Encoding the Building Blocks of Communication

Aleksandar Prokopec
Principal Researcher
Oracle Labs
Switzerland
aleksandar.prokopec@oracle.com

Abstract

Distributed systems are often built from the simplest building blocks such as message sends and RPCs. Since many communication patterns have to be reinvented every time a distributed system is created, implementing a high-level system is usually expensive. The recently proposed *reactor model* alleviates this cost by expressing distributed computations as reusable components, however, encodings for various communications patterns in this model are missing.

This paper investigates how to encode the router, client-server, scatter-gather, rendezvous, two-way communication, reliable communication and the backpressure protocol in the reactor model. These protocols are used to implement the core of a distributed streaming framework, and the performance of these implementations is evaluated.

CCS Concepts • **Computing methodologies → Distributed programming languages**; **Concurrent programming languages**;

Keywords reactor model, communication protocols, backpressure, streaming

ACM Reference Format:
Aleksandar Prokopec. 2017. Encoding the Building Blocks of Communication. In *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3133850.3133865

1 Introduction

A distributed computing model usually starts off with a few primitives, but gets extended as new use-cases arise. Here are some examples. MPI initially defined sends and receives (roughly); today it defines process topologies, parallel I/O, distributed state and memory management, and many other features [16]. Similarly, the actor model originally defined non-blocking sends [6], but it was augmented at least once with futures to support certain communication patterns [2] [22]. Futures were not built in terms of message-passing, and their implementation typically assumes shared memory. For performance, the RPC model is often extended with streaming RPCs, which are not built from regular RPCs, but are an independent extension.

While careful choice likely leads to picking useful extensions, we noticed two problems with this. First, distributed programming models start off simple, but over time become bulky, and difficult to implement or port. As an example, the MPI 3.0 specification is around 850 pages long [16]. Second, high-level distributed systems are in practice built from the simplest primitives, such as message-passing [3] or RPCs [23] [42]. Often, these systems share common patterns (for example, broadcast). Developers, unable to wait for the next release, implement their own application-specific variants of such patterns. At best, effort to discover the wheel is spent many times over. At worst, a rectangle is discovered instead.

In this paper, we investigate if the recently proposed *reactor model* [37] allows building communication protocols. The basic primitives in the reactor model are *channels*, used for sending, and *event streams*, used for receiving data. Rather than adding additional primitives, we examine how to derive new communication patterns from channels and event streams. This keeps the basic model simple, and allows users to partake in identifying the extensions. Since their inception, reactors raised a lot of questions. Does the model incorporate backpressure? Is delivery reliable? Is two-way or rendezvous-style communication possible? Can one do distributed dataflow and streaming? Does the model implement industry standards such as the Reactive Streams [4]?

The main idea in this paper is that, although one could say *yes* to each of these extensions, one should say *no* to all of them. A few basic primitives suffice to build families of communication protocols. The main benefits of having protocols in a library, and not in the programming model implementation, are *simplicitly and portability* – the communication protocols can be expressed as libraries only once, and reused on different platforms, which need to provide only the core communication primitives. Distributed systems need not be built directly from the basic communication primitives (as was often the case – Akka Streams [3], Apache Mesos [23], and Spark [42] were built mostly using message passing and

Onward!'17, October 25–27, 2017, Vancouver, Canada

RPCs, and without many generic middleware components). Rather, protocols can be composed from sufficiently high-level compound building blocks. To validate this, we show an exploratory implementation of a streaming framework that incorporates reliable delivery and backpressure.

This paper recapitulates the reactor model in Section 2, and compares it in Section 6. The paper then identifies a protocol stack in Section 3, and observes the following:

- A protocol is a set of functions and data types on channels and event streams. Every such definition is itself a first-class value, which can be further composed.
- The router protocol and the client-server protocol can be expressed directly in terms of channels and event streams (Sections 3.1 and 3.2).
- Scatter-gather, the rendezvous protocol and two-way communication use the router and the client-server protocol as building blocks (Sections 3.3, 3.4 and 3.5).
- Backpressure is a special form of two-way communication, and the pump and valve data types are used to interact with backpressure (Section 3.7).
- Streaming operations can be expressed as minimalistic kernels – we implement a polymorphic `lift` function that converts a synchronous stream transformation into an equivalent transformation on asynchronous streams with backpressure (Section 4). This allows building streaming frameworks such as Reactive Streams [4], Flink [10], or Spark Streaming [43].
- Abstraction costs of these protocols are relatively small and acceptable in practice (Section 5).

We use Scala [31] in our code examples, and we take care to use a minimal feature set. Local variables are declared with the `var` keyword, and final variables with `val`. Methods are declared with `def`. A lambda is a parameter list and a `=>` symbol, followed by the function body. Tuples are comma-separated expressions enclosed in parentheses `()`. The `case` keyword destructures arguments, such as tuples. Types are usually inferred, but can be explicitly placed after a colon. We sometimes do this for clarity. Polymorphic type parameters come between square brackets `[]`. Type aliases are declared with the `type` keyword. Examples should be easy to follow even if readers have no prior knowledge of Scala.

2 Reactor Model

This section rehashes the reactor model [5] [35] [37] in terms of an extended variant of the polymorphic lambda calculus [19] [32]. Several use-case examples are shown.

The key novelty in the reactor model is that each concurrency unit serially evolves multiple terms. These separate terms encode independent protocols, which helps compose distributed, asynchronous computations [37]. Section 6 highlights the differences between the reactor model and pi-calculus, CSP and the actor model.

`t ::=`	terms:
`x`	identifier
`def f[X](p:P):R = t`	abstraction
`f[T]`	type application
`t(t)`	application
`(t, ..., t)`	tuple
`t._n`	tuple selection
`Left(t)`	left sum
`Right(t)`	right sum
`t match { case Left(x) => t; case Right(x) => t }`	pattern match
`type N[X] = T`	type alias
`spawn[T](f)`	spawn
`t ! t`	send
`t.onReact(f)`	react
`open[T]`	open
`T ::=`	types:
`X`	type variable
`Boolean`	boolean type
`Int`	32-bit integer
`Long`	64-bit integer
`Unit`	unit type
`T => T`	function type
`(T, ..., T)`	product type
`Either[T, T]`	sum type
`N[T]`	type instantiation
`Channel[T]`	channel type
`Events[T]`	event stream type

Figure 1. Reactor model – syntax and types

The syntax of the reactor model is shown in Figure 1. Abstraction and type abstraction are expressed as a single term called *abstraction*, which binds an identifier to the function. Type application and application are standard. For convenience, we include tuples and sum types (`Either` values, which can be pattern matched with the `Left` or the `Right` case), and we add parametric type aliases.

The reactor-specific part consists of the terms that express concurrency aspects: *spawn*, *send*, *react* and *open*. The `spawn` function takes a type `T` and the function `f` that encodes the reactor body. Evaluation of `spawn` starts a concurrent computation (i.e. a reactor), and reduces to a channel value of type `Channel[T]`, used to communicate with the reactor.

Assume that we want to encode a reactor that behaves like a variable, shown in Listing 1. We first define a type alias `Op[T]` as a sum that is either a value of type `T` or a channel of type `Channel[T]`. We then define a function `v` that takes the initial value `x`, and spawns a reactor. The reactor definition calls another function named `cell`, defined shortly.

Listing 1. Spawn example

```
1 type Op[T] = Either[T, Channel[T]]
2 def v[T](x:T):Channel[Op[T]] = spawn[Op[T]] {
3   case (c,e) => cell(x,e) }
```

$$\frac{t = \texttt{spawn[T](f)}\,;\,t' \quad \texttt{f:(Channel[T], Events[T]) => Unit} \quad \texttt{(c, e)} = \texttt{open[T]}}{E \cup (t,i) \mid S \longrightarrow E \cup (\texttt{c}\,;\,t', i) \cup (\texttt{f((c, e))}, (\epsilon, \texttt{c,e} \to \varnothing)) \mid S} \text{ (Spawn)}$$

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

$$\frac{\texttt{v} \in \mathit{values}}{E \cup (\texttt{v}, i) \mid S \longrightarrow E \mid S \cup (\epsilon, i)} \text{ (Sleep)}$$

$$\frac{i = i' \cup (Q \cdot \texttt{x,c,e} \to F) \quad F = \{f_1, f_2, \ldots, f_n\} \quad t = f_1(\texttt{x})\,;\,f_2(\texttt{x})\,;\,\ldots\,;\,f_n(\texttt{x})}{E \mid S \cup (\epsilon, i) \longrightarrow E \cup (t, i' \cup (Q, \texttt{c,e} \to \varnothing)) \mid S} \text{ (Awake)}$$

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

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

$$\frac{t = \texttt{e.onReact(f)}\,;\,t' \quad \texttt{e:Events[T]} \quad \texttt{f:T => Unit}}{E \cup (t, i \cup (Q, \texttt{c,e} \to F)) \mid S \longrightarrow E \cup (t', i \cup (Q, \texttt{c,e} \to F \cup \texttt{f})) \mid S} \text{ (React)}$$

Figure 2. Concurrency-specific part of the operational semantics

Every reactor has one main channel and event stream. As shown in Listing 1, `spawn` passes the main channel `c` and event stream `e` to the reactor definition. The channel and the event stream represent the writing and the reading end, respectively. The `!` operator is used to send values, called *events*, along the channel. Once sent, events are eventually delivered to the corresponding event stream of type `Events[T]`. Only the owner of the event stream can listen to the next event by invoking `onReact` with a callback function. The *first* event that arrives is then passed to the callback.

Listing 2. Example of `onReact`

```
1 def cell[T](x:T,e:Events[Op[T]]) = e.onReact {
2   case Left(y) => cell(y,e)
3   case Right(c) => c ! x; cell(x,e) }
```

In Listing 2, the function `cell` invokes `onReact` on the event stream `e`. If the incoming event is `Left(y)`, then the new value `y` is used in a recursive call to `cell`. If the event is `Right(c)`, then the previous value `x` is sent along the channel `c`, and used in the recursive call to `cell`.

The reactor that *calls* `cell` effectively becomes a variable with loads and stores. Here, the `cell` function is a reusable protocol, and it can be instantiated in different reactors.

The `open` function creates additional channel and event stream pairs. In Listing 3, we use `open` in the `load` function to create a new channel `c`, then send that channel `c` to the variable reactor, and return the corresponding event stream. As seen in Listing 2, the variable must eventually send a value back along the channel `c`. The event stream returned from `load` reacts after the reply arrives, so invoking `load` corresponds to a variable read. The `store` function models assignment – it sends a new value and does not await a reply.

Listing 3. Load and store example

```
1 def load[T](v:Channel[Op[T]]):Events[T] =
2   { val (c,e) = open[T]; v ! Right(c); e }
3 def store[T](v:Channel[Op[T]],y:T):Unit =
4   v ! Left(y)
```

Listing 4 shows that the reactor model (not surprisingly [30]) allows encoding state, so in the rest of the paper we use a shorthand `var` to declare variables.

Listing 4. Variable usage example

```
1 val num = v(1)
2 load(num) onReact { x => store(num,x+1) }
```

Furthermore, in the previous examples and throughout this paper, we rely on n-ary function declarations, currying, value bindings, if-statements, while-statements, pattern matching, lambda expressions, methods, and the list data type. For example, sequencing (`;`) and value bindings (`val`) can be expressed as function applications, and lambda expressions (`=>`) as a combination of abstraction and type application. The exact encodings for these constructs were extensively studied [32], so we do not repeat them here.

Evaluation rules that capture the concurrency in the reactor model [37] are recapitulated in Figure 2. Program state is represented with two sets E and S, denoting currently executing and sleeping reactors, respectively. Each reactor is represented as a pair of the currently evaluated term (for reactors in the sleeping set S always empty, ϵ), and a set of tuples i, where each tuple contains an event queue Q, a channel c, and an event stream $e \to F$, where F is the set of callbacks on the event stream. The program terminates when the executing set E and the event queues are empty.

The evaluation rule Spawn adds a new reactor to the executing set, and replaces the `spawn` term with the new channel. The Open rule reduces the `open` expression to a channel and event stream tuple, and adds that tuple with an empty event queue to reactor's set i. The Sleep rule moves an executing reactor, whose term reduced to a value, to the sleeping set. The Awake rule moves a reactor with a non-empty event queue back to the executing set, and invokes the callbacks on the first event. The Send1 and Send2 rules deliver an event to a channel, and require that the event type does not contain `Events`, as event stream values cannot be shared. The React rule adds a callback to an event stream.

Since every reactor evaluates at most one term at any point, there are no data races on the mutable state of a reactor. This *serializability* property is retained from the actor model [6], and it improves program comprehension [37].

2.1 Combinators and Signals

Additional abstractions can be built from event streams [27] [36], and we borrow some of them in this paper. The `onEvent` function installs a callback to the event stream that reacts to every subsequent event, and not just the first one.

```
def onEvent[T](xs:Events[T],f:T=>Unit) =
  xs.onReact { x => f(x); onEvent(xs,f) }
```

A *signal* is an event stream whose last event is cached. We encode it with the `Signal[T]` type, which is a pair of an event stream and a function that returns the last event.

```
type Signal[T] = (Events[T], ()=>T)
def signal[T](xs:Events[T],z:T):Signal[T] = {
  var last:T = z
  xs.onEvent(x => last = x)
  (xs, ()=>last) }
```

We also use several event stream combinators. Given a function `f` of type `T => S`, the `map` function creates a new event stream such that when the original event stream emits an event of type `T`, the resulting event stream *synchronously* emits an event of type `S`, in the same reactor.

```
def map[T,S](xs:Events[T],f:T=>S):Events[S] = {
  val (c,e) = open[S]
  xs.onEvent(x => c ! f(x))
  e }
```

The `sync` combinator applies a function to a tuple after *both* input streams emit an event. Synchronizing pairs of event streams requires keeping two auxiliary queues.

```
def sync[T,S](xs:Events[T],ys:Events[T],
  f:(T,T)=>S):Events[S] = {
  val (c,e) = open[S]
  val (qxs,qys) = (new Queue[T],new Queue[T])
  def push(q:Queue[T],v:T) {
    q.enqueue(v)
    while (qxs.nonEmpty && qys.nonEmpty)
      c ! f(qxs.dequeue(), qys.dequeue()) }
  xs.onEvent(x => push(qxs,x))
  ys.onEvent(y => push(qys,y))
  e }
```

The second version of the sync combinator converts a list (`Seq`) of event streams into an event stream of lists. It does so by first mapping each event stream into an event stream of lists, and then reducing the event streams pairwise, where the reduction operator is the two-element `sync` from above, and `++` is list concatenation.

```
def sync[T](es:Seq[Events[T]]):Events[Seq[T]] =
  es.map(xs => xs.map(x=>Seq(x))
    .reduceLeft { (reduced,xs) =>
      (reduced sync xs)(_ ++ _) }
```

The `zip` function is the equivalent of `sync` that works on signals – the resulting event stream emits when either of the input signals emits. Here, `_2()` retrieves the cached value.

```
def zip[T,S](xs:Signal[T],ys:Signal[T],
  f:(T,T)=>S):Signal[S] = {
  val (c,e) = open[S]
  xs.onEvent(x => c ! f(x,ys._2()))
  ys.onEvent(y => c ! f(xs._2(),y))
  e }
```

We now have the machinery needed to present the generic communication protocols in the next section.

3 Generic Protocol Components

This section presents a stack of modular communication protocols. In the examples that follow, we name the channel and event stream pair a *connector*. We use a selector `events` to extract the event stream from a connector, and `channel` to extract the channel.

3.1 Router Protocol

One frequent pattern is forwarding events to one or more target destinations. We call this the *router protocol*[1] and encode it as a single function `router`, shown in Listing 5. The router is parametric in its routing policy `p` – the policy is a function that maps each incoming event to an output channel.

Listing 5. Router protocol

```
def router(p:T=>Channel[T]):Channel[T] = {
  val connector = open[T]
  connector.events onEvent { x => p(x) ! x }
  connector.channel }
```

In Listing 6, we show an implementation of a round robin routing policy, used when the router executes simple load-balancing. Such a router protocol is instantiated with the expression `router(roundRobin(chs))`, where `chs` is a list of destinations.

Listing 6. Round-robin router policy

```
def roundRobin(chs:List[Channel[T]]) = {
  var i = -1
  (x: T) => {
    i = (i + 1) % chs.length
    chs(i)
  } }
```

Similarly, the `broadcast` policy in Listing 7 forwards each message to multiple destination channels.

Listing 7. Broadcast router policy

```
def broadcast(chs:Set[Channel[T]]) = {
  val (c,e) = open[T]
  e.onEvent(x => for (c <- chs) c ! x)
  (x:T) => c }
```

[1] We note that the term router is not implied to be a real network router.

Users can provide their own routing policies. For example, when the workload distribution is biased, a random routing policy works better. If workload magnitude can be estimated on a per-request basis, then the *deficit round-robin* can be a more efficient load-balancing policy [38]. Load-balancing can also be made more efficient when destinations provide explicit feedback, while a hash-based policy is more appropriate for sharding and consistent replication.

3.2 Client-Server Protocol

In the client-server protocol, a client sends a request to the server, and the server uses the request to compute a response. To respond, the server needs a channel belonging to the client. Consequently, the client must send not only the request value, but also the channel that accepts the response value. It is helpful to encode these relationships with types, as shown in Listing 8. `Req[T,S]` is a tuple with the request value of type `T`, and the reply channel of type `Channel[S]`. `Server[T,S]` is a channel that accepts request tuples.

Listing 8. Data types in the client-server protocol

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

These types drive the implementation of the `server` protocol – the server first opens a new connector for the request type `Req[T,S]`. Next, the server calls `onEvent` to map each request value with the user-specified function `f`, and send it along the reply channel. Finally, the server returns its server channel of type `Server[T,S]`. This is shown in Listing 9.

Listing 9. Server protocol

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

The client must send a request tuple to the server, and then wait for the reply. This is done in the ? operator shown in Listing 10, which creates a connector of the reply type `S`, sends a request to the server, and returns the reply event stream to the caller.

Listing 10. Client protocol

```
def ?[T,S](s:Server[T,S],x:T):Events[S] = {
  val reply = open[S]
  server ! (x,reply.channel)
  reply.events }
```

In some cases, a server needs to send a batch of replies, or compute the reply asynchronously. Assume that these replies are specified by an event stream `f(x)`, computed from the request `x`. Given such a mapping `f`, the `stream` function in Listing 11 adds a callback to the event stream `f(x)`, which then forwards events back to the client.

Listing 11. Streaming server

```
def stream[T,S](f:T=>Events[S]):Server[T,S] = {
  val c = open[Req[T,S]]
  c.events onEvent {
    case (x,ch) => f(x).onEvent(ch ! x)
  }
  c.channel
}
```

Note that the streaming server from Listing 11 does not take care to prevent overflowing the consumer. We address this problem in Section 3.7.

3.3 Scatter-Gather Protocol

In some applications, information must be disseminated to many destinations, which then process this information and send a result back. This pattern is called *scatter-gather*, and it can be expressed by composing the router protocol with the client-server protocol.

Listing 12 shows the `scatterGather` function, which takes a router policy that maps a request to a server. This policy is used to define a router instance named `scatter`. A streaming server then uses the router to map each incoming list of requests of type `List[T]` to a list of reply event streams of type `List[Events[S]]`. Then, it uses the `sync` combinator to get an event stream of type `Events[List[S]]` – this event stream emits after all the servers reply. After all the results are gathered by `sync`, they are sent back to the client.

Listing 12. Scatter-gather protocol

```
def scatterGather[T,S](
  p: Req[T,S]=>Server[T,S]
): Server[List[T],List[S]] = {
  val scatter:Server[T,S] = router(p)
  stream(xs => sync(xs.map(x => scatter ? x)))
}
```

Given a `servers` list containing `Server[T,S]` channels, the expression `scatterGather(roundRobin(servers))` creates a channel that scatters incoming request batches across multiple servers, and then gathers the results. This pattern is useful for implementing map-reduce-style computations, multicasts and multi-destination queries.

3.4 Rendezvous Protocol

In the rendezvous protocol [29] [33], two processes synchronize and exchange a value. A rendezvous call must provide a value to exchange, and it suspends the process until a matching rendezvous call is made by another process, at which point the two processes continue.

The `rendezvous` method, which creates a rendezvous point, is shown in Listing 13. This method returns a pair of servers for types `T` and `S` of the values being exchanged. The method starts by creating two connectors and two queues for the types `T` and `S`. A helper method `flush` in line 5 checks if both queues are non-empty, and, if so, dequeues and sends one event from each. The method `meet` in line 8 is called

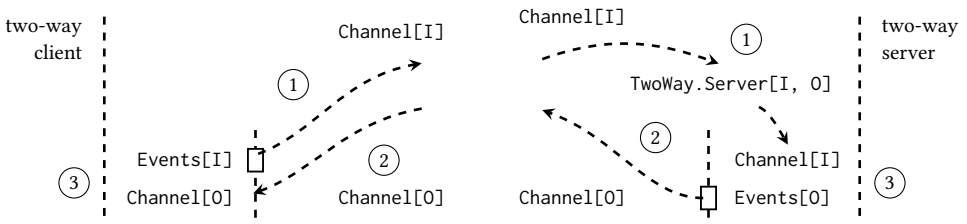

Figure 3. Establishing two-way communication

when either server receives an event, and it appends an element to the queue, calls `flush` and returns the event stream of the opposite type. A pair of streaming servers that use `meet` are created in line 10. When a process evaluates `s?x` on one of the rendezvous servers, the value `x` is placed on the respective queue and kept there until another process does the same. The call to `flush` by the second process then releases the enqueued values, and emits them on the event streams used by the streaming servers. The servers then respond to the pair of processes, resuming them.

Listing 13. Rendezvous protocol

```
1 def rendezvous[T,S]:(Server[T,S],Server[S,T]) =
2 {
3   val (ct,cs) = (open[T], open[S])
4   val (qt,qs) = (new Queue[T], new Queue[S])
5   def flush() = if (qt.nonEmpty && qs.nonEmpty)
6   { ct.channel ! qt.dequeue()
7     cs.channel ! qs.dequeue() }
8   def meet[X,Y](x:X,qx:Queue[X],e:Events[Y]) =
9     { qx.enqueue(x); flush(); e }
10  (stream(t => meet(t, qt, cs.events)),
11   stream(s => meet(s, qs, ct.events)))
12 }
```

The rendezvous protocol is typically invoked at one reactor, and the rendezvous servers are then shared; the ? operator starts the synchronization. Multiple rendezvous instances can be composed into other synchronization primitives, such as barriers or join patterns [17].

3.5 Two-Way Communication

In two-way communication, two parties simultaneously send and receive events, so each uses both a channel and an event stream. To establish such a two-way link, the two parties must first exchange their input channels. One of the parties, called a client, must initiate the link by sharing its input channel, and the other, a link server, completes it by responding.

The client, shown on the left in Figure 3, first creates an event stream of type `Events[I]`, and sends the corresponding channel to the server (1). The client then waits until the server responds (2) with a channel of type `Channel[O]`. Finally, the client uses the two-way link (3). The server, shown on the right, first accepts the incoming channel (1). It then creates a channel of type `Channel[O]`, sends it to the client (2), and starts using the link (3).

These relationships are expressed as types in Listing 14. The client-side two-way link is a tuple with an outgoing channel, incoming event stream and the subscription used to close the link, named `TwoWay[I,O]`, where `I` is the type of incoming events, and `O` is the type of outgoing events. The server-side link then has the type `TwoWay[O,I]`.

Listing 14. Data types in two-way communication

```
1 type TwoWay[I,O] = (Channel[O],Events[I])
2 type TwoWay.Req[I,O] =
3   Req[Channel[I],Channel[O]]
4 type TwoWay.Server[I,O] =
5   (Channel[TwoWay.Req[I,O]],
6    Events[TwoWay[O,I]])
```

The link request type, named `TwoWay.Req[I,O]`, is the client-server request type instantiated at the request type `Channel[I]` and the response type `Channel[O]`. The server state, typed `TwoWay.Server[I,O]`, is a tuple with the request channel, and the event stream that emits established two-way links.

The function `twoWayServer`, shown in Listing 15, creates a new link server. It opens a request connector in line 2, then uses its event stream to respond to incoming events and create a two-way link object in line 6. The server state is returned in line 8.

Listing 15. Two-way server protocol

```
1 def twoWayServer[I,O]():TwoWay.Server[I,O] = {
2   val c = open[TwoWay.Req[I,O]]
3   val links = c.events map { case (in,reply) =>
4     val output = open[O]
5     reply ! output.channel
6     (in,output.events):TwoWay[O,I]
7   }:Events[TwoWay[O,I]]
8   (c.channel,links):TwoWay.Server[I,O]
9 }
```

The `connect` function shown in Listing 16 starts the client protocol from Section 3.2 by sending the input channel. Once the output channel arrives, it is mapped to a two-way link.

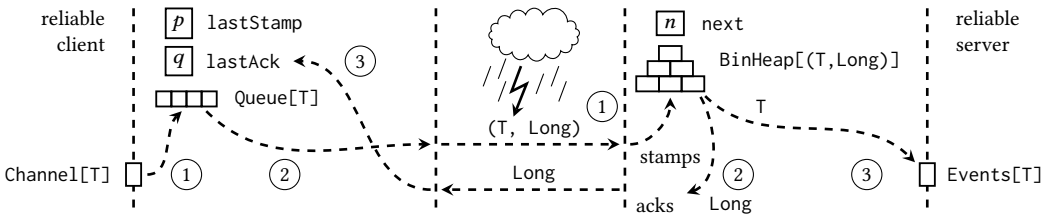

Figure 4. Reliable communication over a transport with arbitrary delays

Listing 16. Two-way client protocol

```
def connect[I,O](s:TwoWay.Server[I,O]) = {
  val in = open[I]
  (s ? in.channel) map { output =>
    (output,in.events):TwoWay[I, O]
  }:Events[TwoWay[I,O]]
}
```

The implementation of the two-way link is somewhat complex, but the usage is simple. For example, a chat server awaits established links, tracks all the clients, and broadcasts messages (recall the `broadcast` policy from Listing 7):

```
val (chat,links) = twoWayServer[String,String]
val clients = new Set[Channel[String]]
val everybody = router(broadcast(clients))
links.onEvent { case (out,in) =>
  clients += out // Add the client's channel.
  in.onEvent(msg => everybody ! msg)
}
```

The `chat` channel from above can now be shared. The chat client invokes `connect`, waits until the link is established, and then forwards standard input to the chat server:

```
connect(chat) onReact { case (out,in) =>
  in.onEvent(println)
  stdin.onEvent(line => out ! line)
}
```

3.6 Reliable Communication

Events traveling between reactors on the same machine are eventually delivered. For reactors on different machines, delivery depends on the underlying *transport implementation*. A transport based on the TCP protocol typically guards against reordering and packet loss. However, a TCP connection *can temporarily break*. When this happens, a user program needs to *manually recover from a broken TCP connection, and resend the lost events*. An application-level reliability protocol helps deliver events transparently, and keep the user unaware of the underlying TCP connection failures.

To keep this section short, we assume that the transport may arbitrarily delay and reorder events, but does not lose, duplicate or corrupt them. A reliable channel ensures that events sent from a client are delivered in order. It can be built from an unreliable two-way link, as shown in Figure 4.

Order is restored by assigning a timestamp to each event. The client requests a link with the `(T,Long)` output type. The input type `Long` is used for acknowledgements. The server maintains a priority queue with the timestamped events, and the next expected stamp. Every incoming event is stored into the priority queue (1). First event in the priority queue is compared against the expected stamp. While these match, an acknowledgement is sent back to the client (2), and the event is removed and delivered (3). The client sends events, but takes care to avoid flooding the server. For this purpose, it tracks the last sent stamp and the last acknowledgement. Each event is first stored into a queue (1). Events are then dequeued and sent (2) as long as the `lastStamp` is less than some *window* value ahead of `lastAck`. When an acknowledgement arrives, the `lastAck` field is incremented.

Listing 17. Data types in reliable communication

```
type Reliable.Req[T] =
  TwoWay.Req[Long,(T,Long)]
type Reliable.Server[T] =
  (Channel[Reliable.Req[T]],Events[Events[T]])
```

Basic data types are encoded in Listing 17. The request type `Reliable.Req[T]` is a special case of `TwoWay.Req`, and the server type `Reliable.Server[T]` is a pair of the request channel, and an event stream of established connections.

Function `reliableServer`, shown in Listing 18, starts a reliable server. It first creates a two-way server, and then maps its two-way links into reliable links. For each two-way link, a new channel `c` is opened, used to deliver events of type `T`. Channel `c`, and the two-way link with `acks` and `stamps`, are passed to the `setupServer` function in line 5.

Listing 18. Reliable server protocol

```
def reliableServer[T]: Reliable.Server[T] = {
  val (s,links) = twoWayServer[Long,(T,Long)]
  val rlinks = links.map { case (acks,stamps)=>
    val (c,e) = open[T]
    setupServer(stamps,acks,c)
    e:Events[T] }:Events[Events[T]]
  (s,rlinks): Reliable.Server[T]
}
```

The `openReliable` function in Listing 19 requests a reliable connection and establishes the writing end of the reliable

channel. When the client invokes `openReliable`, this function invokes `connect` from Section 3.5 to establish a two-way link, and then calls `setupClient` in line 6 to hook incoming acks and outgoing stamped events with user events. The reliable channel is then returned in line 7.

Listing 19. Reliable client protocol

```
1 def openReliable[T](
2   s: Channel[Reliable.Req[T]]
3 ): Events[Channel[T]] =
4   connect(s) map { case (stamps,acks) =>
5     val (c,e) = open[T]
6     setupClient(stamps,acks,e)
7     c:Channel[T]
8 }
```

The functions `setupServer` and `setupClient` must wire up the stamps, the acknowledgements and the user-level events so that the delivery becomes reliable. The policy in Figure 4, which prevents event reordering due to arbitrary delays, is implemented in the Listing 20. The `setupServer` function puts every stamped event on the binary heap in line 6, and then delivers events with corresponding timestamps. The `setupClient` function is not shown, but it has a similar structure.

Listing 20. Reordering reliability policy

```
1 def setupServer[T](stamps:Events[(T,Long)],
2   acks:Channel[Long],c:Channel[T]):Unit = {
3   var next = 1L
4   val q = new BinHeap[(T, Long)]
5   stamps onEvent { case (x, stamp) =>
6     q.enqueue((x, stamp))
7     while (q.nonEmpty && q.head.stamp==next) {
8       acks ! next; next += 1; c ! q.dequeue()
9     }
10  }
11 }
```

3.7 Backpressure Protocol, Valves and Pumps

Protocols shown so far did not incorporate flow control. To ensure that the sender (i.e. the client) does not overflow the receiver (i.e. the server), feedback about the available capacity must be sent in the opposite direction. The client, shown on the left in Figure 5, maintains a budget counter and can send only when this counter is positive (1). When an event is sent, the counter is decremented (2). The counter is incremented when additional budget arrives from the server (3). The server puts all the inbound events into a queue (1). The server must explicitly dequeue events, and only do so after the availability signal becomes `true` (2). When an event is dequeued, additional budget is sent back to the client (3).

The backpressure interface differs from earlier protocols, since the writing end is not always available to its user. The availability of the writing end is dictated by a signal, which acts like a *valve* in fluid flow control. The reading end decides when to deliver events and sends pressure back to the sender, much like a mechanical *pump*.

Listing 21. Data types in the backpressure protocol

```
1 type Valve[T] = (Channel[T],Signal[Boolean])
2 type Pump[T] =
3   (Signal[Boolean],()=>Unit,Events[T])
4 type Backpressure.Server[T] =
5   (Channel[TwoWay.Req[Int,T]],Events[Pump[T]])
```

The `Valve[T]` type in Listing 21 is the writing end of a backpressure channel, and is defined as a pair of an output channel and an availability signal. The availability of the valve is indicated with a `Signal[Boolean]` value. Events can only be sent while this signal is `true`. The `Pump[T]` type is used to deliver events, and is defined as a pair consisting of an availability signal, a dequeue function, and an event stream. If the availability signal is `true`, then invoking the dequeue function emits an event on the event stream. Dequeuing must be explicit, since this allows clients to compose pumps with valves, as explained later in Section 4.

The backpressure server, shown in Listing 22, creates a two-way server, and maps inbound two-way links to `Pump` values. A two-way link consists of a channel `back`, used to send budget to the client, and an event stream `in` with incoming events. For each link, a delivery connector `(c,e)` is created in line 4. Then, inbound events are mapped to the `grow` event stream, which enqueues the event, and produces 1, in line 6. Delivered events from `e` are mapped to the `shrink` event stream, which emits `-1`, in line 7. The `shrink` and `grow` event streams are joined with `union`, and their events are summed using the `scanPast` combinator (which is the stream equivalent of `scanLeft` on collections). The valve is available when this sum is larger than zero, so the sum gets mapped into the `available` signal in line 10. The `deq` function in line 11 removes an event from the queue, delivers it and sends a backpressure token to the writer. The `deq` function, the availability signal, and event stream `e` are used to create the pump in line 12.

Listing 22. Backpressure server protocol

```
1 def backpressureServer[T] = {
2   val (s,links) = twoWayServer[T,Int]()
3   val bplinks = links.map { case (back,in) =>
4     val (c,e) = open[T]
5     val q = new Queue[T]
6     val grow = in.map { x => q.enqueue(x); 1 }
7     val shrink = e.map(_ => -1)
8     val available = (grow union shrink)
9       .scanPast(0)(_ + _)
10      .map(_ > 0).signal(false)
11    val deq = ()=>{ c ! q.dequeue(); back ! 1 }
12    (available,deq,e):Pump[T] }
13  (s,bplinks):Backpressure.Server[T] }
```

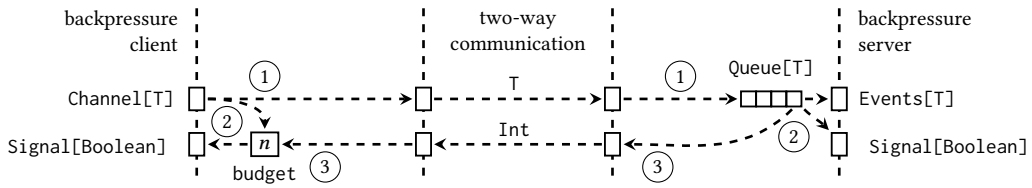

Figure 5. Communication with backpressure

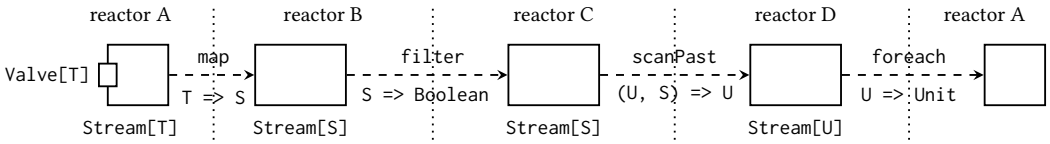

Figure 6. Dataflow in distributed streaming frameworks

The client-side `connectBackpressure` function, shown in Listing 23, invokes the function `connect` from Section 3.5 to establish a two-way link with the server. After the link is established, a new connector `(c,e)` is opened. Events from `e` are forwarded to the outgoing channel, and used to create a `shrink` event stream. The `grow` event stream from line 4 contains the budget sent by the server. The `shrink` and `grow` event streams are again used to define the `available` signal.

Listing 23. Backpressure client protocol

```
1 def connectBackpressure[T](
2   server:Channel[TwoWay.Req[Int, T]]
3 ):Events[Valve[T]] =
4   connect(server) map { case (out,grow) =>
5     val (c,e) = open[T]
6     val shrink = e.map { x => out ! x; -1 }
7     val available = (grow union shrink)
8       .scanPast(initialBudget)(_ + _)
9       .map(_ > 0).signal(true)
10    (c,available):Valve[T] }
```

The usage of the `Valve[T]` data type is different from channel types from the earlier sections, since its channel can only be used when the valve is available. In the following snippet, the client establishes a backpressure channel, then subscribes to the events when the `available` signal becomes `true`, and sends events while the valve is available.

```
1 connectBackpressure(server) onReact {
2   case (c,available) =>
3   available.becomes(true) onEvent { _ =>
4     while (available()) c ! produceNextEvent()
5   } }
```

4 Case Study: Distributed Streaming

In this section, we use the previous protocol components to implement a proof-of-concept streaming framework. Our aim is to show that the core functionality of a streaming framework can be composed so that any synchronous stream operation can be easily *lifted* into an asynchronous stream operation. The goal is not to implement all other aspects of a production streaming framework, such as fusion, persistence, remoting or fault-tolerance. Later, in Section 5, we show that the streaming implementation from this section is as efficient as industry-standard streaming frameworks.

In the distributed streaming context, data elements are processed in a pipeline that is split *across concurrent computations*, as in Figure 6. This is different from normal event stream operations from Section 2.1, such as `map`, `sync` or `scanPast`. Event stream transformations, which we used so far throughout this paper, are *synchronous* and confined to a single reactor, so they do not require reliable delivery or backpressure, whereas in distributed streaming the same operations are *asynchronous* and separated *across* reactors.

In Figure 6, the reactor A creates a source stream element, and then invokes the `map`, `filter` and `scanPast` operations, which creates three new reactors B, C and D, each processing the respective part of the pipeline. Calling `foreach` routes the events coming out of the pipeline back to the reactor A.

Most stream elements have upstream dependencies from which data flows. To establish a backpressure link from Section 3.7, a downstream element must send a backpressure request channel to an upstream element. Thus, a stream request type `Stream.Req[T]` must be a backpressure request channel. A stream, typed `Stream[T]`, is then a channel that accepts stream requests. There exists a special stream element called *source*, which does not have upstream dependencies, but its input is controlled with a `Valve` value. These relationships are expressed as types in Listing 24.

Listing 24. Streaming data types

```
type Stream.Req[T] =
  Channel[TwoWay.Req[Int,T]]
type Stream[T] = Channel[Stream.Req[T]]
type Source[T] = (Events[Valve[T]],Stream[T])
```

The source function, which creates source streams, is shown in Listing 25. It first creates an event stream e that will emit a valve once a downstream connects, and then creates a streaming server channel (s,links). Once a downstream request arrives on links, the connectBackpressure function gets called. After the connection is established, the resulting valve is emitted on the event stream e. The event stream e and the streaming server s comprise a Source value.

Listing 25. Stream source

```
def source[T]: Source[T] = {
  val (c,e) = open[Valve[T]]
  val (s,links) = open[Stream.Req[T]]
  links.onEvent { b =>
    connectBackpressure(b).onEvent(v => c!v) }
  (e,s) }
```

A *sink* is another special stream, which does not have downstream dependencies. Calling the foreach function shown in Listing 26 creates a sink. This function creates a backpressure server and sends it to the upstream self. Once the link is established, events are dequeued from the Pump[T] whenever they are available. Lines 6-11 show typical pump usage. First, a callback is added to the pump's event stream. Then, another callback is added to the pump's availability signal. Similar to a valve, while the pump is available, events are dequeued and emitted on the event stream, which passes them to the user function f.

Listing 26. Stream sink

```
1  def foreach[T](self:Stream[T],f:T=>Unit) {
2    val (server,links) = backpressureServer[T]
3    self ! server
4    links map {
5      case (available,deq,events):Pump[T] =>
6        events.onEvent(f)
7        available.becomes(true) onEvent { _ =>
8          while (available()) deq()
9      }
10   }
11 }
```

A streaming dataflow graph contains intermediate streams that behave as both sources and sinks. Here, the basic constraint is that events can be processed only when the upstream link is ready to deliver them and the downstream dependencies are available. This is the basis of the lift function in Listing 27. This function takes a parent stream upstream, of type Stream[T], and the transformation function f, of type Events[T] => Events[S]. The function f is a transformation on *synchronous* event streams.

Listing 27. Polymorphic lifting from synchronous to asynchronous event streams

```
1  type XSync[T,S] = Events[T]=>Events[S]
2  type XAsync[T,S] = Stream[T]=>Stream[S]
3  def lift[T,S](f:XSync[T,S]):XAsync[T,S] =
4    (upstream:Stream[T])=> spawn[Stream.Req[S]] {
5      (ch,downreqs) =>
6      val (s,uplinks) = backpressureServer[T]
7      upstream ! s
8      uplinks.sync(downreqs) { (p, down) =>
9        connectBackpressure(down) onEvent { v =>
10         val (pready,deq,in) = p
11         val (out,vready) = v
12         val ready = (pready zip vready)(_ && _)
13         f(in).onEvent(x => out ! x)
14         ready.becomes(true) onEvent { _ =>
15           while (ready()) deq()
16         }
17       } } }
```

The lift function spawns a reactor, and then creates a backpressure server in line 6. The backpressure server is sent to the stream's upstream parent in line 7. After *both* the link with upstream is established and the downstream request arrives in line 8, the stream connects with its downstream dependency with the connectBackpressure call in line 9. After the downstream connection is established, the stream defines a new ready signal, which is true only when both the upstream pump and the downstream valve are available.

The lift function converts any synchronous stream transformation into a distributed stream transformation that handles backpressure. For example, given a mapping f of type T=>S, and a map combinator on event streams from Section 2.1, the lift((e:Events[T]) => map(e,f)) expression creates an asynchronous map combinator.

When an equivalent synchronous event stream transformation does not exist, it is instead convenient to use another generic mapping called transform, shown in Listing 28. This function takes a kernel function that specifies how the event is forwarded to the output channel. Function kernel is invoked when there is an event ready for processing and the output channel is available.

Listing 28. Generic stream transformation function

```
def transform[T,S](
  up:Stream[T],kernel:(T,Channel[S])=>Unit
):Stream[S] = {
  lift(xs => {
    val (c,e) = open[S]
    xs.onEvent(x => kernel(x,c))
    e
  })(up)
}
```

Listing 29 shows kernels of several stream operations. The kernel of the map function applies the user-provided mapping function f, and forwards the event to the output

channel `out`. The `batch` function groups events into batches of a given size, the `filter` function applies a predicate to decide whether to forward the event, and `scanPast` updates the accumulation value of type `S` when an event of type `T` arrives.

Other stream operations such as `sync` and `union`, which have multiple upstream dependencies, are similarly implemented, but require variants of `lift` with different arities, which establish multiple upstream links before responding to downstream requests.

Listing 29. Stream operation kernels

```
def map[T,S](up:Stream[T],f:T=>S):Stream[S] =
  transform(up) { (x,out) => out ! f(x) }

def filter[T](
  up:Stream[T],p:T=>Boolean
):Stream[T] =
  transform(up) { (x,out) =>
    if (p(x)) out ! x
  }

def scanPast[T,S](
  up:Stream[T],z:S,op:(S,T)=>S
):Stream[S] = {
  var acc = z
  transform(up) { (x,out) =>
    acc = op(acc,x)
    out ! acc
  }
}

def batch[T](
  up:Stream[T],sz:Int
):Stream[Buffer[T]] = {
  var buff = new Buffer[T]
  transform(up) { (x,out) =>
    buff += x
    if (buff.size == window) {
      out ! buff
      buff = new Buffer[T]
    }
  }
}
```

5 Demonstration of Efficiency

We empirically estimate the overheads of our protocol encodings, and we compare our streaming framework encoding on typical workloads against the state-of-the-art industrial frameworks, such as Akka Streams [3] and Spark Streaming [43]. In all cases, measurements are done on an Intel i7-4900MQ 2.8 GHz quad-core CPU with hyperthreading. To gather accurate results, we use established benchmarking methodologies for the JVM [18], and rely on the ScalaMeter performance testing framework [34].

Abstraction overhead. We created several synthetic workloads that do not execute useful computation and consist entirely from communication. These workloads are deliberately artificial and do not reflect the relative overhead in real applications, but they help quantify the absolute costs, and identify various parts of the overhead.

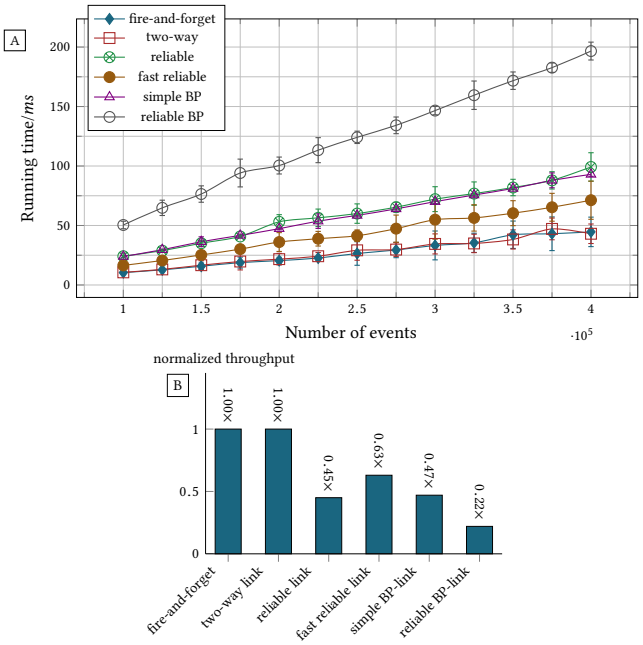

Figure 7. A - Running time comparison between different protocols (no network, lower is better); B - Normalized average throughput of different protocols (no network, higher is better)

In Figure 7, we use different protocols to deliver N events, ranging from $100k$ to $400k$, between two reactors. The receiver discards each event and proceeds to the next one. Both reactors are kept in the same process and no network is used – sending an event amounts to putting it on an event queue. The *fire-and-forget* curve stands for the ! operator, and serves as a baseline. Figure 7A shows that the variance in running time is small for all protocols, usually within 20% of the mean value, and the protocols scale linearly with the number of events sent N. Figure 7B shows the throughput of the protocols, normalized against the *fire-and-forget* baseline, and averaged across different values of N. We test the *two-way link* protocol by sending only in one direction, so its relative throughput is 1.0×. The *reliable link* protocol from Section 3.6 uses a sender-side buffer (to prevent receiver overflow) and creates a stamp object for each delivered event, which makes the relative throughput only 0.45×. We remove the sender-side buffer in the *fast reliable link* variant, but the stamp object allocation keeps throughput at 0.63× of fire-and-forget. The *simple backpressure link* is a variant that incorporates only flow control (it uses two-way links for delivery directly). Relative throughput is 0.47×, due

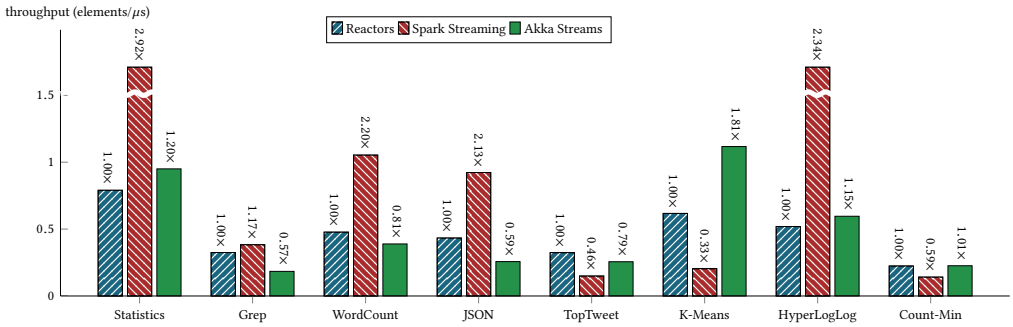

Figure 9. Comparison of streaming frameworks on typical applications, higher is better (note: Spark Streaming is a semi-batching framework, and it is optimized for throughput, but not latency)

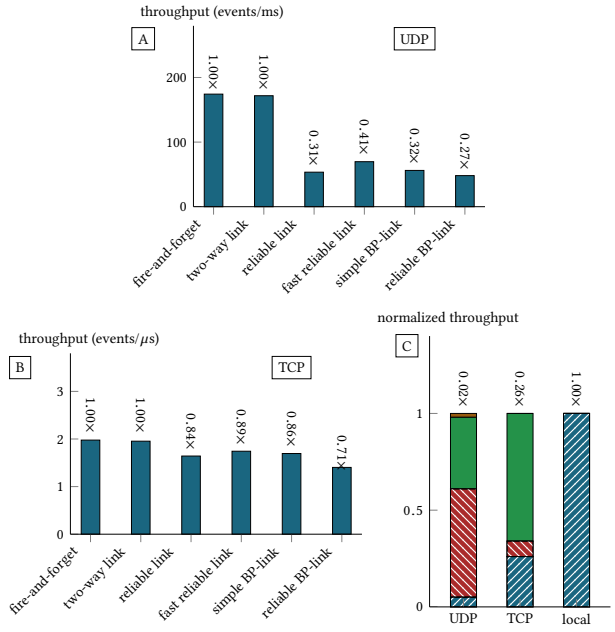

Figure 8. A, B - throughput comparison of UDP and TCP transport (higher is better); C - cost breakdown for UDP and TCP (higher is better; framework serialization socket GC)

to backpressure housekeeping, and allocations in `TwoWay` objects, which are in this case not type-specialized by the Scala compiler [13]. The *reliable backpressure link* uses reliability and must additionally allocate stamp objects, so this reduces its relative throughput to 0.22×.

Figure 8A shows that using UDP on the same workload results in 50× slower delivery, with the throughput penalty of 3−4×. This is because (a) UDP implementation in Reactors uses Java serialization for each event, (b) each UDP packet encodes the destination reactor and channel, and (c) packet-load is much lower than MTU for UDP. The breakdown of these costs, obtained with the YourKit JVM profiler, is shown in Figure 8C, where UDP spends 56% time in serialization, 37% using the network socket, and 2% in garbage collecting the structures allocated during serialization.

We implemented a proof-of-concept TCP-based transport, which serializes events directly to the network socket, and sends them in batches to reduce costs. This is still around 4× slower compared to no network, and the throughput penalty is 10 − 30%, as shown in Figure 8B. In both cases, throughput mostly depends on the data bandwidth, and the cost of writing to the socket obscures other abstraction penalties.

Applications. We compare the streaming implementation from Section 4 against Akka Streams [3] and Spark Streaming [43]. Akka Streams has similar stream transformation combinators as those shown in Section 4, and implements stream elements as separate actors. Akka Streams is a direct counterpart of the implementation in Section 4, the difference being that flow control is encoded directly with message sends. In Spark Streaming, events are batched into RDD collections [41], which are delivered at regular time intervals [43]. The difference between Akka Streams and Spark Streaming is a tradeoff between latency and throughput. While batching events improves throughput, this discretization increases latency, since events produced at the start of a batch are delayed until the end of the respective time interval. Akka Streams and Spark Streaming have two different streaming approaches – the first is optimized for lower latency, while the latter is optimized for throughput.

We chose eight typical streaming-based algorithms to compare the performance of these frameworks, and we report relative throughput in Figure 9. *Statistics* incrementally computes the average and standard deviation from a stream of numbers, and is communication-intensive. The *Grep* benchmark does string regex matching, and has a high allocation pressure. *WordCount* uses a `scan` operation to maintain a word count histogram. *JSON* uses `map` to parse input strings. *TopTweet* maintains a sliding window over the last 32 tweets, and selects the tweet with the most retweets. The streaming *K-Means* clustering algorithm [7] incrementally computes clusters from a stream of 2D points, and is work-intensive.

HyperLogLog cardinality estimation [15] has similar characteristics as *Statistics*, and *Count-Min* Sketch frequency estimation [11] computes 32 hash functions and is work-intensive.

While garbage collection reduces Spark's throughput in *TopTweet*, *K-Means* and *Count-Min*, the batching approach usually results in much higher throughput. Akka Streams is between 2× slower and 2× faster compared to the streaming framework from Section 4, depending on the benchmark. Throughput is in these workloads around 10 – 40× lower compared to the synthetic workload from Figure 7, indicating that useful work dominates abstraction penalties. While we did not replicate Spark's streaming strategy in Reactors, we note that batching only decreases the ratio of communication and useful work further. We conclude that abstraction overheads should not noticeable in user applications.

6 Related Work

A language usually has two tasks. First, it allows expressing patterns. Second, it must be possible to modularize, and then compose those patterns. In the context of distributed computing, numerous communication patterns and algorithms were identified in the past [20] [26], but modest effort was invested into standard libraries for distributed computing. For example, Erlang OTP framework [1] exposes middleware such as the client-server pattern, finite state machines, and actor supervision, but does not contain algorithmic components such as failure detectors, consensus, CRDTs, backpressure or replication, peer-to-peer or gossip. Support for building distributed systems is today either high-level and targeted (a framework such as Spark [42]), or very low-level (RPCs and message sends). A possible cause for this is lack of composability in existing distributed programming models. This prompted us to examine the reactor model more closely.

The reactor model draws inspiration from the traditional actor model [6]. In the actor model, concurrent computations are separated into entities called *actors*, which communicate by asynchronously sending messages between themselves. To receive a message, an actor invokes the `receive` statement, which blocks its execution until a message arrives.

There exist multiple variants of actors. In Erlang [40], actors are called processes, and `receive` must specify the desired message type. Computation suspends until such a message arrives, and is then resumed from the same point. Messages of other types are buffered until a matching `receive` is invoked. Runtimes without efficient continuation support either selectively dedicate threads to actors, as was the case with original Scala actors [21], or only allow a top-level event loop, as is the case with frameworks like Akka [2].

What is common among these actor models is that at most a single term inside an actor can await a message at any point in time. Consequently, protocols that await multiple messages or combinations thereof can be difficult to express or modularize. One potential solution for protocol composition is to implement independent protocol components as separate actors. As analyzed in related work [37], this approach requires separating state across actors, which introduces additional raciness and complexity into the program.

The reactor model [35] [36] [37] relies on two different entities for communication – channels and event streams. To read from an event stream, its owner provides a callback. After an event gets sent along the channel, the callback is eventually invoked. Multiple event streams can be listened to simultaneously, but at most one callback is executed at any point in time. This is the key feature of reactors, which enables protocol composition.

The reactor model inherits several important advantages of the actor model. The first is *serializability* – events received by a reactor are processed serially, one after the other, regardless of the channel they are delivered on. The second useful property is *location transparency* – it does not matter where a computation is located for program correctness.

Pi-calculus [28] is a foundational calculus for distributed systems, consisting of send, receive, spawn and channel creation primitives. Unlike the actor model, a channel value must be specified when sending and receiving. Like the actor model, the receive operation is blocking. At first glance, pi-calculus seems almost identical to the reactor model, but there are three crucial differences. First, while multiple channels can be awaited simultaneously, this has to happen on different processes, hence breaking serializability. Second, channels, which pi-calculus uses for both reading and writing, can be arbitrarily shared, whereas event streams cannot be shared among reactors. Arbitrary reads have an adverse effect on location transparency – tracking information about the current readers and writers in an asynchronous fail-stop model is tremendously difficult [14]. Third, channels are typed in the reactor model, which improves program comprehension, and allows certain static optimizations.

The CSP model [24], which predates pi-calculus, models imperative processes that communicate with messages. Its most important difference with pi-calculus is the nondeterministic choice operator. This operator can be thought of as the `select` system call in Unix – given a set of channels, receive on the channel that first delivers a message. Despite this extension, CSP has composability limitations that are similar to those of pi-calculus.

As an example, the Go language adopts a part of the CSP concurrency model. Receiving on a channel suspends the respective *goroutine*. While suspended, that goroutine cannot receive on other channels. Go provides a `select` call that allows waiting on multiple channels, but using `select` requires that different protocol components agree on the set of channels to wait on, and be mutually aware as a result. This breaks encapsulation – such a `select` call can be encapsulated inside a function, but cannot be further composed.

Many other actor-based languages were proposed [9] [21] [25] [39]. AmbientTalk [12], a distributed programming language that focuses on peer-to-peer and mobile applications, takes a step forward from traditional actor languages in that it exposes features such as *far references* and *lease references* as basic primitives. These AmbientTalk-specific concepts could be expressed as protocols in the reactor model, but this was not yet investigated. The Kompics component model [8] is an actor-like framework that has an ideology similar to the reactor model with respect to decoupling and modularizing components. In Kompics, components are dedicated entities with explicitly declared ports that must be connected with channels. Kompics has more basic primitives and a different composition model compared to reactors, but encodings of the protocols in this paper can likely be ported to Kompics.

Event streams used in the reactor model are similar to `Observable` values from Reactive Extensions [27]. The type `Observable` also provides a large set of transformation operations. While an event stream is confined to a single reactor, the `Observable` objects can transmit events across threads, so their implementations internally need synchronization.

Event streams in the reactor model can be seen as a minimalistic form of the publish-subscribe pattern. Many of the related programming models rely on the this paradigm. A good general overview of publish-subscribe techniques is given by Eugster et al. [14].

7 Conclusion

We showed how to implement several communication protocols in the reactor model. Each protocol was encoded as a set of functions and corresponding data types. Together, these protocols form an abstraction stack, in which complex components use simpler ones as building blocks – two-way communication relies on the client-server protocol, backpressure uses two-way links, and streaming requires backpressure. We showed that overheads are mostly acceptable, and do not affect typical application performance.

Libraries with generic communication protocols are likely to become standard. Rich ecosystems of libraries, which we today see for sequential programs, may become available for distributed programming as well. One aspect that we did not explore deeply in this work is fault tolerance – focusing on identifying and expressing composable communication protocols in the reactor model that guarantee fault-tolerance by construction is essential for distributed systems, and we plan to address this in future work.

References

[1] 2015. Erlang/OTP Documentation. (2015). http://www.erlang.org/

[2] 2016. Akka Documentation. (2016). http://akka.io/docs/

[3] 2016. Akka Streams Documentation. (2016). http://doc.akka.io/docs/akka/2.4.3/scala/stream/index.html

[4] 2016. Reactive Streams. (2016). http://www.reactive-streams.org/

[5] 2016. Reactors.IO website. (2016). http://reactors.io/

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

[7] Nir Ailon, Ragesh Jaiswal, and Claire Monteleoni. 2009. Streaming k-means approximation. In *NIPS*.

[8] Cosmin Arad, Jim Dowling, and Seif Haridi. 2012. Message-passing Concurrency for Scalable, Stateful, Reconfigurable Middleware. In *Proceedings of the 13th International Middleware Conference (Middleware '12)*. Springer-Verlag New York, Inc., New York, NY, USA, 208–228.

[9] J.-P. Briot. 1988. From Objects to Actors: Study of a Limited Symbiosis in Smalltalk-80. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-based Concurrent Programming (OOPSLA/ECOOP '88)*. ACM, New York, NY, USA, 69–72. DOI: http://dx.doi.org/10.1145/67386.67403

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[11] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-min Sketch and Its Applications. *J. Algorithms* 55, 1 (April 2005), 58–75. DOI: http://dx.doi.org/10.1016/j.jalgor.2003.12.001

[12] 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 and Structures, SCI Impact factor in 2013: 0.296, 5 year impact factor 0.329 (to appear)* (2014).

[13] Iulian Dragos and Martin Odersky. 2009. Compiling generics through user-directed type specialization. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. ACM, New York, NY, USA, 42–47. DOI: http://dx.doi.org/10.1145/1565824.1565830

[14] 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. DOI: http://dx.doi.org/10.1145/857076.857078

[15] Philippe Flajolet, ÃĽric Fusy, Olivier Gandouet, and et al. 2007. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *IN AOFA âĂŹ07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS*.

[16] Message Passing Interface Forum. 2012. MPI: A Message-Passing Interface Standard Version 3.0. (09 2012). Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[17] Cédric Fournet and Georges Gonthier. 2002. *The Join Calculus: A Language for Distributed Mobile Programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 268–332. DOI: http://dx.doi.org/10.1007/3-540-45699-6_6

[18] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.* 42, 10 (Oct. 2007), 57–76. DOI: http://dx.doi.org/10.1145/1297105.1297033

[19] J.Y. Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* https://books.google.hr/books?id=IRcVHAAACAAJ

[20] Rachid Guerraoui and Luís Rodrigues. 2006. *Introduction to reliable distributed programming.* Springer.

[21] Philipp Haller and Martin Odersky. 2006. Event-Based Programming without Inversion of Control. In *Proc. Joint Modular Languages Conference (Springer LNCS)*.

[22] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. 2012. Scala Improvement Proposal: Futures and Promises (SIP-14). http://docs.scala-lang.org/sips/pending/futures-promises.html

[23] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011.

Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 295–308. http://dl.acm.org/citation.cfm?id=1972457.1972488

[24] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. DOI: http://dx.doi.org/10.1145/359576.359585

[25] Shams M. Imam and Vivek Sarkar. 2014. Selectors: Actors with Multiple Guarded Mailboxes. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents and Decentralized Control (AGERE! '14)*. ACM, New York, NY, USA, 1–14. DOI: http://dx.doi.org/10.1145/2687357.2687360

[26] Nancy A. Lynch. 1996. *Distributed Algorithms*. MK Publishers Inc., San Francisco, CA, USA.

[27] Erik Meijer. 2012. Your Mouse is a Database. *Commun. ACM* 55, 5 (May 2012), 66–73. DOI: http://dx.doi.org/10.1145/2160718.2160735

[28] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Inf. Comput.* 100, 1 (Sept. 1992), 1–40. DOI: http://dx.doi.org/10.1016/0890-5401(92)90008-4

[29] Thomas D. Newton. 1987. An implementation of Ada tasking. (1987).

[30] Martin Odersky. 2002. *An Introduction to Functional Nets*. Springer Berlin Heidelberg, Berlin, Heidelberg, 333–377. DOI: http://dx.doi.org/10.1007/3-540-45699-6_7

[31] Martin Odersky and al. 2004. *An Overview of the Scala Programming Language*. Technical Report IC/2004/64. EPFL Lausanne, Switzerland.

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

[33] Rob Pike, Dave Presotto, Ken Thompson, and Gerard Holzmann. 1991. Process Sleep and Wakeup on a Shared-memory Multiprocessor. (1991).

[34] Aleksandar Prokopec. 2014. ScalaMeter Website. (2014). http://scalameter.github.io

[35] Aleksandar Prokopec. 2016. Pluggable Scheduling for the Reactor Programming Model. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016)*. ACM, New York, NY, USA, 41–50. DOI: http://dx.doi.org/10.1145/3001886.3001891

[36] Aleksandar Prokopec, Philipp Haller, and Martin Odersky. 2014. Containers and Aggregates, Mutators and Isolates for Reactive Programming. In *Proceedings of the Fifth Annual Scala Workshop (SCALA '14)*. ACM, 51–61. DOI: http://dx.doi.org/10.1145/2637647.2637656

[37] Aleksandar Prokopec and Martin Odersky. 2015. Isolates, Channels, and Event Streams for Composable Distributed Programming. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 171–182.

[38] M. Shreedhar and George Varghese. 1995. Efficient Fair Queueing Using Deficit Round Robin. *SIGCOMM Comput. Commun. Rev.* 25, 4 (Oct. 1995), 231–242. DOI: http://dx.doi.org/10.1145/217391.217453

[39] Sriram Srinivasan and Alan Mycroft. 2008. *Kilim: Isolation-Typed Actors for Java*. Springer Berlin Heidelberg, Berlin, Heidelberg, 104–128. DOI: http://dx.doi.org/10.1007/978-3-540-70592-5_6

[40] Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in ERLANG (2nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

[41] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[42] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

[43] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 423–438. DOI: http://dx.doi.org/10.1145/2517349.2522737