Reactive Caching for Composed Services

Polling at the Speed of Push

SEBASTIAN BURCKHARDT, Microsoft Research, United States
TIM COPPIETERS, Vrije Universiteit Brussel, Belgium

Sometimes, service clients repeat requests in a polling loop in order to refresh their view. However, such polling may be slow to pick up changes, or may increase the load unacceptably, in particular for composed services that disperse over many components. We present an alternative *reactive polling* API and *reactive caching* algorithm that combines the conceptual simplicity of polling with the efficiency of push-based change propagation. A reactive cache contains a summary of a distributed read-only operation and maintains a connection to its dependencies so changes can be propagated automatically.

We first formalize the setting using an abstract calculus for composed services. Then we present a fault-tolerant distributed algorithm for reactive caching that guarantees eventual consistency. Finally, we implement and evaluate our solution by extending the Orleans actor framework, and perform experiments on two benchmarks in a distributed cloud deployment. The results show that our solution provides superior performance compared to polling, at a latency that comes close to hand-written change notifications.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; **Distributed programming languages**; • **Software and its engineering** → *Concurrent programming structures*; Frameworks;

Additional Key Words and Phrases: Services, Distributed Programming, Reactive Programming, Actor Model, Virtual Actors

ACM Reference Format:
Sebastian Burckhardt and Tim Coppieters. 2018. Reactive Caching for Composed Services: Polling at the Speed of Push. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 152 (November 2018), 38 pages. https://doi.org/10.1145/3276522

1 INTRODUCTION

To simplify development and operation, cloud services are often split into component services that offer different functionality and may depend on each other. Some service components provide generic functionality such as durable storage or communication. Others, especially *microservices*, offer an application-specific functionality. Microservices are commonly implemented using actor frameworks [Akka 2016; Armstrong 2010; Chuang et al. 2013; Orbit 2016; Orleans 2016; Sang et al. 2016; SF Reliable Actors 2016] that further partition the service into small, application-defined entities. For example, actors may represent individual user profiles, articles, game sessions, devices, bank accounts, or chat rooms.

Services must often process large volumes of requests, including both external requests and internal requests between components. For example, when a user connects from a browser, a single external service request may in turn issue many more requests among internal component services.

Authors' addresses: Sebastian Burckhardt, Microsoft Research, United States, sburckha@microsoft.com; Tim Coppieters, Vrije Universiteit Brussel, Belgium, coppieters.tim@gmail.com.

2475-1421/2018/11-ART152
https://doi.org/10.1145/3276522

Often, such requests are needed simply to get a cohesive view of application state that is scattered across the micro-services, actors, or partitions.

The challenge we address in this work arises when clients require an *up-to-date* view of the distributed application state, i.e. would like to refresh on changes. The most common solution is to simply repeat the request periodically, i.e. run a polling loop on the client. This is easy to understand and implement. Low-frequency polling is an easy solution, but does not deliver changes quickly. High-frequency polling, on the other hand, produces tremendous extra load on the service, in particular if each request spawns many more internal requests.

To do better, we propose a combination of an efficient push-based caching algorithm with a convenient, polling-like API:

(1) We propose a distributed, fault-tolerant algorithm for maintaining *reactive caches*. A reactive cache maintains a subscription to the data it depends on, so that changes can be *pushed* to the cache.
(2) We provide a client API for reactive polling that resembles a standard polling loop, but is built on top of the reactive caching mechanism, and thus receives push-based change notifications.

Chirper Example. Consider an application where users post messages to their own timeline, and view a timeline containing all messages posted by people they are following. We can implement such a service using two partitioned microservices as shown in Fig. 1, written in a virtual-actor-style imperative pseudocode. Many actor frameworks support the development of microservices in a comparable style [Bernstein et al. 2017; Chuang et al. 2013; Orbit 2016; Orleans 2016; Sang et al. 2016; SF Reliable Actors 2016].

```
1  service Posts partition userid: string
2  {
3    // stores all messages by this user
4    state map⟨time,string⟩;
5    // update operations
6    op Post(t: time, msg: string) {
7      state[t] = msg;
8    }
9    op Unpost(t: time) {
10     state.remove(t);
11   }
12   // read operations
13   op Get(): list⟨pair⟨time,string⟩⟩) {
14     var msgs=new list⟨pair⟨time,string⟩⟩(); 33
15     foreach(var m in state)
16       msgs.add((m.key, m.value));
17     return msgs;
18   }
19 }
```

```
20 service Timeline partition userid: string
21 {
22   // stores set of followed users
23   state set⟨string⟩;
24   // update operations
25   op Follow(id: string) { state.add(id); }
26   op Unfollow(id: string) {state.remove(id);}
27   // read operation
28   op Get(): list⟨pair⟨time,string⟩⟩ {
29     // retrieve posts by this user
30     var msgs = await Posts⟨userid⟩.Get();
31     // incorporate posts of followed users
32     foreach(var f in state) {
34       var fm = await Posts⟨f⟩.Get();
35       // add all posts to the list
36       foreach(var m in fm){msgs.add(m);}
37     }
38     msgs.sort();
39     return msgs;
40   }
41 }
```

Fig. 1. Pseudocode for the Chirper service example.

```
41
42
43  while (interested) {
44    try {
45      var result = Timeline⟨myuserid⟩.Get();
46      display(result);
47      // wait 5 seconds before refresh
48      await delay(5000);
49    } catch(TimeoutException) {
50      display("no response, retrying...");
51    }
52  }
```

```
54  var rp = CreateReactivePoll(
55    // anonymous function
56    () ⇒ Timeline⟨myuserid⟩.Get()
57  )
58  while (interested) {
59    try {
60      var result = await rp.NextResult();
61      display(result);
62    } catch(TimeoutException) {
63      display("no response, retrying...");
64    }
65  }
66  rp.Dispose();
```

Fig. 2. **(a)** left: conventional client-side polling. **(b)** right: proposed reactive poll API.

The Posts service is partitioned by user, with each partition identified by a user-id string. The state of each partition is a collection that maps timestamps to strings, representing all posts by that user. The operations Post and Unpost add or remove posts, respectively, and the operation Get (line 13) returns a list of all posts.

The Timeline service is similarly partitioned by user. The state of each partition is a set of strings, which represent the users followed by this user (line 23). The operations Follow and Unfollow modify that set. The operation Get returns a combined, sorted list of all messages posted by this user and all followed users. First, it makes a remote call to the Posts⟨userid⟩ service partition, to retrieve all posts by this user (line 30). Then, it runs a loop to retrieve the messages of all followed users, adding them to the list. Finally it sorts and returns the list (line 37, line 38). We illustrate a distributed execution of Timeline.Get in Fig. 3.

Polling. Consider a client that wants to refresh the user display whenever the timeline changes (as a consequence of any post, unpost, follow, or unfollow operations). A straightforward polling-based solution is shown in Fig. 2a on the left. It repeatedly queries the timeline (line 45) and updates the display (line 46), then waits for some fixed time interval (line 47).

Polling is easy to understand, and handles failures gracefully (line 50). However, choosing a satisfactory polling interval is not always possible. Infrequent polling means the displayed result can lag significantly behind the current state; but frequent polling dramatically increases the load on the service, requiring us to pay for more servers to keep up. We demonstrate this effect experimentally in section 5.

Reactive Polling API. To take advantage of our new mechanism, developers replace the polling loop in Fig. 2a with the code shown in Fig. 2b on the right. It first creates a ReactivePoll object (line 54), passing the request to execute as a lambda (line 56). Inside the loop, we repeatedly call NextResult (line 60) to get the first and successive results, and display them. When called in the first iteration of the loop, **await** rp.NextResult() behaves just like a normal, asynchronous request; but under the covers, the runtime now tracks what data the request depends on and maintains a connection to it. When called again in subsequent iterations of the loop, **await** rp.NextResult() awaits until the result is actually *different from the last returned result.* Therefore, it is no longer necessary to add a time delay into each iteration - successive loop iterations are now triggered only

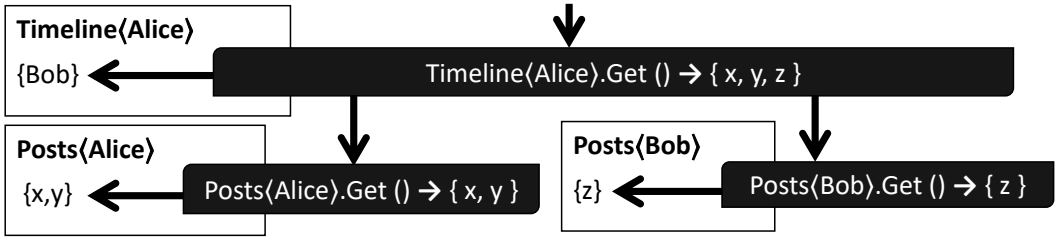

Fig. 3. Illustration of a distributed execution of Timeline.Get, summaries, and dependencies.

by actually differing results, as pushed by the reactive caching mechanism under the hood. When no longer needed, the code must dispose the ReactivePoll object (line 66) to release resources for the tracking. [1]

Actor Framework Integration. While our idea and algorithm are not tied to any particular framework, our implementation uses a virtual actor framework [Bernstein et al. 2014; Orleans 2016] to demonstrate how reactive caching can be provided *automatically* without changing the service code. Note that some actor frameworks, including virtual actors, have altered the basic actor model to emphasize a service-nature of actors. For example, they offer a type-checked RPC interface instead of just one-way messages, and favor stable addresses over dynamic creation and garbage collection. Building a micro-service from virtual actors is thus very similar to building a service from micro-services, but with extra convenience and performance. In particular, the framework can provide automatic runtime support for failure detection and recovery, and for load balancing over elastic clusters.

Alternative Solutions. It is possible to keep clients up-to-date using alternative solutions, but they require significant changes to the original service code (Fig. 1). The *observer pattern* is a common ad-hoc solution. It means that the programmer explicitly adds state, operations, and code to track dependencies and propagate changes at the application level. This can quickly become complex. Even for non-distributed programs, the observer pattern is error-prone [Salvaneschi et al. 2014] and difficult to maintain.

Another common approach is to use *streams* [Orleans 2016; Reactors.IO 2016], or to use *publish-subscribe services* [Eugster et al. 2003]. These allow elegant expression and efficient execution of dataflow computations, and are extremely convenient for many workloads, in particular for analytics. However, they are not a panacea. Handling applications with dynamically changing dependencies can be challenging, e.g. (1) applications that mix state- and event-based paradigms, e.g. want to first read the state but then also subscribe to changes; (2) applications that depend on state dispersed over multiple components in a manner that is nontrivial to convert into a functional join expression; or (3) subscriptions that require data-dependent dynamic switching of streams or topics. In such applications, concurrency- or failure-induced bugs are hard to avoid. Reactive caching solves this problem for the developer: it constructs and maintains a dynamic dataflow graph automatically by tracking dependencies at runtime.

Failures. Failures are a normal occurrence when operating services in the cloud. If using a framework, it will typically detect a failed partition and recover it by restarting it and recovering

[1] Our C# implementation supports the IDisposable pattern, so the application can employ a using clause to automatically dispose the ReactivePoll object when control exits the scope.

its state. However, this process is not fully transparent, as some state may be lost, in particular in-progress requests. Also, requests that update multiple partitions can fail non-atomically. The Chirper application handles failures via idempotence: it expects clients to retry any update operations (Post, Unpost, Follow, or Unfollow) that do not complete successfully.

Note that ReactivePoll objects do not require any special consideration of failures by the developer, since they are built on top of a fault-tolerant reactive caching algorithm.

Recursive Summaries. Our algorithm caches *summaries* that record (1) the result of executing an operation, and (2) what that result depends on. When computing a summary at some service partition, it can call remote operations, which are also summarized. This execution leads to a dependency tree (Fig. 3). Note that dependencies can be detected dynamically during execution, e.g. by instrumenting service calls; there is no requirement for a static analysis. The summaries are shown as black boxes in Fig. 3.

Eventual Consistency. We call a summary *up-to-date* if it reflects an execution of the operation on a snapshot of the current global state. Our algorithm guarantees that any summary that is not up-to-date is eventually recomputed. In particular, any final result is snapshot-consistent. Moreover, because we maintain a dependency graph and immediately propagate changes along its edges, the change propagation happens at the speed of push: it does not have to wait for some arbitrary timer interval to expire, as with polling.

Incremental Repair. To refresh a summary that is not up-to-date, we recompute the operation. During recomputation, we elide nested operation calls for which we have a summary and instead use the cached result. After recomputing a summary, we propagate the new result to dependent summaries *only if the new result is different than the old one*. This means that the summary tree (Fig. 3) is not simply replaced, but is repaired using an incremental and parallel bottom-up process.

Inconsistent Views. Even in a standard polling loop, an operation that reads from more than one partition (e.g. Timeline.Get) is not guaranteed to see a globally consistent snapshot. Developers must thus take care to write such operations robustly. Still, the reads are at least *sequentially consistent*. For example, if an operation reads from partition A first, then from partition B, what it reads from B is at least as fresh as what it read from A.

Consistency Tradeoff. When using reactive caching, the consistency guarantees are *weaker for intermediate results, but stronger for final results*: Intermediate results may be not sequentially consistent, but final results are guaranteed to be snapshot consistent. As intermediate results are recomputed quickly, reactive caching is particularly useful for applications where rapid refresh is more important than the consistency of intermediate results.

1.1 Contributions

Overall, we make the following contributions.

Polling API. We propose an API to replace polling loops (Fig. 2a) with a polling-abstraction (Fig. 2b) that can be optimized under the hood.

Service Calculus. As a foundational contribution, we present a minimal core calculus that models composed, partitioned services communicating asynchronously (§2). This calculus provides a concise reference and means of comparison for virtual actors and related middleware programming models [Chuang et al. 2013; Orbit 2016; Orleans 2016; Sang et al. 2016; SF Reliable Actors 2016]. It also enables us to describe our algorithm precisely. We include a type system, and prove progress and preservation theorems.

Reactive Caching Algorithm. We present a novel distributed, fault-tolerant reactive caching algorithm (§3). By using a distributed, bipartite dependence graph of summaries and caches, it avoids a central point of contention or failure, and can scale elastically. For reference, we include a complete formalization of the algorithm in appendix §C.

Implementation. We have implemented the algorithm as an extension of the Orleans virtual actor runtime [Orleans 2016]. Since the latter already provides failure detection and elasticity, reactive caching is *fully automatic*. In particular, developers who wish to use it do not have to make *any changes* to the service code. The only required change is on the client, replacing polling loops with reactive polling objects (i.e. refactor Fig. 2a to Fig. 2b).

Evaluation. We provide a performance evaluation (§5) in the Orleans context, on two benchmarks. It demonstrates that the propagation latency when using reactive caching is much better than when using polling, and that the throughput when using reactive caching is better than when polling with high frequency.

2 FORMULATION

To describe the problem and our solution precisely, we now define a minimal core calculus for composed services. It models partitioned services, asynchronous communication, and failures, by extending the simply typed lambda calculus with imperative features and failure transitions.

All code executes within the context of a *service partition* $G\langle v\rangle$, each identified by a *partition key* v. A service partition's behavior is defined by a *service definition* G, which defines the partition key type, the type of the internal state, an initial state, and the public operations. For example, we previously showed informal service definitions for Timeline and Posts in Fig. 1, and service partitions Timeline⟨Alice⟩, Posts⟨Alice⟩ and Posts⟨Bob⟩ in Fig. 3.

Unlike objects or conventional actors, service partitions are not created or deleted by the program, but exist perpetually, just like virtual actors [Bernstein et al. 2014]. In fact, service partitions and service definitions are the semantic equivalent of Orleans grain instances and grain classes [Bernstein et al. 2014] that are marked with a [Reentrant] attribute. Hence our use of the letters g and G to represent them.

2.1 Syntax

We show the basic syntax in Fig. 4a on the left. We use a standard call-by-value evaluation, with a minimal syntax based on variables x, functions $\lambda x : t.e$, values v, expressions e, and function application $e\ e'$. The remaining syntax extends this core calculus with types and imperative features as usual (see [Pierce 2002] for an extensive treatment of the topic). For example, we can define syntactic sugar for let-expressions ($\textbf{let}\ x = e\ \textbf{in}\ e') \equiv ((\lambda x : t.e')\ e)$ and sequential composition $(e;\ e') \equiv \textbf{let}\ x = e\ \textbf{in}\ e'$ (where x not free in e').

We include the unit constant () and the *unit* type, but we elide other basic types or constants for simplicity. We distinguish between serializable types s, and general types t. The latter can include function types and task types *task* s which represent the future result of a remote operation call. Non-serializable values cannot be stored in the state, nor sent in messages. Function types $t \rightarrow_\mu t'$ include an effect μ which is either r or u, for read-only or update.

A service definition G is of the form $\langle s\rangle\ v_0\ \overline{(o : f)}$. The type s specifies the partition key type (we can use *unit* for non-partitioned services). The value v_0 defines the initial state of the service partition. The list $\overline{(o : f)}$ is comprised of definitions $(o : f)$ containing an operation name o and a

function f, the implementation. Operation names are distinguished into read-only operations o_r and update operations o_u.[2]

Each service partition has locally mutable state, similar to a mutable reference, and can make asynchronous calls to another partition (of the same or a different service). We extend the expression syntax accordingly. The expression **key** reads the key of the service partition on which the code is executing. The expressions **get** and (**set** e) read and update the state, respectively. The expression $G\langle e\rangle.o(e')$ performs an asynchronous operation call to a remote service partition $G\langle e\rangle$, where e is an expression that evaluates to the partition key, o is the name of the operation, and e' is an expression that evaluates to the argument to be passed to the operation. A remote call immediately returns a special value, a placeholder p of type *task* s, which is commonly called a *future* [Flanagan and Felleisen 2001; Moreau 1970]. Thus, multiple remote calls can be simultaneously in progress. For an expression e of type *task* s, the expression **await** e waits for the placeholder to resolve and returns the result.

For example, we can perform two sequential calls as

$$\textbf{await}\ G_1\langle v_1\rangle.o_1();\ \ \textbf{await}\ G_2\langle v_2\rangle.o_2().$$

Or, we can execute the same two calls in parallel as

$$\textbf{let}\ x = G_1\langle v_1\rangle.o_1()\ \textbf{in}\ (\textbf{let}\ y = G_2\langle v_2\rangle.o_2()\ \textbf{in}\ (\textbf{await}\ x; \textbf{await}\ y)).$$

Parallelizing calls within an operation can often improve latency significantly; for example, we can rewrite the code in Fig. 1 to perform the calls on line 33 in parallel.

[2]Note that it is in fact common practice to include informal, un-enforced purity information in service APIs; for example, REST services use the GET verb to identify read-only operations and enable HTTP caching, and POST or PUT verbs for identifying operations that may mutate service state.

x	::=	$\ldots$	(variable)
c	::=	$() \mid \ldots$	(constant)
f	::=	$\lambda x{:}t.e$	(function)
v	::=	$c \mid x \mid f$	(value)
		$\mid p \mid \textbf{done}\ v$	
s	::=	$unit \mid \ldots$	(serializable type)
t	::=	$s \mid task\ s$	(general type)
		$\mid t \to_\mu t'$	
μ	::=	$r \mid u$	(effect)
e	::=	$v \mid e\ e'$	(expression)
		$\mid \textbf{key}$	(read my key)
		$\mid \textbf{get} \mid \textbf{set}\ e$	(read/write state)
		$\mid G\langle e\rangle.o(e')$	(operation call)
		$\mid \textbf{await}\ e$	(wait for response)
G	::=	$\langle s\rangle\ v_0\ \overline{(o:f)}$	(service definition)
o	::=	$o_r \mid o_u$	(operation name)
o_r	::=	$\ldots$	(read-only operation)
o_u	::=	$\ldots$	(update operation)

$E[\circ]$	::=	$\circ$	(execution context)
		$\mid \textbf{set}\ E[\circ]$	
		$\mid \textbf{await}\ E[\circ]$	
		$\mid E[\circ]\ e \mid v\ E[\circ]$	
		$\mid G\langle E[\circ]\rangle.o(e) \mid G\langle v\rangle.o(E[\circ])$	
p	::=	$g.i/g'.o$	(placeholder)
g	::=	$G\langle v\rangle$	(service partition)
i	::=	$\ldots$	(unique identifier)
m	::=	$call\ r\ v$	(call message)
		$\mid rsp\ r\ v$	(response message)
r	::=	$g.i/g'.o$	(service call)
		$\mid i/g.o$	(external call)
T	::=	$\overline{(r:e)}$	(service task pool)
R	::=	$\sigma\ T$	(service state)
S	::=	$\overline{(g:R)}$	(service collection)
N	::=	$\overline{m}$	(network)

Fig. 4. **(a)** left: Syntax of the service calculus. **(b)** right: Syntax for defining the operational semantics.

$\boxed{e, \sigma, N \ \rightarrow_g \ e', \sigma', N'}$ **Local Step**

$$\text{App}\ \frac{}{E[\lambda x{:}t.e \ \ v], \sigma, N \ \rightarrow_g \ E[e[v/x]], \sigma, N}$$

$$\text{Get}\ \frac{}{E[\mathbf{get}], \sigma, N \ \rightarrow_g \ E[\sigma], \sigma, N} \qquad \text{Set}\ \frac{}{E[\mathbf{set}\ v], \sigma, N \ \rightarrow_g \ E[()], v, N}$$

$$\text{Key}\ \frac{g = G\langle v \rangle}{E[\mathbf{key}], \sigma, N \ \rightarrow_g \ E[v], \sigma, N} \qquad \text{Resolve}\ \frac{}{E[\mathbf{await\ done}\ v], \sigma, N \ \rightarrow_g \ E[v], \sigma, N}$$

$$\text{Call}\ \frac{i \text{ fresh} \qquad r = g.i/g'.o}{E[g'.o(v)], \sigma, N \ \rightarrow_g \ E[r], \sigma, (\mathit{call}\ r\ v)\ N}$$

Fig. 5. Operational semantics: Local steps.

2.2 Semantics

We define the operational semantics using a combination of local steps (which evaluate expressions locally at a service partition) and system steps (which change the state of the entire system). An expression can take a local step if it contains a redex (reducible subexpression). To define which subexpressions can be evaluated, and in what order, we define execution contexts $E[\circ]$ as shown in Fig. 4b on the top right. This definition ensures call-by-value semantics, and for operation calls, it ensures we first evaluate the partition key, and then the operation argument, before calling the operation.

Local Steps. Local steps take place in the context of a particular service partition $g = G\langle v \rangle$. They are defined by a g-local step relation $e, \sigma, N \ \rightarrow_g \ e', \sigma', N'$ as defined by the six rules in Fig. 5, which use the syntax in Figs. 4a and 4b. Note that the service partition g is a suffix to the arrow. Besides transforming the expression e, a local step can also read or update the local state σ, and append messages to the list of outgoing messages N. The (App) rule represents function application as usual and has no other effects. The rule (Key) reads the partition key determined by g. The rules (Get) and (Set) read and update the partition state σ, respectively. The rule (Call) performs an asynchronous operation call. For tracking the resolution of this request both locally and globally, it creates a request identifier $r = g.i/g'.o$, where i is a globally unique identifier. This identifier r is used to construct a request message $(\mathit{call}\ r\ v)$ which is added to N. The same identifier r is also returned to the application (i.e. inserted into the evaluation context) as a placeholder for the final result. As we shall discuss in the next section, a system step replaces placeholders with a final result of the form **done** v when processing a response message for the corresponding request. Thereafter, an expression that awaits this result can resume with that value, as shown in the rule (Resolve).

System Steps. A global system configuration is of the form $S \mid N$, where S defines the state of all service partitions, and N defines the state of the network. System steps are of the form $S \mid N \ \rightarrow \ S' \mid N'$, and are shown in Fig. 6, using the syntax in Figs. 4a and 4b.

The network state N is simply an unordered set of messages m of the form $(\mathit{call}\ r\ v)$ or $(\mathit{rsp}\ r\ v)$. The request identifier r identifies caller and callee, and thus determines the routing. There are no ordering guarantees for message delivery. Requests can also be received from the outside; in fact, there is nothing to evaluate without at least one such external request to start things off. The system step (ExtCall) models an incoming external call (note that the prerequisite $\vdash g : G$ ensures the request targets a well-typed service definition G) and (ExtRsp) models an outgoing response.

We allow infinitely many service partitions and infinite executions. However, at any point of an execution, only finitely many partitions have been active so far, and all others are still in their

$\boxed{S \mid N \;\rightarrow\; S' \mid N'}$ **System Step**

$$\text{ExtCall}\ \frac{i \text{ fresh} \qquad \vdash \mathit{call}\ r\ v \qquad r = i/g.o}{S \mid N \;\rightarrow\; S \mid (\mathit{call}\ r\ v)\ N} \qquad \text{ExtRsp}\ \frac{r = i/g.o}{S \mid (\mathit{rsp}\ r\ v)\ N \;\rightarrow\; S \mid N}$$

$$\text{Activate}\ \frac{g \notin \mathit{dom}\ S \qquad \vdash g : G \qquad G = \ldots v_0 \ldots}{S \mid N \;\rightarrow\; (g{:}v_0)\ S \mid N}$$

$$\text{RcvCall}\ \frac{r = \ldots/g.o \qquad g = G\langle v' \rangle \qquad G = \ldots (o{:}f)\ \ldots}{(g{:}\sigma\ T)\ S \mid (\mathit{call}\ r\ v)\ N \;\rightarrow\; (g{:}\sigma\ (r{:}f\ v)\ T)\ S \mid N}$$

$$\text{RcvRsp}\ \frac{r = g.i/g'.o \qquad T' = T[(\textbf{done}\ v)/r]}{(g{:}\sigma\ T)\ S \mid (\mathit{rsp}\ r\ v)\ N \;\rightarrow\; (g{:}\sigma\ T')\ S \mid N}$$

$$\text{CompleteReq}\ \frac{e, \sigma, N \;\rightarrow_g^+\; e', \sigma', N' \qquad e' = v}{(g{:}\sigma\ (r{:}e)\ T)\ S \mid N \;\rightarrow\; (g{:}\sigma'\ T)\ S \mid (\mathit{rsp}\ r\ v)\ N'}$$

$$\text{TakeTurn}\ \frac{e, \sigma, N \;\rightarrow_g^+\; e', \sigma', N' \qquad e' = E[\textbf{await}\ p]}{(g{:}\sigma\ (r{:}e)\ T)\ S \mid N \;\rightarrow\; (g{:}\sigma'\ (r{:}e')\ T)\ S \mid N'}$$

Fig. 6. Operational semantics: System Steps.

initial state. Therefore, we can represent the system state S as a finite map $\overline{(g{:}R)}$ to store the state R for each accessed service partition g. The state R is of the form $\sigma\ T$, where σ is the user-defined partition state, and T is a task pool which contains tuples $(r : e)$ where r is the identifier of the request that spawned the task, and the expression e is the current evaluation state of that task. As needed, the system step (Activate) can add a not-yet-accessed service partition to this map, in its initial state with an empty task pool.

The rule (RcvCall) models the processing of a call message $(\mathit{call}\ r\ v)$ at a service partition g. It retrieves the corresponding operation definition $(o{:}f)$ from the class definition G and adds a new entry $(r{:}(f\ v))$ to the task pool, where $(f\ v)$, is the application of the function that implements this operation to the operation argument received in the message.

The task pool can contain placeholders p for results of calls made. These match the request identifier r in the messages. In rule (RcvRsp) shows how a response $(\mathit{rsp}\ r\ v)$ is processed by replacing all occurrences of the placeholder r in any expression in the task pool T with **done** v.

The remaining two rules work on an existing entry $(r : e)$ in the task pool. They perform one or more local evaluation steps $e, \sigma, N \;\rightarrow_g^+\; e', \sigma', N'$. If e evaluates all the way to a value $e' = v$, the rule (CompleteReq) applies; it removes the entry $(r : e)$ from the task pool and adds a response message $(\mathit{rsp}\ r\ v)$ to the network. If e evaluates to an expression of the form $e' = E[\textbf{await}\ p]$, i.e. gets stuck waiting for the placeholder p to resolve, then the rule (TakeTurn) applies: it leaves e' in the task pool for future consideration (once the placeholder is replaced). As we shall see later, any well-typed expression that cannot take a local step is either a value v, or of the form $E[\textbf{await}\ p]$, thus the two rules (TakeTurn) and (CompleteReq) cover all maximal local executions.

Failure Steps. We model four types of failures using nondeterministic system steps (Fig. 7). The step (LoseMessage) represents a message being lost. (LosePartition) represents the entire state of a service partition being lost. Note that since services live forever, this is effectively a "restart" from

Failure System Steps

$$\text{LoseMessage}\ \frac{}{S \mid m\ N\ \rightarrow\ S \mid N}$$

$$\text{LosePartition}\ \frac{g = G\langle v' \rangle}{(g{:}\sigma\ T)\ S \mid N\ \rightarrow\ S \mid N} \qquad \text{LoseTasks}\ \frac{g = G\langle v' \rangle}{(g{:}\sigma\ T)\ S \mid N\ \rightarrow\ (g{:}\sigma)\ S \mid N}$$

$$\text{TimeOut}\ \frac{}{(g{:}\sigma\ (r{:}E[\textbf{await}\ p])\ T)\ S \mid N\ \rightarrow\ (g{:}\sigma\ T)\ S \mid N}$$

Fig. 7. Operational semantics: System Steps that represent Failures.

the initial state. (LoseTasks) represents all tasks being lost, but the state surviving; this is typical for many stateful, persistent services (e.g. storage), that can save and recover the durable state σ, but may still lose in-progress requests. (TimeOut) models a request timing out before the response arrives; it simply drops the task from the task pool without sending a response (which, in turn, causes waiting callers to time out).

Executions. We can now define executions as a sequence of steps starting in the initial state. Note that we include infinite executions in the formalization, which is necessary for stating liveness properties such as eventual consistency [Burckhardt 2014].

Definition 2.1. An *execution* X is a finite or infinite alternating sequence of system states and transitions $s_0\ t_0\ s_1\ t_1 ...$, such that (1) each $s_i = S_i \mid N_i$ is a system configuration, (2) s_0 is the empty configuration, and (3) each t_i is a derivation of $S_i \mid N_i\ \rightarrow\ S_{i+1} \mid N_{i+1}$.

For notational convenience, given an execution X, we let $X[0..n]$ denote the prefix execution consisting of the first $n+1$ states and n transitions. Also, we use square brackets to index states $X[i] = s_i$ and parentheses to index transitions $X(i) = t_i$.

2.3 Type System

The type system plays an important role: it ensures that operation arguments, return values, partition states, and partition keys are well-typed and serializable. Moreover, it ensures that distributed read-only operations do not modify the state of any of the service partitions. As a validation of the service calculus, we now prove progress and preservation theorems.

All the type rules are shown in Fig. 8, organized into groups that apply to expression types, service definitions, or system configurations, respectively. For typing expressions, we use a type judgment of the form $\Gamma \vdash_\mu e : t$, where μ is an effect (either r for read-only or u for update). The typing environment Γ is of the form $\overline{(x{:}t)}\ G?$, containing variable typings $(x{:}t)$, and optionally a service definition G, if typing expressions within the context of a particular service.

The rules (TVar), (TAbs) and (TApp) rules are standard, and (TUnit) is obvious. (TAwait) and (TDone) are straightforward, and require or produce task types, respectively. (TOpCall) checks that the class G' is well typed, that the key and argument types match, and that we are not calling an update operation from a read-only effect context. (TPlaceholder) types a placeholder returned by an operation call. The rules (TKey), (TGet), and (TSet) are specific to a service context: thus, the typing context Γ must contain a service definition G, which determines the type of the key and the type of the partition state. The (TSet) operation requires an update effect context.

The service definition type rules (TOperation), (TDefinition) enforce that each operation definition is properly typed for the context G of the partition that contains them, for the correct effect μ as indicated by the operation name (o_r or o_u), and also enforces that only serializable types appear in any argument type, return type, key type, or state type.

$\boxed{\Gamma \vdash_\mu e : t}$ **Expression Typing**

$$\text{TVar}\frac{}{(x:t)\,\Gamma \vdash_\mu x : t} \qquad \text{TAbs}\frac{(x:t)\,\Gamma \vdash_\mu e : t'}{\Gamma \vdash_\mu (\lambda x{:}t.e) : t \to_\mu t'} \qquad \text{TApp}\frac{\Gamma \vdash_\mu e : t \to_\mu t' \quad \Gamma \vdash_\mu e' : t}{\Gamma \vdash_\mu (e\ e') : t'}$$

$$\text{TUnit}\frac{}{\Gamma \vdash_\mu () : \mathit{unit}} \qquad \text{TAwait}\frac{\Gamma \vdash_\mu e : \mathit{task}\ s}{\Gamma \vdash_\mu \textbf{await}\ e : s} \qquad \text{TDone}\frac{\Gamma \vdash_\mu v : s}{\Gamma \vdash_\mu \textbf{done}\ v : \mathit{task}\ s}$$

$$\text{TOpCall}\frac{\vdash G' : s_k, s_\sigma \qquad G' \vdash_\nu (o_\nu : f) : s_a \to_\nu s_r \qquad \Gamma \vdash_\mu e : s_k \qquad \Gamma \vdash_\mu e' : s_a \qquad \nu \le \mu}{\Gamma \vdash_\mu G'\langle e\rangle.o_\nu(e') : \mathit{task}\ s_r}$$

$$\text{TPlaceholder}\frac{\vdash g : G \qquad \vdash g' : G' \qquad G' \vdash_\nu (o_\nu : f) : s_a \to_\nu s_r \qquad \nu \le \mu}{\Gamma\ G \vdash_\mu (g/g'.o_\nu) : \mathit{task}\ s_r}$$

$$\text{TKey}\frac{\vdash G : s_k, s_\sigma}{\Gamma\ G \vdash_\mu \textbf{key} : s_k} \qquad \text{TGet}\frac{\vdash G : s_k, s_\sigma}{\Gamma\ G \vdash_\mu \textbf{get} : s_\sigma} \qquad \text{TSet}\frac{\vdash G : s_k, s_\sigma \qquad \Gamma\ G \vdash_u e : s_\sigma}{\Gamma\ G \vdash_u \textbf{set}\ e : \mathit{unit}}$$

Service Definition Typing

$$\text{TOperation}\frac{G \vdash_\mu f : s \to_\mu s'}{G \vdash (o_\mu : f) : s \to_\mu s'}$$

$$\text{TDefinition}\frac{G = \langle s_k\rangle\ a\ \sigma\ D \qquad \vdash \sigma : s_\sigma \qquad \forall (o_\mu : f) \in D : (G \vdash_\mu (o_\mu : f) : \_)}{\vdash G : s_k, s_\sigma}$$

$\boxed{\vdash S \mid N}$ **Configuration Typing**

$$\text{TConfiguration}\frac{\forall (g:R) \in S : (\vdash (g:R)) \qquad \vdash N}{\vdash S \mid N} \qquad \text{TMessages}\frac{\forall m \in N : (\vdash m)}{\vdash N}$$

$$\text{TPartition}\frac{\vdash G : s_k, s_\sigma \qquad \vdash_\mu v : s_k}{\vdash G\langle v\rangle : G} \qquad \text{TCallMsg}\frac{\vdash g : G \qquad G \vdash (o:f) : s \to s' \qquad \vdash_\mu v : s}{\vdash \mathit{call}\ (\_/g.o)\ v}$$

$$\text{TRspMsg}\frac{\vdash g : G \qquad G \vdash (o:f) : s \to s' \qquad \vdash_\mu v : s'}{\vdash \mathit{rsp}\ (\_/g.o)\ v}$$

$$\text{TPartitionData}\frac{\vdash g : G \qquad \vdash G : s_k, s_\sigma \qquad \vdash_\mu \sigma : s_\sigma \qquad \forall (r:e) \in T : (G \vdash (r:e))}{\vdash (g:\sigma\ T)}$$

$$\text{TTask}\frac{\vdash g : G \qquad G \vdash (o_\mu : f) : s \to s' \qquad G \vdash_\mu e : s'}{G \vdash ((\_/g.o_\mu):e)}$$

$$\text{TLocalState}\frac{\vdash g : G \qquad G \vdash_\mu e : t_e \qquad \vdash_\mu G : s_k, s_\sigma \qquad \vdash \sigma : s_\sigma \qquad \vdash N}{\vdash_\mu (g, e, \sigma, N) : t_e}$$

Fig. 8. Typing judgments for expressions, service definitions, and system configurations.

The system configuration type rules ensure that all pieces (including messages, partition records, and task pools) are well-typed. Note that we take some notational shortcuts with empty collections, which are implicitly well-typed.

Progress and Preservation. The last rule in Fig. 8 types tuples (g, e, σ, N) and is used to concisely state the following preservation and progress guarantees. The proofs are included in §A. Local preservation means that local steps do preserve the type of an expression, the local state, and emit only well-typed messages:

Theorem 2.2 (Local Preservation). *If* $\vdash_\mu (g, e, \sigma, N) : t$, *and* $e, \sigma, N \rightarrow_g e', \sigma', N'$, *then* $\vdash_\mu (g, e', \sigma', N') : t$.

Local progress means that each task in the task pool of a partition has either completed its evaluation, is waiting for a response from a remote call, or can take a local step:

Theorem 2.3 (Local Progress). *If* $\vdash_\mu (g, e, \sigma, N) : t$, *then exactly one of:*

(1) $e = v$ *for some value* v
(2) $e = E[\textbf{\textit{await}}\ p]$ *for some placeholder* p
(3) *there exist* e', σ', N' *such that* $e, \sigma, N \rightarrow_g e', \sigma', N'$.

Global preservation means that all of the state in the system (including all messages, all states, and all expressions in all task pools) remains well-typed during execution:

Theorem 2.4 (Global Preservation). *If* $\vdash S \mid N$ *and* $S \mid N \rightarrow S' \mid N'$, *then* $\vdash S' \mid N'$.

Global progress is also guaranteed, despite divergence of local tasks, lost messages, or failed partitions. It is easy to see (hence we do not formulate a theorem) that a configuration $S|N$ can always take a system step, either adding new requests, losing messages, losing partitions, or losing tasks and/or time them out. Interestingly, timeouts thereby "solve" the problem of various forms of divergence, including divergence of local computations.

2.4 Reactive Caching

We cache results of distributed read-only operations using *summaries*. Formally, a summary is a tuple (g, o_r, v_a, v_r) where g is the service partition called, o_r is the operation name, v_a is the argument, and v_r is the value returned.

Example. Consider again the Timeline.Get operation of the Chirper application in Fig. 3. Its depicted system configuration is

$$S = ((\text{Timeline}\langle\text{Alice}\rangle : \{\text{Bob}\}) \quad (\text{Posts}\langle\text{Alice}\rangle : \{x, y\}) \quad (\text{Posts}\langle\text{Bob}\rangle : \{z\})$$

and the summaries represented by the black boxes are

$$(\text{Timeline}\langle\text{Alice}\rangle, \text{Get}, (), \{x, y\}) \quad (\text{Posts}\langle\text{Alice}\rangle, \text{Get}, (), \{x, y\}) \quad (\text{Posts}\langle\text{Bob}\rangle, \text{Get}, (), \{z\})$$

Intuitively, we consider a summary to be up-to-date if its result is a possible result of executing the operation on a snapshot of the current global state. Formally:

Definition 2.5. Given an execution X and position n, we say the summary (g, o_r, v_a, v_r) is *up-to-date at* n if there exists an execution Y such that

- $Y[0..n] = X[0..n]$
- $Y(n)$ is the (ExtCall) rule for some request id r of the form $i/g.o_r$, and with $v = v_a$
- for all $n' > n$, $Y(n')$ is not an (ExtCall) rule
- for some $n' > n$, $Y(n')$ is an (ExtRsp) rule with matching r and $v = v_r$

Our reactive caching algorithm guarantees that all summary caches are eventually up-to-date. This implies that if there are no more changes, caches stabilize to a final value that is snapshot consistent. We formalize the syntax and the algorithm in appendix C, which is included with the complete version of this paper.

3 DISTRIBUTED ALGORITHM

To make reactive caching practical for distributed systems, we need to tolerate faults, provide elasticity, and provide good performance despite large communication latencies. It is not immediately clear how to do so. For example, a centralized solution for tracking summaries and their dependencies creates a central point of congestion and failure. In contrast, we developed a fully decentralized and fault-tolerant algorithm.

The key idea is to combine summary trees with *summary caching*: rather than letting a summary depend directly on other summaries that may be located on other partitions, summaries can depend only on *local caches* of summaries. This allows us to handle faults, and it greatly improves the performance of recomputing summaries since all their dependencies are stored in local caches. It also allows summary caches to be shared by co-located service partitions. We now explain the mechanism in detail; see the complete version of this paper for a formalization.

3.1 Bipartite Dependence Graph

We maintain two relations between summaries and caches: summaries depend on local caches, and local caches subscribe to remote summaries. Together, these relations form a bipartite dependence graph as shown in Fig. 9. The subscription and dependency relations are maintained and used as follows.

Summary Subscriptions. Each cache is subscribed to the remote summary that it is representing (solid bidirectional arrows). The summary maintains a list of these subscriptions; whenever the summary changes after an execution (i.e. produces a different result than before), this new result is pushed to all the subscribed caches.

Summary Computation. When a summary is computed or recomputed, the computation executes in a special mode. All operation calls are intercepted instead of being executed normally, and the result is instead looked up in the local cache. If found, execution continues immediately with the cached result. If not found, a new cache entry is created, subscribed to the remote summary, and filled in when the result arrives, at which time execution continues.

Cache Dependencies. For each cache, the algorithm tracks all the local summaries that depend on this cache (dotted arrows), i.e. used this cache during execution. Whenever a cache receives a new, different result from the summary it subscribes to, it invalidates all the dependent summaries, and schedules them for recomputation.

With reactive caching in place, support for reactive polling is straightforward. We treat the passed-in lambda like a summary, and (re-)compute it like a summary to obtain the first, as well as successive results.

3.2 Change Propagation

The reactive caching algorithm propagates changes incrementally via invalidation and lazy recomputation of summaries. Whenever the state of a service partition changes, all of its summaries are marked invalid. After recomputation, if a summary still has the same output, no further action is required. Otherwise, the changed summary pushes the new result to its subscribed caches. Those caches then invalidate their dependent summaries, and so on.

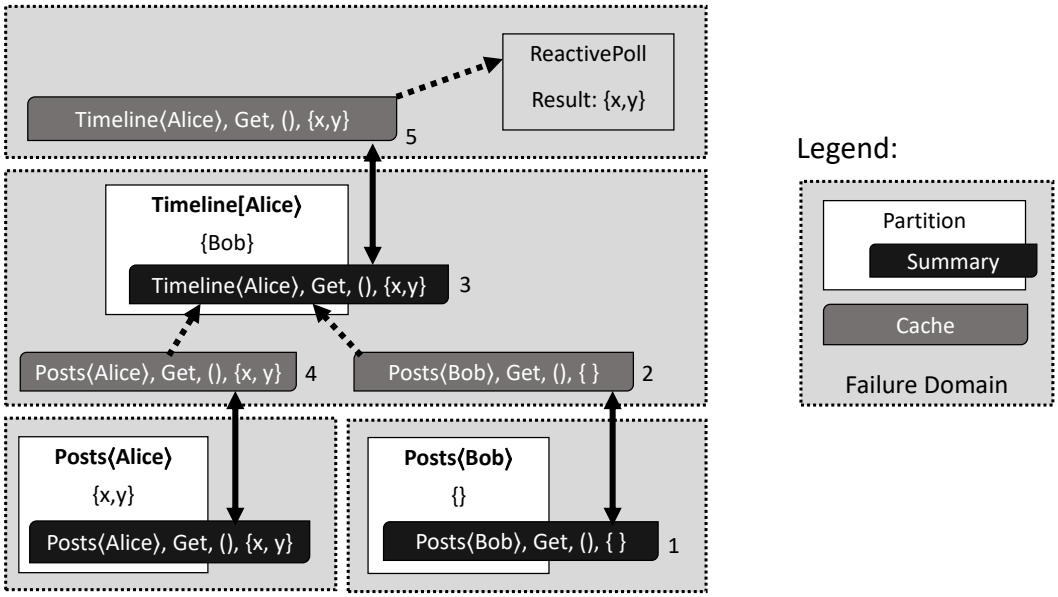

Fig. 9. Illustration of a bipartite dependency graph of summaries and caches. The numbers are used for describing update propgation in §3.2.

Our dependency tracking ensures that whenever a cached summary is not up-to-date, meaning that it is not consistent with a global snapshot of the current state (Def. 2.5), then it is replaced "at the speed of push", i.e. at the speed at which changes are propagated upward the dependency tree. In the typical case where a local recomputation has negligible latency compared to messages between nodes, the propagation latency is approximately (depth of summary tree * message latency).

For example, consider the situation in Fig. 9, and assume Bob posts a new message z, which invalidates summary 1. Then, it takes a total of only *two messages* to propagate all changes and refresh all summaries. We now walk through this in detail. After recomputing summary 1, it returns a new result {z}, which is sent to the subscribed cache 2 (this is the first message). Cache 2 then invalidates the dependent summary 3. When summary 3 recomputes, it can do so locally and quickly, because it reads from the local caches 4 and 2 instead of making remote calls. The result is {x,y,z} which is different than before, so summary 3 sends this new result to the subscribed cache 5 (this is the second message). Cache 5 then invalidates the reactive computation, which recomputes and returns {x,y,z} to the code awaiting NextResult().

Minimizing the number of messages is important because message transport is often a dominating contributor to both latency (because of slow communication between machines) and CPU consumption (because of costly serialization and deserialization).

3.3 Fault Tolerance

To achieve fault tolerance, we exploit that colocated components fail together. For example, if a service partition fails, then its summaries and local caches fail too. We express this using logical failure domains, shown as light gray boxes in Fig. 9. The failure domain at the top resides on a client, where the reactive computation is used. All components within a failure domain fail together.

The key observation is that there is now only one relation that crosses failure domains: the subscription relation (solid bidirectional arrows in Fig. 9). Therefore, fault tolerance can be guaranteed simply by repairing one-sided failures of these subscription edges. In particular, no state needs to be persisted:

- Suppose a cache fails, and the summary detects a dangling subscription. This can be easily repaired by removing the subscription — there is no subscriber listening to updates any more.
- Suppose a summary fails. Then, the cache detects the dangling subscription.[3] In that case, we can simply send a new subscription request, which is delivered to the failed service partition after it recovers. The recovered partition can then compute a new summary and subscribe the cache to the new summary.

3.4 Consistency

The algorithm guarantees that if a summary cache is not up-to-date, there are messages or recomputations already underway that cause it to be replaced. This implies that if the dependencies of an operation no longer change, it reaches a final value that is consistent with the latest global state.

Ephemeral Inconsistency. When computing summaries, we read from local caches, which may not provide a snaphot-consistent, or even sequentially-consistent view. However, a summary that is based on an inconsistent view is also not up-to-date and thus guaranteed to be replaced. Thus, any inconsistencies are ephemeral.

Performance/Consistency Tradeoff. Given its weakened consistency and fast propagation speed, reactive caching is most attractive for applications where propagation latency is essential, i.e. where it is more important to refresh the result of an operation rapidly, than to have a guarantee that intermediate results are sequentially consistent.

Ephemeral Cycles. A summary participating in a cycle represents a nonterminating computation. But nonterminating computations time out, and we do not store them in summaries. Thus, cycles can appear only as temporary effect when reading stale caches, and do not cause progress problems for the algorithm.

Garbage Collection. A summary is deleted if there are no caches subscribed to it. A cache is deleted if no summaries or reactive computations depend on it. Since the dependency graph is (eventually) acyclic, this means that summaries are garbage collected when no longer needed.

3.5 Other Performance Benefits

Batching. Some time may pass in between marking a summary for recomputation and the actual recomputation. In particular, it may be marked multiple times, but recomputed only once. This batching effect is important for maintaining good throughput under high update frequencies, as we demonstrate in the evaluation section.

Sharing. Any number of summaries can depend on the same cache, but there is at most one cache per summary on each machine. Sharing caches in this way can be very space-efficient. For example, an application can use reactive polling to monitor a configuration setting. Even if there are many service partitions "polling" this configuration setting, there will be just a single cache for it per machine.

Large Fan-out. The reactive polling improves performance in situations with a large fan-out (summaries with many subscriptions): rather than sending updates to each summary, it is enough

[3] To detect dangling subscriptions, our implementation (§4) currently uses a 30-second periodic resubscription mechanism. This could be optimized, e.g. by using faster failure detection options within the virtual actor runtime.

to send one update to each cache, one per machine, and the machine then forwards the update locally to the summaries.

Back-Pressure. Our experiments show that in the case of high update rates, it is beneficial to throttle the sending of results to subscribed caches, since only the latest result matters. 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 — that is, we wait for an extra delay equal to the round trip time of the push.

4 IMPLEMENTATION

We have implemented reactive caching as an extension of Orleans, an open-source distributed virtual actor framework for .NET available on GitHub [Orleans 2016]. The Orleans runtime already provides distributed protocols for managing the creation, placement, discovery, recovery, and load-balancing of service partitions, which are called virtual actors, or grains [Bernstein et al. 2014; Bykov et al. 2011]. What we added is (a) extensions to the grain objects to store summaries, (b) interception of grain calls during summary computations, (c) modifications to the grain scheduler to distinguish between normal execution and summary computations, and (d) a silo-wide cache manager.

Detecting Changes. The propagation algorithm described in § 3.2 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 conservatively assume that *all* operations change the grain's state. This is not as wasteful as it may seem at first, because if re-execution of the summary produces the same result, 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 summary computation does not change the grain state, and thus avoid invalidation of summaries in that case.

Re-execution. Our algorithm changes the way grain operations are executed, which can surprise programmers. Any method that is called during a summary computation is 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 polling is meant to be used only with read-only operations. However, currently, we cannot enforce that. For a different host language, one can imagine a type/effect system along the lines of what we show in the calculus. 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).

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

External Dependencies. Some operations are likely to have external dependencies. To support reactive polling across service boundaries, users can implement a façade-grain for each external dependency. Each façade grain can use its own choice of mechanism to refresh its state, e.g. notifications provided by the external service, if supported, or polling, or hybrid approaches such as

Table 1. Parameter combinations. The fanout is the average number of views that depend on an item, and is equal to (#views ∗ #deps)/#items.

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

long-polling. Any changes in the façade-grain are then automatically propagated to the summaries that depend on it.

5 PERFORMANCE EVALUATION

Our evaluation compares the latency and resource consumption of reactive polling to two alternatives: *periodic polling* at various frequencies, and *handwritten propagation* at the application level. Note that these comparisons are not entirely apples-to-apples. For one, normal polling guarantees causal consistency, while reactive polling can (temporarily) return inconsistent results. Also, unlike the other two solutions, the hand-written propagation is not fault tolerant, and requires extensive changes to the service code. We measure it so we know the limit on what performance is achievable within the framework.

To conduct the experiments, we implemented two benchmarks and designed three series of experiments that measure low-load latency (§5.1), variable-load throughput (§5.2), and overall timeliness (§5.3).

Item-View Benchmark. This benchmark models an application using *item* grains that are observed by *view* grains. It is a microbenchmark, i.e. it isolates the mechanism we want to measure (change propagation) and removes all other aspects of the application. Each view depends on a fixed number of items, selected at random at the beginning of the test. Views are updated when items change. 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.

Chirper Benchmark. This benchmark is based on the chirper example introduced earlier (Fig. 1). Random state machines simulate users that choose between actions such as watch, post, follow, unfollow, or delete. All grain state (including followers, and posted messages) is reliably persisted in cloud storage. Service responses are tested for timeliness, giving us a measure of application performance as perceived by the user.

All benchmarks run on five Orleans silos deployed as a Windows Azure cloud service, using A4 machines (8 cores, and 14GB of RAM). The robots (client simulations) run on 10 load generator servers. 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. We observed that for different deployments, absolute numbers can vary up to 10% (since the actual machine specifications can vary), but for a single deployment, the relative performance of the various solutions was stable.

5.1 Latency Experiments

Our latency experiments use the low-load parameters (see Table 1) to eliminate delays caused by queueing and contention. We measure two types, called *query latency* and *propagation latency*.

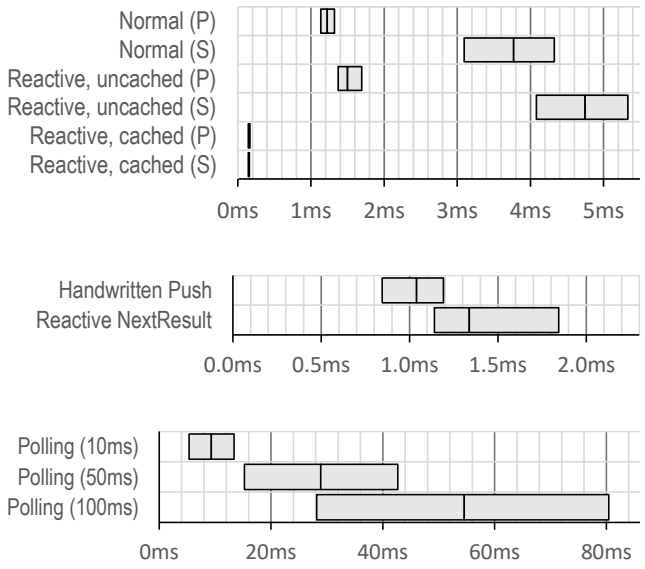

Fig. 10. Measured latencies under low load, in milliseconds. The line in the middle of each box is the median, and the left and right edge are the first and third quartile. **(top)** Latencies for parallel and sequential queries; **(middle)** latencies for mutation response, handwritten application-level propagation, and propagation via reactive computation; **(bottom)** propagation latencies for the polling solution at various frequencies.

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

5.1.1 Query Latency. Each view offers a distributed read-only operation that calls four items (either sequentially or in parallel) and aggregates the returned values. We compare

- the latency of the read-only operation executed directly
- the latency of the *first* NextResult() of a reactive polling loop for the same read-only operation

The normal latency for the sequential and parallel versions are shown in the top two rows of Fig. 10. The median latency is about 1.2ms for the parallel query and 3.8ms for the sequential query. This is consistent with the round-trip time of a typical grain call taking a bit less than 1ms.

For the reactive polling, 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. 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 NextResult() 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) reactive caching, even without reactive polling, (i.e. even if calling NextResult() only once) can dramatically improve the latency of a read-only operation.

5.1.2 Propagation Latency. For measuring propagation latency, a view calls an update operation on one of the items it depends on, and measures how much time elapses until it receives the change propagation.

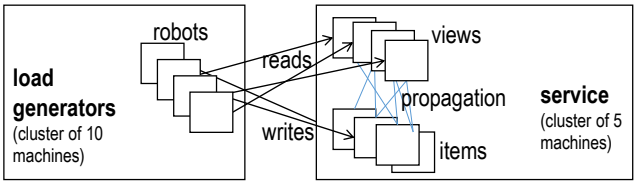

Fig. 11. Experimental setup for throughput experiments.

First, we measured the speed of handwritten propagation at the application level, where each item sends a direct notification message to all dependent views when mutated. The results show that the propagation message arrives about 1ms after the mutation is inititiated (see middle of Fig. 10), which represents the baseline round-trip-time for messages sent between grains in Orleans.

Second, we measured the speed of change propagation provided by the reactive caching algorithm (§3.2), i.e. the time elapsed until NextResult returns the changed result. In that case, the propagation takes about 300μs longer, due to the scheduling overhead of our implementation.

Finally, we looked at the propagation speed of a polling-based solution. Fig. 10, at the 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 as frequently as every 100ms is usually not advisable or even possible (we show impact of polling on throughput in §5.2). Reasonable polling intervals are more typically between 1 and 30 seconds, with a median propagation latency that is easily three orders of magnitude worse than for handwritten change propagation or reactive polling.

Conclusions. The results show that the latency of change propagation for reactive polling is much better than for a normal polling, and competitive with hand-written change propagation.

5.2 Throughput Experiments

For the throughput experiments, we generate external load as shown in Fig. 11. The load generator contains up to 2000 robots, distributed over 10 machines. Each robot simulates a user, by running 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. 12; 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 machine resources are available for handling requests. Here, we reach close to 120k requests per second. But as soon as we use a change propagation or polling mechanism, the throughput is lower, because it consumes machine resources that are diverted from processing requests: hand-written propagation (dotted lines) reaches about 45k, propagation by reactive computations 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

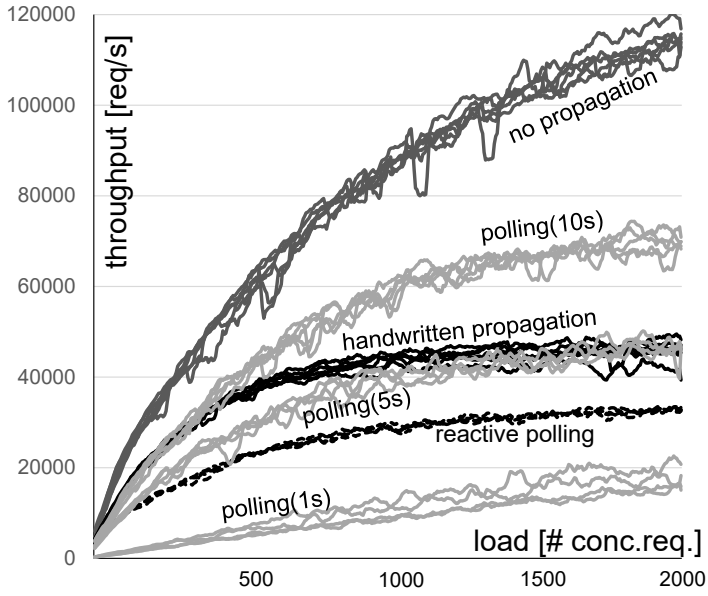

Fig. 12. Throughput response of different propagation mechanisms, for fanout-20 with 10% updates.

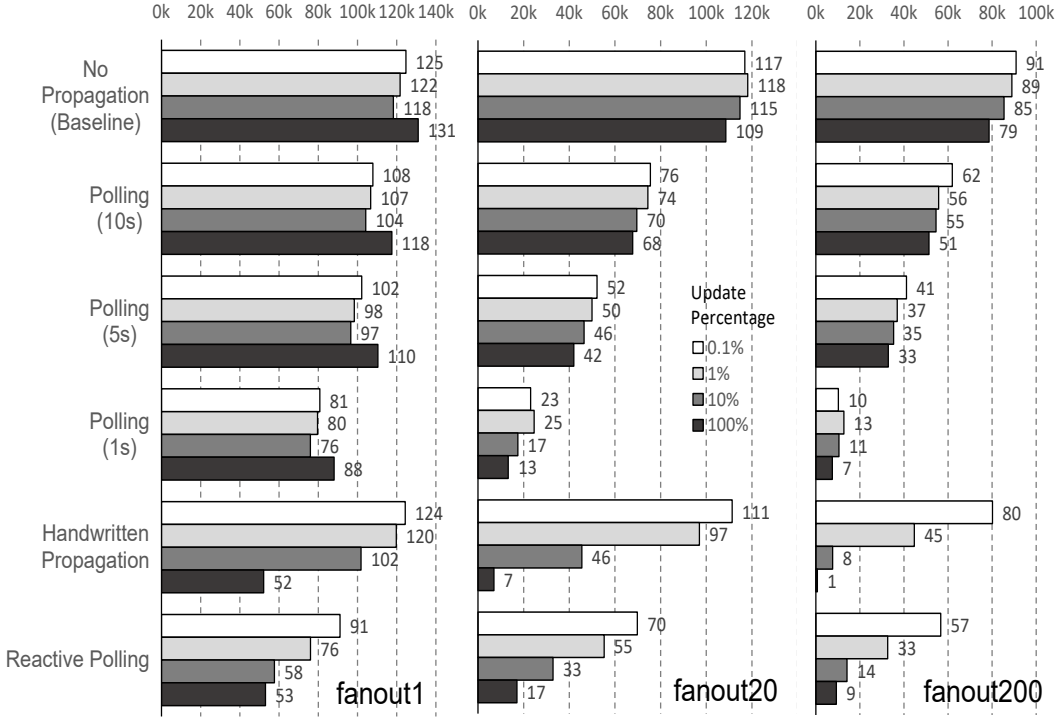

Fig. 13. Measured throughput (in thousands of requests per second) for varying configurations (Table 1), propagation mechanisms, and percentage of updates.

in Table 1) for a while at the end, to measure average peak throughput for that load. The results are shown in Fig. 13. 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. 12.

Baseline. With update propagation turned off (top row), 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.

Hand-Written Propagation. The extra work incurred by hand-written propagation at the application level 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.

Reactive Polling. There are two main differences to be expected compared to hand-written propagation: (1) reactive computations incur a bit more work due to scheduling indirection, re-execution of summaries, and management of reactive caches; and (2) reactive computations can adapt to back-pressure and reduce the number of updates sent. We can observe these effects: throughput for reactive computations is generally lower than for hand-written propagation, except for high update rates and/or fanout where hand-written propagation suffers, because it is sending more messages.

Conclusions. The results show that reactive polling is generally competitive with hand-written 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, while terrible when using a high frequency, remains an acceptable solution under low frequency, i.e. for applications that do not require quick propagation time. It can be tuned to reliably consume little resources, and does not suffer under high update rates.

5.3 Chirper Experiments

Our final experiment measures user-perceived timeliness of the chirper application. We simulate users by state machines that pick randomly from a list of actions, starting a new action immediately upon completion of a previous one. Each test starts with 100 users and adds 400 users every 2 seconds. The cumulative activity of the users creates the messages that appear in timelines. The user actions and their probability weight are: *Follow* (3): pick a user at random and follow (we use a bias to create an asymmetric distribution, reflecting that popularity concentrates on influential users). *Unfollow* (1): stop following a user. *Post* (3): post a random message. *Delete* (1) delete a

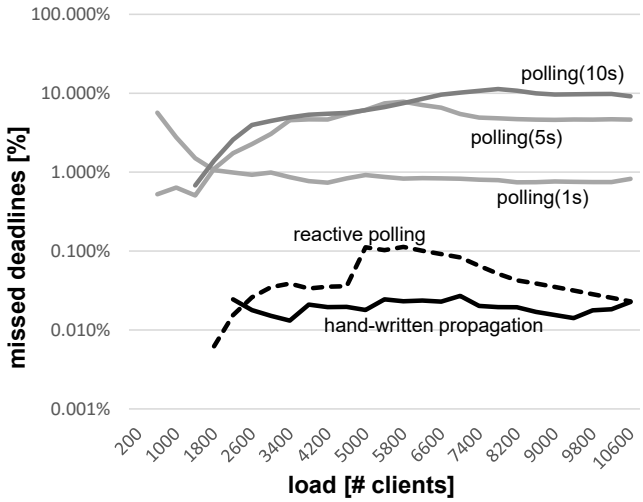

Fig. 14. User-observed timeliness of chirper benchmark under various propagation mechanisms. Note the exponential scale on the y-axis.

random message. *Watch* (3): initiate a 15-20s period of watching the current timeline, using one of the available mechanisms (polling, reactive polling, or hand-written propagation).

To measure timeliness, we keep track of two kinds of "deadlines": (a) any request issued to the service should receive its response within 1s; and (b) if the timeline changes during a watching period, any new message should have a timestamp no older than 1s back. We then measure timeliness as the percentage of missed deadlines.

Fig. 14 shows the results. Note that the y-axis is logarithmic. We make the following observations. For one, if using polling, a significant percentage of deadlines are missed, unless the polling is very frequent. Second, explicit propagation is still noticeably better than reactive computations at low load, but with the gap closing when the load increases. These observations are consistent with the conclusions we reached for the item-view benchmark.

6 RELATED WORK

Self-Adjusting Computation. Our techniques are inspired by, and closely related to, previous work on self-adjusting computation [Acar 2009; Acar et al. 2008, 2010, 2009; Hammer et al. 2009], imperative reactive programming [Demetrescu et al. 2011], and incremental concurrent revisions [Burckhardt et al. 2011]. While they target quite different environments, these papers have a common theme: to improve performance of repeated computations by incrementally repairing an in-memory representation of the computation when inputs change. This same idea underlies our use of summaries and reactive caches. The difference is that here, we use the encapsulation afforded by service composition as a means to decompose the computation. Moreover, our algorithm executes in a distributed setting and is fault-tolerant. More generally, the theme of incremental update propagation is also found in other reactive programming models [Demetrescu et al. 2016], and the model-view paradigm [Burckhardt et al. 2013; React 2016].

Actor Frameworks. Actor frameworks [Akka 2016; Armstrong 2010; Bernstein et al. 2017; Chuang et al. 2013; Orbit 2016; Orleans 2016; Sang et al. 2016; SF Reliable Actors 2016] directly embody the composed-services paradigm for building scalable distributed systems. Actors are not only

suitable for formal study, but can also deliver excellent performance in practice, and distribute easily over elastic clusters since they do not share memory and communicate asynchronously. Many of the actor frameworks mentioned above, and in particular virtual actors, go beyond basic actors: they provide a programming model where actors behave more like miniature services. Our service calculus captures this practice in a form suitable for formal study.

Functional Reactive Programming. In functional reactive programming (FRP) [Acar et al. 2006; Cooper and Krishnamurthi 2006; Courtney 2001; Czaplicki and Chong 2013; Elliott and Hudak 1997; Hudak et al. 2003; Meyerovich et al. 2009; Wan and Hudak 2000], views are expressed as signals that depend on event streams, and defined using a vocabulary of functional operators or combinators [Acar et al. 2006; Cooper and Krishnamurthi 2006; Czaplicki and Chong 2013; Elliott and Hudak 1997; Wan and Hudak 2000]. FRP, as well as related approaches, can deliver superior ease of use and unrivaled performance for the type of queries that are convenient to express.

However, note that it is in general *not* straightforward to obtain a functional query that is equivalent to requests composed in an imperative style (e.g. the code in Fig. 1). In particular, if the dependencies are data-dependent, this may require the use of nontrivial join operators. Such join operators are not only more difficult to write, but can also be challenging for the runtime to execute efficiently. Imperative code as in Fig. 1, in contrast, gives the programmer full control and transparency over how the distributed computation is broken down.

GUI Construction. Much research has focused on reactive programming for graphical user interfaces [Burckhardt et al. 2013; Czaplicki and Chong 2013; Meyerovich et al. 2009] on the client. These solutions do typically not address the service side, or assume a single server. An exception is distributed ReScala [Drechsler et al. 2014] which parallelizes update propagation across multiple servers. However, unlike our solution, it does not handle faults or provide elasticity, and 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 stark contrast to our workload that exhibits a large volume of concurrent independent updates on actors.

Cloud Scale. Reactive techniques have been successfully applied to large-scale data-parallel computations used for data analytics [Dean and Ghemawat 2008; Zaharia et al. 2012]. Some systems incrementalize mapreduce queries [Bhatotia et al. 2011; Condie et al. 2010] or queries constructed from a vocabulary of functional operators [Gunda et al. 2010], possibly including fixpoints as in Naiad [McSherry et al. 2013]. Frameworks for stream programming [Flink 2016] have also gained popularity - they are more low-level and very flexible. In all those solutions, the user assumes responsibility for expressing and tracking dependencies, by publishing or subscribing to streams, which requires significant changes to the service code. In contrast, our solution tracks dependencies and pushes changes automatically.

Semantics and Consistency. Change propagation can be 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. Since we are emulating a polling loop, not a globally consistent snapshot, we are not preventing glitches. Avoiding glitches in a distributed system may add significant latency, but it would be interesting to figure out just how much. This general tradeoff is similar to variations of eventual consistency [Burckhardt 2014].

At the other end of the spectrum are synchronous reactive languages [Benveniste et al. 2000; Berry and Gonthier 1992; Caspi et al. 1987; Gautier et al. 1987], where time is explicit, and systems

make very strong semantic guarantees. It is hard to imagine an efficient implementation of such models in our context, given the high cost of distributed coordination.

Other. The problem of combining object-oriented and reactive paradigms is not new [Salvaneschi and Mezini 2013], as there is often a desire to connect object-oriented GUI frameworks with functional-reactive backends [Ignatoff et al. 2006]. 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 [McDirmid and Hsieh 2006] 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[Cutsem et al. 2014] also introduces reactive programming into the actor model. But its focus is not on fault-tolerant elastic 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.

7 CONCLUSION

We have motivated, explained, implemented, and evaluated a new reactive caching mechanism that allows latency-sensitive clients to replace polling loops with a similarly looking, but more efficiently executing reactive caching algorithm that uses push-based change propagation.

Given the novelty of our approach, many more questions remain to be investigated in future work. We would like to gather more experience with early adopters; we have already started a collaboration with a game developer team. A very interesting endeavor is to explore the performance cost of guaranteeing causality, which would make reactive polling semantically equivalent to polling loops. Moreover, we would like to explore how to combine reactive caching with other mechanisms, such as streams and event sourcing. Also, for collection data types, using diffing optimizations may further improve performance. Finally, we are working on extending our formal development of the service calculus and the reactive caching algorithm, in particular adding formulations and proofs of the consistency guarantees.

A TYPE SYSTEM

A.1 Proof of Theorem 2.2

Starting with derivations

(a) $e, \sigma, N \;\rightarrow_g\; e', \sigma', N'$
(b) $\vdash_\mu (g, E[e], \sigma, N) : t$
(c) $G \vdash_\mu e : t$ (from (b) using lemma A.2a)

for some $g = G\langle v_k \rangle$. We proceed below by structural induction on (c), with a case distinction over the inference rule used for the conclusion. In each case, we need to construct a derivation for $\vdash_\mu (g, e', \sigma', N') : t$. We discharge each case by showing that the following derivations exist

(x) $G \vdash_\mu e' : t$
(y) $\vdash_\mu \sigma' : s_\sigma$ (only needed if $\sigma' \neq \sigma$)
(z) $\vdash N'$ (only needed if $N' \neq N$)

which is then sufficient to construct the required derivation using Lemma A.2b.

- (TVar), (TAbs), (TUnit), (TDone), (TPlaceholder): these are not applicable because it would mean e is already a value, and cannot take an evaluation step.
- (TKey): Then $e = \mathbf{key}$ and $e' = v_k$. (c) is (TKey) and ensures that $t = s_k$, and (b)'s prerequisite $\vdash g : G$ means that $\vdash_\mu k : t_k$, which gives us (x) using the lemma A.4.

- (TGet): Then $e = \textbf{get}$ and $e' = \sigma$. (c) is (TGet) and ensures that $t = s_\sigma$ and (b) ensures that $\vdash_\mu \sigma : s_\sigma$, which gives us x using the lemma A.4.
- (TSet): Then $e = \textbf{set}\ v$, $e' = ()$, and $\sigma' = v$. Since $t = \mathit{unit}$ thus (x) is direct by (TUnit). Moreover, $\sigma' = v$, and (TSet) says that $G \vdash_\mu v : s_\sigma$, which gives us (y) using lemma A.5 and lemma A.3.
- (TAwait): Then $e = \textbf{await done}\ v$ and $e' = v$. (c) must decompose into (TAwait) and (TDone) and have a prerequisite $G \vdash_\mu v : t$ which is (x).
- (TOpCall): Then $e = g'.o(v_a)$ and $e' = r = g.i/g'.o$ and $N' = \mathit{call}\ r\ v_a N$. For (x) we need (TPlaceholder), which we can assemble from the prerequisites of (b) and (c). Also, for (z) we need the message typing (TMessages) which we get using (TCall) and the prerequisites of (b) and (c) as well as the same placeholder typing, and using the lemmas A.5 and A.3.
- (TApp): Then $e = \lambda x{:}t.e_b\ \ v$ and $e' = E[e[v/x]]$. We get (x) from (b) using the substitution lemma A.1.

A.2 Proof of Theorem 2.4

We start with $\vdash S \mid N$, which provides typing derivations for all messages in N and all service partition data in S, including for each partition a key, state, and task pool. We then need to show that whatever is modified/added by the global step $S \mid N \longrightarrow S' \mid N'$ remains typable, to prove $\vdash S' \mid N'$. We distinguish by step.

- (ExtCall) adds a new message which is well typed thanks to the prerequisite.
- (ExtRsp) only removes a message.
- (Activate) the prerequisites ensure the partition reference g and initial value v_0 are well typed, so the added partition data $(g{:}v_0)$ is as well (TPartitionData).
- (RcvCall) the added task pool entry is well-typed according to (TTask) because the incoming message is well typed according to (TCallMsg), giving exactly the required prerequisites.
- (RcvRsp) Using lemma A.6.
- (CompleteReq) Using local preservation theorem.
- (TakeTurn) Using local preservation theorem.
- (Failure Transitions): only remove things.

A.3 Proof of Theorem 2.3

We need to show that if $G \vdash_\mu E[e] : t$, then exactly one of:

(1) $e = v$ for some value v
(2) $e = E[\textbf{await}\ p]$ for some placeholder p
(3) e can take a step.

Clearly, not more than one can be true since v and $\textbf{await}\ p$ are different and neither is a redex. We prove that at least one is true using structural induction over the type derivation $G \vdash_\mu e : t$.

- (TVar), (TAbs), (TUnit), (TDone), (TPlaceholder): claim is immediate since in that case e is a value.
- (TKey), (TGet), (TSet): Can take a local step.
- (TAwait): Then $e = \textbf{await}\ e'$, $t = \mathit{task}\ s$ and $G \vdash_\mu e' : s$. Now, use induction hypothesis and distinguish cases:
  - if e' can take a step so can e
  - if $e' = E[\textbf{await}\ e'']$ then also $e = E[\textbf{await}\ e'']$.
  - if e' is a value, it has to be $\textbf{done}\ v'$ (then e can take a step) or a placeholder (then $e = \textbf{await}\ p = E[\textbf{await}\ p]$) because those are the only values of type $\mathit{task}\ t'$

- (TApp): Then $e = (e_f e_a)$ and $G \vdash_\mu e_f : t_f$ and $G \vdash_\mu e_a : t_a$. Use induction hypothesis on e_f. If it can take a step, so can e. If it is of the form $E[\textbf{await } p]$, then so is e. If it is a value v_f, then use induction hypothesis on e_a. If it can take a step, so can e. If it is of the form $E[\textbf{await } p]$, then so is e. If it is a value v_a, then $e = (v_f\ v_a)$ can take an (App) step, since the typing constraints guarantee that v_f is a lambda.
- (TOpCall): Then $e = G\langle e\rangle.o(e')$. Do a case distinction analogous to what we did for (TApp).

A.4 Lemmas

Expressions of same type can be substituted without breaking type derivations:

Lemma A.1. *If* $(x:s)\Gamma \vdash_\mu e : t$ *and* $\Gamma \vdash_\mu e' : s$, *then* $\Gamma \vdash_\mu e[e'/x] : t$.

Similarly, expression contexts allow substitution:

Lemma A.2. *If* $G \vdash_\mu E[e] : t$, *then*
(a) $G \vdash_\mu e : t'$ *for some type* t'
(b) *if* $G \vdash_\mu e' : t'$, *then* $G \vdash_\mu E[e'] : t$

Expression contexts do not evaluate expressions inside bound contexts:

Lemma A.3. *If* $G \vdash_\mu E[e] : t$, *then* e *has no free variables.*

Serializable value typing is independent of context:

Lemma A.4. $\vdash_\mu v : s$ *implies* $\Gamma \vdash_\mu v : s$ *for any* Γ

Lemma A.5. *if* v *is not a variable, then* $\Gamma \vdash_\mu v : s$ *implies* $\vdash_\mu v : s$

Placeholders can be substituted without breaking type derivations:

Lemma A.6. *If* $\Gamma \vdash_\mu e : t$, *and* $\vdash g' : G'$, *and* $G' \vdash (o_v : f) : s_a \rightarrow_v s_r$, *and* $\vdash_\mu v : s_r$, *then* $\Gamma \vdash_\mu e[(\textbf{\textit{done}}\ v)/(g.i/g'.o)] : t$.

B ELASTICITY AND PERSISTENCE

In this section we show how the grain calculus can be implemented on an elastic cluster with storage separation, i.e. built from a combination of unreliable compute servers and reliable storage. We start with a version that supports only volatile grains (i.e. does not provide persistence), and then show a model that demonstrates how to add persistence.

B.1 Volatile Silo Cluster Model

This model is a straightforward modification of the previous one, where system configuration group grains into so-called *silos*. Each silo is identified by a unique silo identifier. To express this, we add the following syntax:

$$\begin{array}{lll} L & ::= \overline{(s:X)} & \text{(silos)} \\ s & ::= \ldots & \text{(silo identifier)} \\ X & ::= \overline{(g:R)} & \text{(grainstore)} \end{array}$$

System configurations are now of the form $L \mid N$, where N is the network as before, but L is a collection of silos. Each silo is identified by a unique silo identifier s and contains a collection of grains X, where X is a collection of grains (just like S was before). Most rules are adapted straightforwardly to use a "nested collection" L of grains, instead of a flat collection S. However, the failure rules are different now: rather than failing individual grains, (FailSilo) fails an entire silo. All state of the grains in the silos is lost.

$\boxed{L \mid N \;\rightarrow\; L' \mid N'}$ **Elastic Silo Cluster, Volatile Grains**

$$\text{ExtCall}\ \frac{i \text{ fresh} \qquad \vdash \mathit{call}\ r\ v \qquad r = i/g.o \qquad g = G\langle v\rangle}{L \mid N \;\rightarrow\; L \mid (\mathit{call}\ r\ v)\ N} \qquad \text{Fail-Message}\ \frac{}{L \mid m\ N \;\rightarrow\; L \mid N}$$

$$\text{NewSilo}\ \frac{s \notin \mathit{dom}\ L}{L \mid N \;\rightarrow\; (s{:}\,)\ L \mid N} \qquad \text{FailSilo}\ \frac{}{(s{:}X)\ L \mid N \;\rightarrow\; L \mid N}$$

$$\text{RcvCall}\ \frac{r = \ldots/g.o \qquad g = G\langle v'\rangle \qquad G = \ldots(o{:}f)\ \ldots}{(s{:}(g{:}\sigma\ T)\ X)\ L \mid (\mathit{call}\ r\ v)\ N \;\rightarrow\; (s{:}(g{:}\sigma\ (r{:}f\ v)\ T)\ X)\ L \mid N}$$

$$\text{RcvRsp}\ \frac{r = g.i/g'.o \qquad T' = T[(\textbf{done}\ v)/r]}{(s{:}(g{:}\sigma\ T)\ X)\ L \mid (\mathit{rsp}\ r\ v)\ N \;\rightarrow\; (s{:}(g{:}\sigma\ T')\ X)\ L \mid N}$$

$$\text{Activate}\ \frac{g \notin \mathit{dom} X \cup \mathit{active}(L) \qquad \vdash g : G \qquad G = \langle t\rangle\ a\ \sigma\ D}{(s{:}X)\ L \mid N \;\rightarrow\; (s{:}(g{:}\sigma)\ X)\ L \mid N}$$

$$\text{TakeTurn}\ \frac{e, \sigma, N \;\rightarrow^{+}_{g}\; E[\textbf{await}\ p], \sigma', N'}{(s{:}(g{:}\sigma\ (r{:}e)\ T)\ X)\ L \mid N \;\rightarrow\; (s{:}(g{:}\sigma'\ (r{:}E[\textbf{await}\ p])\ T)\ X)\ L \mid N'}$$

$$\text{CompleteReq}\ \frac{e, \sigma, N \;\rightarrow^{+}_{g}\; v, \sigma', N' \qquad g = G\langle v\rangle}{(s{:}(g{:}\sigma\ (r{:}e)\ T)\ X)\ L \mid N \;\rightarrow\; (s{:}(g{:}\sigma'\ T)\ X)\ L \mid (\mathit{rsp}\ r\ v)\ N'}$$

If we define observations as traces of call and response transitions relating to external requests, and if all grain classes are volatile, then this model is observationally equivalent to the grain calculus shown earlier.

B.2 Persistent Silo Cluster Model

To handle persistent grains, we can modify the previous model by adding persistent storage, which is a simple map from grain references to values:

$$P ::= \overline{(g{:}\sigma)} \quad \text{(persistent store)}$$

Then, we define system configurations $L \mid N \mid P$ that include a persistent store. The store is updated whenever persistent grains are modified (write-through).

$\boxed{L \mid N \mid P \;\rightarrow\; L' \mid N' \mid P'}$ **Elastic Silo Cluster w/ Persistent Store**

$$\text{ExtCall}\;\frac{i \text{ fresh} \qquad \vdash \mathit{call}\; r\; v \qquad r = i/g.o}{L \mid N \mid P \;\rightarrow\; L \mid (\mathit{call}\; r\; v)\; N \mid P} \qquad \text{Fail-Message}\;\frac{}{L \mid m\; N \mid P \;\rightarrow\; L \mid N \mid P}$$

$$\text{NewSilo}\;\frac{s \text{ fresh}}{L \mid N \mid P \;\rightarrow\; (s{:}\;)\; L \mid N \mid P} \qquad \text{FailSilo}\;\frac{}{(s{:}X)\; L \mid N \mid P \;\rightarrow\; L \mid N \mid P}$$

$$\text{RcvCall}\;\frac{r = \ldots/g.o \qquad g = G\langle v' \rangle \qquad G = \ldots (o{:}f)\;\ldots}{(s{:}(g{:}\sigma\; T)\; X)\; L \mid (\mathit{call}\; r\; v)\; N \mid P \;\rightarrow\; (s{:}(g{:}\sigma\; (r{:}f\; v)\; T)\; X)\; L \mid N \mid P}$$

$$\text{RcvRsp}\;\frac{r = g.i/g'.o \qquad T' = T[(\textbf{done}\; v)/r]}{(s{:}(g{:}\sigma\; T)\; X)\; L \mid (\mathit{rsp}\; r\; v)\; N \mid P \;\rightarrow\; (s{:}(g{:}\sigma\; T')\; X)\; L \mid N \mid P}$$

$$\text{Activate-V}\;\frac{g \notin \mathit{dom} X \cup \mathit{active}(L) \qquad \vdash g : G \qquad G = \ldots\; \mathit{volatile}\; \sigma\;\ldots}{(s{:}X)\; L \mid N \mid P \;\rightarrow\; (s{:}(g{:}\sigma)\; X)\; L \mid N \mid P}$$

$$\text{TakeTurn-V}\;\frac{e, \sigma, N \;\rightarrow_g^+\; E[\textbf{await}\; p], \sigma', N' \qquad g = G\langle v \rangle \qquad G = \ldots\; \mathit{volatile}\;\ldots}{(s{:}(g{:}\sigma\; (r{:}e)\; T)\; X)\; L \mid N \mid P \;\rightarrow\; (s{:}(g{:}\sigma'\; (r{:}E[\textbf{await}\; p])\; T)\; X)\; L \mid N' \mid P}$$

$$\text{CompleteReq-V}\;\frac{e, \sigma, N \;\rightarrow_g^+\; v, \sigma', N' \qquad g = G\langle v \rangle \qquad G = \ldots\; \mathit{volatile}\;\ldots}{(s{:}(g{:}\sigma\; (r{:}e)\; T)\; X)\; L \mid N \mid P \;\rightarrow\; (s{:}(g{:}\sigma'\; T)\; X)\; L \mid (\mathit{rsp}\; r\; v)\; N' \mid P}$$

$$\text{Activate-P}\;\frac{g \notin \mathit{dom}\, P \qquad \vdash g : G \qquad G = \ldots\; \mathit{persistent}\; \sigma\;\ldots}{L \mid N \mid P \;\rightarrow\; L \mid N \mid (g{:}\sigma)\; P}$$

$$\text{Load-P}\;\frac{g \notin \mathit{dom} X \cup \mathit{active}(L)}{(s{:}X)\; L \mid N \mid (g{:}\sigma)\; P \;\rightarrow\; (s{:}(g{:}\sigma)\; X)\; L \mid N \mid (g{:}\sigma)\; P}$$

$$\text{TakeTurn-P}\;\frac{e, \sigma, N \;\rightarrow_g^+\; E[\textbf{await}\; p], \sigma', N' \qquad g = G\langle v \rangle \qquad G = \ldots \mathit{persistent}\;\ldots}{(s{:}(g{:}\sigma\; (r{:}e)\; T)\; X)\; L \mid N \mid (g{:}\sigma)\; P \;\rightarrow\; (s{:}(g{:}\sigma'\; (r{:}E[\textbf{await}\; p])\; T)\; X)\; L \mid N' \mid (g{:}\sigma')\; P}$$

$$\text{CompleteReq-P}\;\frac{e, \sigma, N \;\rightarrow_g^+\; v, \sigma', N' \qquad G = \ldots\; \mathit{persistent}\;\ldots}{(s{:}(g{:}\sigma\; (r{:}e)\; T)\; X)\; L \mid N \mid (g{:}\sigma)\; P \;\rightarrow\; (s{:}(g{:}\sigma'\; T)\; X)\; L \mid (\mathit{rsp}\; r\; v)\; N' \mid (g{:}\sigma')\; P}$$

C REACTIVE CACHING ALGORITHM

In this section, we present a detailed model to document our distributed fault-tolerant algorithm for reactive caching. This model matches the description of summaries and caches from §3, and documents a somewhat abstracted version of our algorithm precisely.

C.1 Overview

We start by summarizing the overall mechanism.

- Silos represent failure domains, i.e. contain things that all fail together.
- A silo can contain any number of service partitions.
- A service partition can be located on any silo, but no more than one silo at a time.
- A partition can store a summary for any of its cacheable operations and call argument.
- Each silo can store a local cache for any summary.
- Caches subscribe to summaries.
- A summary keeps a list of subscribed caches, and sends any new results to all subscribed caches.
- When a partition asks for the first or next result of a reactive computation, it looks in the cache, and may return a cachekey placeholder.
- When a cache receives a new result, it replaces all cachekey-placeholders in the silo with the new result.
- A summary can have a value (the last result computed) or not (no result computed yet).
- If a summary has a value, it also has a read set (the dependencies) and a dirty flag.
- A summary is marked dirty if a cache in its read set receives a new result.
- If the state of a partition is changed, all its summaries are marked dirty.
- Summaries are computed or recomputed using a special type of task called summary computation.
- A summary computation is started if the summary has no value yet, or if it is dirty.
- Only one summary computation can be active per summary.
- Summary computations track a read set and have a dirty flag.
- A summary computation is marked dirty if a cache in its read set receives a new result.
- If the state of a partition is changed, all its summary computations are marked dirty.
- When a summary computation completes, the summary assumes its result value, read set, and dirty flag.
- If a silo is lost, all contained caches and summaries are lost.
- The loss of a subscribed cache is detected when a result is pushed to it.
- The loss of a summary to which a cache is subscribed is detected during the periodic renew.
- Renewing a subscription to a lost summary means the partition is reactivated and the summary is recomputed.

C.2 Formalization Choices

For the formalization, we make the following simplifications or assumptions:

- A reactive computation is not a general anonymous function, but must be a single operation call. This is already the case for the example shown in Fig. 2b, and can be easily achieved by defining an auxiliary operation if desired.
- We are not modeling result trackers directly; rather, we represent their function using two expressions *first* and *next* which can be called to obtain a first, or a successive result, respectively.
- We consider all partitions to be volatile. Adding persistence is an orthogonal issue.

All timing is replaced by simple nondeterministic transitions. Also, cast-out of cache entries is a simple nondeterministic transition; we are not modeling details about when this should happen (in the actual implementation, cache entries are cast out once the last result tracker is disposed).

C.3 Syntax

The modified syntax is shown in Fig. 15. Reactive computations, caches, and summaries are identified by a *cache key* of the form $g/o_c/v$ composed of a partition, operation, and argument value. These cache keys are used to label messages, as keys to index the cache and summary collections, and as a new kind of placeholder that gets substituted whenever a cache receives a new value.

Expression Syntax. At the top left, we show the changes to the expression syntax. It includes two new asynchronous expressions returning task types. The expression **first** $G\langle e\rangle.o_c(e)$ is a reactive computation, containing a single operation call. It returns **done** v immediately if the local cache already contains a cached result v; otherwise it returns a placeholder that resolves once a result is received. The expression **next** $G\langle e\rangle.o_c(e)$ returns a placeholder that resolves to **done** v when the local cache receives a new value v, or its first value v. In both cases, the placeholder is not a request identifier r as before, but a cache key k that gets substituted whenever the corresponding cache receives a new value.

Configurations. The system configurations are analogous to the volatile cluster model, i.e. comprised of silos and a network. However, silos now have additional data structures (caches, summaries, summary computations) and the network may contain new types of messages.

Messages. There are four new messages; *sub s k*, *renew s k*, and *unsub s k* are sent from a cache entry on silo s to the corresponding summary k to subscribe, renew, or cancel a subscription, respectively. *push s k v* is sent from summary k to a silo s in order to update the cached result of that summary on that silo to the latest value v.

Caches. Each silo has a cache C that stores cache entries w consisting of a summary key k, and either a value v or no value if none has arrived yet.

Summaries. Partition records now contain a collection Z of summaries. Each summary z is of the form $(k : Y\ d)$ where k is the summary key, Y is a list of silos that have subscribed to this summaries,

e	$::= \ldots$	(expression)
	$\mid$ **first** $G\langle e\rangle.o_c(e)$	(reactive computation)
	$\mid$ **next** $G\langle e\rangle.o_c(e)$	(reactive change)
p	$::= r \mid k$	(placeholder)
k	$::= g/o_c/v$	(cache key)
m	$::= \ldots$	(message)
	$\mid$ *sub s k*	(subscribe)
	$\mid$ *renew s k*	(renew)
	$\mid$ *unsub s k*	(unsubscribe)
	$\mid$ *push s k v*	(push update)

L	$::= \overline{(s : X\ C)}$	(silos)
C	$::= \overline{w}$	(cache)
w	$::= (k : v) \mid (k :)$	(cache entry)
X	$::= \overline{(g : R)}$	(partition store)
R	$::= \sigma\ T\ Z$	(partition data)
T	$::= \overline{u}$	(task pool)
u	$::= (r : e)$	(normal task)
	$\mid (k : e\ Q)$	(comp. task)
Q	$::= \overline{k}\ \mathit{dirty}?$	(read set)
Z	$::= \overline{z}$	(summaries)
z	$::= (k : Y\ d)$	(summary)
Y	$::= \overline{s}$	(subscribed silos)
d	$::= (\mathit{val}\ v\ Q)?$	(summary state)

Fig. 15. Syntax extensions for reactive caching, and for silo configurations including summaries and caches.

and d is the state of this summary, which is either empty, or a combination $(val\ v\ Q)$ of a value and a read set (also possibly dirty).

Task pools. The task pool for each partition record now contains an additional type of task, called summary computation, which is keyed by the summary key k. For each such entry $(k : e\ Q)$, we track not only the current expression e, but also a read set Q contains summary keys this this summary computation depends on, and possibly a flag *dirty* which means at least one dependency has already changed and the reactive computation will have to be restarted to compute the latest result.

$\boxed{e, \sigma, N, C \ \rightarrow_{s,g}\ e', \sigma', N', C}$ **Local Step (outside reactive computation)**

$$\text{SimpleStep}\ \frac{e, \sigma, N \ \rightarrow_{g}\ e', \sigma', N'}{e, \sigma, N, C \ \rightarrow_{s,g}\ e', \sigma', N', C}$$

$$\text{Cache-Miss1}\ \frac{(e = \textbf{first}\ g'.o_c(v)) \vee (e = \textbf{next}\ g'.o_c(v)) \qquad k = g'/o_c/v \qquad k \notin dom\, C \qquad C' = (k :)\ C}{E[e], \sigma, N, C \ \rightarrow_{s,g}\ E[k], \sigma, (sub\ s\ k)\ N, C'}$$

$$\text{Cache-Miss2}\ \frac{(e = \textbf{first}\ g'.o_c(v)) \vee (e = \textbf{next}\ g'.o_c(v)) \qquad k = g'/o_c/v \qquad C = (k :)\ C'}{E[e], \sigma, N, C \ \rightarrow_{s,g}\ E[k], \sigma, N, C}$$

$$\text{Cache-Hit1}\ \frac{k = g'/o_c/v \qquad (k : v') \in C}{E[\textbf{first}\ g'.o_c(v)], \sigma, N, C \ \rightarrow_{s,g}\ E[\textbf{done}\ v'], \sigma, N, C}$$

$$\text{Cache-Hit2}\ \frac{k = g'/o_c/v \qquad (k : v') \in C}{E[\textbf{next}\ g'.o_c(v)], \sigma, N, C \ \rightarrow_{s,g}\ E[k], \sigma, N, C}$$

$\boxed{e, N, C, Q \ \rightarrow_{s,g,\sigma}\ e', N', C', Q'}$ **Local Step (inside reactive computation)**

$$\text{SimpleStep}\ \frac{e, \sigma, N \ \rightarrow_{g}\ e', \sigma', N'}{e, \sigma, N, C \ \rightarrow_{s,g}\ e', \sigma', N', C}$$

$$\text{Cache-Hit}\ \frac{k = g'/o_c/v \qquad (k : v') \in C \qquad Q' = k\ Q}{E[g'.o_c(v)], N, C, Q \ \rightarrow_{s,g,\sigma}\ E[\textbf{done}\ v'], N, C, Q'}$$

$$\text{Cache-Miss1}\ \frac{k = g'/o_c/v \qquad k \notin dom\, C \qquad C' = (k :)\ C}{E[g'.o_c(v)], N, C, Q \ \rightarrow_{s,g,\sigma}\ E[k], (sub\ s\ k)\ N, C', Q}$$

$$\text{Cache-Miss2}\ \frac{k = g'/o_c/v \qquad C = (k :)\ C'}{E[g'.o_c(v)], N, C, Q \ \rightarrow_{s,g,\sigma}\ E[k], N, C, Q}$$

Fig. 16. Local steps outside of reactive computations (top) and inside reactive computations (bottom).

C.4 Operational Semantics

As before, partitions have a task pool to process incoming operation calls. However, there are additional mechanisms involved. We give a brief description here before inspecting the corresponding rules in more detail.

This summary tracks what caches are subscribed to it, and pushes any freshloy As before, partitions have a task pool to process incoming operation calls. However, they also may have one or more summary tasks executing, which compute summary results. Summary results are pushed to the subscribed We have broken the operational semantics into four separate sections, to facilitate explanation. In §C.4.2 we describe the standard execution semantics.

C.4.1 Local Steps. The local execution of a local step is now distinguished into two modes: inside and outside of reactive computations. We show these in Fig. 16.

Outside. For local steps outside of reactive computations, the transition rule contains a new component C which is the cache. The (SimpleStep) rule imports all the same rules for local steps from Fig. 5, ignoring the cache C. The other three rules define the semantics of the **first** α and **next** α expressions for some reactive computation α, based on whether a cache entry for α already exists or not. If a cache entry does not exist, (Cache-Miss1) applies and handles both expressions the same way: a new cache entry (with no value) is created, a subscription message is sent, and a cache-key placeholder is returned. If a cache entry exists but has no value, (Cache-Miss2) applies which does not send a subscription message but is otherwise the same. If a cache entry exists, (Cache-Hit1) returns a first result immediately. (Cache-Hit2) returns a placeholder, thus blocking execution until the cache changes in response to a received latest result.

Inside. For local steps outside of reactive computations, the transition rule contains an additional component Q which is the read set. When a call to another partition (which must be to a cacheable operation) is performed, and it hits in the cache, (Cache-Hit) returns the cached value and records the dependency in the read set. If it misses, (Cache-Miss1) or (Cache-Miss2) returns a placeholder. It does not record the dependency; this will instead happen at the time the value is substituted.

C.4.2 Normal System Steps. The system steps corresponding to regular execution of operation calls are shown in Fig. 17. They are analogous to the steps of the volatile cluster model, with one difference: the rules (TakeTurn) and (CompleteReq) now perform *invalidation* if the partition state is changed by the local execution (i.e. if $\sigma \neq \sigma'$). The invalidation judgment $\text{inv}(T\ Z, T'\ Z')$ traverses thread pool T and summaries Z and marks all summary computations and all summaries as dirty.

C.4.3 Summary Computations. The system steps corresponding to summary computations are shown in Fig. 18. A new summary can be created for any well-typed combination of operation and argument, by (NewSummary). We made this rule nondeterministic for simplicity, but it would usually happen if there is a corresponding subscription request. (DisposeSummary) can remove the summary, but only if it does not have an active summary computation and no subscribed silos.

(CompStart) starts a new summary computation, adding it to the task pool. It applies only if there is not one already in the task pool. (CompTurn) executes a summary up until the point where it blocks awaiting a placeholder. When a summary finishes, we distinguish based on whether the computed result is different. If it is the same as before, (CompDoneSame) applies, and updates the summary with the read set including dirty flag (which may be different even if the result is the same), but does not send any notification. Otherwise, (CompDone) applies, which sends notifications to all subscribed silos.

C.4.4 Pushing. The system steps corresponding to summary computations are shown in Fig. 19. If the cache entry does not exist, (RcvPush-Gone) applies and sends an unsubscription message

$\boxed{L \mid N \;\rightarrow\; L' \mid N'}$ **System Steps (1/4)**

$$\textsc{ExtCall}\ \frac{i \text{ fresh} \qquad \vdash \mathit{call}\ r\ v \qquad r = i/g.o \qquad g = G\langle v\rangle}{L \mid N \;\rightarrow\; L \mid (\mathit{call}\ r\ v)\ N} \qquad \textsc{Fail-Message}\ \frac{}{L \mid m\ N \;\rightarrow\; L \mid N}$$

$$\textsc{NewSilo}\ \frac{s \text{ fresh}}{L \mid N \;\rightarrow\; (s{:}\ )\ L \mid N} \qquad \textsc{FailSilo}\ \frac{}{(s{:}\ldots)\ L \mid N \;\rightarrow\; L \mid N}$$

$$\textsc{Activate}\ \frac{g \notin \mathit{domX} \cup \mathit{active}(L) \qquad \vdash g : G \qquad G = \langle t\rangle\ a\ \sigma\ D}{(s{:}X\ C)\ L \mid N \;\rightarrow\; (s{:}(g{:}\sigma)\ X\ C)\ L \mid N}$$

$$\textsc{RcvCall}\ \frac{r = \ldots/g.o \qquad g = G\langle v'\rangle \qquad G = \ldots(o{:}f)\ \ldots}{(s{:}(g{:}\sigma\ T\ Z)\ X\ C)\ L \mid (\mathit{call}\ r\ v)\ N \;\rightarrow\; (s{:}(g{:}\sigma\ (r{:}f\ v)\ T\ Z)\ X\ C)\ L \mid N}$$

$$\textsc{RcvRsp}\ \frac{r = g.i/g'.o \qquad T' = T[(\textbf{done}\ v)/r]}{(s{:}(g{:}\sigma\ T\ Z)\ X\ C)\ L \mid (\mathit{rsp}\ r\ v)\ N \;\rightarrow\; (s{:}(g{:}\sigma\ T'\ Z)\ XC)\ L \mid N}$$

$$\textsc{TakeTurn}\ \frac{e, \sigma, N, C \;\rightarrow^{+}_{s,g}\; E[\textbf{await}\ p], \sigma', N', C' \qquad \text{if } \sigma = \sigma' \text{ then } T\ Z = T'\ Z' \text{ else } \mathrm{inv}(T\ Z, T'\ Z')}{(s{:}(g{:}\sigma\ (r{:}e)\ T\ Z)\ X\ C)\ L \mid N \;\rightarrow\; (s{:}(g{:}\sigma'\ (r{:}E[\textbf{await}\ p])\ T'\ Z')\ X\ C')\ L \mid N'}$$

$$\textsc{CompleteReq}\ \frac{e, \sigma, N, C \;\rightarrow^{+}_{s,g}\; v, \sigma', N', C' \qquad \text{if } \sigma = \sigma' \text{ then } T\ Z = T'\ Z' \text{ else } \mathrm{inv}(T\ Z, T'\ Z')}{(s{:}(g{:}\sigma\ (r{:}e)\ T\ Z)\ X\ C)\ L \mid N \;\rightarrow\; (s{:}(g{:}\sigma'\ T'\ Z')\ X\ C')\ L \mid (\mathit{rsp}\ r\ v)\ N'}$$

$\boxed{\mathrm{inv}(T\ Z, T'\ Z')}$

$$\frac{\mathrm{inv}(T, T') \qquad \mathrm{inv}(Z, Z')}{\mathrm{inv}(T\ Z, T'\ Z')} \qquad \frac{}{\mathrm{inv}(,)}$$

$$\frac{\mathrm{inv}(z, z') \qquad \mathrm{inv}(Z, Z')}{\mathrm{inv}(z\ Z, z'\ Z')}$$

$$\textsc{InvSummary}\ \frac{}{\mathrm{inv}((k{:}\ldots(\mathit{val}\ v\ Q)\ldots), (k{:}\ldots(\mathit{val}\ v\ (Q\ \mathit{dirty}))\ldots))}$$

$$\frac{\mathrm{inv}(u, u') \qquad \mathrm{inv}(T, T')}{\mathrm{inv}(u\ T, u'\ T')} \qquad \textsc{RegularTask}\ \frac{}{\mathrm{inv}((r{:}e), (r{:}e))}$$

$$\textsc{InvComp}\ \frac{}{\mathrm{inv}((k{:}e\ Q), (k{:}e\ (Q\ \mathit{dirty})))}$$

Fig. 17. System steps (1/2): execution of normal operations.

back. Otherwise, (RcvPush) applies . It updates the cache with the latest value. Any placeholder appearing in a normal task is replaced with the received result value, by rule (Task). Any placeholder appearing in a summary computation is replaced with the received result value and the read set is extended to record the dependency, by rule (CompRead). Finally, there is an invalidation effect on summaries or summary computations whose read set already includes the updated key k: any such read set is marked dirty by the rules (CompInv) and (SummaryInv), respectively.

C.4.5 Subscription Management. The system steps for subscription managements are in Fig. 20.

C.5 Consistency Guarantees

Informally, the guarantee is that any stale cache (i.e. cache whose current value is not consistent with an atomic execution of the corresponding computation on a global snapshot of the current state) eventually receives a new result, unless the silo that contains the cache fails before that point. A more precise formulation and proof of the consistency guarantees remains future work.

REFERENCES

Umut A. Acar. 2009. Self-adjusting computation (an overview). In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*.

Umut A. Acar, Amal Ahmed, and Matthias Blume. 2008. Imperative self-adjusting computation. In *Principles of Programming Languages (POPL)*.

$\boxed{L \mid N \;\longrightarrow\; L' \mid N'}$ **System steps (2/4)**

$$\text{NewSummary}\;\frac{k = g'/o_c/v \qquad \vdash k \qquad k \notin domZ \qquad Z' = (k:)\;Z}{(s:(g:\sigma\;T\;Z)\;X\;C)\;L \mid N \;\longrightarrow\; (s:(g:\sigma\;T\;Z')\;X\;C)\;L \mid N}$$

$$\text{DisposeSummary}\;\frac{Z = (k:d)\;Z' \qquad \text{nocomp}(k,T)}{(s:(g:\sigma\;T\;Z)\;X\;C)\;L \mid N \;\longrightarrow\; (s:(g:\sigma\;T\;Z')\;X\;C)\;L \mid N}$$

$$\text{CompStart}\;\frac{k = g'/o_c/v \qquad \text{nocomp}(k,T) \qquad \vdash g : G \qquad G \vdash (o_c : f)}{(s:(g:\sigma\;T\;((k:Y\;d)\;Z))\;X\;C)\;L \mid N \;\longrightarrow\; (s:(g:\sigma\;((k:f\;v)\;T)\;(z\;Z))\;X\;C)\;L \mid N}$$

$$\text{CompTurn}\;\frac{e, N, C, Q \;\rightarrow^{+}_{s,g,\sigma}\; E[\textbf{await}\;p], N', C', Q'}{(s:(g:\sigma\;(k:e\;Q)\;T\;Z)\;X\;C)\;L \mid N \;\longrightarrow\; (s:(g:\sigma\;(k:E[\textbf{await}\;p]\;Q')\;T\;Z)\;X\;C')\;L \mid N'}$$

$$\text{CompDone}\;\frac{e, N, C, Q \;\rightarrow^{+}_{s,g,\sigma}\; v, N', C', Q' \qquad \text{notify}_{k,v}(Y, N', N'')}{(s:(g:\sigma\;(k:e\;Q)\;T\;(k:Y\;d)\;Z)\;X\;C)\;L \mid N \;\longrightarrow\; (s:(g:\sigma\;T\;(k:Y\;val\;v\;Q')\;Z)\;X\;C')\;L \mid N''}$$

$$\text{CompDoneSame}\;\frac{e, N, C, Q \;\rightarrow^{+}_{s,g,\sigma}\; v, N', C', Q' \qquad v = v'}{(s:(g:\sigma\;(k:e\;Q)\;T\;(k:Y_{\,val}\,v'\;Q'')\;Z)\;X\;C)\;L \mid N \;\longrightarrow\; (s:(g:\sigma\;T\;(k:Y_{\,val}\,v\;Q')\;Z)\;X\;C')\;L \mid N''}$$

$\boxed{\text{nocomp}(k,T)}$

$$\frac{}{\text{nocomp}(k,)} \qquad \frac{\text{nocomp}(k,T)}{\text{nocomp}(k,(r:e)\;T)}$$

$$\frac{k' \neq k \qquad \text{nocomp}(k,T)}{\text{nocomp}(k,(k':e\;Q)\;T)}$$

$\boxed{\text{notify}_{k,v}(Y,N,N')}$

$$\frac{}{\text{notify}_{k,v}(,N,N)} \qquad \frac{\text{notify}_{k,v}(Y,N,N')}{\text{notify}_{k,v}(z\;Y,N,(push\;s\;k\;v)\;N')}$$

Fig. 18. System steps for summary computations.

$\boxed{L \mid N \;\rightarrow\; L' \mid N'}$ **System Steps (3/4)**

$$\text{RcvPush}\ \frac{\mathrm{upd}_{k,v}(X, X')}{(s{:}X\ ((k{:}\_)\ C))\ L \mid (\mathit{push}\ s\ k\ v)\ N \;\rightarrow\; (s{:}X'\ ((k{:}v)\ C))\ L \mid N}$$

$$\text{RcvPush-Gone}\ \frac{k \notin \mathit{dom}\, C}{(s{:}X\ C)\ L \mid (\mathit{push}\ s\ k\ v)\ N \;\rightarrow\; (s{:}X\ C)\ L \mid (\mathit{unsub}\ s\ k)\ N}$$

$\boxed{\mathrm{upd}_{k,v}(X, X')}$

$$\frac{}{\mathrm{upd}_{k,v}(,)} \qquad \frac{\mathrm{upd}_{k,v}(T, T') \qquad \mathrm{upd}_{k,v}(Z, Z') \qquad \mathrm{upd}_{k,v}(X, X')}{\mathrm{upd}_{k,v}((g{:}\sigma\ T\ Z)\ X, (g{:}\sigma\ T'\ Z')\ X')}$$

$$\frac{\mathrm{upd}_{k,v}(u, u') \qquad \mathrm{upd}_{k,v}(T, T')}{\mathrm{upd}_{k,v}(u\ T, u'\ T')} \qquad \frac{\mathrm{upd}_{k,v}(z, z') \qquad \mathrm{upd}_{k,v}(Z, Z')}{\mathrm{upd}_{k,v}(z\ Z, z'\ Z')}$$

$$\text{Task}\ \frac{e' = e[(\textbf{done}\ v)/k]}{\mathrm{upd}_{k,v}((r{:}e), (r{:}e'))} \qquad \text{CompInd}\ \frac{k \notin Q \qquad k \notin e}{\mathrm{upd}_{k,v}((k'{:}e\ Q), (k'{:}e\ Q))}$$

$$\text{CompRead}\ \frac{k \notin Q \qquad k \in e \qquad e' = e[(\textbf{done}\ v)/k] \qquad Q' = Q\ k}{\mathrm{upd}_{k,v}((k'{:}e\ Q), (k'{:}e'\ Q'))}$$

$$\text{CompInv}\ \frac{k \in Q \qquad e' = e[(\textbf{done}\ v)/k] \qquad Q' = Q\ \mathit{dirty}}{\mathrm{upd}_{k,v}((k'{:}e\ Q), (k'{:}e'\ Q'))} \qquad \text{SummaryEmpty}\ \frac{z = (k'{:}Y)}{\mathrm{upd}_{k,v}(z, z)}$$

$$\text{SummaryInd}\ \frac{z = (k'{:}Y\ \mathit{val}\ v'\ Q) \qquad k \notin Q}{\mathrm{upd}_{k,v}(z, z)} \qquad \text{SummaryInv}\ \frac{z = (k'{:}Y\ \mathit{val}\ v'\ Q) \qquad k \in Q}{\mathrm{upd}_{k,v}(z, (k'{:}Y\ \mathit{val}\ v'\ (Q\ \mathit{dirty})))}$$

Fig. 19. System steps for receiving a push message on a silo.

Umut A. Acar, Guy Blelloch, Ruy Ley-Wild, Kanat Tangwongsan, and Duru Türkoğlu. 2010. Traceable data types for self-adjusting computation. In *Programming Language Design and Implementation (PLDI)*.

Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. 2009. An experimental analysis of self-adjusting computation. *Transactions on Programming Languages and Systems (TOPLAS)* 32 (November 2009), 3:1–3:53. Issue 1.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive Functional Programming. *ACM Trans. Program. Lang. Syst.* 28, 6 (Nov. 2006), 990–1034.

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

Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (Sept. 2010), 68–75.

Albert Benveniste, Benoit Caillaud, and Paul Le Guernic. 2000. Compositionality in Dataflow Synchronous Languages. *Inf. Comput.* 163, 1 (Nov. 2000), 125–171.

Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report MSR-TR-2014-41. Microsoft Research.

Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. 2017. Geo-Distribution of Actor-Based Services. *Proc. ACM Program. Lang.* 1 (October 2017), 107:1–107:26.

Gérard Berry and Georges Gonthier. 1992. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.* 19, 2 (Nov. 1992), 87–152.

Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. 2011. Incoop: MapReduce for Incremental Computations. In *Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, Article 7,

$\boxed{L \mid N \;\longrightarrow\; L' \mid N'}$ **System Steps (4/4): Subscription Management**

$$\text{CastOut}\ \frac{k \notin X}{(s{:}X\ (k{:}v)\ C)\ L \mid N \;\longrightarrow\; (s{:}X\ C)\ L \mid (\mathit{unsub}\ s\ k)\ N}$$

$$\text{RenewSubscription}\ \frac{}{(s{:}X\ (k{:}v)\ C)\ L \mid N \;\longrightarrow\; (s{:}X\ (k{:}v)\ C)\ L \mid (\mathit{renew}\ s\ k)\ N}$$

$$\text{ResendSubscription}\ \frac{}{(s{:}X\ (k{:})\ C)\ L \mid N \;\longrightarrow\; (s{:}X\ (k{:})\ C)\ L \mid (\mathit{sub}\ s\ k)\ N}$$

$$\text{Unsubscribe}\ \frac{k = g/o_c/v \qquad X' = (g{:}\sigma\ T\ (k{:}Y'\ d)\ Z)\ X \qquad Y' = Y \setminus s'}{(s{:}(g{:}\sigma\ T\ (k{:}Y\ d)\ Z)\ X\ C)\ L \mid (\mathit{unsub}\ s'\ k)\ N \;\longrightarrow\; (s{:}X'\ C)\ L \mid N}$$

$$\text{UnsubscribeFailed}\ \frac{k = g/o_c/v \qquad s' \text{ failed} \qquad X' = (g{:}\sigma\ T\ (k{:}Y'\ d)\ Z)\ X \qquad Y' = Y \setminus s'}{(s{:}(g{:}\sigma\ T\ ((k{:}Y\ d)\ Z))\ X\ C)\ L \mid N \;\longrightarrow\; (s{:}X'\ C)\ L \mid N}$$

$$\text{Subscribe}\ \frac{k = g/o_c/v \qquad z = (k{:}Y) \qquad z' = (k{:}s'\ Y)}{(s{:}(g{:}\sigma\ T\ (z\ Z))\ X\ C)\ L \mid (\mathit{sub}\ s'\ k)\ N \;\longrightarrow\; (s{:}(g{:}\sigma\ T\ (z'\ Z))\ X\ C)\ L \mid N}$$

$$\text{Subscribe-Send}\ \frac{k = g/o_c/v \qquad z = (k{:}Y\ \mathit{val}\ v'\ Q) \qquad z' = (k{:}(s'\ Y)\mathit{val}\ v'\ Q)}{(s{:}(g{:}\sigma\ T\ (z\ Z))\ X\ C)\ L \mid (\mathit{sub}\ s'\ k)\ N \;\longrightarrow\; (s{:}(g{:}\sigma\ T\ (z'\ Z))\ X\ C)\ L \mid (\mathit{push}\ s\ k\ v')\ N}$$

$$\text{Renew-Ok}\ \frac{k = g/o_c/v \qquad z = (k{:}Y\ d) \qquad s' \in Y}{(s{:}(g{:}\sigma\ T\ (z\ Z))\ X\ C)\ L \mid (\mathit{renew}\ s'\ k)\ N \;\longrightarrow\; (s{:}(g{:}\sigma\ T\ (z\ Z))\ X\ C)\ L \mid N}$$

$$\text{Renew-Resubscribe}\ \frac{k = g/o_c/v \qquad z = (k{:}Y) \qquad s' \notin Y \qquad z' = (k{:}s'\ Y)}{(s{:}(g{:}\sigma\ T\ (z\ Z))\ X\ C)\ L \mid (\mathit{renew}\ s'\ k)\ N \;\longrightarrow\; (s{:}(g{:}\sigma\ T\ (z'\ Z))\ X\ C)\ L \mid N}$$

$$\text{Renew-Resend}\ \frac{k = g/o_c/v \qquad z = (k{:}Y\mathit{val}\ v'\ Q) \qquad s' \notin Y \qquad z' = (k{:}(s'\ Y)\mathit{val}\ v'\ Q)}{(s{:}(g{:}\sigma\ T\ (z\ Z))\ X\ C)\ L \mid (\mathit{renew}\ s'\ k)\ N \;\longrightarrow\; (s{:}(g{:}\sigma\ T\ (z'\ Z))\ X\ C)\ L \mid (\mathit{push}\ s\ k\ v)\ N}$$

Fig. 20. System steps for subscription management.

14 pages.

Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1-2 (Oct. 2014), 1–150.

Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's Alive! Continuous Feedback in UI Programming. In *Programming Language Design and Implementation (PLDI)*. 95–104.

Sebastian Burckhardt, Daan Leijen, Jaeheon Yi, Caitlin Sadowski, and Tom Ball. 2011. Two for the Price of One: A Model for Parallel and Incremental Computation. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *ACM Symposium on Cloud Computing (SOCC '11)*. Article 16, 14 pages.

Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. LUSTRE: A Declarative Language for Real-time Programming. In *Principles of Programming Languages (POPL)*. 178–188.

Wei-Chiu Chuang, Bo Sang, Sunghwan Yoo, Rui Gu, Milind Kulkarni, and Charles Killian. 2013. EventWave: Programming Model and Runtime Support for Tightly-coupled Elastic Cloud Applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 21, 16 pages. https://doi.org/10.1145/2523616.2523617

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce Online. In *Networked Systems Design and Implementation (NSDI)*. USENIX Association, Berkeley, CA, USA, 21–21.

Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-value Language. In *European Symposium on Programming (ESOP)*. Springer-Verlag, Berlin, Heidelberg, 294–308.

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

Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. 2014. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures* 40, 3-4 (2014), 112–136. https://doi.org/10.1016/j.cl.2014.05.002

Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Programming Language Design and Implementation (PLDI)*. 411–422.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

Camil Demetrescu, Sebastian Erdweg, Matthew A. Hammer, and Shriram Krishnamurthi. 2016. Programming Language Techniques for Incremental and Reactive Computing (Dagstuhl Seminar 16402). *Dagstuhl Reports* 6, 10 (2016), 1–12. https://doi.org/10.4230/DagRep.6.10.1

Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. 2011. Reactive Imperative Programming with Dataflow Constraints. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 407–426.

Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 16.

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *International Conference on Functional Programming (ICFP) (ICFP '97)*. 263–273.

Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* 35, 2 (June 2003), 114–131.

Cormac Flanagan and Matthias Felleisen. 2001. The Semantics of Future and Its Use in Program Optimization. (09 2001).

Flink 2016. Apache Flink. https://flink.apache.org/.

Thierry Gautier, Paul Le Guernic, and Löic Besnard. 1987. SIGNAL: A Declarative Language for Synchronous Programming of Real-time Systems. In *Functional Programming Languages and Computer Architecture*. 257–277.

Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *Operating Systems Design and Implementation (OSDI) (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 75–88.

Matthew A. Hammer, Umut A. Acar, and Yan Chen. 2009. CEAL: a C-based language for self-adjusting computation. In *Programming Language Design and Implementation (PLDI)*.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. *Arrows, Robots, and Functional Reactive Programming*. Springer Berlin Heidelberg, 159–187.

Daniel Ignatoff, Gregory H. Cooper, and Shriram Krishnamurthi. 2006. *Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 259–276.

Sean McDirmid and Wilson C. Hsieh. 2006. *SuperGlue: Component Programming with Object-Oriented Signals*. Springer, 206–229.

Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential dataflow. In *Proceedings of CIDR 2013*.

Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 1–20.

Luc Moreau. 1970. The Semantics of Scheme with Future. 31 (02 1970).

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

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

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

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

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

Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. 2014. An Empirical Study on Program Comprehension with Reactive Programming. In *International Symposium on Foundations of Software Engineering (FSE)*. 564–575.

Guido Salvaneschi and Mira Mezini. 2013. Reactive Behavior in Object-oriented Applications: An Analysis and a Research Roadmap. In *International Conference on Aspect-oriented Software Development (AOSD)*. New York, NY, USA, 37–48.

Bo Sang, Gustavo Petri, Masoud Saeida Ardekani, Srivatsan Ravi, and Patrick Eugster. 2016. Programming Scalable Cloud Services with AEON. In *Proceedings of the 17th International Middleware Conference (Middleware '16)*. ACM, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2988336.2988352

SF Reliable Actors 2016. 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/.

Zhanyong Wan and Paul Hudak. 2000. Functional Reactive Programming from First Principles. *SIGPLAN Not.* 35, 5 (May 2000), 242–252.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 15–28.