# Isolates, Channels, and Event Streams for Composable Distributed Programming

Aleksandar Prokopec
École polytechnique fédérale de Lausanne, Switzerland
aleksandar.prokopec@alumni.epfl.ch

Martin Odersky
École polytechnique fédérale de Lausanne, Switzerland
martin.odersky@epfl.ch

## Abstract

The actor model has been a model of choice for building reliable distributed systems. On one hand, it ensures that message-processing is serialized within each actor, preserving the familiar sequential programming model. On the other hand, programs written in the actor model are location-transparent. The model is sufficiently low-level to express arbitrary message protocols. Composing these protocols is the key to high-level abstractions. Unfortunately, it is difficult to reuse or compose message protocols with actors.

*Reactive isolates*, proposed in this paper, simplify protocol composition with first-class typed channels and event streams. We compare reactive isolates and the actor model on concrete programs. We identify obstacles for composition in the classic actor model, and show how to overcome them. We then show how to build reusable, composable distributed computing components in the new model.

***Categories and Subject Descriptors*** Software [*Programming Techniques*]: Distributed Programming; Programming Languages [*Language Classifications*]: Concurrent, distributed, and parallel languages

***General Terms*** distributed programming, message passing

***Keywords*** actors, distributed, event streams, isolates

## 1. Introduction

Today, there exists a gap between high-level distributed computing frameworks and low-level distributed programming models. High-level frameworks such as Map-Reduce [10], distributed file-systems and databases, and peer-to-peer networks [30] are used to build programs for specific domains. Low-level distributed programming models, such as remote procedure calls (RPCs) [33], and actors [5], form the basis for building distributed systems. There does not seem to be a strong middle ground – a set of reusable intermediate components is missing. High-level frameworks are complex systems, built from low-level primitives during countless engineer hours, whose efforts are repeated every time a new distributed system is created.

This gap between the high-level and the low-level distributed computing did not significantly decrease in the last 30 years, since the formalization of the actor model [5]. Where sequential programmers today build their programs from iterators, monads, zippers, generic collection frameworks, parser combinators, I/O libraries, and UI toolkits, distributed systems engineers think in terms of low-level RPCs and message passing. While sequential programming paradigms realized the importance of structured programming and high-level abstractions long ago [11] [21], distributed computing has still not moved far from message passing – its own assembly. To date, there exist middleware frameworks tailored towards specific tasks, such as Erlang/OTP [3], various message queues or generic application servers, but no unified standard library of reusable high-level components. This situation is rooted in the following: *existing low-level distributed programming models expose primitives that do not compose well.*

Distributed systems consist of message exchanges. *Message protocols* are specific patterns of message exchanges between concurrent computations. In the past, researchers identified an entire ecosystem of different message protocols [13] [19]. These message protocols should be exposed as reusable components of distributed programs, and should be composable. Protocols compose if they can be implemented independently, and combined into more complex protocols.

In this paper, we present constructive criticism of the actor programming model, and propose a solution that addresses those complaints. Our goal is not to rebuke the established distributed programming methodologies, but to suggest the necessary next steps in their evolution. Concretely:

- We propose a programming model based on isolates, typed event channels and event streams. The model is minimalistic, but sufficient to build stronger abstractions.

*Onward!'15*, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3688-8/15/10...$15.00
http://dx.doi.org/10.1145/2814228.2814245

- We compare reactive isolates model with the actor model, and identify specific obstacles for composability, outlined in Section 2, and studied in Section 4. We show how to overcome these obstacles with reactive isolates.
- We show how to build a stack of distributed programming abstractions in a composable way using reactive isolates.

Semantics of event streams are explained in previous work, which shows how to compose event streams and use them in incremental data structures in the context of *single-threaded reactive programming* [28]. This paper shows detailed semantics of isolates and channels as foundational abstractions for *distributed programming*, and shows how channels and event streams compose into already established software components for distributed computing.

Our framework [4] is implemented in the Scala programming language [22]. Scala is a statically typed language running on the JVM and is similar to most mainstream languages. Variables are declared with `var` like in JavaScript or C#, and their type is inferred from the initialization. Methods are defined with the keyword `def` like in Python or Ruby, and type ascriptions come after a `:` sign. First-class functions consist of an argument list followed by a body after `=>`, similar to Java 8 lambdas. Keyword `val` denotes fields and local variables that cannot be reassigned. The keyword `trait` denotes an `interface`, but traits can also have concrete members. Generic types come in `[]` brackets, similar to `<>` in other languages. Type `Unit` is the analogue of the `void` type in C or Java. Statements in the body of a `class` are parts of its primary constructor. In several places we refer to specific collection types, such as maps, sets, sequences and buffers. Although these are a part of the Scala collection library [23], their equivalents exist in most other languages.

The paper is organized as follows. In Section 2, we present two existing actor models, and show the reactive isolates in Section 3. In Section 4, we compare actors and reactive isolates, and outline our vision in Section 5. We survey related work in Section 6, and conclude in Section 7.

## 2. Actor Model

Erlang is a programming language used for building fault-tolerant applications [32]. Units of concurrent execution are called *processes* in Erlang. Processes communicate only by exchanging messages – shared memory synchronization is very limited. Each process has a process ID, which is required by the *tell* operation (`!`) to send messages. In the following, we send a message `"Hello"` to the process `Server`:

```
Server ! "Hello"
```

Processes are created with the `spawn` method. Every process can call the `receive` method, which suspends execution until one of the matching messages arrives. In the following, we start the `Server` process that suspends until it receives the `"Hello"` message:

```
server() ->
  receive
    "Hello" -> io::format("Hola.~n");
  end.
Server = spawn(?MODULE, server, []);
```

Akka [2] is an actor framework for the JVM, which borrows much of its design from Erlang. Units of concurrent execution are called *actors*, and can send messages with `!`. Each actor has an actor reference, an equivalent of Erlang's process ID. Every actor has a top-level message-handling loop – while `receive` calls can nest in Erlang, an Akka actor only suspends after the top-level `receive` block completes:

```
class Server extends Actor {
  def receive = {
    case "Hello" => println("Holla.") } }
```

Each `receive` block has a set of cases – an actor resumes when an incoming message can be matched to some case. Messages not conforming to the specified cases are either discarded, as is the case in Akka, or buffered until the actor decides to process them, as is the case in Erlang. Since in Akka messages can arrive before their message-handling case is installed, and in Erlang messages can be buffered without being received, these approaches are a tradeoff between potential race conditions and memory leaks.

The actor model has multiple advantages, and we outline those that are relevant for this paper. The first advantage is *location-transparency* – actor programs can be implemented on a single machine and then deployed on multiple machines connected with a network. Although program performance depends on the network layout and can be compromised when the actors are distributed on different machines, program correctness typically does not change.

The second advantage is that incoming messages are processed *serially* within each actor. Although programmers need to reason about different message delivery orders, they do not need to worry about race conditions when manipulating the state of an actor. This is illustrated in the following:

```
class Counter extends Actor {
  var count = 0
  def receive = {
    case "inc" => count += 1
  }
}
val counter = actorOf(Counter())
class Inc() extends Actor {
  def receive = PartialFunction.empty
  counter ! "inc"
}
for (i <- 0 until 2) actorOf(Inc())
```

Above, the two actors of type `Inc` simultaneously send an `"inc"` message to the `Counter` actor, which increments the `count` variable each time it receives a message. The messages can be received in any order, but they are processed one after the other. The `count` variable is never accessed concurrently, and is always `2` when the program ends.

| Signature | Description |
|---|---|
| `onEvent(f: T => Unit): Unit` | Executes the function `f` when an event arrives. |
| `onCase(pf: PartialFunction[T, Unit]): Unit` | Executes the function `pf` if the event matches a case. |
| `on(b: =>Unit): Unit` | Executes the function `b`, which ignores the event value. |
| `map[S](f: T => S): Events[S]` | Returns an event stream with events of type `S`, where every event is mapped using the function `f`. |
| `zip[S](that: Events[S]): Events[(T, S)]` | Returns an event stream with events of type `(T, S)`, triggered after an event is emitted on both `this` and `that`. |
| `union[T](xs: Seq[Events[T]]): Events[T]` | Returns an event stream that forwards events from all the event streams in the `xs` collection. |

**Table 1.** Examples of Event Stream Operations

Another important advantage of the actor model is that message sends are *non-blocking*. After an actor sends a message, it does not suspend execution until the message reaches its destination. Asynchronous message sends are one of the preconditions for scalability [7].

The actor model also has some limitations. First, in the basic actor model, actors cannot simultaneously contain *multiple message entry points*. Separate protocols within the actor must be encoded in a single message-handling construct, and consequently need to be aware of each other.

Second, in the basic actor model, actors cannot *await specific combinations of messages*. The `receive` block suspends computation until one of the specified message types arrives, but cannot suspend until a pair of messages arrives.

Third, `receive` is not a *first class entity*. Instead of a value that can be passed to and returned from functions, `receive` is a static construct.

These limitations, which we study closely in Section 4, do not entirely prevent modularity in actor programs. Separation of concerns is typically achieved by dividing responsibilities across different actors, while composition is achieved through message passing – actors are clustered into groups that exchange messages. However, as we argue in Section 4, separating responsibilities across actors makes programs complicated and error-prone. At the same time, these limitations make protocol composition within a single actor cumbersome, convolute abstractions and restrict code reuse.

## 3. Reactive Isolates Model

In this section, we present the reactive isolates model, whose goal is to address the aforementioned limitations of the actor model. The model retains properties such as asynchronicity, location-transparency and serial message handling. To avoid ambiguities with actors, we refer to messages as *events*.

The reactive isolates model describes programs in terms of three basic abstractions: *isolates*, *channels* and *event streams*. An isolate specifies a concurrent computation and the associated program state. To communicate, isolates send events to channels, which represent communication paths. Every channel is owned by a single isolate. Any isolate can send an event to a channel, but only the channel owner can process that event. Communication is split into inter-isolate event propagation based on channels, and intra-isolate event propagation based on event streams. Event streams are entities that propagate events within an isolate, and they cannot be shared between isolates. An event stream is associated to every channel. Event streams can be composed with declarative combinators, and are used to implement operations on incremental data structures, as shown in previous work [28].

An isolate, represented with the type `Iso[T]`, is a set of channel and event stream pairs that manipulate the isolate state. A new isolate is created with the `isolate` function. With respect to the actor model, an isolate corresponds to an actor. The following program declares an isolate `Hello`, which prints to the standard output after it starts.

```
class Hello extends Iso[Unit] {
  sysEvents onCase {
    case IsoStarted =>
      println("Hello␣World!")
  }
}
sys.isolate(Hello)
```

When created, an isolate comes with the default channel and event stream pair, called `channel` and `events`, which handle events of type `T`. Additionally, an isolate has a system channel and event stream pair, called `sysChannel` and `sysEvents`, used for events such as the start or the termination of the isolate. The `Hello` isolate extends `Iso[Unit]`, so it can only receive `Unit` events from `events`.

A channel, represented with the type `Channel[T]`, is used to send events of type `T` to the channel owner. Events are sent using the channel's `!` method. An isolate can gain additional channels with the `open` method, which returns a new channel and event stream pair, called a *connector*:

```
def open[T]: (Channel[T], Events[T])
```

Channels can be sent to other isolates, enabling them to communicate with the channel owner. Owner closes the channel by calling `seal`. An isolate terminates after its channels are sealed and pending events processed. With respect to the actor model, channels correspond to actor references.

$$\frac{t = \texttt{isolate(f)}\,;\,t' \quad \texttt{f: (Id, (Channel[T], Events[T])) => Unit} \quad \texttt{(c, r)} = \texttt{open[T]}}{E \cup (t, i) \mid S \longrightarrow E \cup (\texttt{c}\,;\,t', i) \cup (\texttt{f(fresh, (c, r))}, \emptyset) \mid S} \qquad (\textsc{Spawn})$$

$$\frac{t = \texttt{open[T]}\,;\,t' \quad \texttt{c: Channel[T]} \quad \texttt{r: Events[T]}}{E \cup (t, i) \mid S \longrightarrow E \cup (\texttt{(c, r)}\,;\,t', i \cup (\epsilon, \texttt{c}, \texttt{r})) \mid S} \qquad (\textsc{Open})$$

$$\frac{}{E \cup (\texttt{v}, i) \mid S \longrightarrow E \mid S \cup (\epsilon, i)} \qquad (\textsc{Sleep})$$

$$\frac{i = i' \cup (Q \cdot \texttt{x}, \texttt{c}, \texttt{r}) \quad r = \{f_1, f_2, \ldots, f_n\} \quad t = f_1(\texttt{x})\,;\,f_2(\texttt{x})\,;\,\ldots\,;\,f_n(\texttt{x})}{E \mid S \cup (\epsilon, i) \longrightarrow E \cup (t, i' \cup (Q, \texttt{c}, \texttt{r})) \mid S} \qquad (\textsc{Awake})$$

$$\frac{t = \texttt{c ! x}\,;\,t' \quad \texttt{c: Channel[T]} \quad \texttt{x: T} \quad \nexists \texttt{X.Events[X]} \in \texttt{T}}{E \cup (t, i) \mid S \cup (\epsilon, j \cup (Q, \texttt{c}, \texttt{r})) \longrightarrow E \cup (t', i) \mid S \cup (\epsilon, j \cup (\texttt{x} \cdot Q, \texttt{c}, \texttt{r}))} \qquad (\textsc{Send1})$$

$$\frac{t = \texttt{c ! x}\,;\,t' \quad \texttt{c: Channel[T]} \quad \texttt{x: T} \quad \nexists \texttt{X.Events[X]} \in \texttt{T}}{E \cup (t, i) \cup (u, j \cup (Q, \texttt{c}, \texttt{r})) \mid S \longrightarrow E \cup (t', i) \cup (u, j \cup (\texttt{x} \cdot Q, \texttt{c}, \texttt{r})) \mid S} \qquad (\textsc{Send2})$$

$$\frac{t = \texttt{r onReaction f}\,;\,t' \quad \texttt{r: Events[T]} \quad \texttt{f: T => Unit}}{E \cup (t, i \cup (Q, \texttt{c}, \texttt{r})) \mid S \longrightarrow E \cup (t', i \cup (Q, \texttt{c}, \texttt{r} \cup \texttt{f})) \mid S} \qquad (\textsc{React})$$

**Figure 1.** Operational Semantics of Reactive Isolates

The following program creates two isolates that execute the *ping* protocol – `Pingy` isolate sends a `"ping"` event, to which `Pongy` must respond with a `"pong"` event.

`Pongy` accepts `(String, Channel[String])` pairs, with a string and the sender channel. When `Pongy` receives an event `"ping"`, it sends back a `"pong"` and seals its channel.

```
class Pongy
extends Iso[(String, Channel[String])] {
  events onCase {
    case ("ping", sender) =>
      sender ! "pong"
      channel.seal()
  }
}
```

`Pingy` takes `Pongy`'s channel as an argument `p`. After `Pingy` starts, it sends a pair with `"ping"` and its channel to `Pongy`. When `Pingy` receives an answer `"pong"`, it seals its channel and the program terminates.

```
class Pingy(
  val p: Channel[(String, Channel[String])]
) extends Iso[String] {
  sysEvents onCase {
    case IsoStarted => p ! ("ping", channel)
  }
  events onCase {
    case "pong" => channel.seal()
  }
}
sys.isolate(Pingy(sys.isolate(Pongy)))
```

Event streams have the type `Events[T]`, where `T` is the type of their events. An event stream exists only within a specific isolate. Two event streams in different isolates concurrently propagate events, but at most one event stream simultaneously propagates events within one isolate. An event stream corresponds to the actor message-handling loop.

Whenever an event stream propagates an event, we say that it *reacts*. At any point, an event stream can *unreact*, indicating that it will not propagate any additional events. A `Events[T]` has a method `onReaction` that takes a pair of functions `T => Unit` and `() => Unit` invoked when the event stream reacts or unreacts, respectively. The `onReaction` method is used to implement convenience methods on `Events[T]`, such as `onCase`, which takes a *partial function*, defined only for some events:

```
def onCase(pf: PartialFunction[T, Unit]) =
  onReaction(
    ev => if (pf.isDefinedAt(ev)) pf(ev),
    () => {})
```

The `onReaction` method is also used in methods that transform event streams. For example, the `map` method, given an event stream of type `Events[T]` and a function `T => S`, creates an event stream of type `Events[S]`:

```
def map[S](f: T => S): Events[S] = {
  val (ch, result) = open[S]
  this.onReaction(x => ch ! f(x), () => {})
  result }
```

The following `Logger` isolate transforms events into strings, and appends them into a buffer with the `+=` method.

```
class Logger[T] extends Iso[T] {
  val log = new Buffer[String]()
  events.map(x => x.toString).onEvent(log +=)
}
```

We use several other event stream transformers throughout the paper. A list of relevant transformers and their meaning is shown in Table 1.

So far, we have left out some details. For example, in our implementation, the `isolate` statement accepts a `Scheduler` argument, which decides when the isolate executes. We do not study such features, as they are not essential for understanding the programming model.

### 3.1 Operational Semantics

To precisely define the reactive isolates model, we present its operational semantics as a set of reduction rules. Since our programming model is implemented as a Scala library, these reduction rules can extend existing Scala core calculi [6] [8]. Alternatively, they can be used to extend more general models, such as the simply typed lambda calculus [24].

We show formal operational semantics of a simplified reactive isolates model in Figure 1, which introduces the `isolate`, `open`, `!` and `onReaction` statements. Isolate programs are represented as a pair of isolate sets called *executing* and *sleep*, denoted $E$ and $S$. An isolate is represented as a tuple with the currently executing term $t$, and the set $i$ of event queue $Q$, channel $s$ and event stream $r$ triples. Event streams are represented as sets of callback functions. The sleep set contains only isolates whose term is empty ($\epsilon$). The program terminates when the executing set $E$ is empty, and all the isolates in the sleep set $S$ have empty event queues $Q$.

The evaluation rule SPAWN reduces the `isolate` invocation by adding a new isolate to $E$. The OPEN rule reduces the `open` invocation into a channel and event stream tuple, and adds the same tuple with an empty event queue to the set $i$. The SLEEP rule is triggered when the isolate reduces its term to a value. SLEEP moves the isolate to $S$. The AWAKE rule does the opposite – if there is a non-empty event queue, it invokes the corresponding callbacks on the first element in the queue. The SEND1 and SEND2 rules send an event `x` to a channel `c`. They can only trigger if `Events` is not a part of the event – event streams cannot be shared across isolate boundaries. Finally, the REACT rule adds a callback function into the event stream.

The rest of the rewrite rules deal with sequential term reduction, which is unrelated to concurrency aspects of reactive isolates, and treated elsewhere [6] [8] [24].

## 4. Comparison Between Actors and Reactive Isolates

In this section, we study aspects of program composition in actors and reactive isolates. We expect that the protocols are *isolated*. Two separately implemented protocols should work correctly within the same actor or isolate. Then, expressing protocols that involve *multiple parties* should be concise and straightforward. Finally, protocols should *compose* – it should be possible to combine protocol instances to form more complex protocols.

For comparison, we need to choose a specific actor model. Although we will use Akka to compare the two programming models, we note that a similar comparison can be made with Erlang and other actor models.

### 4.1 Protocol Isolation

To build distributed systems in a scalable way, separate message protocols in the same actor must be isolated. In this section, we consider how to achieve this isolation on an example of name resolution protocol.

In the *request-reply* communication pattern, a machine sends a request to another machine, which then sends back a response. Request-reply is the basic part of the client-server model. In Akka, we declare a generic server actor as follows:

```
class Server[T, S](f: T => S) extends Actor {
  def receive = { case x:T => sender ! f(x) }
}
```

A generic server is an actor that receives requests of type `T`, and replies with objects of type `S`, where `T` and `S` are type parameters. The `Server` actor uses the function `f` and the special value `sender` to send back a response to the client.

We similarly define a client that takes the actor reference `server`, a request of type `T` and a function `g` that accepts the reply. The `preStart` method is overridden to send a request to the server, and `receive` invokes `g` with the reply.

```
class Client[T, S](
  server: ActorRef, x: T, g: S => Unit
) extends Actor {
  override def preStart() { server ! x }
  def receive = { case y: S => g(y) }
}
```

The *identify actor* protocol is a specific instance of the request-reply protocol, where the server keeps a map from actor names to actor references, and provides them to clients, much like a DNS server. In the following, the `names` map is a function of type `String => ActorRef`, so we use it to create a `Server` actor. We then define the `Client` actor that requests the reference for the actor called `"/p"`, and prints it.

```
val names = Map[String, ActorRef]()
val reg = actorOf(Server(names))
val p = actorOf(Client(reg, "/p", println))
```

This pattern is not very reusable. Let's say that we want to address contention on `Server` with a `Cache` actor that caches a particular actor reference `"/p"`. `Cache` is a specific instance of the `Client` actor, which stores the server response into a variable `cached`. Occasionally, the name server changes, so `Cache` receives a reference to the new server, in which case it needs to update its cache.

```
class Cache(var cached: ActorRef = null)
extends Client(reg, "/p", r => cached = r) {
  def receive = super.receive orElse {
    case newReg: ActorRef => newReg ! "/p"
  }
}
```

`Cache` uses the `orElse` method to add a new message-handling case to `receive` defined in `Client`. In doing so, `Cache` encodes two protocols: a request-reply to obtain the actor reference, and a *push* protocol that updates its state.

However, `Cache` implementation is not correct. `Cache` cannot correctly disambiguate between the actor reference sent from the name server `reg`, and the new name server reference. The `super.receive` consumes all `ActorRef` messages, and the case defined in `Cache` is never invoked. The two protocols are not isolated.

To avoid this, the actual Akka implementation uses `ActorIdentity` objects to encapsulate the identify actor protocol. Only a name server can send an `ActorIdentity` message. In the following, the `Printer` actor receives an `ActorIdentity` message with an actor reference `ref`.

```
class Printer extends Actor {
  def receive = {
    case ActorIdentity("print", Some(ref)) =>
      println(ref)
  }
}
```

Actor models achieve protocol isolation through protocol-specific wrapper types, such as `ActorIdentity` above. This approach is lacking when there are multiple instances of the *same* protocol in the same actor. For example, if two protocols both expect `ActorIdentity`, there is a danger that one will consume the message intended for the other. This is why the `ActorIdentity` object in the previous program also contains a `"print"` value, called a *tag*. Different protocols must take care not to reuse the same tag value.

In reactive isolates there is no need for protocol-specific wrappers or message tags. To show this, we encode the request-reply pattern as a channel that takes request object and reply channel pairs:

```
type Req[T, S] = Channel[(T, Channel[S])]
```

Method `server`, given a request-reply function `f`, creates a new channel `ch` and event stream `events`. The event stream computes a reply using `f` and sends it on the specified channel. Finally, `server` returns the channel `ch`.

```
def server[T, S](f: T => S): Req[T, S] = {
  val (ch, events) = open[(T, Channel[S])]
  events onEvent { case (x, c) => c ! f(x) }
  ch
}
```

The `Req[T, S]` channel is used with the `?` operator, which sends a event of type `T` to the server, and returns an event stream of type `Events[S]` with the server reply:

```
def ?[T, S](req: Req[T, S], x: T) = {
  val (chan, events) = open[(T, Channel[S])]
  req ! (x, chan)
  events
}
```

The *identify channel* protocol retrieves channels given their names. It is a specific instance of the request-reply protocol. We define a `NameServer`, which instantiates the request-reply channel from channel names to channels.

```
class NameServer
extends Iso[(String, Channel[Channel[_]])] {
  val s = server(Map[String, Channel[_]]())
  events.onEvent(x => s ! x)
}
val nameServer = sys.isolate(NameServer)
```

The `Printer` isolate uses the `?` operator to send a name request to the `nameServer` instance. The resulting event stream `response` of type `Events[Channel[_]]` eventually emits the channel and prints it:

```
class Printer extends Iso[Unit] {
  sysEvents onCase {
    case IsoStarted =>
      val response = nameServer ? "/p"
      response.onEvent(println)
  }
}
```

Above, `response` is tied to the specific invocation of the `?` operator. Only events corresponding to that invocation are delivered to `response`, and different protocols invoking `?` cannot see each other's events.

Actors provide weaker protocol isolation, because they have a *single message entry point*. Their protocol reuse is burdened with protocol-specific types and message tags.

## 4.2 Multiple Party Protocols

Some message exchange protocols involve more than two participants. Consider an `Authenticator` actor that receives a `Login` message, and then simultaneously sends two messages `GetCert` and `GetAuth` to the key server and the authorization server, respectively. Upon receiving both a certificate response `Cert` and an authorization `Auth`, the server computes a token and returns it to the requester. In Akka, one way to implement `Authenticator` is as follows:

```
class Authenticator extends ActorRef {
  var user: ActorRef = null
  var cert: Cert = null
  var auth: Auth = null
  val tokens = Map[ActorRef, String]()
  def check() {
    if (cert != null && auth != null) {
      tokens(user) = compute(cert, auth)
      user ! tokens(user) } }
  def receive = {
    case Login(u) =>
      user = u
      keyCenter ! GetCert(u)
      authServer ! GetAuth(u)
    case c: Cert => cert = c; check()
    case a: Auth => auth = a; check() } }
```

This implementation manually stores the `user` actor reference, and whether the `Cert` and `Auth` messages have arrived. To avoid this, Akka defines its own `?` operator, which sends a message to the target actor, and returns an object called *a future*. A future is a placeholder for asynchronously created values [15]. The `?` operator returns a future object containing the reply from the destination actor:

```
(kc ? GetCert(u)): Future[Any]
```

Sending the `GetCert` message to the `keyCenter` returns a future of type `Future[Any]`, where `Any` is the top type. This is because the type of the reply is not known in advance. The `mapTo` operation casts the future to a specific type:

```
(kc ? GetCert(u)).mapTo[Cert]: Future[Cert]
```

We use futures to express `Authenticator` more elegantly. Upon creating the futures `fc` and `fa` with the certificate and the authorization, we use them in a *for-comprehension* to create a future that contains a user, certificate and authorization triple. Parts of this for-comprehension execute asynchronously, when `fc` and `fa` are completed. Since futures are an external concurrency framework that is unaware of the serial message-handling in actors, modifying the `tokens` map from the for-comprehension body is illegal, as it possibly executes concurrently with the actor. Instead, we have to use the `pipeTo` operator to send the triple in the resulting future `ft` back to the actor. The actor then processes the triple as part of its message-handling loop:

```
def receive = {
  case Login(u) =>
    val fc = keyCenter ? GetCert(u)
    val fa = authServer ? GetAuth(u)
    val ft = for {
      cert <- fc.mapTo[Cert]
      auth <- fa.mapTo[Auth]
    } yield (u, cert, auth)
    ft.pipeTo(self)
  case (u: ActorRef, c: Cert, a: Auth) =>
    tokens(user) = compute(c, a)
    user ! tokens(user)
}
```

Omitting the call to `pipeTo` is a source of subtle errors in Akka programs, which manifest themselves as data races when accessing the shared state of the actor.

In reactive isolates, different event streams can independently receive events, and compose into complex event streams that trigger on combinations of events. Serial event stream semantics eliminate the possibility of a data race within an isolate. In the `Authenticator` isolate, we use the `zip` combinator that produces pairs of events arriving from the two input event streams. Events are emitted serially, so there is no danger of concurrently accessing the isolate state:

```
events onCase {
  case Login(u) =>
    val ec = keyCenter ? GetCert(u)
    val ea = authServer ? GetAuth(u)
    (ec zip ea) onCase { case (cert, auth) =>
      tokens(user) = compute(cert, auth)
      user ! tokens(user)
    }
}
```

The actor implementation is verbose because an actor cannot await for *specific combinations of message types*. Futures are a step in the right direction to address this problem, but they break the serial semantics and can only receive a single reply. Isolates use event stream composition to overcome the need for a dedicated multi-receive construct.

## 4.3 Protocol Composition

We next consider how to compose protocols in the reactive isolates model. Composition allows building complex systems from independently developed components. The fact that channels and event streams are first class entities simplifies composition. Consider the `server` method defined in Section 4.1 – it satisfies the following type:

```
type Server[T, S] = (T => S) => Req[T, S]
```

This reads: given a function from `T` to `S`, return a request-reply channel. Generally, a protocol can be expressed as a function type, and its implementation as a function instance. Instantiating a protocol implementation corresponds to applying the function to its arguments. However, we choose an alternative, arguably more intuitive representation of message protocols, based on Scala traits. Traits are similar to Java interfaces, but can additionally have concrete members. The alternative encoding of `Server` using traits is as follows:

```
trait Server[T, S] {
  val mapping: T => S
  val channel: Req[T, S]
}
```

We define a generic protocol as a trait with a method that retrieves the isolate system, and a method that retrieves the unique ID of the isolate. All other protocols extend this trait:

```
trait Protocol {
  def sys: IsoSystem
  def id: Id
}
```

In what follows, we encode several classic distributed computing algorithms as protocols in the reactive isolates model to show that they compose. Recall that a distributed system spans multiple machines, some of which may at any point stop working. A distributed system is *fault-tolerant* if it can continue to function after some of the machines crash. A *failure detector* [13] is one of the basic components of most distributed systems, and we start with its specification. Failure detector protocol defines a boolean event stream `failed` that emits `true` when it suspects that a specific isolate crashed. The failure detector can decide that its previous suspicion was incorrect, in which case it emits `false`.

```
trait FailureDetector extends Protocol {
  val failed: Events[Boolean]
}
```

The `HeartbeatFailureDetector` is a simple implementation of the `FailureDetector` protocol. The request channel `target` is the requirement of this protocol, so it is undefined. The protocol opens a connector `beats` to send periodic *heartbeat* events. It also opens the connector `failures` to signal failures. State of the failure detector is defined by two boolean flags `seen`, which is `true` if there was a heartbeat event during the last second, and `suspect`, which denotes what the detector previously reported.

```
trait HeartbeatFailureDetector
extends FailureDetector {
  val target: Req[Unit, Unit]
  val beats = open[Unit]
  val failures = open[Boolean]
  val failed = failures.events
  var (seen, suspect) = (false, false)
  sys.timer(1.second) on {
    target ! ((), beats.channel)
    if (seen) seen = false
    else if (!suspect) {
      failures.channel ! true
      suspect = true
    }
  }
  beats.events on {
    if (suspect) failures.channel ! false
    suspect = false
    seen = true
  }
}
```

The detector creates a `timer` event stream, that produces an event every second. On each of these events, the detector sends a heartbeat request to `target`. The heartbeat request specifies that the reply should come on the `beats` channel. If the target was seen since the last beat request, `seen` is reset to `false`. If not and the failure was not reported, a `true` event is sent to the `failures` channel and `suspect` is set to `true`. When `beats` delivers an event, the detector concludes that the target did not fail, so it resets the flags and emits a `false` event if necessary.

To instantiate this protocol, we provide a channel of type `Req[Unit, Unit]`. In the following, isolate `Monitored` instantiates the `heartbeat` channel and registers it with the name server. The isolate `Monitor` requests the heartbeat channel from the name server, and then uses it to instantiate the failure detector.

```
class Monitored extends Iso[Unit] {
  val heartbeat = server[Unit, Unit](u => u)
  nameServer ! ("/heartbeat", heartbeat)
}
class Monitor extends Iso[Unit] {
  (nameServer ? "/heartbeat") onCase {
    case heartbeat: Req[Unit, Unit] =>
      val fd = new HeartbeatFailureDetector {
        val target = heartbeat
      }
      fd.failed.onEvent(println)
  }
}
```

Note that the `FailureDetector` protocol reports the failure of a single isolate. We can use this protocol to define a bulk failure detector protocol, which uses a map of isolate IDs and corresponding failure detectors, and reports the ID of the failing isolate. The `BulkFailureDetector` creates a set of event streams of type `Events[(Id, Boolean)]`, and then calls `union` to produce an event stream that collects all of the events from that collection.

```
trait BulkFailureDetector {
  val detectors: Map[Id, FailureDetector]
  val failed: Events[(Id, Boolean)] =
    union(for ((id, d) <- detectors)
      yield d.failed.map(x => (id, x)))
}
```

*Best-effort broadcast* [13] is another distributed programming protocol. For a set of isolate IDs that participate in the protocol, best-effort broadcast exposes `channel`, used to send events to other participants, and `events`, which emits events sent from other participants. The `events` stream guarantees to deliver an event sent to `channel` to all the participants if the sender does not fail, hence the name *best-effort*.

```
trait BestEffortBroadcast[T]
extends Protocol {
  val targets: Set[Id]
  val events: Events[T]
  val channel: Channel[T]
}
```

`BestEffortBroadcast` takes a type parameter `T`, denoting the type of events it broadcasts. `BasicBroadcast[T]` implements best-effort broadcast – given a set of participant channels, and an `events` stream corresponding to one of the channels in that set, the protocol exposes a `channel` that forwards events to all participants:

```
trait BasicBroadcast[T]
extends BestEffortBroadcast[T] {
  val channels: Set[Channel[T]]
  val events: Events[T]
  val targets = channels.keys
  val (channel, sends) = open[T]
  sends onEvent { x =>
    for (ch <- channels) ch ! x
  }
}
```

Above, if the sender fails in the middle of the for-loop, only a subset of all the isolates receive the event. *Regular reliable broadcast* [13] is a stronger primitive, which guarantees that the same set of messages are delivered to all the participants, even if some of the machines fail.

```
trait RegularReliableBroadcast[T]
extends BestEffortBroadcast[T]
```

`LazyReliableBroadcast[T]` in Figure 2 is a concrete implementation of regular reliable broadcast. This protocol requires a bulk failure detector `fd` and a best-effort broadcast `beb` that delivers events of type `(Id, T)`. Lazy reliable broadcast uses two connectors `(channel, sends)` and `(receives, events)`, the set of correct isolates called `correct`, initially containing all isolates, and a map `from` of all the events received for each isolate. Sending is sim-

```
trait LazyReliableBroadcast[T]
extends RegularReliableBroadcast[T] {
  val fd: BulkFailureDetector
  val beb: BestEffortBroadcast[(Id, T)]
  val targets = beb.targets
  val (channel, sends) = open[T]
  val (receives, events) = open[T]
  val correct = Set(fd.targets.keys)
  val from = Map[Id, Set[T]]()
  for (i <- targets) from(i) = Set()

  sends.onEvent(x => beb.channel ! (id, x))

  beb.events.onCase { case (i, x) =>
    if (!(from(i) contains x)) {
      receives ! x
      from(i).add(x)
      if (!correct(i)) beb.channel ! (i, x)
    }
  }

  fd.failed onCase {
    case (i, true) if correct(i) =>
      correct.remove(i)
      for (x <- from(i)) beb.channel ! x
  }
}
```

**Figure 2.** Lazy Reliable Broadcast

ple – all the events from `channel` are tupled with the isolate ID and forwarded to the underlying best-effort broadcast protocol. If an isolate *I* fails in the middle of executing the best-effort protocol, then the failure detector ensures that the isolates that received the event also resend it. Hence, every event could be delivered to `beb` multiple times. When `beb` delivers an ID `i` and the event, the protocol checks `from` to see if the message was already delivered. If not, the event is delivered to `events`, added to `from`, and optionally resent if `i` is not in the `correct` set. When the failure detector reports a failure for a correct isolate `i`, the ID `i` is removed from the `correct` set, and all the events from `i` are resent.

A *replicated sequence* is a fault-tolerant distributed ordered dataset. Informally, a replicated sequence guarantees that the participants eventually observe the same dataset despite machine failures. While a sequence from sequential programming associates an integer index to each element, `ReplicaSeq[T]` protocol is more general – it does not have a concrete index representation. Instead, `ReplicaSeq[T]` protocol represents the indices with the abstract type `Tag`.

```
trait ReplicaSeq[T] extends Protocol {
  type Tag
  def apply(): Seq[(Tag, T)]
  def remove(t: Tag): Unit
  def insert(t: Tag, x: T): Unit
}
```

To obtain a sequence of elements, `ReplicaSeq[T]` defines the method `apply`, which returns an ordinary sequence

```
trait TreedocSeq[T] extends ReplicaSeq[T] {
  type Tag = (List[Id], Id)

  trait Op
  case class Non() extends Op
  case class Ins(t: Tag, x: T) extends Op
  case class Rem(t: Tag) extends Op

  class Node(path: List[Id]) {
    val values = SortedMap[Id, Op]()
    val children = SortedMap[Id, Node]()
    for (t <- targets) values ++= t -> Non()
  }

  val rb: RegularReliableBroadcast[Op]
  val targets = rb.targets + dummyMaxId

  def remove(t: Tag) {
    val (path, i) = t
    val node = find(root, path)
    node.values(i) = Rem(t)
    rb.channel ! node.values(i)
  }

  def insert(t: Tag, x: T) {
    val node = findLessThan(root, t)
    node.values(id) = Ins((node.path, id), x)
    rb.channel ! node.values(id)
  }

  def apply(): Seq[(Tag, T)] =
    filterIns(root)

  rb.events onCase {
    case Rem((path, i)) =>
      val n = ensure(path)
      n.children(i) = Rem((path, i))
    case op @ Ins((path, i), x) =>
      val n = ensure(path)
      n.children(i) match {
        case Non() => n.children(i) = op
        case _ => // do nothing
      }
  }
}
```

**Figure 3.** Treedoc CRDT

of pairs of tags and elements. Clients can traverse this sequence to inspect the elements and the respective tags. To remove an element, clients call `remove` with the previously observed tag. To insert an element before a specific tag, clients call `insert`.

We use a conflict-free replicated data type [31], or CRDT, to implement the replicated sequence protocol. CRDTs are distributed data structures that achieve *strong eventual consistency* – replicas of the data structure that have observed the same updates are in the same state [31]. This consistency is achieved by commutative update operations. After an isolate updates its copy of the data structure, it broadcasts the

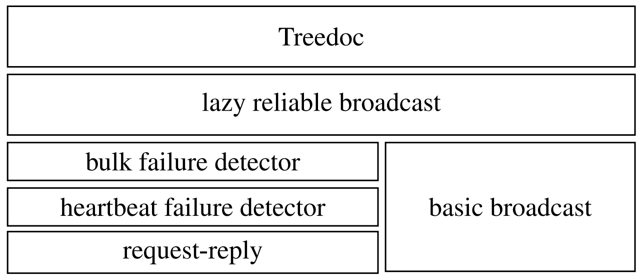

**Figure 4.** Protocol Stack for the Treedoc CRDT

update to other participants. Due to commutativity, the order in which participants receive the updates does not matter.

We implement a CRDT called Treedoc [26], shown in Figure 3. Treedoc maintains an $n$-ary tree, where $n$ is the number of participating isolates. Each `Node` contains a list of IDs called `path`. Every child of a node is associated with an ID of the participating isolates, so `path` uniquely describes a path to a node in the tree. Each `Node` also contains $n$ elements, and each element is uniquely identified with the path and its ID, which together comprise Treedoc's `Tag` type:

```
type Tag = (List[Id], Id)
```

An isolate can only insert an element at a tag that ends with its own ID by replacing the `Non` value with an `Ins` value. An isolate can remove any `Ins` from the tree by replacing it with a `Rem` value, which serves as a tombstone. Removing takes precedence over insertion. As shown before, operations defined this way are commutative [26].

Treedoc uses regular reliable broadcast to broadcast operations performed by each isolate. The `remove` operation uses `find` to locate the node with the specified `path`, and sets the value at the specified ID to `Rem`. It then broadcasts the update. Similarly, the `insert` operation uses `findLessThan` to find the node immediately preceding the specified tag, sets `values` at current isolate ID to `Ins`, and broadcasts the update. Finally, `apply` filters all values in the tree whose last operation is `Ins`. For brevity, we do not show implementations of `find`, `findLessThan` and `filterIns` methods.

Note that the efficiency of the protocol implementations in this section can be improved. The reliable broadcast and CRDT memory consumption is unbounded, the failure detector can overwhelm the target with messages, the reliable broadcast starts using $O(n^2)$ messages after the first spurious failure, and the Treedoc CRDT can become unbalanced. However, these issues can be addressed at the cost of conciseness. For example, we can use exponential backoff when the failure detector does not receive a reply, rely on vector clocks [12] along with false-positive events to remove old entries in the `from` map, or use consensus [13] [18] to rebalance Treedoc and garbage collect removed nodes.

Our goal is not to present optimal algorithms, but to demonstrate that these algorithms compose – the preceding protocol specifications form an abstraction stack, shown in Figure 4. Each of these protocol implementations can be reasoned about, developed, tested, debugged and replaced in isolation. A working system can be prototyped using suboptimal protocol components, and then made robust by improving these components independently.

Protocol composability is a consequence of having channels and event streams as first-class values. First-class primitives can be passed to and returned from functions, or specified as module members. Event streams and channels specify requirements at the layer boundaries in the protocol stack. Although we did not show an equivalent actor-based implementation of the example protocol stack, we note that actor references are the first-class equivalent of channels and event streams. As a consequence, components can be separated across different actors in the actor model. However, this approach has several drawbacks:

1. *Efficiency*: Passing messages is less efficient than method calls – every message that crosses the protocol boundary must be buffered, with a context switch between actors. Conversely, event propagation between protocols within the same isolate is done by method invocation [28].
2. *Type-safety*: Type information is lost on the protocol boundaries in the actor model, as actors can receive and send messages of any type. As a result, type-related errors only manifest at runtime, and protocol correctness is compromised.
3. *Program comprehension*: Concurrent dataflow is harder to debug and comprehend. A stack dump of an event propagation within an isolate directly coincides with the protocol stack. On the other hand, understanding the message origin requires retaining traces across actors. Similarly, program analysis tools and IDEs can easily track protocol dependencies, but are less effective at determining the actor referred to by a specific actor reference.

## 5. Future Work and Vision

In Section 4, we studied the protocol stack on an example of several distributed algorithms. We envision a more general stack of reusable components, which forms a *standard distributed computing library*. This standard library should be divided into the following layers:

1. *Foundational layer*: This layer constitues the core programming model, based on event streams for intra-isolate event propagation, and channels for inter-isolate communication, backed by facilities such as name resolution, schedulers, networking infrastructure and system isolates such as timers and I/O. Other abstractions are built in terms of these canonical primitives.
2. *Algorithm layer*: These are the basic distributed algorithms, such as failure detectors, broadcast and consensus primitives, resource allocation algorithms, vector clocks, leader election and gossip protocols. These abstractions

solve fundamental distributed computing problems, but are too low-level to describe application logic directly.

3. *Component layer*: This is the set of out-of-the-box components with standardized interfaces, which can be composed into applications. They include lease references, CRDTs, distributed caches, distributed hash tables, reactive data stores, and peer-to-peer primitives.
4. *Middleware layer*: Components in this layer can be directly used as applications, or main parts thereof. They include peer-to-peer networks, streaming and map-reduce engines, logging modules, distributed message queues, distributed file systems and databases.

Components in one layer are built mostly in terms of components in the same or preceding layer. Existing components can be reused or modified, as is the case with standard libraries for sequential programming.

## 6. Related Work

Actors were studied and formally described by Gul Agha [5], and have appeared in different forms in many languages and programming frameworks. Languages like D, E, Go, Scala, and Rust support both message-passing and shared-memory synchronization. The recent Dart language [1] completely separates different computations into isolates, which communicate exclusively by message passing.

Erlang [32] is a general purpose concurrent programming language and runtime for implementing distributed, fault-tolerant applications. The actor model in Erlang is based on the `receive` statement, which suspends execution until one of the specified message types arrive. The `receive` statement must encode the message types of all the protocols currently executing in the actor, limiting composition.

Akka [2] is an actor framework for the JVM with frontends in both Java and Scala. It differs from the Erlang actor model in that its `receive` statement cannot be called at arbitrary points in the actor. Instead, Akka handles the next message only after the call stack in `receive` unwinds, since it is unable to capture the program continuation like Erlang. Some actor frameworks on the JVM attempt to remedy this [14], but are unable to emulate the exact Erlang semantics. To switch the message-handling loop in the `receive` statement, Akka instead uses the `become` statement.

*Selectors* [16] are actors with multiple untyped mailboxes, which allow specifying which mailboxes can deliver the next message, and which must buffer it. Optional buffering allows synchronization patterns that rely on the message delivery order, such as join patterns and bounded buffers. Multiple mailboxes are akin to different channels in reactive isolates, but do not allow receiving specific message combinations directly. Selectors do not have first-class event streams, and their protocol composition is limited.

Rx [20] is a programming framework for event-based asynchronous programming, initially developed at Microsoft. Rx uses first class event streams to express asynchronous programs. These event streams are transformed using a large corpus of declarative combinator methods. Rx implementations exist in many different languages, including Java, Scala, JavaScript, Python, C++ and Objective C. One difference is that event streams in reactive isolates are by default *push-based*, and Rx event streams are *pull-based*. In the pull-based model, events are propagated through the dataflow graph once per each subscriber, whereas, in the push-based model, events are propagated exactly once. Concurrency in Rx is achieved with the `schedule` combinator, which transfers event propagation to another thread.

FlowPools [29] is a deterministic dataflow framework, which defines similar combinators used to transform event streams. Events are buffered in mailboxes associated with each event stream, and handled asynchronously, on separate computation threads. FlowPools are based on a lock-free queue data structure, similar to SnapQueue [27].

AmbientTalk [9] [25] is an actor-based, domain-specific language incorporating primitives such as *lease references*, and eventually consistent distributed collections. We envision its high-level abstractions as parts of the protocol stack, as they can be implemented with actors or reactive isolates.

High-level frameworks provide abstractions such as *resilient distributed data types* from Apache Spark [34], Google's MapReduce [10], or publish-subscribe systems like Apache Kafka [17]. These frameworks are complex systems built in terms of low-level RPCs, and do not expose a reusable stack of distributed programming components.

## 7. Conclusion

We introduced a foundational distributed programming model called *reactive isolates*. The model expresses concurrent computations with isolates, which use typed channels for inter-isolate communication, and event streams for intra-isolate event propagation. Our model retains serial event processing and location-transparency associated with the actor model. As opposed to traditional actor models, channels and event streams in reactive isolates are first-class objects. By implementing failure detectors, several broadcast primitives, and a conflict-free replicated data type, we showed that first-class channels and event streams improve separation between protocols, and enhance protocol reuse.

The proposed model enables building a protocol stack of reusable distributed computing components. These components specify requirements in terms of lower-level components, and expose their own capabilities. Separation into components improves composition in distributed programs, and helps bridge the gap between low-level and high-level frameworks for distributed computing.

## References

[1] Dart programming language, 2011. http://www.dartlang.org/.

[2] Akka documentation, 2015. http://akka.io/docs/.

[3] Erlang/OTP documentation, 2015. http://www.erlang.org/.

[4] Reactive Collections, 2015. https://reactive-collections.com.

[5] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[6] N. Amin, A. Moors, and M. Odersky. Dependend object types. In *19th Internation Workshop on Foundations of Object-Oriented Languages*, 2012.

[7] J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden, December 2003.

[8] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for scala type checking. In *Proceedings of the 31st International Conference on Mathematical Foundations of Computer Science*, MFCS'06, pages 1–23, Berlin, Heidelberg, 2006. Springer-Verlag.

[9] T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter. AmbientTalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, SCCC '07, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

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

[11] E. W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, Mar. 1968.

[12] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[13] R. Guerraoui and L. Rodrigues. *Introduction to reliable distributed programming*. Springer, 2006.

[14] P. Haller and M. Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, Springer LNCS, 2006.

[15] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic. Scala improvement proposal: Futures and promises (SIP-14). 2012.

[16] S. M. Imam and V. Sarkar. Selectors: Actors with multiple guarded mailboxes. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*, pages 1–14, 2014.

[17] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[18] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[19] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[20] E. Meijer. Your mouse is a database. *Commun. ACM*, 55(5):66–73, 2012.

[21] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer-Verlag, 1991.

[22] M. Odersky and al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[23] M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *FSTTCS 2009*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 427–451, Dagstuhl, Germany, 2009.

[24] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[25] K. Pinte, A. Lombide Carreton, E. Gonzalez Boix, and W. Meuter. Ambient Clouds: Reactive asynchronous collections for mobile ad hoc network applications. In J. Dowling and F. Taïani, editors, *Distributed Applications and Interoperable Systems*, volume 7891 of *Lecture Notes in Computer Science*, pages 85–98. Springer Berlin Heidelberg, 2013.

[26] N. M. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*, pages 395–403, 2009.

[27] A. Prokopec. Snapqueue: lock-free queue with constant time snapshots. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala 2015, Portland, OR, USA, June 15-17, 2015*, pages 1–12, 2015.

[28] A. Prokopec, P. Haller, and M. Odersky. Containers and aggregates, mutators and isolates for reactive programming. In *Proceedings of the Fifth Annual Scala Workshop*, SCALA '14, pages 51–61. ACM, 2014.

[29] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction. In *LCPC*, pages 158–173, 2012.

[30] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings of the First International Conference on Peer-to-Peer Computing*, P2P '01, pages 101–, Washington, DC, USA, 2001. IEEE Computer Society.

[31] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Jan. 2011.

[32] R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[33] J. White. High-level framework for network-based resource sharing. RFC 707, Dec. 1975.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.