Reactive Computations for Distributed Actors

Anonymous

Abstract

Many online applications connect users in real-time, be it for games, social interactions, or collaboration. However, architecting services to power these applications while providing elasticity and fault tolerance is difficult. To simplify this task, we propose *reactive computations*, an extension of the virtual actor programming model. A reactive computation computes a result not once, but continuously - the runtime automatically detects dependencies and propagates changes to the client, using a fault-tolerant distributed algorithm. Our cloud experiments show that the added overhead is minor: reactive computations perform similarly to solutions that directly push changes, but use a simpler programming model and provide fault tolerance.

Categories and Subject Descriptors D1.3 [*Concurrent Programming*]: Distributed Programming

General Terms term1, term2

Keywords Distributed Reactive Programming, Actor Model

1. Introduction

Applications that run on client devices and cloud services are ubiquitous today. Since the early days of the internet, such applications have evolved from simple page-delivery engines to distributed reactive applications that let users share the experience and interact socially. Whether it is about games, collaborative editing, or online conversations: connecting users in real-time via a scalable and fault-tolerant service remains a significant and exciting challenge.

Reactive Programming. Research on reactive programming has proposed many interesting solutions, and the community has been experimenting with many programming languages, language extensions, and frameworks. Simply speaking, a reactive programming model combines (1) a convenient way to express a view of some state or event source, and (2) a corresponding efficient mechanism that can update the view in response to state updates or events. It is a rich design space including many different philosophies and implementations.

For example, in *functional-reactive programming (FRP)*, we express views as signals that depend on event streams, and define them using a vocabulary of functional operators or combinators [3, 21, 23, 27, 46]. Another approach, which we call *model-view computation*, is to define views as the result of a computation (functional or imperative) that depends on the model state [17, 41]. In either case, propagating small changes selectively and efficiently is key. This can be achieved not only using FRP-style language support, but (alternatively or additionally) by runtime algorithms (such as virtual DOMs [17, 41], one-way dataflow constraints [25], or self-adjusting computation [1]) that can isolate which parts of a computation need to be updated in response to a change.

An important difference is that with FRP, all intermediate values matter (streams must not lose events), while for model-view computation, only the latest state of the model is relevant for computing the view. This distinction gains relevance when considering fault tolerance.

Actor-Based Cloud Services. Much research has focused on reactive programming for graphical user interfaces [17, 23, 37] on the client. But often, the observed state or events are located not on the client, but in the service. Providing a reactive programming model for services is much harder than for a single machine. Performance is more elusive because of the high communication latency between machines. Moreover, a good service provides *elastic scalability* (runs on a cluster that can be grown or shrunk at runtime), *fault tolerance* (automatically handles individual server failures) and *compute-storage separation* (persists application data in a separate cloud storage system). *None* of these requirements is guaranteed by *any* current reactive programming model, other than for large-scale data-parallel analytics which are no match for low-latency interactive services like games or collaborative editing.

Actors have emerged as a viable model for developing such services, because they can be distributed over a cluster [6, 8, 39]. In particular, *virtual actor* runtimes [11, 38, 39, 45] provide the necessary distributed protocols for persistence, load balancing, and fault detection. But how can we support reactive programming on virtual actor systems?

A common ad-hoc solution is to employ some sort of *observer pattern*. The programmer takes full responsibility for tracking dependencies and propagating changes between actors. But even for non-distributed programs, observer patterns can lead to complex and error-prone code [44]; this only gets worse when we have to tolerate failures and provide persistence.

Another common approach is to add support for *streams* [39, 42]. However, combining actors and streams can be confusing, as the FRP paradigm (state is an event stream) is at odds with the actor paradigm (state is encapsulated by asynchronously communicating actors). It is difficult to keep programs that combine actors and streams both fault tolerant and performant — persisting every event in a stream is costly and often redundant. Similarly, combining actors with a publish-subscribe mechanism [28] can ameliorate some of the drawbacks of observers, but does complicate fault tolerance and persistence.

Reactive Computations. In this work, we propose a novel approach to distributed reactive programming that does not replace the actor paradigm, but extends it with a new feature called *reactive computations*. Any client of the service can start such a reactive computation, which computes some result that depends on one or more actors of the service. But, as in self-adjusting computation [2], this result is computed not just once, but *continuously*: the latest result is pushed to the client whenever it changes.

Notably, the service code remains exactly the same — all the programmer has to do is to use a reactive computation on the client. The reactive magic is not in the program, but in the runtime: it executes the computation in a special mode that intercepts all actor calls, including nested calls, and constructs a distributed dependency graph. Whenever the state of any actor involved in the computation changes, the change is automatically propagated, i.e. pushed along the edges of the dependency graph. This happens for as long as the reactive computation remains active.

1.1 Contributions

We make the following contributions:

1. We propose a new programming model for reactive elastic services, which extends the virtual actor model (§2) with runtime support for reactive computations (§3).
2. We present a new fault-tolerant distributed reactive caching algorithm (§4), and implement it as an extension of the Orleans virtual actor runtime (§5).
3. We provide a performance evaluation (§6) that demonstrates that the latency and throughput of our automatic change propagation compares favorably with alternative solutions, such as manual change propagation and polling.

2. Virtual Actor Model

Actors have emerged as a useful abstraction for the middle tier of scalable service applications that run on virtualized cloud infrastructure in a datacenter . In such systems, each actor is an object with a user-defined meaning, identity, state, and operations. For example, actors can represent user profiles, articles, game sessions, devices, bank accounts, or chat rooms. Actors resemble miniature servers: they do not share memory, but communicate asynchronously, and can fail and recover independently. One of the main benefit of actor systems is that they *scale horizontally*, because the actor instances can be distributed across a cluster of servers.

In a traditional bare-bones actor system, the developer remains responsible for the creation, placement, discovery, recovery, and load-balancing of actors. A newer generation of actor models [38, 39, 45], called virtual actor models, automate all of these aspects. The developer specifies only a unique key for identifying each actor, and how to save and load the actor state to/from external storage, if persistence is desired. As virtual actor systems can activate and deactivate actors based on use, they resemble distributed caches [36] and provide similar performance benefits.

Grains. Adopting the terminology [11, 18], we call the virtual actors *grains*. For each grain type, the code defines a (1) key that identifies instances, (2) fields that define the grain state, and (3) operations supported by the grain. The operations of a grain can read and modify the grain state, and can call operations on other grains, but cannot directly read or write the state of other grains (encapsulation).

When a grain calls another grain, it sends an asynchronous message containing the operation name and all arguments, which are copied or serialized (caller and callee cannot share data structures); and when an operation completes, a message is sent back to the caller, containing the return value or an exception. Messages are delivered at most once, and there are no ordering guarantees.

2.1 Example: Chirper

As a running example, we use a service called *Chirper* (based on the Orleans sample with the same name). It models a typical social application where users post messages to their own timeline, and view a timeline containing all messages posted by people they are following. The code for the chirper service is shown in Fig. 1. We use imperative pseudocode and an imaginary actor DSL (domain-specific language) to focus on the programming model and minimize distractions (such as framework details or C# language features).

There is only one type of grain, called User. A user grain encapsulates data associated with a user, and is identified by a string called userid (line 1). Each user grain stores the messages posted by that user in a map data structure Messages (line 4), keyed by timestamp. The current list of followed users is stored in Follows (line 5). The grain state

```
1  grain User[userid: string]
2  {
3    // grain state
4    state Messages: map⟨time,string⟩;
5    state Follows: set⟨string⟩;
6    // update operations
7    op Post(t: time, msg: string) { Messages[t] = msg; }
8    op Unpost(t: time) { Messages.remove(t); }
9    op Follow(id: string) { Follows.add(id); }
10   op Unfollow(id: string) { Follows.remove(id); }
11   // query operations
12   op GetMessages(from: time) : list⟨pair⟨time,string⟩⟩ {
13     var msgs = new list⟨pair⟨time,string⟩⟩();
14     foreach(var m in Messages)
15       if (m.key ⩾ from)
16         msgs.add((m.key, m.value));
17     return msgs;
18   }
19   op GetTimeline(from: time): list⟨pair⟨time,string⟩⟩ {
20     var msgs = GetMessages(from);
21     parallel foreach(var f in Follows) {
22       // retrieve messages from user f by remote call
23       var fm = User[f].GetMessages(from);
24       foreach(var m in fm) { msgs.add(m); }
25     }
26     msgs.sort();
27     return msgs;
28   }
29 }
```

Figure 1. Pseudocode for the Chirper service example.

can change in response to update operations Post, Unpost, Follow, Unfollow (lines 7–10) , which are invoked by the chirper client in response to clicking respective buttons.

The operation GetMessages(from) (line 12) returns all messages by this user since the time indicated by from; it straightforwardly computes the result by iterating over Messages.

The operation GetTimeline(from) (line 19) is more interesting. It returns a sorted list of all messages posted by the user *and all followed users* since the time indicated by from. It computes the result as follows: first, it retrieves the locally stored messages (line 20). Then, *in parallel* (line 21), it calls GetMessages on all followed users. These calls target other grains, by constructing a grain reference User[f] to the grain for userid f (line 23). The returned messages are all added to the list msgs (line 24). After all the parallel tasks complete, the messages are sorted (line 26) and returned (line 27).

2.2 Failure Model

A grain may fail at any time, even when in the middle of executing an operation, and this can be observed by the application. The runtime does provide some support for simplifying the failure handling at the application level.

```
1  while (interested) {
2    try {
3      var result = Grain[myuserid].GetTimeline();
4      display(result);
5    } catch(TimeoutException) {
6      display("server is not responding, retrying...");
7    }
8    await delay(5000); // wait for 5 seconds between refresh
9  }
```

Figure 2. Possible solution using client-side polling.

Recovery. If a grain is hosted on a server and that server fails, the runtime activates a fresh instance of the grain the next time it is accessed.

Persistence. Optionally, programmers can declare grains to be *persistent*, which means their state is saved in external persistent storage. The state of a persistent grain is saved after each operation that updates the state, and loaded from storage when a grain is activated. The User grains in the chirper example are persistent.

Timeouts. All grain operations throw a special timeout-exception if no response is received after a configurable time limit (the default is 30 seconds). This helps to avoid hangs and deadlocks at the application level if a grain fails before responding, or becomes unavailable for any reason.

If an operation times out, the caller does not know whether its effect was performed or not. To handle this issue, applications often design update operations to be *idempotent* [40], so they can be safely retried after encountering a timeout-exception. For example, all the operations of the User grain in Fig. 1 are idempotent.

3. Reactive Computations

Suppose now that we want our chirper client to not just display a frozen snapshot of GetTimeline, but to react to changes in the application state and refresh the user display accordingly. In the absence of any special support for reactive programming, our options are to use polling (§3.1) or to implement change propagation explicitly at the application level (§3.2). Next, we briefly discuss those two solutions, and then show the solution based on runtime support for reactive computations (§3.3).

3.1 Client-side Polling

A straightforward solution is to call GetTimeline periodically on the client to retrieve the latest state (Fig. 2). This solution is easy to understand, requires no changes to the service code, and gracefully handles transient exceptions (such as caused by high load or server failures). However, it confronts us with a unpleasant tradeoff when choosing the *polling frequency*: infrequent polling means the displayed result can lag significantly behind the latest result; but frequent polling

```
// new additions to grain state
state Observers: map⟨Observer,time⟩;
state Followers: set⟨userid⟩;

// new update operations
op AddFollower (userid: string)
  { Followers.Add(userid); }
op RemoveFollower (userid: string)
  { Followers.Remove(userid); }
op AddObserver (client: Observer, from: time)
   : list⟨pair⟨time,string⟩⟩{
  Observers[client] = time;
  return GetTimeline(from);
}
op RemoveObserver (client: Observer)
  { Observers.Remove(client); }

// modifications to existing update operations
op Post(tstamp: time, msg: string) {
  ...
  NotifyFollowers(); NotifyObservers();
}
op Unpost(tstamp: time) {
  ...
  NotifyFollowers(); NotifyObservers();
}
op Follow(userid: string) {
  ...
  User[userid].AddFollower(this.userid);
  NotifyObservers();
}
op Unfollow(userid: string) {
  ...
  User[userid].RemoveFollower(this.userid);
  NotifyObservers();
}

// new propagation operations
op NotifyObservers() {
  foreach(var (client,from) in Observers) {
    var latest = GetTimeline(from);
    client.Notify(latest);
  }
}
op NotifyFollowers() {
  parallel foreach(var f in Followers) { f.NotifyObservers(); }
}
```

Figure 3. Cautionary sketch of what it may take to apply the observer pattern to the User grain from Fig. 1.

dramatically increases the load on our service. We demonstrate this tradeoff experimentally in section 6 .

3.2 Explicit Propagation

A more challenging, but possibly better-performing solution is to add code to our service that explicitly tracks dependencies and pushes changes to the client. Fig. 3 shows a sketch

```
1  var rc = CreateReactiveComputation(
2                    () ⇒ Grain[myuserid].GetTimeline()  );
3  var resulttracker = rc.GetResultTracker();
4
5  while (interested) {
6    try {
7      var result = await resulttracker.NextResult();
8      display(result);
9    } catch(TimeoutException) {
10     display("server is not responding, retrying...");
11   }
12 }
```

Figure 4. Proposed solution: runtime support for reactive computations.

of how we can apply the observer design pattern to the User grain (many other variations are possible). Clients can register themselves as observers (line 11), which means they receive notifications containing the latest result. Each grain tracks observing clients and followers and notifies them. *But this solution leaves much to be desired.* We have polluted the grain state with information about observers, and it is unclear whether this state should be persisted (and thereby pollute our storage as well) or not (and thereby cause issues when grains fail). Also, as written, changes are still not propagated very efficiently — there remain many potential optimizations, and opportunities for introducing subtle bugs.

3.3 Automatic Propagation

By adding support for reactive computations into the virtual actor runtime, we can spare the programmer considerable hassle, and let them benefit from an automatic, optimized distributed algorithm for dependency tracking and change propagation.

We propose a novel API for reactive computations over virtual actors (Fig. 4). It is purposefully similar to the client-side polling solution (Fig. 2) and likewise does *not require any changes to the service code*.

First, the programmer creates a *reactive computation* object (line 1), passing an anonymous function that describes the computation. This instructs the runtime to construct a dependency graph which continuously tracks the result of that computation and propagates changes via pushing, until the rc object is disposed.

To consume the results, the programmer simply creates a *result tracker* object (line 3). The result tracker operates like an enumerator that produces both the initial result, and any successive changed results, when we call NextResult (line 7). Note that the code in the loop (lines 5–12) looks just like the client-side polling solution (Fig. 2), and has the same failure and exception semantics, but we no longer need to specify an arbitrary polling delay — instead, the loop simply waits for a new result to arrive, and then delivers it immediately. With compiler support for **await**, as in C# [14], this is just as

efficient as using callbacks, because the compiler constructs continuations that do not block the thread.

Exactly how the client processes the results is outside the scope of this work, and likely depends on the chosen client framework. For chirper we used react [41]; it lets us simply assign the result to the model, and automatically does the diffing.

4. Reactive Caching Algorithm

To determine when a result of a reactive computation changes, we track all the grains it depends on. Tracking these dependencies is done entirely at runtime and does not require any static analysis: rather, we modify the virtual actor runtime to intercept grain calls and construct a *dependency graph*.

We explain the dependency graph in two stages: first, we describe it as a directed acyclic graph of summaries (§4.1). Then, we show to represent it as a bipartite graph of summaries and caches (§4.2) to improve performance and handle failures in a distributed setting.

4.1 Summaries and Dependencies

During execution, we record and store a *summary* for each executed operation. A summary is a pair consisting of (1) the invoked operation (including grain identity, method name, and all parameters), and (2) the value or exception returned at the end. For each summary, we record what other summaries it depends on, and what summaries depend on it. The result is a directed *dependency graph* that represents the computation that was performed.

Fig. 5(a) shows an example of a dependency graph of summaries. Two clients perform reactive computations Alice.GetTimeline(t) and Chris.GetTimeline(t) for the same time parameter t. Summaries are shown as black boxes, and reside either in a grain or in a reactive computation. Summary dependencies are shown as black arrows.

Reuse. If a summary for a particular operation already exists, we reuse it. For example, [GetMessages(t)→ {y}] of Bob's grain is shared: two other summaries depend on it.

Re-execution. If a summary is stale (see §4.1.1), it is marked for re-execution. When re-executed, the dependencies may change, and are updated accordingly.

Garbage Collection. A summary for a grain method is deleted if there are no other summaries that depend on it. A summary for a reactive computation is deleted when the reactive computation object is disposed. Since the dependency graph is acyclic (a cycle would represent a query with infinite recursion, which is prevented by the runtime), this means that summaries are guaranteed to be collected when no longer needed.

4.1.1 Change Propagation

We call a summary *stale* if a re-execution of the computation or operation would necessarily yield a different result. Our change propagation algorithm guarantees that *any stale summary is eventually re-executed*. Because grains cannot share any state, summaries can become stale for only two reasons: either (1) the state of their grain changes, or (2) a summary they depend on becomes stale.[1] Thus, the following is sufficient:

1. Whenever a grain operation changes the state of a grain, we mark all summaries of that grain for re-execution.
2. After a summary for a grain operation is re-executed, we compare the new result to the previous result. If it is the same, no further action is needed. Otherwise, we mark all dependent summaries for re-execution.

For example, executing the operation Chris.Unfollow("Bob") will mark both of the summaries in Bob's grain for re-execution. Re-execution of [GetMessages(t)→ {}] yields the same result, and propagation stops. But re-execution of [GetTimeline(t)→ {y}] yields a different result {}, which means we mark the dependent summary in Reactive Computation 2, [Chris.GetTimeline(t)→ {y}], for re-execution. After it re-executes, the latest result reaches the client as desired.

Ephemeral Inconsistency. It is possible for the result of a computation to be inconsistent, in the sense that it is based on versions of grain states that never existed at the same time, or even on different versions of the same grain's state. However, if the result of a computation is inconsistent, then it is also ephemeral: an inconsistent summary must be stale, and is thus guaranteed to be re-executed and replaced.

Batching. Some time may pass in between marking a summary for re-execution and the actual re-execution. In particular, it may be marked multiple times, in which case it only needs to be executed once. This batching effect is important for maintaining good throughput under high update frequencies, as we demonstrate in the evaluation section.

4.2 Reactive Caching

Under the hood, Orleans' virtual actor runtime load-balances grains across different physical machines called silos. The set of silos can change when administrators choose to increase or decrease the number of servers, or when servers fail (which is detected automatically). By design, the application layer is unaware of the existence of these silos. However, to make our algorithm performant and fault-tolerant, the spatial distribution of grains over silos is relevant, because (1) the latency of a remote call is orders of magnitude higher than the latency of a local call, and (2) silos can fail independently. Thus, we build an extra *caching layer* into the dependency graph.

Rather than summaries that depend directly on other summaries, we now have summaries depend on local caches of summaries. Each summary maintains a connection to all its

[1] We assume that the grain operations used in reactive computations depend only on grain states, and not external state such as clocks or arbitrary I/O.

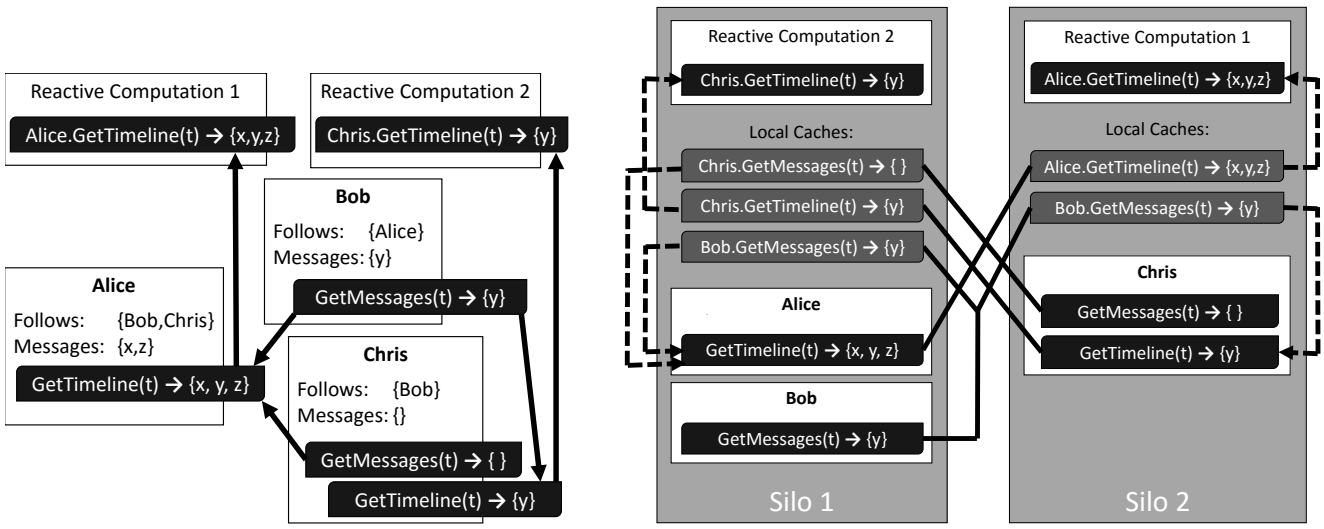

Figure 5. (a) (on the left) example of a dependency graph of summaries, for two reactive computations; **(b)** (on the right) example of a corresponding bipartite dependency graph of summaries and local caches, distributed over two silos.

caches on the various silos, and pushes changes to those caches whenever a re-execution produces a different result. This *reactive caching* mechanism improves performance because it can elide many slow remote accesses in favor of fast lookups in a silo-local hash table.

As an example, consider Fig. 5(b). It shows a bipartite graph of summaries and caches that corresponds to the dependency graph in Fig. 5(a). We show two types of edges: summary dependencies (dotted arrows), which are always within a silo, and cache connections (solid lines), which may cross silo boundaries.

Extending change propagation and garbage collection to the bipartite graph is straightforward: when a cache receives a new value, it marks dependent summaries for re-execution. Caches are removed if they have no summaries that depend on them, and summaries that are not part of a reactive computation are removed if they are not connected to any caches.

Fast Re-execution. When a summary is re-executing, and its dependencies have not changed, all of its grain operations hit in the local cache. Therefore, the overhead of re-execution of summaries is quite low: the latency of our change propagation mechanism is very close to the performance of a hand-written change propagation solution (see §6).

Large Fan-out. The reactive caching improves performance in situations with a large fan-out (summaries with many observers): rather than sending updates to each observer directly, it is enough to send one update to each silo, and the silo then forwards the update locally.

Back-Pressure. Our experiments show that in the case of high update rates, it is beneficial to throttle the sending of updates (only the latest result matters, so intermediate results can be dropped). Our mechanism achieves this by sending results to caches one at a time, and measuring the response time. If above a configurable threshold, we back off (wait for an extra delay equal to the round trip time of the push).

4.2.1 Fault Tolerance

To tolerate faults, we need to make sure that we correctly handle the disappearance of any components that are visible to components on other silos.[2] In our case, the only thing we need to be concerned about is the connection between summaries and caches (this is readily apparent in Fig. 5b: the only lines that cross silo boundaries are the solid edges).

Each summary maintains a list of connected caches. To maintain this connection, a cache periodically *re-subscribes* every 30 seconds. This re-subscription allows us to handle one-sided failures as follows:

- if a cache fails, it will no longer resubscribe. If a summary does not hear from a cache for 90 seconds, it removes it from the list.
- if a summary fails, the cache sends a re-subscription after some time — but the summary, and its associated grain, no longer exist. In that case, the standard virtual actor mechanism re-activates the grain on some silo, and loads its state from persistent storage. Then we recreate the summary and execute it. The cache is now connected to the new instance of the same summary.

5. Implementation

We have implemented reactive computations as extensions to Orleans, an open-source distributed actor framework for .NET available on GitHub [39]. The Orleans runtime already provides the needed distributed protocols for managing the

[2] We need not worry about failure of components that are not visible on other silos, because whenever they fail, all components that are aware of them also fail.

creation, placement, discovery, recovery, and load-balancing of grains [11, 18]. What we added is (a) extensions to the grain objects to store summaries, (b) interception points for grain calls, (b) modifications to the grain scheduler to distinguish between reactive and normal execution mode, and (c) a silo-wide cache manager.

5.1 Language and Framework

So far, our exposition of the programming model and algorithm has avoided delving into particulars of the Orleans framework or of C#. We now discuss some of these details.

***Disposal by* using.** Until the programmer disposes the reactive computation object, the runtime incurs the cost of tracking summaries and propagating updates. To avoid leaking these computational resources accidentally, programmers can use the C# using clause as shown in Fig. 6 (line 1). This guarantees that Dispose is called automatically on the reactive computation when control leaves the block (lines 3–17), either normally or due to an exception (and it works correctly in combination with await).

Disposing the reactive computation removes the summary associated with the reactive computation, and any local caches it depends on (unless those have other dependents). As described in (§4.1), this eventually guarantees that all summaries and caches that have no observers are removed. It can take some time because summaries are not removed until 90 seconds have passed since the last subscription by a cache (§4.2.1).

Asynchronous I/O. ResultTrackers are well suited for situations where updating the display requires I/O, such as when communicating with a remotely connected client device: (1) result trackers may skip intermediate versions: only the latest result matters, and (2) result trackers return a task that can be efficiently awaited without blocking the thread. (using C# language support for async/await [14]).

It is possible to use multiple result trackers for the same reactive computation, and each one can consume results at its own speed (for example, over different websockets).

Exception Handling. As can be seen from Fig. 6, whenever an exception occurs somewhere down the reactive computation, we don't simply stop the computation and deconstruct the dependency graph. Instead we assume the exception just occurred due to some current state of the grains, but following states might produce results again. The exception can be caught (1) either somewhere upstream in the reactive computation itself, or (2) on the NextResult() call of a tracker if it wasn't caught. This provides a uniform way of handling both faults (timeout exceptions) (Fig. 6 line 9) and computational exceptions (Fig. 6 line 13).

5.2 Effects, Dependencies, and Determinism

Detecting Changes. The propagation algorithm described in § 4.1.1 requires that we detect whenever the state of a grain changes. Unfortunately, in Orleans, we cannot easily detect whether an operation has side effects, because grains are C# objects, and the use of heap and libraries obfuscates the presence of side effects. Therefore, we overapproximate and conservatively assume that *all* operations change the grain's state. This is not as expensive as it may seem at first, because if the grain state has not changed, or changed in ways that are irrelevant, re-execution of the summary produces the same result, and propagation stops.

Programmers can annotate an operation with a [ReadOnly] attribute to avoid the re-execution overhead; also, we assume that any operation called as part of a reactive computation does not change the grain state, and thus avoid summary re-execution in that case.

To execute or not to execute. Our algorithm changes the way grain operations are executed, which can surprise programmers. Any method that is called from within a reactive context is now prone to being re-executed without the programmer explicitly performing the call. Conversely, calling any such method may skip the execution entirely and instead return a cached result. In theory, this is fine as long the method does not modify any state, and if it does not have an external dependency that is invisible to our staleness detection (e.g. read a clock or do arbitrary I/O).

These conditions are usually satisfied as reactive computations are meant to be used for some kind of *query* over the distributed state. However, currently, we do not enforce them. For a different host language, one can imagine a type-/effect system to this end. However, note also that a naive rigid enforcement is not advisable, because we want to allow harmless side effects (such as writing a timestamped message to a log).

```
1  using(var rc = CreateReactiveComputation(
2                    () ⇒ Grain[myuserid].GetTimeline()))
3  {
4    using(var resulttracker = rc.GetResultTracker())
5    {
6      while (interested) {
7        try {
8          var result = await resulttracker.NextResult();
9          display(result);
10       } catch(TimeoutException) {
11         display("server is not responding, retrying...");
12       } catch(DivisionByZeroException) {
13         display("can't divide by zero");
14       }
15     }
16   }
17 }
```

Figure 6. Proposed solution, including the `using` constructs that make sure the graph is garbage collected when no longer required.

Determinism. We do *not* require that grain operations are deterministic. For example, it is o.k. for an operation to call two other grains concurrently, and return the first of the two results returned. By definition, a cached value is considered stale only if a re-execution *must* return a different result, not if it *may* return a different result.

6. Performance Evaluation

When operating a service, the overall performance objective is to provide acceptable service latency at minimal cost. The cost can be lowered by running fewer servers - but this increases the latency because requests spend more time waiting for contended resources (such as network, storage, CPU, threads, locks, and so on). This tradeoff is an essential challenge for developers writing an elastic service application.

To clarify the value that our automatic, fault-tolerant change propagation provides to service architects, we now quantify both (a) its latency and (b) its resource consumption, by comparing them to alternative solutions, including periodic polling (§3.1), which is also fault-tolerant, and explicit change propagation at the application level (§3.2), which is not fault-tolerant. To this end, we designed two series of experiments that measure low-load latency (§6.1), and variable-load throughput (§6.2).

Application Model. We model the application using *item* grains that are observed by *view* grains. Each view depends on a fixed number of items, selected at random at the beginning of the test. Views are updated when items change, in a manner that depends on the chosen propagation solution. Moreover, we vary the number of items and views to simulate different workloads (Table 1). For example, a high *fan-out* (= average number of views that depend on an item) means that whenever an item is mutated, many views need to be updated.

The items and views run on five Orleans silos deployed as a Windows Azure cloud service. The robots run on 10 load generator servers. All processors have 8 cores, 14GB of RAM, and run at 1.6 GHz. To account for unexpected variations, we made sure to run each experiment series on at least 2 different datacenters, on at least 3 different days, and running the experiments in different order. Between runs, absolute numbers can vary up to 10%, but the relative performance of the various solutions were stable. Note that we achieved this only after investing substantial work into our experimentation framework.

Note that by design, this is a microbenchmark that isolates the mechanism we want to measure (change propagation) and removes all other aspects. If deployed as part of a complete service, including item persistence, the performance differences between the various solutions are likely to be less pronounced.

Name	#items	#views	#deps.	max robots
low-load	600	20	4	n/a
fanout-1	20,000	20,000	1	2,000
fanout-20	10,000	20,000	10	2,000
fanout-200	1,000	20,000	10	1,000

Table 1. Parameter combinations we used.

```
1  grain View
2  {
3    state Deps: Item[numdeps];
4    op Query(sequential: bool) : int[]
5    {
6      var result = new int[numdeps];
7      if (sequential) {
8        for (0 <= i < numdeps)
9          result[i] = Deps[i].GetValue();
10     } else {
11       parallel for (0 <= i < numdeps)
12         result[i] = Deps[i].GetValue();
13     }
14     return result;
15   }
16   op OneTimeReactiveQuery(sequential: bool) : int[]
17   {
18     var rc = CreateReactiveComputation(
19                          () ⇒ Query(sequential));
20     var result = await rc.GetResultTracker.NextResult();
21     rc.Dispose();
22     return result;
23   }
24 }
```

Figure 7. Queries expressing how views depend on items.

6.1 Latency Experiments

Our latency experiments use the low-load parameters (Table 1) to eliminate delays caused by queueing and contention. We measure two types, called *query latency* and *propagation latency*. Latencies are measured 4000 times each (200 times per view, separated by 500ms). We describe the results using the median and lower and upper quartiles.[3]

6.1.1 Query Latency

For measuring query latency, we execute a query on a view that calls each of four items (either sequentially or in parallel) to collect some value and returns the values in an array (Fig. 7). We now compare the latency when executed normally (Query) to the latency when executed as a reactive computation (OneTimeReactiveQuery). The normal latency for the sequential and parallel versions are shown in the first two rows of Fig. 8 (top). The median latency is about 1.2ms for the parallel query and 3.8ms for the sequen-

[3] Average and standard deviation are unsuitable statistics because of the long tail of the distribution.

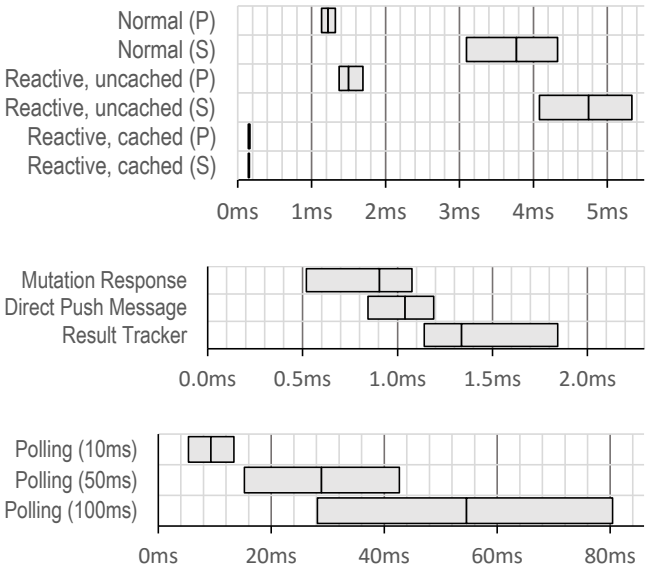

Figure 8. Measured latencies under low load, in milliseconds. The line in the middle of each box is the median, and the left and right edge are the first and third quartile. **(top)** Latencies for parallel and sequential queries; **(middle)** latencies for mutation response, application-level propagation using direct messages, and automatic propagation using result trackers; **(bottom)** propagation latencies for the polling solution at various frequencies.

tial query. This is consistent with the round-trip time of a typical grain call taking a bit less than 1ms. For the reactive queries, we distinguish two cases. If the relevant summaries are not already cached on the silo, the query takes about 25% longer than normal (rows 2,3). This overhead is caused by the installation and removal of the summary caches, and by scheduling overhead of our reactive caching implementation. However, if summaries for the items are already cached on the silo (for example, if another view is tracking the same items), the latency of OneTimeReactiveQuery is less than 200μs (rows 4,5) because remote calls can be completely avoided.

Conclusions. The results demonstrate that (1) the latency overhead of constructing the dependency graph is modest, and (2) the caching effect alone can improve latency, even if not using the reactive features.

6.1.2 Propagation Latency

For measuring propagation latency, a view sends an update message to an item it depends on, and measures how much time elapses (a) until it receives the mutation response, and (b) until it receives the change propagation.

First, we measured the speed of explicit propagation, where each item sends a notification message to all dependent views when mutated. This establishes a lower bound on propagation speed in the Orleans framework. The results show that the propagation message arrives right after the mutation completes: close to 1ms after calling the mutation operation (rows 2,3 of Fig. 8 (middle)). This is consistent with the underlying system sending the mutation response message and the propagation message at about the same time.

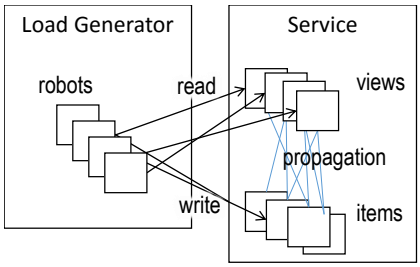

Figure 9. Experimental setup for throughput experiments.

Second, we measured the speed of automatic propagation provided by our reactive caching algorithm. In that case, the propagation takes about 300μs longer (row 3), due to the scheduling overhead of our implementation.

Finally, we looked at the propagation speed of a polling-based solution. Fig. 8 (bottom) shows the measured latencies, using a sequential query and various polling intervals. As expected, we see a median propagation time in the neighborhood of half of the polling interval plus the query latency, and a wide inter-quartile distance. However, polling every 100ms is usually not advisable (we show impact of polling on throughput in §6.2). Reasonable polling intervals are more typically between 1 and 30 seconds, with a median propagation speed that is easily three orders of magnitude worse than for explicit or automatic change propagation.

Conclusions. The results show that our automatic propagation performs much better than a polling solution, while having the same simple programming model; and is not much slower than explicit change propagation, despite offering the advantage of being fault-tolerant, and much simpler to use.

6.2 Throughput Experiments

For the throughput experiments, we generate external load as shown in Fig. 9. The load generator contains up to 2000 robots; each runs a continuous loop that sends requests to the service, either to read a view, or to update an item. The percentage of updates in the mix is configurable.

Each experiment gradually increases the robots and measures the throughput over time. The result is a curve that shows how throughput (number of requests handled per second) responds to load (number of requests concurrently in flight). For example, for the fanout20 configuration and a request mix containing 10% updates, we obtained the curves shown in Fig. 10; each line corresponds to one experiment, and each bundle of similar lines corresponds to several experiments using the same propagation mechanism.

The best possible throughput is achieved when change propagation is turned off entirely - because all resources are used for handling requests. Here, we reach close to 120k re-

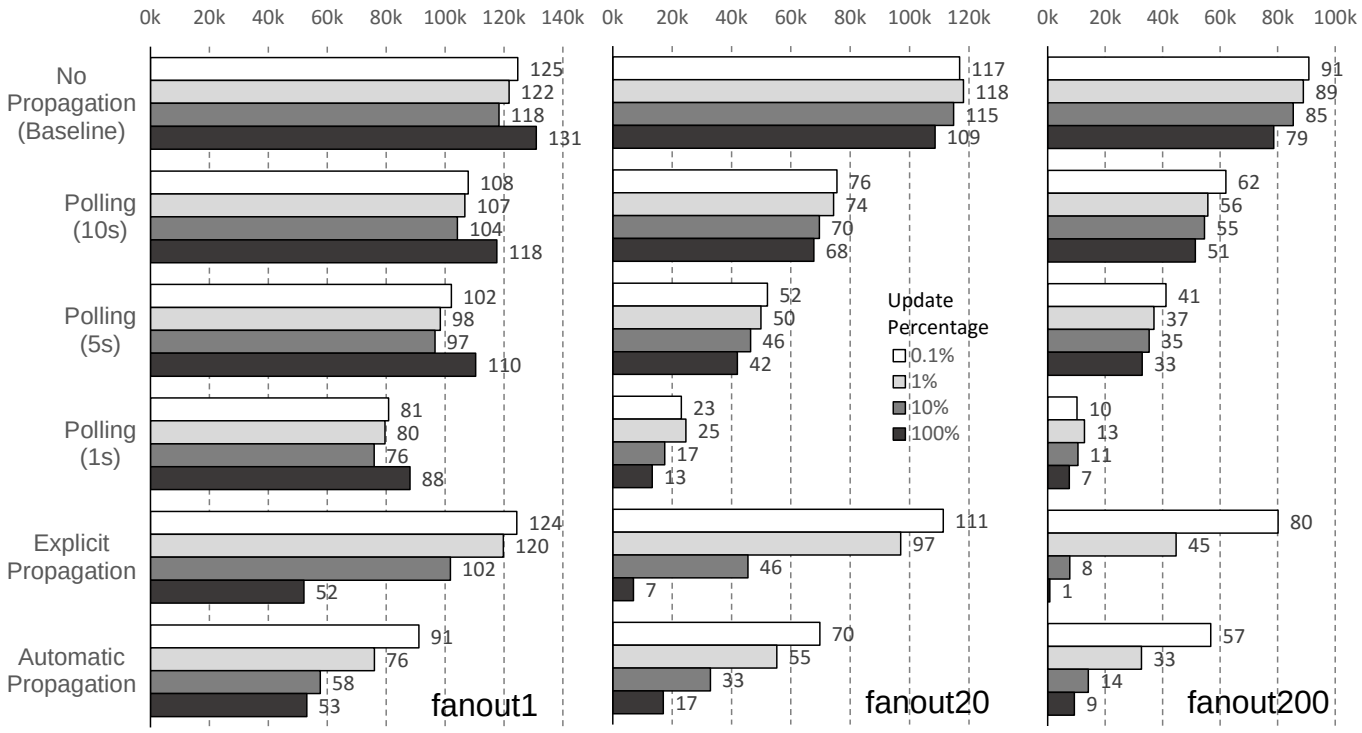

Figure 11. Measured throughput (in thousands of requests per second) for varying configurations (Table 1), propagation mechanisms, and percentage of updates.

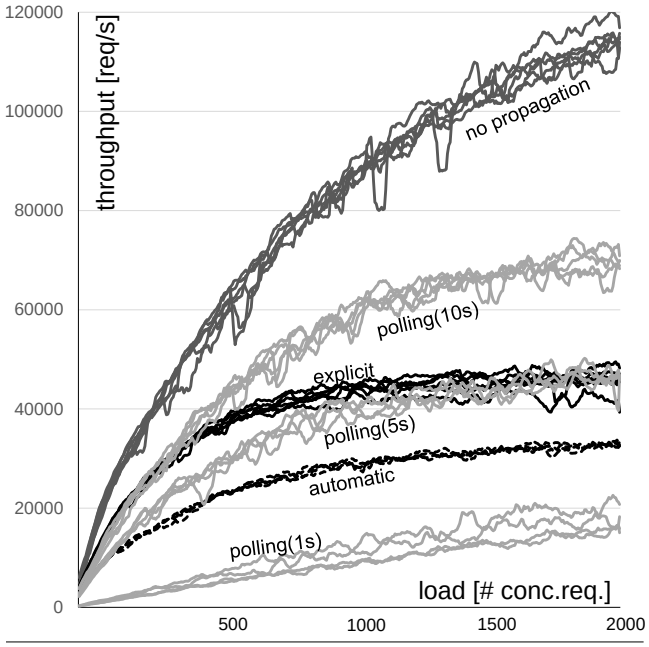

Figure 10. Throughput response of different propagation mechanisms, for fanout-20 with 10% updates.

quests per second. But as soon as we use change propagation mechanism, the throughput is lower, because resources diverted to the propagation mechanism: explicit propagation (dotted lines) reaches about 45k, automatic propagation reaches about 35k. For 10s-polling, the throughput reaches about 70k (better than automatic or manual propagation), but for 1s-polling, it reaches only about 20k (worse than automatic or manual propagation).

To compare the solutions across different configurations and update ratios, we ran this type of experiment for all combinations, but extended to run fixed load (the maximum number of robots as in Table 1) for a while at the end, to measure average peak throughput for that load. The results are shown in Fig. 11. Each column corresponds to a configuration in Table 1); each row corresponds to a choice of propagation mechanism; and each bar color corresponds to update percentage. For example, the dark gray bars (10% updates) in the middle column (fanout20) correspond to the peak throughput in Fig. 10

Baseline. With update propagation turned off (top row, labeled *None*), all requests are simple operations on a single grain. We reach a throughput in the neighborhood of 120k for the fanout1 and fanout20 configurations (under a load of 2000 concurrent requests), and near 90k for the fanout200 configuration (under load of 1000 concurrent requests). Though throughput is largely consistent, the numbers show some unexpected variation: throughput degrades somewhat with higher update percentages, and jumps up for

the specific combination of fanout1 and 100% updates). We suspect they are caused by load balancing differences within the Orleans runtime regarding items, views, and requests.

Polling. The extra work incurred by polling is (a) inversely proportional to the polling interval, and (b) proportional to the number of items a view depends on. The reduction in throughput (relative to the baseline) is thus modest for views that depend on only 1 item (left column), especially for large polling intervals, but if the view depends on 10 items (middle and right column), the throughput reduction is significant.

Explicit Propagation. The extra work incurred by explicit propagation is proportional to both the percentage of updates and the fan-out. The results confirm this: (1) we see o.k. throughput results for update percentages up to 1%, and (2) we see terribly low throughput for the combination of high fanout and high update percentage, as low as 1k (lowest of all) for fanout200 and 100% updates.

Automatic Propagation. There are two main differences between the work required for automatic and explicit propagation: (1) automatic propagation incurs a bit more work due to scheduling indirection, re-execution of summaries, and management of reactive caches; and (2) automatic propagation can adapt to back-pressure and reduce the number of updates sent. We can observe these effects: throughput for automatic propagation is generally lower than for explicit propagation, except for high update rates and/or fanout where explicit propagation suffers more.

Conclusions. The results show that automatic propagation is generally competitive with explicit propagation, despite the added benefit of fault tolerance and a simpler programming model. It is even better in cases where there are many updates to be propagated, thanks to its batching optimization. Polling remains an acceptable solution for applications that do not require quick propagation time; it can be tuned to reliably consume little resources, even under high update rates.

7. Related Work

As discussed in the introduction, reactive programming is a rich area of research that spans language design, algorithms and implementation architecture (see the survey [9]).

7.1 Comparison to FRP

A fundamental difference between our solution and much prior work on FRP [3, 21–23, 27, 32, 37, 46] is that our computations do not require the addition of special operators/combinators to the host language, but execute just like regular code (imperative, object-oriented C# in our case). This reduces the mental overhead for programmers who are unfamiliar with FRP, or in situations where the FRP operators lack expressiveness. Incrementality is not achieved via compilation, but at runtime: the trick is to decompose the computation into subcomputations that can be selectively re-evaluated if their dependencies change. This is the principle used by self-adjusting computation [1, 2, 4, 5, 31], imperative reactive programming [25], or incremental concurrent revisions [16]. The difference is that here, we use the encapsulation afforded by the *actor model* as a means to decompose the computation.

7.2 Semantics and Consistency

Change propagation is semantically subtle. A so-called *glitch* occurs if an observer sees two observables A, B that have inconsistent state, meaning that the set of updates propagated to A is different from the set of updates propagated to B. Many reactive systems strive to eliminate glitches (e.g. using topological ordering of dependencies), but some embrace them.

How to best compromise between consistency and performance is highly dependent on the architecture and workload. Avoiding glitches in a distributed actor system like ours is likely to add significant latency overhead: updates are only partially ordered to begin with, and dependencies are detected dynamically. For the applications we have in mind, latency is usually more important (i.e. it is preferrable to quickly display a glitchy result and then quickly correct it, than to wait). Consequently, we chose an algorithm that does not avoid glitches or causality violations categorically, but guarantees that they are ephemeral (§4.1.1). This tradeoff is similar to variations of eventual consistency [15].

At the other end of the spectrum are synchronous reactive languages [10, 12, 19, 29], where time is explicit, and systems make very strong semantic guarantees. It is hard to imagine an efficient implementation of such models in the context of a scalable service built from asynchronously communicating independent actors, given the high cost of coordination.

7.3 Cloud scale

Reactive techniques have been successfully applied to large-scale data-parallel computations used for data analytics [24, 47]. Some systems incrementalize mapreduce queries[13, 20] or queries constructed from a FRP-like vocabulary of operators [30], possibly including fixpoints [35]. Frameworks for stream programming [7] have also gained popularity - they are more low-level and very flexible, as the user assumes responsibility for plumbing the streams together to achieve the desired effect.

While fault tolerance, compute-storage separation, and elasticity are common guarantees for large-scale analytics, we are not aware of any system for programming reactive services that guarantees them. In fact, most such systems don't even consider the use of multiple servers, but assume a single-machine configuration, a configuration involving multiple clients and a single server, or consider the service to be part of the environment. An exception is distributed ReScala[26] which parallelizes update propagation across multiple servers. Unlike our solution, it needs to know

the dataflow dependencies in advance. It guarantees glitch-freedom without coordination, but only under the limiting assumption that updates are not concurrent, in start contrast to our workload that exhibits a large volume of concurrent independent updates on actors.

7.4 Other

The problem of combining object-oriented and reactive paradigms is not new [43], as there is often a desire to connect object-oriented GUI frameworks with functional-reactive backends [33]. In some sense our problem is the exact opposite: adding reactivity to an object-oriented (or rather, actor) back-end to support a reactive user interface. SuperGlue [34] is another example that adds reactivity to objects. In our experience, actor models provide a much better home for reactivity than mainstream object-oriented programs, because actors completely prevent the passing of shared data structures as arguments and return values. Our dependency tracking algorithm crucially relies on this fact when computing and caching summaries.

AmbientTalk/R also introduces reactive programming into the actor model. But its focus is not on services, but on providing reactivity within a mobile ad-hoc network. So-called *ambient behaviours* can be exposed, which allow actors to share behaviours using an intentional description of the behaviour.

8. Conclusion

We have motivated, explained, implemented, and evaluated a new mechanism called *reactive computations* that makes it easier for developers of actor-based services to quickly and easily propagate changes in actor states to clients that depend on it. Our results show that even though its API is as simple as polling, our fault-tolerant distributed reactive caching algorithm can push changes fully automatically and efficiently. Therefore, it is an attractive alternative to complex and error-prone solutions that implement push-based change propagation explicitly at the application level.

Much work remains to be done. We believe the performance of our implementation can be further improved, by identifying hot-spots and improving the scheduler. Also, we would like to gather more experience in an industrial context; we have already started a collaboration with a game developer team. Moreover, we would like to explore how to combine reactive computations with other mechanisms, such as streams and event sourcing. Another interesting question is to explore the performance cost of making stronger consistency guarantees, such as causality. For collection data types, using diffing optimizations may further improve performance. Finally, we are working on formalizations for describing the virtual actor model, the reactive caching algorithm, and the consistency guarantees.

Acknowledgments

omitted for anonymity.

References

[1] U. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Principles of Programming Languages (POPL)*, 2008.

[2] U. A. Acar. Self-adjusting computation (an overview). In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2009.

[3] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, Nov. 2006. ISSN 0164-0925.

[4] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *Transactions on Programming Languages and Systems (TOPLAS)*, 32:3:1–3:53, November 2009.

[5] U. A. Acar, G. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Türkoğlu. Traceable data types for self-adjusting computation. In *Programming Language Design and Implementation (PLDI)*, 2010.

[6] Akka - Actors for the JVM. Apache 2 License, https://github.com/akka/akka, 2016.

[7] Apache Flink. https://flink.apache.org/, 2016.

[8] J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, Sept. 2010.

[9] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013.

[10] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages. *Inf. Comput.*, 163 (1):125–171, Nov. 2000. ISSN 0890-5401.

[11] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, Microsoft Research, March 2014.

[12] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, Nov. 1992. ISSN 0167-6423.

[13] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. ACM.

[14] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause 'n' play: Formalizing asynchronous C#. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 233–257. Springer, 2012.

[15] S. Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, Oct. 2014. ISSN 2325-1107.

[16] S. Burckhardt, D. Leijen, J. Yi, C. Sadowski, and T. Ball. Two for the price of one: A model for parallel and incremental computation. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[17] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! Continuous feedback in UI programming. In *Programming Language Design and Implementation (PLDI)*, pages 95–104, 2013.

[18] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *ACM Symposium on Cloud Computing*, SOCC '11, pages 16:1–16:14, 2011.

[19] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for real-time programming. In *Principles of Programming Languages (POPL)*, pages 178–188, 1987.

[20] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Networked Systems Design and Implementation (NSDI)*, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[21] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming (ESOP)*, pages 294–308, Berlin, Heidelberg, 2006. Springer-Verlag.

[22] A. Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages (PADL)*, pages 29–44. Springer, 2001.

[23] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *Programming Language Design and Implementation (PLDI)*, pages 411–422, 2013.

[24] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51 (1):107–113, 2008.

[25] C. Demetrescu, I. Finocchi, and A. Ribichini. Reactive imperative programming with dataflow constraints. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 407–426, 2011.

[26] J. Drechsler, G. Salvaneschi, R. Mogk, and M. Mezini. Distributed REScala: An update algorithm for distributed reactive programming. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[27] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP)*, ICFP '97, pages 263–273, 1997.

[28] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[29] T. Gautier, P. Le Guernic, and L. Besnard. SIGNAL: A declarative language for synchronous programming of real-time systems. In *Functional Programming Languages and Computer Architecture*, pages 257–277, 1987.

[30] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Operating Systems Design and Implementation (OSDI)*, OSDI'10, pages 75–88, Berkeley, CA, USA, 2010. USENIX Association.

[31] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *Programming Language Design and Implementation (PLDI)*, 2009.

[32] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. *Arrows, Robots, and Functional Reactive Programming*, pages 159–187. Springer Berlin Heidelberg, 2003.

[33] D. Ignatoff, G. H. Cooper, and S. Krishnamurthi. *Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages*, pages 259–276. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-33439-2.

[34] S. McDirmid and W. C. Hsieh. *SuperGlue: Component Programming with Object-Oriented Signals*, pages 206–229. Springer, 2006.

[35] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Proceedings of CIDR 2013*, January 2013.

[36] Memcached. Available under BSD 3-clause license. https://github.com/memcached/memcached, 2016.

[37] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–20, 2009.

[38] Orbit - Virtual Actors for the JVM. BSD 3-clause license. https://github.com/orbit/orbit, 2016.

[39] Orleans - Distributed Virtual Actor Model for .NET. MIT license. https://github.com/dotnet/orleans, 2016.

[40] G. Ramalingam and K. Vaswani. Fault tolerance via idempotence. In *Principles of Programming Languages (POPL)*, 2013.

[41] React - A declarative JavaScript library for building user interfaces. Available under BSD 3-clause license. https://github.com/facebook/react, 2016.

[42] Reactors.IO. Available under BSD 3-clause license. https://github.com/reactors-io/reactors, 2016.

[43] G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: An analysis and a research roadmap. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 37–48, New York, NY, USA, 2013.

[44] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini. An empirical study on program comprehension with reactive programming. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 564–575, 2014.

[45] Service Fabric Reliable Actors. Available for the Windows Azure platform, see https://azure.microsoft.com/en-us/documentation/articles/service-fabric-reliable-actors-get-started/, 2016.

[46] Z. Wan and P. Hudak. Functional reactive programming from first principles. *SIGPLAN Not.*, 35(5):242–252, May 2000.

[47] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Networked Systems Design and Implementation (NSDI)*, pages 15–28. USENIX, 2012.