building<nBuilding>::Room<nRoom>]
    bool moveOut(int kidID, int nRoom, int nBuilding){...}
  [Building<nBuilding>::Hallway]
    bool moveToHallway(int kidID, int nBuilding){...}
}
```

Figure 8: A code snippet written in EVENTWAVE syntax for handling client requests

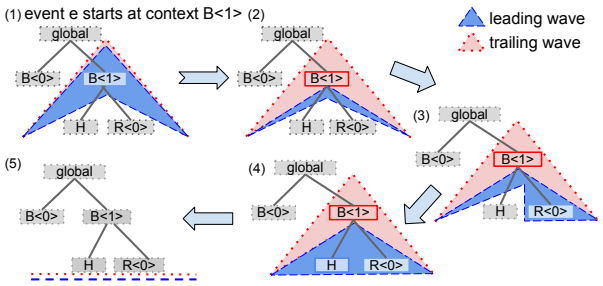


Figure 9: Event waves and execution transitions of an event e. Contexts B⟨0⟩/B⟨1⟩ are abbreviation for Building⟨0⟩ and Building⟨1⟩,respectively. The context H is short for Building⟨1⟩::Hallway. The context R⟨0⟩ is short for Building⟨1⟩::Room⟨0⟩

access to the context. However, once an event releases write access to a context, it cannot reacquire access to that context (though it may still read its state). A context c cannot be downgraded unless the event has already released access to all of c's ancestors in the hierarchy. Absent downgrades, an event only releases access to its contexts when it completes.

We note the similarity of the context access restrictions to two-phase locking approaches for isolation. In the absence of downgrades, acquiring contexts is analogous to strict two-phase locking [5], while the addition of downgrades produces a protocol analogous to tree locking [2]. As in two-phase locking, the ordering imposed on the acquisition of contexts by the context hierarchy helps prevent deadlock (as we shall see in the next section). Downgrades complicate the model somewhat, but the ordering on contexts still suffices to enable safe, parallel execution.

Intuitively, the execution of an event can be reasoned about in terms of two *event waves*: the *leading* wave and *trailing* wave. The region occupied by the leading wave corresponds to the contexts that the event has write access to; the region occupied by the trailing wave corresponds to the contexts that the event has downgraded, but can still read. The trailing wave can not pass by the leading wave, as that would violate the model.

Figure 9 plots an event executing the methods in Figure 8, and shows how context acquires and downgrades affect the event waves. When an event starts in the deliver method (as shown in step ①), it only has write access to context Building⟨1⟩. By downgrading the context, the trailing wave moves down in step ②. It then calls moveOut() method synchronously to acquire the context Building⟨1⟩::Room⟨0⟩, moving the leading wave down (shown in step ③). Subsequently, it calls moveToHallway() synchronously to acquire the context Building⟨1⟩::Hallway (step ④) Finally, the event releases all contexts in step ⑤ when it finishes.

Events can create new events by calling methods *asynchronously*. An asynchronous method invocation, conceptually, is identical to starting a new event and immediately calling the target of the asynchronous invocation. The new event is a separate unit of computation from its parent event, and only owns the context(s) associated with the initiating method. Asynchronous methods are roughly analogous to asynchronous calls such as "spawns" in Cilk [6] or "async" in X10 [9]; unlike in those languages, however, events triggered by asynchronous methods in EVENTWAVE are fully decoupled from their parent event after the parent event commits.

3.3 Parallel execution model

The EVENTWAVE context and event model is coupled with a parallel execution model that constrains the collective behavior of multiple events executing simultaneously. When an event is initiated (*e.g.*, the message triggering it is received, or the asynchronous method initiating it is called), it is given a logical, monotonically-increasing timestamp that orders it with respect to all other events in the system (in the case of a distributed program, with respect to all other events on a particular node). EVENTWAVE events can execute in parallel as long as their behavior corresponds to the atomic event model: the behavior of each event should be consistent with an execution where the events executed sequentially according to their timestamp order. In other words, events appear to execute atomically, in isolation, and in timestamp order.

Because contexts are self-contained collections of state and code, an event executing in one context cannot affect the behavior of a different event executing in another context. Hence, EVENTWAVE allows events to execute in parallel as long as any context is held by at most one event at a time.

In terms of the waves of events, consider events e and e', where e' has a later timestamp than e. For basic correctness and limited parallelism, the leading wave for e' must always be above the trailing wave for e, as event e can *read* the contexts in the trailing wave, and must not read any modifications made by e'.

To improve parallelism, we note that once an event

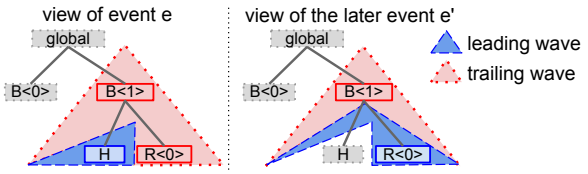


Figure 10: Simultaneous view of two events e and e' in an execution that adheres to EVENTWAVE.

e downgrades from a context, moving down its trailing wave front, it only needs read access to the context. Hence, the EVENTWAVE runtime can take a snapshot of the current state of the downgraded context before moving down the wave front. This is similar to snapshot isolation in databses [3]. The difference is that snapshot isolation reads the committed data when the transaction starts, but in EVENTWAVE, a snapshot of a context is taken when the event releases the lock. The event model ensures that once the trailing wave front moves down, the event will never modify the context. Hence it is safe for a later event, e', to move its own leading wave front below the context—i.e., to begin accessing the context. Future reads of the context by *e* will read from the snapshot, preserving isolation. Effectively, this relaxes the restrictions on events so that now an event's leading wave must only remain above earlier events' leading wave, rather than above the trailing wave. Figure 10 illustrates two concurrent events which obey the model.

Finally, to preserve timestamp order, an event that has completed execution cannot *commit* until all events with earlier timestamps have committed. In particular, any new events triggered by an event's execution (*e.g.*, sent messages, asynchronous calls) are delayed until after the event commits. Note that this property means that an event triggered by an asynchronous call logically starts and completes *after* its parent event finishes.

A simple strategy to ensure sequential semantics is to force all events to start at the global context, and make their way down through the context hierarchy to perform their operations. Because of the restrictions on events' wave fronts, later events will remain "above" earlier events in the hierarchy, and events will appear to execute in sequential order. Because EVENTWAVE allows events to be initiated at deeper contexts in the hierarchy, when an event starts at context *c*, a "dummy" event begins at the global context and implicitly attempts to acquire the global context, then acquire lower-level contexts and immediately downgrade them until *c* is reached. It is clear that this strategy introduces some scalability issues (every event must essentially access every context). Section 4.3 presents a mechanism to mitigate this issue.

The next section discusses how the execution model outlined above can be satisfied in a runtime to distribute a logical node over multiple physical nodes and/or parallel-executing threads while still preserving the illusion that a logical node is running on a single physical node with one thread. Section 5 discusses how the same mechanism can be used to support migration of contexts during elastic execution.

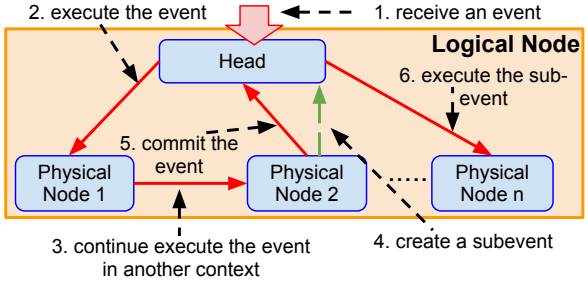


Figure 11: Events processed by logical node distributed over multiple physical nodes

4 Distributed execution model and run-time

This section presents the EVENTWAVE distributed runtime system, which implements the execution model described in Section 3.3.

When running a EVENTWAVE application, the straightforward approach is to request one physical node per logical node. Execution can proceed as in the standard, inelastic event-driven model. If the physical node has multiple threads, the EVENTWAVE runtime can take advantage of events in disjoint contexts to provide additional throughput. If, however, the load on the system increases beyond the capability of a single physical node to process them, the EVENTWAVE distributed runtime system supports distributing the execution of a single logical node across multiple physical nodes. In other words, an application written to run on n logical nodes can be run on m physical nodes, where $m > n$, providing additional throughput. When combined with dynamic migration (Section 5), which supports changing m over the course of execution, the runtime enables elastic execution of distributed applications.

For the remainder of this section, we will assume that the EVENTWAVE application is written for a single logical node; the mechanisms easily generalize to applications with multiple logical nodes.

4.1 Overview

A high level overview of the EVENTWAVE distributed runtime system, and how it processes events, is shown in Figure 11. The execution of a single logical node is distributed across multiple physical nodes. A single *head* node serves as the representative of the logical node: all communication with the outside world is intermediated by the head node (*e.g.*, all messages sent to this logical node are routed to the head node, and all messages sent by this logical node are sent from the head node).

All events processed by the logical node are initiated by the head node. Upon receiving an event (①), the head node assigns the event a timestamp and signals a worker node to process the event (②). Note that deciding which worker node gets an event is discussed in the next section. As the event executes, the worker may "pass" the event to a different worker to continue execution (③). If the event makes an asynchronous call, this event is enqueued at the head node (④). Once the event's execution is complete, the worker currently handling the event signals the head node that the event is ready to commit (⑤). The head, having a global view of the logical node's execution, can ensure that all events will commit in timestamp order. As in the basic parallel execution model, once an event commits, any new events it may have triggered are initiated. In particular, any asynchronous methods called by the event will now begin, starting from the head node as with other events (⑥).

Requiring a single physical node to mediate all the operations of the logical node introduces a bottleneck: the throughput of the logical node is constrained by how efficiently the head node can manage execution. We note that most of the work performed by the head node is bookkeeping, while most computation will be performed by other physical nodes, somewhat mitigating this drawback. A runtime system that allows the head node's execution to be distributed across multiple physical nodes is an important topic for future work.

4.2 Context-based distribution

When a head node begins processing an event, how does it determine which worker node to send the event to? There are many issues to consider: locality (events should be assigned to workers containing the state they must access), communication (events should be distributed to minimize their transfer between worker nodes), and load balance (worker nodes should perform similar amounts of computation).

To address these issues, EVENTWAVE uses *context-based* distribution. The global context is mapped to the head node, but other contexts are mapped to worker nodes. When an event needs to execute in a particular context, it is sent to the worker node to which that context is mapped. If an event accesses multiple contexts during its execution, its execution will be split among multiple worker nodes, being passed back and forth according to the context it is currently in.

Because contexts capture both a portion of the program's data and the computations that act on them, this mapping strategy naturally accomplishes several of the goals laid out above. Locality is achieved by the data-centric nature of the mapping. Rather than a task-based mapping, where an event is assigned to a worker node, and the data may need to be accessed remotely,

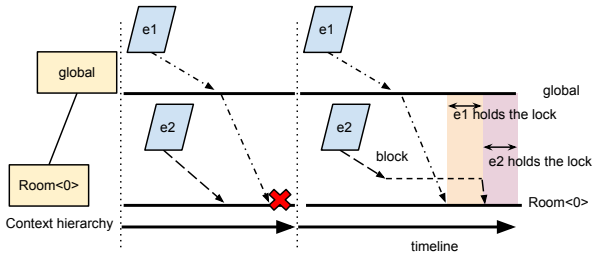


Figure 12: Left: e_1 acquires the context synchronously after e_2, and violates the model. Right: e_2 waits for e_1. e_2 acquires the lock successfully after e_1 downgrades.

in EVENTWAVE, *events are sent to the data*. Hence an event *always* accesses only local data.

Context mapping The mapping of contexts to nodes affects communication and load balance. A simple heuristic for reducing communication cost is by considering the hierarchical nature of contexts: an event in a context c is more likely to also interact with a context c' that is a descendant of c. Hence, c and c' should be mapped to the same physical node.

If the distribution of events visiting each context is known *a priori*, then contexts can also be mapped to nodes to achieve load balance. However, specific heuristics for mapping are beyond the scope of this paper.

4.3 Parallel execution of distributed events

Once the mapping of contexts to physical nodes is accomplished, the final step is to ensure that the atomic event model is preserved by distributed execution. While the head node dispatches multiple events to worker nodes simultaneously, sequential commit order is guaranteed by the mediation of the head node in every event's execution. Because events return to the head node before committing, the head node can delay an event's completion until all earlier timestamped events complete. The trick, then, is to ensure that the parallel execution of (ordered) events preserves the atomic event model. In particular, we must ensure that (a) only one event is in a context at a time; and (b) events access contexts in an order consistent with their timestamp order.

Accomplishing the first goal is straightforward. Every context has a lock associated with it. When an event enters a context by calling a method associated with that context, it must first acquire the lock on that context. That lock is held until the event either completes or explicitly downgrades from that context.

More challenging is ensuring that events acquire contexts according to their timestamp order. In particular, given events e_1 and e_2, with e_1 logically *earlier* than e_2, for any context c that both e_1 and e_2 want to access, e_1 must acquire a lock on c before e_2; e_2 cannot access c until e_1 releases c. While it may seem that the hierarchical nature of contexts enforces this ordering (there seems to be no way for e_2 to get to c before e_1 does without

"passing" e_1 in the context hierarchy), we note that an event can *start* lower in the hierarchy as mentioned in Section 3.3. Considering the context hierarchy from Figure 7, e_1 could be executing in the global context, while a second event e_2 *begins* in context Room ⟨ 0 ⟩. If e_1 then wants to descend to Room ⟨ 0 ⟩, it would arrive at the context after e_2 and violate the atomic event model, as illustrated in Figure 12.

As described in Section 3.3, this problem can be solved using dummy events, but which clearly introduces scalability issues, as every event must touch every context. To avoid this, dummy events are not eagerly propagated through the context hierarchy, but are instead "batched" and propagated through the tree in a group.

To implement this strategy, each context has a *ticket booth* that contains an integer counter that indicates which events (identified by timestamp) have accessed the context already. Hence, the current value of the ticket booth represents the next event the context "expects" to see. When an event e starts in context c, a dummy event with the same timestamp is issued to the global context. If the global context does not yet expect the dummy event, it is enqueued at the global context. When an event that the global context expects begins, it not only moves through the context tree itself, but carries with it all dummy events whose timestamps immediately follow it, and increments the ticket booth at the global context appropriately. For example, suppose the global context expects an event e with timestamp 7, while dummy events with timestamps 8, 9 and 11 are enqueued. When e executes, it propagates the events with timestamps 8 and 9, but not the one with timestamp 11. The ticket booth at the global context now expects timestamp 10.

The same procedure is used at all contexts. If a group of dummy events, with timestamps $x+1 \ldots x+k \ldots$ batched with an event e (with timestamp x) reach a context c where an actual event e' with timestamp $x+k$ is waiting, the group is split, with dummy events $x+1 \ldots x+k-1$ continuing through the context hierarchy with e, while dummy event $x+k$ terminates, and events $x+k+1 \ldots$ are now batched with the actual event e', which can begin executing once e releases access to c.

5 Dynamic migration

With the distributed run time system described in the previous section, EVENTWAVE programs written for n logical nodes can run on m physical nodes. However, the ability to run a distributed program across additional nodes is only one part of the elasticity story. Because the goal of elastic programs is to increase their throughput in response to demand, and throughput is increased by running a program on additional logical nodes, we must have a mechanism for *dynamically* changing the number of physical nodes that a logical node is running on, migrating computation and state to new physical nodes. Furthermore, this migration should be *transparent*. The application must be able to change its physical footprint without affecting any users; throughput and responsiveness might vary during migration, but users should not observe any difference beyond increased latency. Finally, the developer's programming model and mental model of the application should not have to explicitly account for migration. By supporting dynamic, transparent migration in the existing EVENTWAVE programming model, the runtime provides the infrastructure necessary to develop elastic applications.

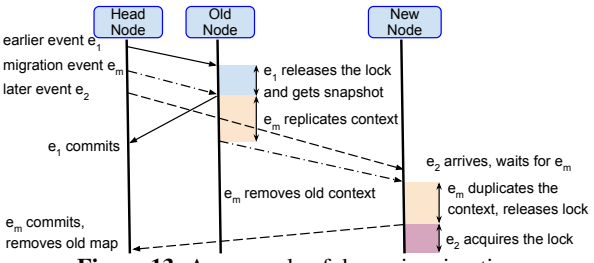


Figure 13: An example of dynamic migration

The basic approach that EVENTWAVE takes to elasticity is to support *dynamic migration* of contexts. Namely, during execution, a context's physical location can move. Because we want this migration to occur transparently, there are a couple of problems that must be addressed. First, because events execute within contexts, the distributed runtime must account for migration correctly to ensure that events are forwarded to the appropriate location. Second, because a migration might occur *while an event has access to a context*, the runtime must make sure that events can safely deal with a context that might move during their execution.

Migration algorithm

To achieve dynamic, transparent migration of a context to a new physical node, the runtime must do two things: (i) replicate the state of the context to the new node; and (ii) update the context-node mapping to reflect the new context location. The first task is a prerequisite to migration: because the goal is to move computation to another physical node, the data associated with that computation must be moved as well. The second task is necessary to ensure that future events that access the context are routed to the correct physical node.

Figure 13 shows the execution of dynamic migration. A migration creates a special "migration event," When a migration request is issued, the head node creates a special "migration event." It is inserted into the same event queue as normal application-generated events. Conceptually, all events *before* the migration event will use the old context mapping, while all events *after* the migration event will see the new context mapping.

The migration event changes the mapping of contexts

to physical nodes, but instead of updating the context mapping immediately, the runtime system keeps several versions of the mapping at the same time, so that every event before the migration event will use the old mapping, and the ones after the migration will use the new version. By keeping multiple version of mapping, it avoids the pitfall mentioned above: events that already have access to the context will not observe the mapping, and hence migration will be transparent. After the new context map is created, it is sent to every physical node. Just as any other event, the migration event accesses the context to be migrated, acquiring a lock on it. Once it gains access to the context (ensuring that *earlier* events are done modifying the context), it replicates the state of the context and transfers it to the new physical node.

After transferring state, the migration event removes the old context. Note that this is safe: the runtime will ensure that events later than the migration event will read from the new maps, and hence those events will access the replicated version of the context on the new physical node. *Earlier* events have already released the migrated context and taken a snapshot. Thus, even though the context is now at a new physical node, those earlier events will simply read from their snapshots and continue without interruption. Finally, when the migration event commits, it removes the old mapping. At this point, all events that may have needed the old mapping will have committed, so it is no longer necessary.

With the mechanism sketched above, a critical challenge is to develop the *policy* for migration. We discuss application-specific policies in Section 7, but a general policy is beyond the scope of this paper.

6 Fault Tolerance

One pressing problem that arises in elastic programs is fault tolerance. If a program initially run on n physical nodes is expanded to run on $2n$ physical nodes, the likelihood of a node failure increases commensurately. If the failure of a single node were able to bring down the entire program, elasticity would result in more failure-prone applications. A valuable property for an elastic system to have is that the failure probability of an application remains fixed, regardless of how many physical nodes an application's logical nodes are distributed over.

Our current implementation of the EVENTWAVE model does not provide this guarantee. However, in earlier work, we described how existing EVENTWAVE mechanisms can be readily repurposed to guard against physical node failures and integrated with existing systems to recover from logical node failures [10]. We briefly summarize this approach here for completeness.

Recall that our runtime system implements three features to support distributed, parallel execution of events: (i) events execute transactionally and in sequential order to preserve atomic-event semantics; (ii) events take snapshots of contexts as they execute; and (iii) all externally-visible communication is handled by the "head" node, with incoming messages triggering events, and outgoing messages deferred until event commits. These features directly enable adding physical-node fault tolerance with little change to the runtime, by implementing a basic checkpoint-and-rollback system: When an event commits at the head node, the context snapshots associated with the event are used to create checkpoints. If a physical node fails, the contexts on the failed node are rolled back using the commited snapshot.

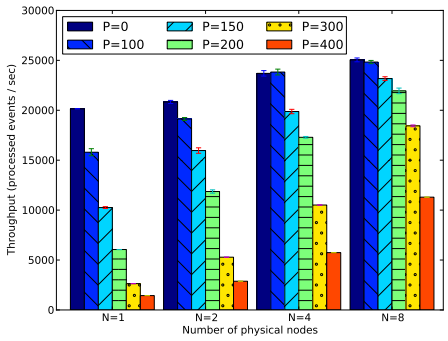
The head node must also be fault-tolerant, since it provides necessary coordination; if it fails, the logical node will fail. Since the head node looks like the entire logical node to the rest of the system, any application- or logical-node-level fault tolerance approaches, can be used to mask the head node failure. MaceKen [32] is a Mace extension integrating the Ken reliability protocol which can potentially mask head node failure. It records committed events to its local persistent storage. When a head node fails, it is simply restarted and its state is restored using the local persistent storage. The state of the entire logical node is thus rolled back to the last committed event. The Ken protocol guarantees that external observers cannot distinguish a restarted logical node from a slowly-responding logical node: an external message is explicitly acknowledged, and any unacknowledged messages are retransmitted until being acknowledged.

7 Evaluation

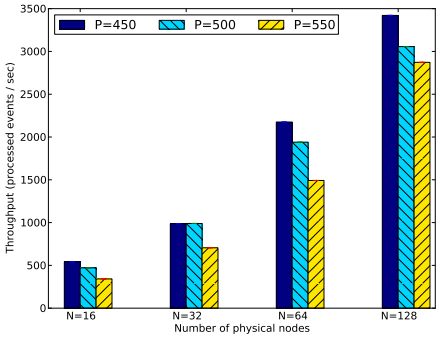
We implemented EVENTWAVE as an extension of the Mace runtime [22] by adding 15,000 lines of code written in C++. We also modified the Mace compiler to parse the context definitions and annotations, and then translate them into runtime API calls. The modification added 5,000 lines of Perl code.

We note that the EVENTWAVE language eases development effort. As one example, *the elastic key-value store application described below was written in less than 20 lines of code, while an equivalent C++ implementation is about 6,000 LOC*; the EVENTWAVE programming model allows programmers to concentrate on application semantics, leaving the complexity of distribution, migration and elasticity to the runtime.

As a new programming model for elastic applications, there are no standard benchmarks against which to evaluate EVENTWAVE. Instead, we evaluate EVENTWAVE in two phases. First, we use a synthetic microbenchmark that allows us to vary the number and size of independent contexts, to evaluate the performance of the EVENTWAVE system under various scenarios. We then use two application case studies, a key-value store and a game server, to study EVENTWAVE's ability to use elas-

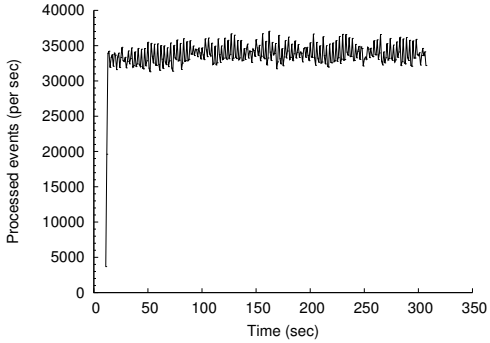


(a) Smaller scale from 1 node to 8 nodes

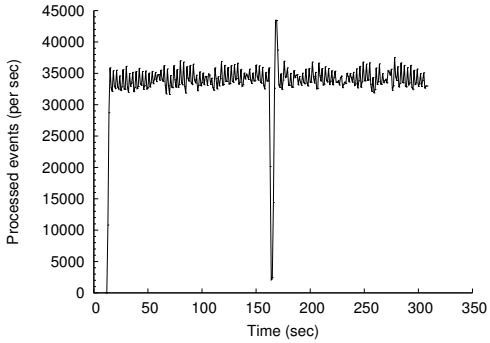


(b) Larger scale microbenchmark from 16 nodes up to 128 nodes

Figure 14: Microbenchmark throughput



(a) context size=0MB



(b) context size=100MB

Figure 15: Throughput change before/after migration at time 160.

ticity to maintain performance under dynamic load. The result of microbenchmark and the key-value store experiments was obtained using our lab cluster. Each node has eight 2.33 Ghz cores Intel Xeon and 8GB RAM connected to 1Gbps ethernet.

7.1 Microbenchmark

We evaluate three different aspects of EVENTWAVE with a microbenchmark. The microbenchmark consists of a generator that produces a series of events that each choose one of 160 independent contexts in a round-robin fashion and perform a specified amount of work.

Throughput Figure 14a plots the throughput of the microbenchmark (events processed per second) with different amounts of work (P) and on different numbers of physical nodes (N). $P = 0$, implies no real work in the event, and the throughput effectively measures the maximum throughput the EVENTWAVE runtime supports. As the number of physical nodes increases, throughput does not drop; in fact, it increases as even with no work to be performed, processing an event still has some parallelizable components. When P is increased, each event does more work, so the overall throughput of the system drops, as seen when $N = 1$. Increasing the number of physical nodes increases the computational resources available to the application and hence recovers the lost performance, until the application is once again running at maximum performance. As the amount of work increases, it takes more physical nodes to recover maximum performance, but the overall trend is stable. We also show microbenchmark result for larger scale up to 128 nodes using larger workload in Figure 14b.

We can draw two conclusions: (i) the maximum throughput of EVENTWAVE applications does not drop as a logical node is spread over more physical nodes; and (ii) EVENTWAVE is able to effectively harness the computational resources of multiple physical nodes to maintain performance under higher levels of load.

Migration Overhead Figure 15 plots the overhead of migrating a single context from one physical node to another at time 160. When the context is small (Figure 15a), migration has no impact on average throughput. When it is large (Figure 15b), the migration event must serialize the context, send it to the destination physical node and deserialize it. Even though other events accessing different contexts continue to execute in parallel with the migration event, they cannot commit until the migration event does. Hence, events back up behind the migration event, leading to a transient drop in throughput during migration. Once the migration event commits, throughput temporarily exceeds the long-run

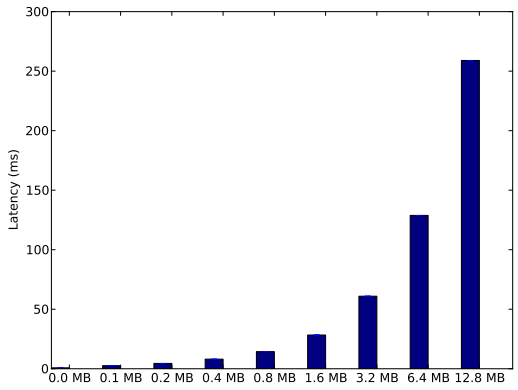


Figure 16: Migration latency of different context size

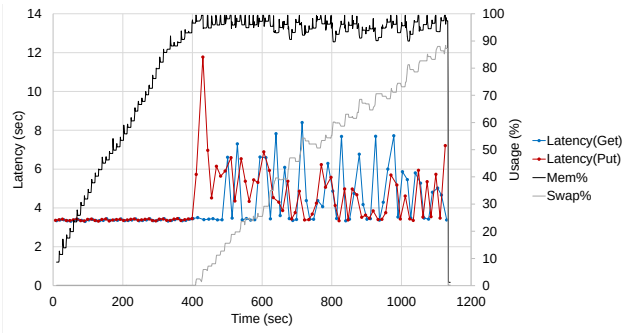
average throughput as all of the events that backed up can be quickly committed. In both cases, migration does not have a long-term impact on throughput.

Migration Latency Figure 16 plots the latency of migration events corresponding to different context sizes. Migration latency is proportional to the serialized size of context, and is largely determined by network throughput and the speed of serialization/deserialization. Since migration does not require global synchronization, it is fast. For contexts of size less than 1MB, the latency is negligible. We note that virtual machine live migration protocols such as VNSnap [20] may also be applied to reduce service disruption. Interestingly, context-based application state distribution may help live migration protocols because a context represents a fine-grained chunk of application state that can be migrated independently of other contexts.

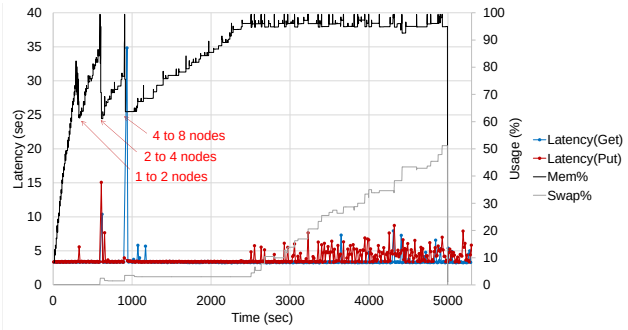
7.2 Elastic Key-Value Store

We implemented an elastic key-value store implementation shown in Figure 18 using EVENTWAVE to evaluate its performance as an application-level benchmark. Note that we claim no novelty of the key-value store. Instead, with the programming model and runtime support described, we show that it is easy to write an elastic key-value store using less than 20 lines of code, whereas an equivalent C++ implementation is about 6,000 LOC. This evaluation demonstrates how elasticity can greatly help a memory-constrained system. The application consists of two logical nodes. One is the client (1 node) and the other is the server. On the server side, one physical node is used as the head node while one or more physical nodes hold the key-value pairs within their memory. The keys are grouped into buckets using a hash function, and each bucket gets its own context; in this way, operations on independent buckets proceed in parallel.

The experiment examines the behavior of performing a series of puts/gets on the key-value store. As the experiment runs and more puts are performed, the size of the key-value store increases. As a result, the nodes maintaining the store eventually exceed their physical memory capacity, and swapping occurs. Figures 17a and 17b plot the latency of put and get requests (measured by the client) over time and physical memory usage. In Figure 17a, an inelastic configuration is used: the key-value store is kept on a single physical node, and once swapping begins, the latency of operations unsurprisingly becomes more variable, and on average much higher.



(a) Single node key-value store without migration



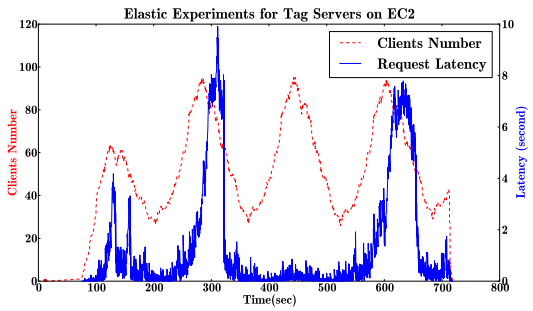
(b) Single node key-value store with migration

Figure 17: Key-value store

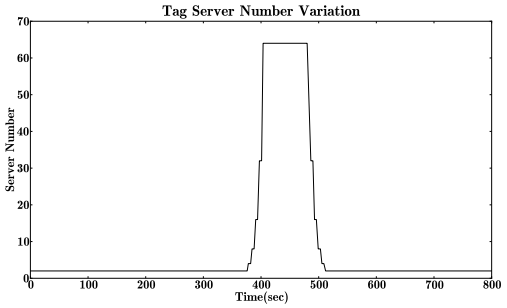
```
context Bucket<int> {map<string, string> kvmap;}
messages {
 Get { string k; }
 GetReply { string k; string v; }
 Put { string k; string v; }
}
transitions {
  [Bucket<hash(msg.k)>] deliver(src, dest, Get& msg) {
    route(src,GetReply(msg.k,kvmap[msg.k]));
  }
  [Bucket<hash(msg.k)>] deliver(src, dest, Put& msg) {
    kvmap[ msg.k ] = msg.v;
  }
}
```

Figure 18: An elastic key-value store implementation

Figure 17b demonstrates the power of EVENTWAVE's elasticity. Because different buckets in the key-value store are different contexts, the EVENTWAVE runtime can easily migrate some buckets to other physical nodes, dynamically expanding the total amount of physical memory available to the application. We implement a simple migration policy: when the system's physical



(a) Latency and # of clients over time



(b) # of physical nodes used by server over time

Figure 19: Performance of game server with and without elasticity.

memory usage exceeds 80%, the number of physical nodes is doubled and each existing node migrates half its contexts to a new node. The figure demonstrates three such migration events, at which point the application is using eight physical nodes. We note several points: (i) the sequential commit policy of EVENTWAVE means that some events that get "trapped" behind a migration event exhibit much longer latency, as they must wait for the migration to complete before committing; (ii) after migration, physical memory usage drops commensurately; (iii) even as the key-value store is spread over more physical nodes, the latency of operations does not increase; and (iv) the elastic application is able to avoid paging, and hence sustain low latencies for much longer than the inelastic version.

7.3 Multiplayer game server

Our second case study concerns the ability to elastically adapt to changing loads of the EVENTWAVE multiplayer game example of Section 3. The game is implemented as a single logical node representing the game server, and multiple logical nodes representing the clients. The game state is organized into contexts as in Figure 6. We deploy 128 clients over 16 EC2 Small Instances to generate workload. To balance the cost and latency, the server head node resides on an Extra Large instance, but physical nodes are Small Instances.

We generate an artificial "load" for the game server in the form of players that are randomly distributed among the buildings, hallways and rooms, and move randomly around the world. We simulate real-world gamer behavior by using a Gaussian distribution to have clients randomly join and leave the game server. Figure 19a shows the number of connected clients at a given time; the number of connected players varies from 0 to around 90, in a periodic pattern. We evaluate four full periods of this behavior, measuring the average latency experienced by clients as they attempt to move around the game world.

Figure 19b shows the number of physical nodes used by the server. For the first, second and the fourth period, the elasticity mechanism is not activated, and the server uses but a single physical node. As Figure 19a demonstrates, the average latency experienced by the clients increases and decreases with the load. At high loads, the server is unable to process client requests fast enough, and average latency increases dramatically.

For the third period, we use a simple elasticity policy. If the number of clients exceeds some threshold, the server doubles the number of physical nodes used, migrating half its contexts to the new physical nodes. This process continues up to a maximum of 64 physical nodes. If the number of clients drops below a certain level, elasticity is exploited in the opposite direction, and the number of physical nodes is decreased. As Figure 19b demonstrates, the number of servers used therefore varies proportional to the load of the system. As expected, this dynamic migration provides more computational resources to the server rapidly, and as a consequence, the average latency experienced by clients in the third period is much reduced. Similarly, when the connected clients leaves, the application scales down and releases extra nodes. With this elasticity support, we could guarantee low request latency as well as minimizing the resource usage under dynamic workload.

8 Related Work

Parallel event execution Recent research on automated sequential execution has been moving towards the systems where programmers specify parallelism to allow the compiler take advantage of it. One notable example is Bamboo [34]. This approach enables parallelism on shared memory multicore systems by utilizing object parameter guards and then compile the language into locks. It is similar to our EVENTWAVE programming model and runtime implementation in that both provides parallelism annotations that direct the compiler to generate an appropriate locking scheme to support parallel execution of events. However, unlike EVENTWAVE, Bamboo does not support distributed or elastic execution.

Computation offloading One line of research with similar goals to EVENTWAVE is *computation offloading*, where an application is partitioned between a client

and server (or multiple servers) to improve performance [16, 23, 26, 29, 31]. These approaches are similar to EVENTWAVE's distribution of a single logical node's computation across multiple physical nodes. However, there are key differences. Some of these approaches use static partitioning—the program is analyzed at compile time and a fixed partition across client and server is computed—and hence cannot provide dynamic elasticity. Others perform dynamic partitioning, allowing them to respond to changing environments and load. However, these approaches mostly target applications either with a single thread of control, or where the programmer has explicitly added parallelism and synchronization. EVENTWAVE aims to supports parallelism while allowing programmers to reason sequentially.

Pilot job frameworks *Pilot job* systems support the execution of a set of tasks on an elastic set of computational resources [7, 11, 15, 25, 27, 30]. The underlying commonality of these systems is that an application must be broken up into a set of isolated tasks. These tasks can be organized either as a "bag of tasks," where the tasks can execute independently in any order [7, 15, 25], or as a DAG of tasks, where the completion of one (or more) tasks enables the execution of later tasks [11, 27, 30]. These models are fundamentally more restrictive than EVENTWAVE's: while tasks are roughly analogous to EVENTWAVE's events, tasks have very limited interaction and cannot communicate with one another, while events can be organized in arbitrary ways, and can communicate through context state. Furthermore, pilot job frameworks use, effectively, a computation-centric approach to elasticity, where tasks are the basic unit of distribution, and adding resources affects how tasks are distributed. EVENTWAVE, in contrast, uses a data-centric approach to elasticity, where state is the basic unity of distribution. This facilitates state-based interactions between events and also leads to better locality.

Actor model The Actor Model is the basis for several systems [18, 28]. Actors are collections of state and code that communicate via message passing, with each actor behaving atomically. There is a clear connection between an actor and a context: the actor has implicit parallelism because each entity is independent of each other. Much like EVENTWAVE, Actors adopt a data-centric approach to distribution, with computation being co-located with its associated data. The primary difference between the two models is that EVENTWAVE provides event atomicity across multiple contexts, rather than treating contexts as independent entities.

Orleans extends the Actor Model to allow transactional execution across multiple actors and to support elasticity [8]. However, the elasticity model of Orleans is different from EVENTWAVE. In EVENTWAVE, elasticity is achieved by partitioning state across different resources, while Orleans achieves elasticity through state replication, allowing parallel execution of the same actor at multiple physical nodes. Orleans' programming model trades off flexibility for consistency: Orleans' transactional events have no restrictions on their execution, unlike EVENTWAVE's event model. However, Orleans' replication-based approach to elasticity does not provide sequential consistency. We note, however, that EVENTWAVE may be complementary to Orleans. An Orleans actor could be implemented using EVENTWAVE, allowing the use of EVENTWAVE's elasticity mechanism and hence providing stronger semantics.

Scalable databases Scalable databases, notably ElasTras [12], MegaStore [1] and Cloud SQL Server [4] typically employ two-level structure to partition the data into shards. ACID properties is guaranteed inside a shard.

At high level, EVENTWAVE and these systems use the same basic design principles: state (or data) is partitioned and hosted by a set of nodes. The *global partition manager* in Cloud SQL Server and the *dynamic partitioning* mechanism of ElasTras are both similar in principle to how EVENTWAVE maps contexts to nodes, but both approaches aim at failure recovery.

Live migration Both virtual machine live migration and database live migration [14] aim at redistributing the state of a distributed system. At high level, their redistribution is similar to EVENTWAVE's migration of context state. However, these systems solve a different problem, attempting to achieve load balance in multi-tenant environments, rather than providing elasticity.

9 Conclusions

Developing elastic cloud applications that can dynamically scale is hard, because the elasticity complicates the program's logic.

We described EVENTWAVE, a new programming model for tightly-coupled, stateful, distributed applications that provides transparent elasticity while preserving sequential semantics. An application is written as a fixed number of logical nodes and the runtime provides elastic execution on arbitrary numbers of physical nodes.

Our case studies suggest EVENTWAVE eases the development effort for elastic cloud applications, and EVENTWAVE applications scale efficiently.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and suggestions, and our shepherd, Phil Bernstein, for his many useful suggestions on how to improve this paper.

References

[1] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta informatica*, 9(1):1–21, 1977.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[4] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting microsoft sql server for cloud computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1255–1263. IEEE, 2011.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP'95*, pages 207–216, 1995.

[7] B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson. The portable batch scheduler and the maui scheduler on linux clusters. In *Usenix, 4th Annual Linux Showcase & Conference*, 2000.

[8] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 16. ACM, 2011.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices*, 40(10):519–538, 2005.

[10] W.-C. Chuang, B. Sang, C. Killian, and M. Kulkarni. Programming model support for dependable, elastic cloud applications. In *Proceedings of the Eighth USENIX conference on Hot Topics in System Dependability*, pages 9–9. USENIX Association, 2012.

[11] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. Workflow in condor. In *In Workflows for e-Science, Editors: I.Taylor, E.Deelman, D.Gannon, M.Shields*. Springer Press, 2007.

[12] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic transactional data store in the cloud. *USENIX HotCloud*, 2, 2009.

[13] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04*, pages 10–10. USENIX Association, 2004.

[14] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 301–312. ACM, 2011.

[15] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[16] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive offloading for pervasive computing. volume 3, pages 66–73. IEEE, 2004.

[17] C. Hewitt. Actor model of computation: Scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[18] C. Hewitt. Tutorial for actorscript (tm) extension of c sharp (tm), java (tm), and objective c (tm): iadaptive (tm) concurrency for anticloud (tm) privacy and security. 2010.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[20] A. Kangarlou, P. Eugster, and D. Xu. Vnsnap: Taking snapshots of virtual networked environments with minimal downtime. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 524–533. IEEE, 2009.

[21] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Detecting liveness bugs in systems code. In *NSDI*, 2007.

[22] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI '07*, pages 179–188. ACM, 2007.

[23] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *CASES'01*, pages 238–246. ACM, 2001.

[24] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *ICDCS'88*, 1988.

[25] A. Luckow, L. Lacinski, and S. Jha. Saga bigjob: An extensible and interoperable pilot-job abstraction for distributed applications and systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 135–144. IEEE, 2010.

[26] P. McGachey, A. Hosking, and J. Moss. Class transformations for transparent distribution of java applications. volume 10, 2011.

[27] J. T. Moscicki. Diane-distributed analysis environment for grid-enabled simulation and analysis of physics data. In *Nuclear Science Symposium Conference Record, 2003 IEEE*, volume 3, pages 1617–1620. IEEE, 2003.

[28] M. D. Noakes, D. A. Wallach, and W. J. Dally. The j-machine multicomputer: an architectural evaluation. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 224–235. ACM, 1993.

[29] S. Ou, K. Yang, and A. Liotta. An adaptive multiconstraint partitioning algorithm for offloading in pervasive systems. In *PerCom'06*, pages 10–pp. IEEE, 2006.

[30] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework.

In *SC'07*, pages 1–12. IEEE, 2007.

[31] L. Wang and M. Franz. Automatic partitioning of object-oriented programs for resource-constrained mobile devices with multiple distribution objectives. In *ICPADS'08*, pages 369–376. IEEE, 2008.

[32] S. Yoo, C. Killian, T. Kelly, H. K. Cho, and S. Plite. Composable reliability for asynchronous systems. In *USENIX Annual Technical Conference (ATC)*, 2012.

[33] S. Yoo, H. Lee, C. Killian, and M. Kulkarni. Incontext: simple parallelism for distributed applications. HPDC '11, pages 97–108. ACM, 2011.

[34] J. Zhou and B. Demsky. Bamboo: a data-centric, object-oriented approach to many-core software. PLDI '10, pages 388–399. ACM, 2010.