Distributed System Development with ScalaLoci

PASCAL WEISENBURGER, Technische Universität Darmstadt, Germany
MIRKO KÖHLER, Technische Universität Darmstadt, Germany
GUIDO SALVANESCHI, Technische Universität Darmstadt, Germany

Distributed applications are traditionally developed as separate modules, often in different languages, which react to events, like user input, and in turn produce new events for the other modules. Separation into components requires time-consuming integration. Manual implementation of communication forces programmers to deal with low-level details. The combination of the two results in obscure distributed data flows scattered among multiple modules, hindering reasoning about the system as a whole.

The ScalaLoci distributed programming language addresses these issues with a coherent model based on placement types that enables reasoning about distributed data flows, supporting multiple software architectures via dedicated language features and abstracting over low-level communication details and data conversions. As we show, ScalaLoci simplifies developing distributed systems, reduces error-prone communication code and favors early detection of bugs.

CCS Concepts: • **Software and its engineering → Distributed programming languages**; Domain specific languages; • **Theory of computation →** *Distributed computing models*;

Additional Key Words and Phrases: Distributed Programming, Multitier Programming, Reactive Programming, Placement Types, Scala

ACM Reference Format:
Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 129 (November 2018), 30 pages. https://doi.org/10.1145/3276499

1 INTRODUCTION

Despite a long history of research, developing distributed applications remains challenging. Among the sources of complexity, we find that distributed applications require transferring both data and control among different hosts [Thekkath et al. 1994] and are often event-based [Carzaniga et al. 2001; Meier and Cahill 2005]. These two aspects complicate reasoning about the distributed system because its run time behavior depends on the interaction among separate modules via events, whose occurrences can be unpredictable and potentially interleaving [Edwards 2009; Fischer et al. 2007]. First, separate development of each module limits programmers to only a local view, which hinders understanding the interactions in the entire system. Second, keeping track of potential events, control flows and data flows among components may become cumbersome.

Authors' addresses: Pascal Weisenburger, Technische Universität Darmstadt, Hochschulstraße 10, Darmstadt, Hessen, 64289, Germany, weisenburger@cs.tu-darmstadt.de; Mirko Köhler, Technische Universität Darmstadt, Hochschulstraße 10, Darmstadt, Hessen, 64289, Germany, koehler@cs.tu-darmstadt.de; Guido Salvaneschi, Technische Universität Darmstadt, Hochschulstraße 10, Darmstadt, Hessen, 64289, Germany, salvaneschi@cs.tu-darmstadt.de.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/11-ART129
https://doi.org/10.1145/3276499

These issues have been addressed by *multitier* (sometimes called *tierless*) *programming*, which aims to lower the effort of developing distributed applications, and *reactive programming*, a technique which allows direct data flow specification.

Multitier languages aim to bring the development of distributed systems closer to programming single-host applications providing means to abstract over distribution and remote communication among different components. With multitier languages, programmers use a *single language* and mix functionalities that belong to different *tiers*, e.g., the client and the server, *inside the same compilation unit*. The compiler automatically splits such compilation unit into the modules to deploy on each tier and generates the necessary communication code [Cooper et al. 2007; Neubauer and Thiemann 2005; Serrano et al. 2006]. As a result, developers do not need to handle details like data conversion and network communication and can focus on the actual application logic.

Reactive programming (RP) is an approach to develop reactive and event-based applications which leads to code that is more composable, more compact [Salvaneschi and Mezini 2014] and easier to understand [Meyerovich et al. 2009; Salvaneschi et al. 2014a]. Earlier RP solutions focused on purely functional languages [Elliott and Hudak 1997] investigating sound [Hudak et al. 2003] and efficient [Elliott 2009] implementations. More recent research developed RP for mainstream languages, seeing significant industrial adoption. Examples include Reactive Extensions / Rx [Meijer 2010] (available for a number of languages), FrTime [Cooper and Krishnamurthi 2006] (Scheme), Flapjax [Meyerovich et al. 2009] (JavaScript), Scala.React [Maier et al. 2010] and REScala [Salvaneschi et al. 2014b] (Scala).

The techniques above currently do not achieve their full potential for supporting the development of distributed systems. Existing multitier languages target the client–server model (mostly in web applications), lacking support for *generic* distributed architectures. These languages [Chlipala 2015; Cooper et al. 2007; Philips et al. 2014; Radanne et al. 2016; Rajchenbach-Teller and Sinot 2010; Reynders et al. 2014; Serrano et al. 2006] focus on issues like the impedance mismatch between server-side code and JavaScript, integrating database queries and manipulating the DOM. Also, the web setting allows certain assumptions, like client-driven interaction or stateless REST servers, which do not hold for generic distributed systems.

In addition, RP and multitier programming are not properly integrated. RP, which focuses on value propagation through data flows, is a natural fit for distributed applications since they are in many cases reactive [Eugster et al. 2003; Pietzuch and Bacon 2002]. Events, e.g., network messages or user input, transfer data among hosts and trigger state changes or new events. Yet, existing RP abstractions do not cross the boundaries of components and existing multitier languages, e.g., Hop [Serrano et al. 2006] or Links [Cooper et al. 2007], do not support RP abstractions. Multitier languages that provide RP features, e.g., Scala Multi-Tier FRP [Reynders et al. 2014], are limited to the web domain. Also, some multitier languages that feature RP abstractions, e.g., Ur/Web [Chlipala 2015], confine reactive values to a single component and do not allow defining data flows over multiple hosts.

We propose ScalaLoci[1], a multitier reactive language with a novel combination of abstractions for distributed applications. First, we propose **placement types** to associate locations *to data* – rather than *to computations only*, like existing approaches. Our solution allows going beyond the web domain and enables static reasoning about placement. Second, we support **multitier reactives** – placed abstractions for RP – which let developers compose data flows spanning over multiple distributed components. Thanks to this combination of features, we believe that ScalaLoci provides a significant advance in tackling the complexity of distributed system development.

[1] http://scala-loci.github.io

In summary, this paper makes the following contributions:

- We present ScalaLoci's design to support distributed applications, including in-language specification of distributed architectures, placement types and reactive abstractions for data flows over multiple hosts.
- We develop a core calculus for ScalaLoci with a type system to check that the interaction of remote values and local values is sound and that the application does not violate the architectural specification. We mechanize the proofs in Coq.
- We provide an implementation based on Scala that compiles multitier code to distributed components and retains compatibility with existing IDEs.
- We evaluate our approach with case studies – including Apache Flink, Apache Gearpump and 22 variants of smaller case studies – and show that ScalaLoci applications exhibit better design and are safer. Microbenchmarks and system benchmarks on an Amazon EC2 distributed deployment show that these advantages come at negligible performance cost.

The paper is structured as follows: Section 2 presents the abstractions of ScalaLoci. Section 3 demonstrates ScalaLoci through examples. Section 4 presents ScalaLoci's fault tolerance mechanism. Section 5 describes the execution model. Section 6 presents a formalization. Section 7 outlines the implementation. Section 8 discusses the evaluation. Section 9 presents related work. Section 10 concludes.

2 PROGRAMMING ABSTRACTIONS OF SCALALOCI

In this section, we introduce the abstractions supported by ScalaLoci. The next section demonstrates how their combination simplifies developing distributed systems.

2.1 In-Language Architecture Definitions

In ScalaLoci, the architectural scheme of a distributed system is expressed using *peers* and *ties*. Peers represent the different kinds of components of the system. Ties specify the kind of relation among peers. Remote access is only possible between tied peers. For instance, the client–server architecture is defined by a *server* peer and a *client* peer:[2]

```
trait Client extends Peer { type Tie <: Single[Server] }
trait Server extends Peer { type Tie <: Single[Client] }
```

```
peer Client ties single[Server]
peer Server ties single[Client]
```

Both peers have a *single tie* to each other, i.e., clients are always connected to a single server instance and each corresponding server instance always handles a single client. A variant of the client–server model, where a single server instance handles multiple clients, is modeled by a *single tie* from client to server and a *multiple tie* from server to client:

```
trait Client extends Peer { type Tie <: Single[Server] }
trait Server extends Peer { type Tie <: Multiple[Client] }
```

```
peer Client ties single[Server]
peer Server ties multiple[Client]
```

We introduce ScalaLoci's abstractions with a P2P chat example, where nodes connect directly among themselves and every node maintains a one-to-one chat with every other connected remote node. In a P2P architecture, every peer instance can be represented by the same `Node` peer – in P2P, peers are homogeneous. The `Node` peer has a *multiple tie* to itself since nodes in a P2P system can maintain connections to arbitrary other nodes. A `Registry` peer is used to discover other nodes.

[2] ScalaLoci is implemented as an embedded Scala DSL. The presented code is valid Scala. To help the reader abstracting over the rough edges of the embedding concerning the type-level architecture definition, Scala code is complemented with pseudocode in violet.

After discovery, nodes do not need to remain connected to the registry. Hence, their relation to the registry is an *optional tie*:

```
trait Registry extends Peer { type Tie <: Multiple[Node] }
trait Node extends Peer { type Tie <: Multiple[Node] with
                                      Optional[Registry] }
```

```
peer Registry ties multiple[Node]
peer Node ties multiple[Node] and optional[Registry]
```

In case a node has more than one tie, ties are expressed by a compound type, e.g., `Multiple[Node] with Optional[Registry]` is a *multiple tie* to a `Node` peer and an *optional tie* to a `Registry` peer. Thus, peers and ties can specify complex schemes. *Multiple* ties are the most general case. *Optional* ties model a channel that may not be open for the complete up time of the application, forcing the developer to explicitly deal with such case in the application code. For instance, in our P2P chat example, nodes do not need to stay connected to the registry for chatting with other nodes, hence their tie is optional. Restricting the tie to *single* removes the need of handling the case that no remote peer instance is connected.

A peer abstracts over instances of the same kind. Yet, during execution, multiple peer instances, e.g., multiple nodes of peer type `Node`, can inhabit the system and dynamically connect to other peer instances at run time. ScalaLoci allows distinguishing among them using *remote peer references*. Peer instances of the same type can be accessed uniformly via *value aggregation* (Section 2.4.1).

2.2 Placement Types

Based on the peers defined in the system architecture, ScalaLoci allows specifying *placed* data and computations. Placement is statically defined and part of a value's type.

2.2.1 Placing Data. In ScalaLoci, placed values of type `T on P`[3] represent a value of type `T` placed on a peer `P`. For example, in our P2P chat application, each `Node` peer defines an event stream of outgoing messages, i.e., the stream is *placed* on the `Node` peer:

```
val messageSent: Event[String] on Node = /* stream of outgoing messages */
```

Event streams of type `Event[T]` and time-changing values of type `Signal[T]` are part of ScalaLoci's reactive abstractions described in Section 2.3. The `messageSent` stream is accessible remotely from other peers to receive the chat messages. Remote visibility of placed values can be regulated: *Shared* placed values denoted by the type `T sharedOn P` specify values placed on a peer `P` that can be accessed from other peers. This is the default case (`T on P` is an alias for `T sharedOn P`). *Local* placed values denoted by the type `T localOn P` specify values that can only be accessed locally from the same peer instance. In the P2P chat, the `Registry` peer maintains an index of all participants. The index is defined local to the `Registry` peer to prevent participants from directly accessing the index:

```
val participantIndex: Index localOn Registry = /* users registry */
```

2.2.2 Accessing Placed Data. Accessing remote values (values on another peer instance) requires the `asLocal` syntactic marker. With `asLocal`, we remind developers that a remote access creates a *local* representation of the value (e.g., by transmitting it over the network or by establishing a remote dependency) that can then be used locally. There are two reasons for making remote communication explicit: First, we want to raise the developers' awareness of whether they are accessing a local or a remote value since (i) local and remote accesses have completely different performance characteristics regarding latency and (ii) remote invocation can potentially fail due to network communication failures. Second, having explicit syntax for remote access allows selecting different aggregation schemes, i.e., different ways of abstracting over multiple remote peer instances.

[3]The Scala compiler treats `T on P` and `on[T, P]` equivalently.

For example, we use `asLocal` to access the remote value of a single remote peer instance and `asLocalFromAll` to access the remote values of multiple remote peer instances in case of a multiple tie (Section 2.4.1).

The following example demonstrates the remote access to the `messageSent` event streams of the chat partners. We assume the existence of a `joinMessages` function, which collects all incoming messages into a growing list representing the log of received messages:

```
val receivedMessages: Signal[List[String]] on Node = placed[Node] { joinMessages(messageSent.asLocalFromAll) }
```

Thus, this line of code makes the messages sent from remote peer instances (`messageSent`) locally available (`asLocalFromAll`) as the list of received messages (`receivedMessages`). Since `receivedMessages` is placed on `Node`, the expression `joinMessages(messageSent.asLocalFromAll)` is evaluated on each `Node` instance. Placed declarations are initialized with `placed[P] { e }` expressions. Either the type argument for `placed` or the explicit type ascription `T on P` for the declaration can be left out since the other can be inferred by the Scala type checker. Either of them is required to specify the placement – we want developers to consciously decide on placement. As for the previous declaration, `messageSent` is also placed on `Node`. Thus, every `Node` instance provides a `messageSent` event stream. Since the architecture defines a multiple `Node`-to-`Node` tie, the remote access from a `Node` instance to another `Node` instance using `asLocalFromAll` is correct. In ScalaLoci, peers, ties, and placements are known statically and the compiler checks that access to remote values is consistent with the architecture definition. For instance, the following code shows an invalid remote access, which is statically rejected by the compiler:

```
val remoteIndex: Index on Registry = placed[Registry] { participantIndex.asLocal }
```

The remote access via `asLocal` from the `Registry` peer to the `participantIndex`, which is itself placed on `Registry`, does not type-check for two reasons: (i) `participantIndex` is not accessible remotely since it is a local placed value and (ii) the remote access violates the architecture specification, which does not specify a `Registry`-to-`Registry` tie.

Accessing non-reactive remote values features copy semantics – changes are not reflected remotely. Accessing reactive remote values establishes a remote *depends-on* relation (Section 2.3).

2.3 Multitier Reactives

In the running example of our P2P chat, we already defined the `messageSent` *event* and the `receivedMessages` *signal*. Events and signals are ScalaLoci's *reactive abstractions* (a.k.a. *reactives*) in the style of systems like REScala [Salvaneschi et al. 2014b]. Events model discrete changes. The following code defines an event stream of type `Event[String]`, which is a sequence of events carrying a `String` each, and pushes a new event into the stream:

```
val messageSent: Event[String] = Event[String]()
messageSent.fire("some message")
```

Signals model continuous time-changing values which are automatically updated by the language runtime. For instance, a signal `val s3: Signal[Int] = Signal { s1() + s2() }` depends on input signals `s1` and `s2`. The *signal expression* `s1() + s2()` is recomputed to update `s3` every time `s1` or `s2` changes. Signals inside `Signal {...}` expressions require `()` to access their current value and register them as dependencies. Events and signals can interoperate. The following snippet defines a signal that holds a list of all sent messages:

```
val sentMessages: Signal[List[String]] = messageSent.fold(List.empty[String]) { (log, message) => message :: log }
```

Event streams support operations such as folding, mapping, filtering and handler registration [Maier and Odersky 2013]. For example, the `list` operator can be used to define the signal `sentMessages` as `messageSent.list` instead of using the `fold` operator to fold over the event stream

as shown in the code snippet above. Defining reactives based on other reactives establishes a *depends-on* relation between them (e.g., `sentMessages` depends on `messageSent`), allowing the definition of complex data flow graphs with transitive dependencies.

ScalaLoci embraces asynchronous remote communication, which is the default in many distributed systems for performance and decoupling. Remote access to a reactive via `asLocal` creates a local representation of the reactive and extends the data flow graph without blocking. The interface to the reactive system is safe since accessing the current value of a reactive from imperative code returns a future to account for asynchronicity and registered handlers are invoked asynchronously. Asynchronicity only becomes visible when imperative code interfaces the reactive system, e.g., a method invocation that fires an event may return before the event reaches remote dependents. Instead, in the RP style, propagating values asynchronously is transparent to the user. Our consistency model corresponds to the one commonly used in actor systems and implemented by Akka [2009a]: remote reactives feature at-most-once delivery and order preservation for sender–receiver pairs (cf. Section 5). Local propagation is *glitch-free* [Cooper and Krishnamurthi 2006].

2.4 Peer Instances

Architecture definitions specify peer *types* and their relation, but the number of connected peer *instances* can change over time. Since peers abstract over multiple instances of the same peer type, remote access to a value `T on P` may refer to the values of multiple instances of type `P`. ScalaLoci offers two options to handle such multiplicity: (i) use an aggregated value over all remote peer instances or (ii) access a single value of the desired peer instance. A specific instance of type `P` is identified by a remote peer reference of type `Remote[P]`.

2.4.1 Aggregation. Accessing a remote value reads a value from each connected peer instance. In case of a multiple tie, reading a remote value abstracts over connected remote peer instances and aggregates their value. The aggregation scheme depends on the type of the value. Accessing remote primitives and standard collections returns a value of type `Map[Remote[P], Future[T]]` providing a future – which is part of Scala's standard library – for each remote peer. The future values account for network latency and possible communication failures by representing a value which may not be available immediately, but will become available in the future or produce an error.

Accessing remote reactives (i.e., events or signals) creates a dependency to each accessed reactive. Upon remote access, changes of the remote reactives are propagated to the accessing peer. Accessing a remote reactive yields a value of type `Signal[Map[Remote[P], Signal[T]]]` for signals and of type `Signal[Map[Remote[P], Event[T]]]` for events. The outer signal changes when the list of connected peers does. The inner reactive corresponds to the reactive on each peer instance `Remote[P]`. Reactives subsume the functionalities of futures, i.e., the propagation of remote values is asynchronous (cf. Section 2.3) and reactives can propagate failures (cf. Section 4).

To make aggregation explicit, we require an `asLocalFromAll` variant to access multiple remote instances uniformly. The following code snippet accesses the `messageSent` event stream of our P2P chat to aggregate all incoming messages from all the chat partners:

```
val incoming: Signal[Map[Remote[Node], Event[String]]] on Node = placed[Node] { messageSent.asLocalFromAll }
```

We additionally provide `asLocalFromAllSeq` for placed event streams, which aggregates the event occurrences of all connected remote peer instances into a single local event stream, providing all event occurrences sequentially in the single stream. This access pattern is often more convenient for event streams, which represent discrete occurrences:

```
val incomingSeq: Event[(Remote[Node], String)] on Node = placed[Node] { messageSent.asLocalFromAllSeq }
```

In Section 3, we demonstrate how aggregation simplifies treatment of several remote instances of equal type (Listing 6).

2.4.2 Instance-Based Access. Complementing aggregation, the `from` operator is used to access a remote value from a specific peer instance only. The following example shows the remote access to the `messageSent` stream of the single peer instance given by `node`. The messages are then stored in a time-changing list of type `Signal[List[String]]` using the `list` operator on event streams:

```
def messageLog(node: Remote[Node]): Signal[List[String]] on Node = placed[Node] { (messageSent from node).asLocal.list }
```

Sometimes, selecting data *on the receiver*, like `from` does, is inefficient or insecure. Hence, dually to `from`, ScalaLoci provides declarative sender-side selection via *subjective* values. By default, every peer instance accessing a shared value reads the same content. In contrast, subjective values exhibit a different value based on the accessing peer instance. Specifically, the `sbj` modifier binds an identifier holding a reference to the peer instance that accesses the subjective value. Using this identifier, user code can filter the value on a per remote peer instance basis before the value leaves the local instance. In the P2P chat, every participant can take part in multiple chats simultaneously, but messages typed by a user should be sent only to the currently selected partner. We achieve this goal by declaring the `messageSent` event stream of outgoing messages *subjective*:

```
val messageSent = placed[Node].sbj { node: Remote[Node] => ui.messageTyped filter { msg => ui.isSelectedChat(node) } }
```

The `node` identifier is used to filter the `ui.messageTyped` event stream based on the accessing peer instance for defining the `messageSent` stream that only contains the messages for the `node` chat partner. Crucially, when accessing a subjective declaration, each peer instance "sees" a different `messageSent` event stream containing only the messages directed to it.

2.4.3 Dynamic Connections. The `remote[P].connected` signal provides a time-changing list of currently connected peer instances of type *P*. User code is informed about established or lost connections via the `remote[P].joined` and `remote[P].left` events. It is possible to setup connections programmatically at run time using `remote[P] connect` and providing the address at which the remote peer instance is reachable.

2.5 Offloading Work: Remote Blocks

Listing 1. Remote blocks.

```
1 val requestedId = /* ... */
2
3 remote[Registry].capture(requestedId).sbj { requesting: Remote[Node] =>
4   val requested = participantIndex.getRemote(requestedId)
5   val address = participantIndex.getAddress(requesting)
6   remote.on(requested).capture(address) {
7     remote[Node] connect address } }
```

Distributed systems often require offloading data processing to other hosts. As an ingredient of this mechanism, ScalaLoci provides *remote blocks*, i.e., (sub-)expressions executed on remote peer instances. The value from the remote evaluation of the block is returned to the instance issuing the block. A remote block expression `remote[P] { e }` runs the computation *e* on every connected peer instance of type *P* and `remote.on(p) { e }` runs the computation on the given instance *p*. Remote blocks only capture (*close* around) values stated explicitly via `capture` and – like remote access via `asLocal` (Section 2.2.2) – feature copy semantics for non-reactive values. Implicit captures, that may be unintentional, are compilation errors.

Listing 1 shows an excerpt from our P2P chat example to setup the connection between two peer instances dynamically at run time. The initiating node transfers the connection request to the requested node through the central registry server, which maintains a connection to both nodes. The initiating node passes the value of `requestedId` to the registry via a remote block (Line 3). The remote block is dispatched *subjectively* (Section 2.4.2), i.e., the `requesting` identifier (Line 3) is bound to a peer remote reference to the initiating node. The registry resolves the remote reference for the

requested peer and the address at which the node is reachable from an internal `participantIndex` (Lines 4 and 5). The registry runs a remote block on the requested peer instance (Line 6), which connects to the initiating node (Line 7) by setting up a dynamic connection via `remote[P] connect` (Section 2.4.3).

Remote blocks do not pose a security threat as their semantics does not entail mobile code. They delimit a functionality executed on another peer instance in a concise way. No code is sent around, only the values exchanged between the block and the surrounding environment. Also, the code that a remote block executes remotely is statically known. For `placed[PeerA] { val v = x; remote[PeerB].capture(v) { val y = v } }`, for example, the code separation into peer-specific code places `val v = x` on `PeerA` and `val y = v` on `PeerB`. This placement is fixed after compilation. Only the value `v` (which is explicitly captured) is sent from the instance of `PeerA` to an instance of `PeerB`.

3 SCALALOCI AT WORK

In this section, we demonstrate, with several *complete* applications, how the abstractions introduced in Section 2 help defeating the complexity of developing distributed systems. Multitier code is syntactically valid and type-correct Scala code. The `@multitier` annotation gives placement types special semantics to enable peer-based splitting (Section 7).

Chat Application: Messaging. Listing 2 concludes the P2P chat example showing the complete messaging logic. We only leave out the logic for dynamically setting up connections, which is already in Listing 1. The application logs messages that are sent and received by each participant as a composition of the data flow from the local UI and from remote chat partners. In the example, nodes are connected to multiple remote nodes and maintain a one-to-one chat with each. Users can select any chat to send messages. The `messageSent` (Line 6) event is defined as subjective value (Section 2.4.2) filtering the `ui.messageTyped` messages from the UI (Line 7) for the currently active chat partner `node`. The `messageLog` (Line 9) signal contains the chat log for the chat between the local peer instance and the remote `node` given as parameter. It merges the remote stream for the chat messages *from the remote instance* `node` (Line 11) and the local stream *subjective to the remote instance* `node` (Line 12) via the `||` operator. The event stream resulting from such merge fires whenever either of both operands fires. The chat log is a signal created using `list`, which extends the list by an element for each new event occurrence in the merged stream. The `chatLogs` signal folds (Line 15) the `remote[Node].joined` event stream (cf. Section 2.4.3), which is fired for each newly connected chat partner, into a signal that contains the chat logs for every chat partner generated by calling `messageLog` (Line 16).

Listing 2. Messaging logic for multiple P2P chats.

```
1  @multitier object P2PChat {
2    trait Node extends Peer { type Tie <: Multiple[Node] }
3
4    val ui: UI on Node = placed { UI() }
5
6    val messageSent = placed[Node].sbj { node: Remote[Node] =>
7      ui.messageTyped filter { _ => ui.isSelectedChat(node) } }
8
9    def messageLog(node: Remote[Node])
10   : Signal[List[String]] on Node = placed {
11     ((messageSent from node).asLocal ||
12      (messageSent to node)).list }
13
14   val chatLogs: List[Signal[List[String]]] on Node = placed {
15     remote[Node].joined.fold(List.empty[Signal[List[String]]]) {
16       (chats, node) => messageLog(node) :: chats } }
17 }
```

Tweets. Next, we show how the operators in a processing pipeline can be placed on different peers (Listing 3) to count the tweets that each author produces in a tweet stream. The application receives a stream of tweets on the `Input` peer (Line 7), selects those containing the `"multitier"` string on the `Filter` peer (Line 8), extracts the `author` for each tweet on the `Mapper` peer (Line 9), and stores a signal with a map counting the tweets from each author on the `Folder` peer (Line 11).

Listing 3. Tweet Processing Pipeline.

```
1  @multitier object TweetProcessing {
2    trait Input extends Peer { type Tie <: Single[Filter] }
3    trait Filter extends Peer { type Tie <: Single[Mapper] with Single[Input] }
4    trait Mapper extends Peer { type Tie <: Single[Folder] with Single[Filter] }
5    trait Folder extends Peer { type Tie <: Single[Mapper] }
6
7    val tweetStream: Event[Tweet] on Input = placed { retrieveTweetStream() }
8    val filtered: Event[Tweet] on Filter = placed { tweetStream.asLocal filter { tweet => tweet.hasHashtag("multitier") } }
9    val mapped: Event[Author] on Mapper = placed { filtered.asLocal map { tweet => tweet.author } }
10   val folded: Signal[Map[Author, Int]] on Folder = placed {
11     mapped.asLocal.fold(Map.empty[Author, Int].withDefaultValue(0)) { (map, author) => map + (author -> (map(author) + 1)) } }
12 }
```

Email Application. Listing 4 shows a client–server e-mail application. The server stores a list of e-mails. The client can request the e-mails received in the n previous days containing a given word. The client user interface displays the e-mails broken into several pages. If the word is not in the current page, the user is informed.

The definition of the `word` signal of type `Signal[String] on Client` (Line 5) defines a signal carrying strings placed on the `Client` peer. Thanks to multitier reactives, the client-side signal `inCurrentPage` is defined by the composition of the local client-side signal `word` and the remote server-side signal `filteredEmails` (Line 20). The latter (Line 12) is defined as a composition of a local signal (Line 14) and two remote signals (Lines 15 and 16).

Listing 4. Email application.

```
1  @multitier object MailApp {
2    trait Server extends Peer { type Tie <: Single[Client] }
3    trait Client extends Peer { type Tie <: Single[Server] }
4
5    val word: Signal[String] on Client = placed { /* GUI input */ }
6
7    val days: Signal[Int] on Client = placed { /* GUI input */ }
8
9    val allEmails: Signal[List[Email]] localOn Server =
10     placed { /* e-mail collection */ }
11
12   val filteredEmails: Signal[List[Email]] on Server =
13     placed { Signal {
14       allEmails() filter { email =>
15         (email.date >= Date.today() - days.asLocal()) &&
16         (email.text contains word.asLocal()) } } }
17
18   val inCurrentPage: Signal[Boolean] localOn Client =
19     placed { Signal {
20       isCurrentFirstPage(word(), filteredEmails.asLocal()) } }
21 }
```

Token Ring. We model a token ring (Listing 5), where every node in the ring can send a token for another node. Multiple tokens can circulate in the ring simultaneously until they reach their destination. Every node has exactly one predecessor and one successor. We define a `Prev` and a `Next` peer and specify that a `Node` itself is both a predecessor and a successor and has a single tie to its own predecessor and a single tie to its successor. Using *multiple* ties would allow nodes to join and leave updating the ring dynamically but is not discussed further. Tokens are passed from predecessors to successors, hence nodes access the tokens sent from their predecessor. For this reason, values are placed on the `Prev` peer. Every node has a unique ID (Line 7). The `sendToken` event (Line 9) sends a token along the ring to another peer instance. The `recv` event stream (Line 12) provides the data received by each peer instance. Each

Listing 5. Token ring.

```
1  @multitier object TokenRing {
2    trait Prev extends Peer { type Tie <: Single[Prev] }
3    trait Next extends Peer { type Tie <: Single[Next] }
4    trait Node extends Prev with Next {
5      type Tie <: Single[Prev] with Single[Next] }
6
7    val id: Id on Prev = placed { Id() }
8
9    val sendToken: Event[(Id, Token)] localOn Prev = placed {
10     Event[(Id, Token)]() }
11
12   val recv: Event[Token] localOn Prev = placed {
13     sent.asLocal collect {
14       case (receiver, token) if receiver == id => token } }
15
16   val sent: Event[(Id, Token)] on Prev = placed {
17     (sent.asLocal \ recv) || sendToken }
18 }
```

Listing 6. Master–worker.

```
1  @multitier object MasterWorker {
2    trait Master extends Peer { type Tie <: Multiple[Worker] }
3    trait Worker extends Peer { type Tie <: Single[Master] }
4  
5    case class Task(v: Int) { def exec: Int = 2 * v }
6  
7    val taskStream: Event[Task] localOn Master = placed { Event[Task]() }    // to add tasks: `taskStream.fire(Task(42))`
8  
9    val assocs: Signal[Map[Remote[Worker], Task]] localOn Master = placed {
10     (taskStream || taskResult.asLocalFromAllSeq).fold(Map.empty[Remote[Worker], Task], List.empty[Task]) {
11       (taskAssocs, taskQueue, taskChanged) => assignTasks(taskAssocs, taskQueue, taskChanged, remote[Worker].connected) } }
12  
13   val deployedTask = placed[Master].sbj { worker: Remote[Worker] =>       // Signal[Task]
14     Signal { assocs().get(worker) } }
15   val taskResult = placed[Worker] {                                        // Event[Int]
16     deployedTask.asLocal.changed collect { case Some(task) => task.exec } }
17   val result = placed[Master] {                                            // Signal[Int]
18     taskResult.asLocalFromAllSeq.fold(0) { case (acc, (worker, result)) => acc + result } }
19 }
```

node fires `recv` when it receives a token addressed to itself, i.e., when the `receiver` equals the node ID (Line 14) and forwards other tokens. The expression `sent.asLocal \ recv` (Line 17) evaluates to an event stream of all events from `sent.asLocal` for which `recv` does not fire. Merging such stream (of forwarded tokens) with the `sendToken` stream via the `||` operator injects both new and forwarded tokens into the ring.

Master–worker. We now show a ScalaLoci implementation of the master–worker pattern (Listing 6) where a master node dispatches tasks – double a number, for simplicity – to workers. The `taskStream` on the master (Line 7) carries the tasks (Line 5) as events. The `assocs` signal (Line 9) contains the assignments of workers to tasks. It folds over the event stream `taskStream || taskResult.asLocalFromAllSeq` that fires for every new task (`taskStream`) and every completed task (`taskResult.asLocalFromAllSeq`). The `assignTasks` method (Line 11) assigns a worker to the new task (`taskAssocs`), or enqueues the task if no worker is free (`taskQueue`) based on the folded event (`taskChanged`) and the currently connected worker instances (`remote[Worker].connected`). The `deployedTask` signal (Line 13) subjectively provides every worker instance with the task it is assigned. Workers provide the result in the `taskResult` event stream (Line 15), which the master aggregates into the `result` (Line 17) signal. The signal is updated for every event to contain the sum of all values carried by the events.

4 FAULT TOLERANCE

To handle failures, we provide a mechanism that unifies reactives and supervision *à la* actors. The key idea is that the *depends-on* relation between reactives establishes a *supervisor-of* relation, where the supervisor is notified if a supervised reactive fails. For example, in `e.filter(_ > 10).map(_.toString)` the `map` reactive depends on the `filter` reactive and `map` supervises `filter`. One can take two perspectives on this mechanism.

From the RP perspective, reactives propagate Err upon failure. Signals and events can carry the successfully computed value v or Err. If, during reevaluation, a reactive accesses an Err value of a reactive it depends on, the recomputation *fails* (i.e., the computation is skipped, like in Akka [2009c] actors), and the reactive also emits Err. Propagation of errors along the data flow graph is in line with the approaches taken by Rx [Meijer 2010] and Reactive Streams [2014] (and the Reactive Streams implementations RxJava and Akka Streams). A reactive can (i) ignore computation failures (`e.recover{ err => Drop }`) or (ii) handle them replacing Err by a v (`e.recover{ err => Revive(v) }`).

A reactive carrying Err holds a success value v again as soon as it reevaluates to v upon new input. Thus, a failed computation does not prevent processing further input.

The other perspective is about supervision relations. To supervise a reactive computation *r*, one simply declares another reactive that depends on *r*, which then becomes the supervisor, like in actors, and receives Err if the supervised reactive fails. A supervisor can (i) ignore notifications of failure or (ii) handle them implementing a recovery strategy.

Our mechanism supports the most common cases for reactives, ScalaLoci's main communication abstraction, still retaining the full generality of supervision relations that proved effective in the actor model. This mechanism allows monitoring reactive computations that are not necessarily arranged as trees – trees are a special case. Similarly, Akka supports monitoring schemes (beside supervision trees) that allow arbitrary monitoring relations [Akka 2009e]. For streams, for example, one can neglect the cases that produce a failure (e.g., with spurious data in big data analytics), generate default values that track failed cases, or check the successful processing of an event through a complete stream pipeline by creating a stream of acknowledgments (or Err) from the sink to the source (cf. Section 5, Listing 7), similar to how bolts acknowledge the successful processing of a tuple in Apache Storm [2011]. Failures in the generated communication layer occur in case a remote connection breaks or cannot be established. Accessing a remote reactive which is not connected (anymore) also propagates an error value `Err`, thus making user code aware of communication failures. Generality is achieved building on top of the Err propagation/supervision mechanism, to allow custom fault handling strategies. For example, a supervisor can emit an event inducing all reactives in the system to reset their internal state, the equivalent of the `one_for_all` Erlang recovery strategy [Erlang 1987b].

We demonstrate our approach augmenting the master–worker example (Listing 6). A first improvement is that the master simply ignores tasks that cause a worker to fail. This is achieved by dropping Err before it propagates to `fold` in Line 18. The `result` signal depends on the workers' `taskResult` using `taskResult.asLocalFromAllSeq`, which performs event aggregation (Section 2.3) *and* establishes a supervision relation. A small change to the `result` signal (in gray) suffices:

```
taskResult.asLocalFromAllSeq .recover{ err => Drop }.fold(...)
```

After merging events from all workers, all `Err` values are dropped from the stream, and `fold` processes only successfully computed values v. A second improvement is to introduce a stream of failed tasks for diagnostics:

```
val failedTasks: Report on Master = placed {
  taskResult.asLocalFromAllSeq
    .recover { err => Revive(Report(assocs().get(err.remote))) }
    .collect { case report @ Report(_) => report } }
```

The `failedTasks` stream first replaces events carrying `Err` by a `Report` value (using the `recover` operator with `Revive`). The report contains the task associated with the disconnected worker remote peer instance (via `assocs`). Finally, reports are collected, filtering out successfully completed tasks.

Machine failures or network connection losses affect all reactives on the lost peer instance. To react to a disconnection, peer instances monitor the peer instances they are connected to. A peer instance is informed about a disconnection via the `remote[P].left` event (Section 2.4.3) and can take countermeasures. This mechanism is similar to how Akka detects that the communication to a remote actor fails. The monitoring actor receives a `Terminated` message for the monitored actor [Akka 2009d]. In the third improvement, the master reassigns the task that was running on the disconnected worker to another worker via an event stream of tasks to redeploy:

```
val redeploy = placed[Master] { remote[Worker].left map { worker => assocs().get(worker) } }
```

The `redeploy` event stream maps every disconnected worker remote reference (provided by `remote[Worker].left`) to the task to which the respective worker was assigned (via `assocs`), resulting in a stream of tasks that were assigned to disconnected worker instances. The `assocs` signal (Listing 6, Line 9) computing the assignments of workers to tasks also needs to be updated to consider the tasks to be redeployed, additionally folding over the `redeploy` event stream (Line 10):

```
(taskStream || taskResult.asLocalFromAllSeq || redeploy).fold(...)
```

5 EXECUTION MODEL AND LIFE CYCLE

Consistency. ScalaLoci adopts the consistency model found in many actor systems: (1) *at-most-once* delivery (i.e., delivery is not guaranteed and dropped messages are lost) for remote reactives and (2) order preservation for sender–receiver pairs [Akka 2009b; Erlang 1987a]. Similar to actors, a developer can implement stronger consistency manually on top of ScalaLoci. For example, to achieve *at-least-once* delivery, events can be re-sent until they are acknowledged. In Listing 7, a `send` event on `PeerA` transfers some payload of type `Data` to `PeerB`. A `timeout` event is regularly fired to implement the timeout for pending acknowledgments. The `msg` event (Line 10), which is a `send` event or a `resend` event after the timeout expired, is read on `PeerB` (Line 12). In the example, the `msg` event is sent back wrapped in an `Ack` acknowledgment (Line 12). On `PeerA`, we fold over the `send || ack.asLocal` event stream (Line 2), adding all sent events to the set of events awaiting acknowledgment (Line 4) and removing all acknowledged events from the set (Line 3). Every time `timeout` fires, the `resend` event (Line 6) fires for one of the events that are not acknowledged yet. For simplicity, in the example, we do not preserve ordering.

Listing 7. At-least-once delivery on top of ScalaLoci.

```
1  val missing: Signal[Set[Data]] on PeerA = placed {
2    (send || ack.asLocal).fold(Set.empty[Data]) {
3      case (missingAcks, Ack(v: Data)) => missingAcks - v
4      case (missingAcks, v: Data) => missingAcks + v } }
5
6  val resend: Event[Data] on PeerA = placed {
7    timeout collect { case _ if missing().nonEmpty =>
8      missing().head } }
9
10 val msg: Event[Data] on PeerA = placed { send || resend }
11 val ack: Event[Ack[Data]] on PeerB = placed {
12   msg.asLocal map { v => Ack(v) } }
```

Given the abstraction level of ScalaLoci (reactives, multitier programming), however, we expect that developers do not implement higher consistency levels themselves. Hence, ScalaLoci allows developers to choose among different reactive systems with different levels of consistency. For simplicity, we do not cover this modular architecture in the paper. We just assume the single consistency model based on (1) and (2). Yet, we currently support several backends, e.g., Rx [Meijer 2010] as well as one that provides stronger consistency guarantees [Drechsler et al. 2014].

Cycles. ScalaLoci allows defining software that entails distributed cycles, such as in the Token Ring application (Listing 5). In this example, a node in the ring receives a token via the `sent` event from its predecessor and (i) either emits it again via its `sent` event to be processed by the successor node (forming a cycle of `sent` events along the nodes in the ring) or (ii) processes the token and removes it from cycling through the ring. The reactive model used by ScalaLoci allows sending events along peer instances that are arranged in a cycle since messages are sent from one instance to another asynchronously (similar to messages in actor systems). After sending an event to a remote instance, the local instance continues processing incoming events. In particular, incoming events may be events that were originally sent by the local instance and are reaching the local instance again through a cycle. With this model, events can cycle around in the system being passed on to the next node until they are consumed and removed from the cycle.

Execution Order and Concurrency. Placed values are initialized at bootstrap on each peer instance. The evaluation of placed expressions adheres to the standard Scala semantics: variable declarations

(`val` or `var`) are evaluated in defintion order, lazy values (`lazy val`) are evaluated upon first access and method definitions (`def`) are evaluated on each access. ScalaLoci, by default, uses a single thread for evaluating values and computing remote blocks initiated by remote instances. This behavior can be adapted, e.g., using a thread pool to improve concurrent access. Sequential remote accesses from the same remote instance are evaluated in the same order but there are no guarantees for the remote access by different instances.

Bootstrapping. A multitier application is bootstrapped by calling `multitier.start[P]`, starting an instance of peer P on the local host and automatically setting up the specified connections. Listing 8 shows the setup for a client–server application. The `Server` *listens* on the tie towards the `Client`. The `Client` *connects* via the tie towards the `Server`. For each tie, a peer instance can either initiate the communication (*connect*) or wait for incoming connections (*listen*). A peer instance can *connect* to a single remote instance on a single or optional tie and to an arbitrary number of remote instances on a multiple tie. In case a peer instance listens on a single or optional tie, a new local peer instance is created for each incoming connection. This is commonly the case in the client–server setting: a new server instance starts upon each request. In contrast, when a peer listens on a multiple tie, a single local peer instance handles all incoming connections.

Listing 8. Automatic connection setup.

```
1  @multitier object MyApp {
2    trait Server { type Tie = Single[Client] }
3    trait Client { type Tie = Single[Server] }
4  }
5
6  object ServerApp extends App {
7    multitier.start[MyApp.Server] {
8      listen[MyApp.Client] { TCP(1099) } }
9  }
10
11 object ClientApp extends App {
12   multitier.start[MyApp.Client] {
13     connect[MyApp.Server] { TCP("example.com", 1099) } }
14 }
```

The dynamic counterpart to the automatic connection setup for establishing connections at run time is introduced in Section 2.4.3, e.g., using `remote[Server] connect TCP("example.com", 1099)`.

Deployment. To deploy a ScalaLoci application on the JVM, we generate Java bytecode that can be packaged into JAR files as usual. For web deployment, we generate Java bytecode for the server and JavaScript code for the client. The client code connects to the server using HTML5 WebSockets. ScalaLoci can potentially work with any web server to answer the initial HTTP request from the browser and serve the generated JavaScript code. We implemented example applications using Akka HTTP [2016] and the Play Framework [2007] for web applications as HTTP servers.

ScalaLoci peers on different hosts can be updated independently as long as there are no changes to the signature of placed values accessed by remote instances. Changing the signature of placed values requires that all affected peers are updated to avoid incompatibilities.

6 FORMALIZATION

We formalize a core calculus for ScalaLoci that models peers, placement, remote access, remote blocks and reactives (only signals, for simplicity). The formalization describes the core concepts of ScalaLoci and is a basis to prove our system sound regarding static types and placement. We implemented the calculus in Coq [Coq Development Team 2016] and mechanized the proofs.

6.1 Syntax

The syntax is in Figure 1. Types are denoted by T. Types of values that can be accessed remotely and transmitted over the network are denoted by U. Placement types S (cf. Section 2.2) are defined based on a numerable set of peer type names $\mathcal{P}$. Since Scala has no native notion of *peers*, we encode peers by subclassing `Peer` in the Scala embedding. $\mathcal{P}$ corresponds to the set of all `Peer` subclasses in the embedding. Besides standard terms, t includes remote access (cf. Section 2.2.2, remote blocks (cf. Section 2.5) and reactives (cf. Section 2.3). Remote access via asLocal is explicitly

Syntax	
l ::= $(\mathcal{T}, \mathcal{S}, \mathcal{I}, s)$	program
s ::= placed x: S = t in s | end	placement term
t ::= $\lambda x{:}\,T.\,t$ | $t\ t$ | x | unit |	standard term
none of T | some t | nil of T | cons $t\ t$ |	
asLocal x: S [from t] |	remote access term
asLocal block x: T = t in t : S [from t] |	remote block term
signal t | var t | now t | set t := t | p |	reactive term
r | ϑ | asLocal t : S | asLocal t : S [from t]	intermediate term
v ::= $\lambda x{:}\,T.\,t$ | unit | end | p | r | ϑ |	value
none of T | some v | nil of T | cons $v\ v$	
T ::= Option T | List T | Signal T | Var T | $T \to T$ | U	type
U ::= Option U | List U | Signal U | Remote P | Unit	transmittable type
S ::= T on P	placement type
$P \in \mathcal{P}$	peer
$\mathcal{T}\ :\ \mathcal{P} \times \mathcal{P} \to$ {multiple, optional, single, none}	ties
$\mathcal{S}\ :\ \mathcal{I} \to \mathcal{P}$	peer instance type
$p = \{i\} \subseteq \vartheta \subseteq \mathcal{I}$	peer instance

Fig. 1. Syntax.

ascribed with a type S for the accessed value. We model both aggregation over all remote values of connected peer instances (cf. Section 2.4.1) – but we do not distinguish syntactically between different variants of asLocal for aggregation – and selecting a specific peer instance using the from operator (cf. Section 2.4.2). A program $l = (\mathcal{T}, \mathcal{S}, \mathcal{I}, s)$ consists of the architecture defined via the ties $\mathcal{T}$ (cf. Section 2.1), the peer types $\mathcal{S}$ for peer instances $\mathcal{I}$ and the definition of placed values, modeled as nested end-terminated s-terms binding t-terms to names. Thus, s-terms express placed bindings and t-terms are placed expressions, which evaluate to the value that is bound (cf. Section 2.2.1). Placement defined by a term s binds an identifier x of type S to a term t, where S specifies the placement. We consider a fixed set $\mathcal{I}$ of peer instances that can participate in the evaluation of the program and $\mathcal{S}$ a mapping from peer instances to their peer type P. There can be multiple individual peer instances $p \in \mathcal{I}$ of a peer type P (cf. Section 2.4). A remote peer reference, which is typed as `Remote[P]` in the Scala embedding, is given the type Remote P in the formal development. $\mathcal{T}$ specifies the tie multiplicity of each two peers. We adopt the notation $P_0 \overset{*}{\rightarrowtail} P_1$ iff $\mathcal{T}(P_0, P_1)$ = multiple, $P_0 \overset{?}{\rightarrowtail} P_1$ iff $\mathcal{T}(P_0, P_1)$ = optional, $P_0 \overset{1}{\rightarrowtail} P_1$ iff $\mathcal{T}(P_0, P_1)$ = single, $P_0 \overset{0}{\rightarrowtail} P_1$ iff $\mathcal{T}(P_0, P_1)$ = none and $P_0 \leftrightarrow P_1$ iff $\mathcal{T}(P_0, P_1) \neq$ none and $\mathcal{T}(P_1, P_0) \neq$ none. Ties between two peers $P_0, P_1 \in \mathcal{P}$ are statically defined and directly correspond to ties between `Peer` subclasses in the embedding, defined through `type Tie` at the type level. Reactives include Vars, which can be set explicitly, and Signals. Both can be accessed to read the current value. Syntactic forms that are not part of the program language and arise only in the evaluation are highlighted in gray.

6.2 Dynamic Semantics

We first introduce the auxiliary construct $\mathcal{I}^{[P]}$ used in the rest of the formalization to denote the set of all peer instances of type P, i.e., $\mathcal{I}^{[P]} = \{p \in \mathcal{I} \mid \mathcal{S}(p) = P\}$. Note that $\mathcal{I}^{[P_1]}$ and $\mathcal{I}^{[P_2]}$ are disjoint for two distinct P_1 and P_2. Ties $\mathcal{T}$ are statically known and constrain the run time connections in a system, e.g., there can only be a single connection for a single tie but an arbitrary number for a multiple one. For simplicity, we do not model dynamic connections, assuming *fixed* connections along the defined ties $\mathcal{T}$. Each peer instance is connected to all remote instances $\mathcal{I}^{[P]}$ for every tied peer type P. We constrain the number of peer instances for every peer type as follows:

Definition 1. For all $P_0, P_1 \in \mathcal{P}$ holds (i) $P_0 \overset{1}{\rightarrowtail} P_1 \implies \left|\mathcal{I}^{[P_1]}\right| = 1$ and (ii) $P_0 \overset{?}{\rightarrowtail} P_1 \implies \left|\mathcal{I}^{[P_1]}\right| \leq 1$.

Figure 2 shows the auxiliary functions to transmit and aggregate over remote values. ζ models remote value transmission from peer instances ϑ of peer type P_1, all of which provide the remote value v of type T. Traditional values, such as options or lists, do not change during transmission. Signals are transmitted by creating a local signal that reevaluates the remote signal remotely on the peer instances on which it is placed. When accessing a remote value on peer instances of type P_1 from local peer instances of type P_0, the aggregation results either in a list of all remote values,

$$\begin{aligned}
\zeta(P_1, \vartheta, \text{none of } T', T) &= \text{none of } T' \\
\zeta(P_1, \vartheta, \text{nil of } T', T) &= \text{nil of } T' \\
\zeta(P_1, \vartheta, \text{unit}, T) &= \text{unit} \\
\zeta(P_1, \vartheta, \vartheta', T) &= \vartheta' \\
\zeta(P_1, \vartheta, \text{some } v, \text{Option } T) &= \text{some } \zeta(P_1, \vartheta, v, T) \\
\zeta(P_1, \vartheta, \text{cons } v_0\, v_1, \text{List } T) &= \text{cons } \zeta(P_1, \vartheta, v_0, T)\ \zeta(P_1, \vartheta, v_1, \text{List } T) \\
\zeta(P_1, \vartheta, r, \text{Signal } T) &= \text{signal asLocal block } x\colon \text{Unit} = \text{unit in now } r : T \text{ on } P_1 \text{ from } \vartheta \qquad \text{with } x \text{ fresh}
\end{aligned}$$

$$\Phi(P_0, P_1, T) = \begin{cases} \text{List } T & \text{for } P_0 \overset{*}{\rightarrowtail} P_1 \\ \text{Option } T & \text{for } P_0 \overset{?}{\rightarrowtail} P_1 \\ T & \text{for } P_0 \overset{1}{\rightarrowtail} P_1 \end{cases}$$

$$\frac{P_0 \overset{?}{\rightarrowtail} P_1 \qquad t' = \zeta(P_1, p, v, T)}{\varphi(P_0, P_1, p, v, T) = \text{some } t'} \ \text{(A-Some)}$$

$$\frac{P_0 \overset{*}{\rightarrowtail} P_1 \qquad t' = \zeta(P_1, p, v, T) \qquad t = \varphi(P_0, P_1, \vartheta, v, T)}{\varphi(P_0, P_1, p \,\dot{\cup}\, \vartheta, v, T) = \text{cons } t'\ t} \ \text{(A-Cons)}$$

$$\frac{P_0 \overset{1}{\rightarrowtail} P_1 \qquad t' = \zeta(P_1, p, v, T)}{\varphi(P_0, P_1, p, v, T) = t'} \ \text{(A-Value)}$$

$$\frac{P_0 \overset{?}{\rightarrowtail} P_1}{\varphi(P_0, P_1, \varnothing, v, T) = \text{none of } T} \ \text{(A-None)}$$

$$\frac{P_0 \overset{*}{\rightarrowtail} P_1}{\varphi(P_0, P_1, \varnothing, v, T) = \text{nil of } T} \ \text{(A-Nil)}$$

Fig. 2. Auxiliary functions ζ for transmission and φ and Φ for aggregation.

in an option of the remote value or in the remote value directly depending on the tie between P_0 and P_1. φ constructs a term t' that represents the aggregated result and Φ specifies its type.

Reduction Rules. To preserve the single multitier *flavor* of ScalaLoci, we model the evaluation as a single thread of execution annotating the reduction relation with a set of peer instances ϑ on which an evaluation step takes place, i.e., the reduction step takes place on all peer instances in ϑ. The reduction relation $s; \rho \overset{\vartheta}{\twoheadrightarrow} s'; \rho'$ reduces the placement term s and the reactive system ρ to s' and ρ' taking a single step on a set of peer instances ϑ. The reactive system ρ stores the reactive values created during the execution. More details about ρ are only relevant to the rules dealing with reactive terms (Figure 3d, described below). The reduction relation $\vartheta\colon P \triangleright t; \rho \overset{\vartheta'}{\longrightarrow} t'; \rho'$ for a term t placed on the peer instances ϑ of peer type P and a reactive system ρ evaluates to a term t' and a reactive system ρ' by taking a step on a set of peer instances ϑ'. The rules are in Figure 3.

The rules in Figure 3a for reducing a term t are standard except that they are extended with the reactive system ρ and the peer instances where the evaluation takes place. E-App steps on ϑ when evaluating on the peer instances $\vartheta\colon P$. E-Context evaluates $E[t]$ on ϑ when t steps on ϑ.

The reduction rules for placement terms s are in Figure 3b. A term placed $x\colon T$ on $P = t$ in s defines a *placed value* by binding the value of t to x in scope of s (cf. Section 2.2.1). E-Placed evaluates a placed term t on all instances $\mathcal{I}^{[P]}\colon P$ of the peer type given by the placement type T on P. The set ϑ denotes the peer instances where the evaluation steps. E-PlacedVal substitutes an evaluated placed value v in all instances $\mathcal{I}$.

The reduction rules for remote access terms t are in Figure 3c. The variants of asLocal model remote access to placed values and remote blocks (cf. Section 2.2.2). E-AsLocal accesses the remote value v on peer instances of type P_1 from the local instances ϑ of type P_0. The result is an aggregation φ (Figure 2) over all remote values (cf. Section 2.4.1). By assuming that every peer instance is connected to all instances of a tied peer type and Definition 1, the values of all peer instances $\mathcal{I}^{[P_1]}$ are aggregated. The evaluation steps on ϑ to provide the aggregated remote value to all instances ϑ. Similarly, E-AsLocalFrom provides the remote value v from the remote instances ϑ' to all local instances ϑ (cf. Section 2.4.2). E-Block applies the value v to a remote block t computed remotely on the peer instances of type P_1. A remote block term asLocal block $x\colon T_0 = t_0$ in $t_1 : T_1$ on P evaluates t_0, binds the result to x in the scope of t_1 and evaluates t_1 remotely on the instances of P (cf. Section 2.5). The resulting value of the remote evaluation of type T_1 is provided to the invoking

$$E ::= E[\cdot] \mid E\ t \mid v\ E \mid \mathsf{some}\ E \mid \mathsf{cons}\ E\ t \mid \mathsf{cons}\ v\ E \mid$$
$$\mathsf{asLocal}\ t\colon S\ \mathsf{from}\ E \mid \mathsf{asLocal\ block}\ x\colon T = E\ \mathsf{in}\ t : S \mid \mathsf{asLocal\ block}\ x\colon T = t\ \mathsf{in}\ t : S\ \mathsf{from}\ E \mid$$
$$\mathsf{asLocal\ block}\ x\colon T = E\ \mathsf{in}\ t : S\ \mathsf{from}\ v \mid \mathsf{var}\ E \mid \mathsf{now}\ E \mid \mathsf{set}\ E := t \mid \mathsf{set}\ v := E$$

$$\frac{\vartheta\colon P \rhd t;\rho \xrightarrow{\vartheta'} t';\rho'}{\vartheta\colon P \rhd E[t];\rho \xrightarrow{\vartheta'} E[t'];\rho'}\ \text{(E-Context)} \qquad \frac{}{\vartheta\colon P \rhd \lambda x\colon T.\ t\ v;\rho \xrightarrow{\vartheta} [x \mapsto v]t;\rho}\ \text{(E-App)}$$

(a) Standard terms.

$$\frac{\mathcal{I}^{[P]}\colon P \rhd t;\rho \xrightarrow{\vartheta} t';\rho'}{\mathsf{placed}\ x\colon T\ \mathsf{on}\ P = t\ \mathsf{in}\ s;\rho \overset{\vartheta}{\twoheadrightarrow} \mathsf{placed}\ x\colon T\ \mathsf{on}\ P = t'\ \mathsf{in}\ s;\rho'}\ \text{(E-Placed)}$$

$$\frac{}{\mathsf{placed}\ x\colon T\ \mathsf{on}\ P = v\ \mathsf{in}\ s;\rho \overset{\mathcal{I}}{\twoheadrightarrow} [x \mapsto v]s;\rho}\ \text{(E-PlacedVal)}$$

(b) Placement terms.

$$\frac{t' = \varphi(P_0, P_1, \mathcal{I}^{[P_1]}, v, T)}{\vartheta\colon P_0 \rhd \mathsf{asLocal}\ v\colon T\ \mathsf{on}\ P_1;\rho \xrightarrow{\vartheta} t';\rho}\ \text{(E-AsLocal)}$$

$$\frac{t' = \zeta(P_1, \vartheta', v, T)}{\vartheta\colon P_0 \rhd \mathsf{asLocal}\ v\colon T\ \mathsf{on}\ P_1\ \mathsf{from}\ \vartheta';\rho \xrightarrow{\vartheta} t';\rho}\ \text{(E-AsLocalFrom)}$$

$$\frac{t' = \zeta(P_0, \vartheta, v, T_0)}{\vartheta\colon P_0 \rhd \mathsf{asLocal\ block}\ x\colon T_0 = v\ \mathsf{in}\ t : T_1\ \mathsf{on}\ P_1;\rho \xrightarrow{\mathcal{I}^{[P_1]}} \mathsf{asLocal}\ [x \mapsto t']t : T_1\ \mathsf{on}\ P_1;\rho}\ \text{(E-Block)}$$

$$\frac{t' = \zeta(P_0, \vartheta, v, T)}{\vartheta\colon P_0 \rhd \mathsf{asLocal\ block}\ x\colon T = v\ \mathsf{in}\ t : S\ \mathsf{from}\ \vartheta';\rho \xrightarrow{\vartheta'} \mathsf{asLocal}\ [x \mapsto t']t : S\ \mathsf{from}\ \vartheta';\rho}\ \text{(E-BlockFrom)}$$

$$\frac{\mathcal{I}^{[P_1]}\colon P_1 \rhd t;\rho \xrightarrow{\vartheta'} t';\rho'}{\vartheta\colon P_0 \rhd \mathsf{asLocal}\ t : T\ \mathsf{on}\ P_1;\rho \xrightarrow{\vartheta'} \mathsf{asLocal}\ t' : T\ \mathsf{on}\ P_1;\rho'}\ \text{(E-Remote)}$$

$$\frac{\vartheta''\colon P_1 \rhd t;\rho \xrightarrow{\vartheta'} t';\rho'}{\vartheta\colon P_0 \rhd \mathsf{asLocal}\ t : T\ \mathsf{on}\ P_1\ \mathsf{from}\ \vartheta'';\rho \xrightarrow{\vartheta'} \mathsf{asLocal}\ t' : T\ \mathsf{on}\ P_1\ \mathsf{from}\ \vartheta'';\rho'}\ \text{(E-RemoteFrom)}$$

(c) Remote access terms.

$$\frac{r \notin \mathrm{dom}(\rho)}{\vartheta\colon P_0 \rhd \mathsf{var}\ v;\rho \xrightarrow{\vartheta} r;(\rho, r \mapsto v)}\ \text{(E-SourceVar)} \qquad \frac{r \notin \mathrm{dom}(\rho)}{\vartheta\colon P_0 \rhd \mathsf{signal}\ t;\rho \xrightarrow{\vartheta} r;(\rho, r \mapsto t)}\ \text{(E-Signal)}$$

$$\frac{}{\vartheta\colon P_0 \rhd \mathsf{set}\ r := v;\rho \xrightarrow{\vartheta} \mathsf{unit};[r \mapsto v]\rho}\ \text{(E-Set)} \qquad \frac{t = \rho(r)}{\vartheta\colon P_0 \rhd \mathsf{now}\ r;\rho \xrightarrow{\vartheta} t;\rho}\ \text{(E-Now)}$$

(d) Reactive terms.

Fig. 3. Operational semantics.

peer instances via E-AsLocal. E-Remote takes a step in a remote block on the peer instances $\mathcal{I}^{[P_1]}$ of type P_1. Similarly, E-BlockFrom and E-RemoteFrom evaluate remote blocks on a single remote instance p.

The rules for reactive terms t in Figure 3d step on the peer instances ϑ where t is placed and model a pull-based reactive system, where reactives are given semantics as store locations in ρ that contain values v for Vars and thunks t for Signals. The pull-based scheme recomputes the value of a signal r and the signals on which r depends upon each access to r. Designing new propagation systems, e.g., push [Maier et al. 2010] push-pull [Elliott 2009], memory-bounded [Krishnaswami et al. 2012], glitch-free [Drechsler et al. 2014], fair [Cave et al. 2014], is ongoing research. We leave extending our model with such approaches for future work.

6.3 Static Semantics

The type system guarantees that cross-peer access is safe and consistent with the architecture defined through ties $\mathcal{T}$. It rejects programs where remote values are mixed with local values without converting them via asLocal or where a remote value is accessed on a peer that is not tied.

$$\frac{x{:}\,T \in \Gamma \quad \vee \quad x{:}\,T \text{ on } P \in \Delta}{\Psi; \Delta; \Gamma; P \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Psi; \Delta; \Gamma, x{:}\,T_1; P \vdash t : T_2}{\Psi; \Delta; \Gamma; P \vdash \lambda x{:}\,T_1.\, t : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{\Psi; \Delta; \Gamma; P \vdash t_1 : T_2 \to T_1 \qquad \Psi; \Delta; \Gamma; P \vdash t_2 : T_2}{\Psi; \Delta; \Gamma; P \vdash t_1\, t_2 : T_1} \quad \text{(T-App)}$$

(a) Standard terms.

$$\frac{\Psi; \Delta, x{:}\,T \text{ on } P \vdash s \qquad \Psi; \Delta; \varnothing; P \vdash t : T}{\Psi; \Delta \vdash \text{placed } x{:}\,T \text{ on } P = t \text{ in } s} \quad \text{(T-Place)}$$

$$\frac{}{\Psi; \Delta \vdash \text{end}} \quad \text{(T-End)}$$

(b) Placement terms.

$$\frac{\vartheta \subseteq \mathcal{I}^{[P_1]}}{\Psi; \Delta; \Gamma; P_0 \vdash \vartheta : \text{Remote } P_1} \quad \text{(T-Peer)}$$

$$\frac{\Psi; \Delta; \varnothing; P_1 \vdash t : U \qquad P_0 \leftrightarrow P_1 \qquad T = \Phi(P_0, P_1, U)}{\Psi; \Delta; \Gamma; P_0 \vdash \text{asLocal } t{:}\,U \text{ on } P_1 : T} \quad \text{(T-AsLocal)}$$

$$\frac{\Psi; \Delta; \varnothing; P_1 \vdash t_0 : U \qquad P_0 \leftrightarrow P_1 \qquad \Psi; \Delta; \Gamma; P_0 \vdash t_1 : \text{Remote } P_1}{\Psi; \Delta; \Gamma; P_0 \vdash \text{asLocal } t_0{:}\,U \text{ on } P_1 \text{ from } t_1 : U} \quad \text{(T-AsLocalFrom)}$$

$$\frac{\Psi; \Delta; \Gamma; P_0 \vdash t_0 : U_0 \qquad \Psi; \Delta; x{:}\,U_0; P_1 \vdash t_1 : U_1 \qquad P_0 \leftrightarrow P_1 \qquad T = \Phi(P_0, P_1, U_1)}{\Psi; \Delta; \Gamma; P_0 \vdash \text{asLocal block } x = t_0{:}\,U_0 \text{ in } t_1 : U_1 \text{ on } P_1 : T} \quad \text{(T-Block)}$$

$$\frac{\Psi; \Delta; \Gamma; P_0 \vdash t_0 : U_0 \qquad \Psi; \Delta; x{:}\,U_0; P_1 \vdash t_1 : U_1 \qquad P_0 \leftrightarrow P_1 \qquad \Psi; \Delta; \Gamma; P_0 \vdash t_2 : \text{Remote } P_1}{\Psi; \Delta; \Gamma; P_0 \vdash \text{asLocal block } x = t_0{:}\,U_0 \text{ in } t_1 : U_1 \text{ on } P_1 \text{ from } t_2 : U_1} \quad \text{(T-BlockFrom)}$$

(c) Remote access terms.

$$\frac{T_0 \text{ on } P = \Psi(r) \qquad T_0 = \text{Signal } T_1 \ \vee \ T_0 = \text{Var } T_1}{\Psi; \Delta; \Gamma; P \vdash r : T_0} \quad \text{(T-Reactive)}$$

$$\frac{\Psi; \Delta; \Gamma; P \vdash t : T}{\Psi; \Delta; \Gamma; P \vdash \text{signal } t : \text{Signal } T} \quad \text{(T-Signal)}$$

$$\frac{\Psi; \Delta; \Gamma; P \vdash t : T}{\Psi; \Delta; \Gamma; P \vdash \text{var } t : \text{Var } T} \quad \text{(T-SourceVar)}$$

$$\frac{\Psi; \Delta; \Gamma; P \vdash t : T_0 \qquad T_0 = \text{Signal } T_1 \ \vee \ T_0 = \text{Var } T_1}{\Psi; \Delta; \Gamma; P \vdash \text{now } t : T_1} \quad \text{(T-Now)}$$

$$\frac{\Psi; \Delta; \Gamma; P \vdash t_1 : \text{Var } T \qquad \Psi; \Delta; \Gamma; P \vdash t_2 : T}{\Psi; \Delta; \Gamma; P \vdash \text{set } t_1 := t_2 : \text{Unit}} \quad \text{(T-Set)}$$

(d) Reactive terms.

Fig. 4. Typing rules.

The typing judgment $\Psi; \Delta \vdash s$ states that s is well-typed under Ψ and Δ. The typing judgment $\Psi; \Delta; \Gamma; P \vdash t : T$ for terms t says that t is well-typed under Ψ, Δ and Γ in the peer context P, i.e., the peer on which term t is placed. $\Gamma ::= \varnothing \mid \Gamma, x{:}\,T$ is the typing environment for variables, $\Delta ::= \varnothing \mid \Delta, x{:}\,S$ is the typing environment for placed variables. We require that the name x is distinct from the variables bound by both Γ and Δ, which can always be achieved by α-conversion. Ψ is the typing environment for reactives. It ranges over mappings from reactives to placement types S. The values held by a reactive always have the same type, which is fixed at creation time of the reactive. The typing rules are in Figure 4.

The typing rules for terms t in Figure 4a are standard except for T-Var where the type for x is looked up in both Γ and Δ. When in the peer context P, the local x – a locally scoped variable in Γ or a value placed on P in Δ – is accessed simply through x. We omit the standard rules T-Unit, T-Cons, T-Nil, T-Some and T-None.

The typing rules for placement terms s are in Figure 4b. T-Place types the term t of type T on P in the peer context P and extends the environment for placed variables Δ with the type of x. Placed values are modeled as a series of nested s-terms ending in end typed by T-End.

The typing rules for remote access terms t are in Figure 4c. T-Peer types peer instances ϑ of peer type defined by P_1. T-AsLocal types remote access to a term t placed on peer P_1 in the peer context P_0. The rule ensures that the type ascription U on P_1 is correct for the placed term t by deriving type U for t in the peer context P_1. The rule ensures that P_0 is tied to P_1. Remote access aggregates over remote values (cf. Section 2.4.1) and the type of the aggregation is defined by Φ

(Figure 2). Similarly, T-AsLocalFrom types remote access to the remote instances given by t_1 of type Remote P_1 (cf. Section 2.4.2). The rule ensures that t_1 refers to remote peer instances of peer type P_1, where t_0 of type U is placed. T-Block types the application of t_0 to the remote block t_1 on peer P_1. The term t_0 is typed in the context of the local peer P_0. The block t_1 is typed in the context of the remote peer P_1. The environment Γ for typing t_1 consists only of the typing for x to prevent the block from implicitly closing over remote values, i.e., variables must be passed explicitly (cf. Section 2.5). Similarly, T-BlockFrom types a remote bock on a single remote instance.

The typing rules for reactive terms t are in Figure 4d. T-Reactive types a reactive r of the type defined by Ψ. T-Signal types signal expressions and T-SourceVar types Var instantiations. T-Set requires that the term t_1 to be set to a new value is a Var. T-Now requires that the term t to be read is either a Var or a Signal.

6.4 Type Soundness

Proving type soundness requires some auxiliary definitions and lemmas. First, we show that a transmitted remote value as modeled by ζ (Figure 2) can be typed on the local peer:

Lemma 1 (Transmission). *If $P_0 \leftrightarrow P_1$ for two peers $P_0, P_1 \in \mathcal{P}$ and $\Psi;\Delta;\Gamma;P_1 \vdash v : U$ and $t = \zeta(P_1, \vartheta, v, U)$ for some $\vartheta \in \mathcal{I}^{[P_1]}$, then $\Psi;\Delta';\Gamma';P_0 \vdash t : U$ for any Δ' and Γ'.*

Proof. By induction on v. □

Second, we show that aggregation modeled by φ yields the type given by Φ:

Lemma 2 (Aggregation). *If $P_0 \leftrightarrow P_1$ for two peers $P_0, P_1 \in \mathcal{P}$ and $\Psi;\Delta;\Gamma;P_1 \vdash v : U$ and $t = \varphi(P_0, P_1, \vartheta, v, U)$ and $T = \Phi(P_0, P_1, U)$ for some $\vartheta \in \mathcal{I}^{[P_1]}$, then $\Psi;\Delta';\Gamma';P_0 \vdash t : T$ for any Δ' and Γ'.*

Proof. By induction on ϑ in the case $P_0 \overset{*}{\rightarrowtail} P_1$ and the transmission lemma. □

Next, we provide a definition of typability for a reactive system ρ. We denote with refs(ρ) the set of all reactive references allocated by ρ:

Definition 2. A reactive system ρ is well-typed with respect to typing contexts Δ, Γ and a reactive typing Ψ, written $\Psi;\Delta;\Gamma \vdash \rho$, iff $\text{refs}(\rho) = \text{dom}(\Psi)$ and $\Psi;\Delta;\Gamma;P \vdash \rho(r) : T$ with $\Psi(r) = \text{Var}\,T$ on P or $\Psi(r) = \text{Signal}\,T$ on P for every $r \in \text{refs}(\rho)$.

We prove type soundness based on the usual notion of progress and preservation [Wright and Felleisen 1994], meaning that well-typed programs do not get stuck during evaluation. We first formulate progress and preservation for terms t:

Theorem 1 (Progress on t-terms). *Suppose t is a closed, well-typed term (that is, $\Psi;\varnothing;\varnothing;P \vdash t : T$ for some T, P and Ψ). Then either t is a value or else, for any ϑ and any reactive system ρ such that $\Psi;\varnothing;\varnothing \vdash \rho$, there is some term t', some ϑ' and some reactive system ρ' with $\vartheta{:}\,P \rhd t;\rho \xrightarrow{\vartheta'} t';\rho'$.*

Proof. By induction on the typing derivation. Case T-AsLocal steps with E-Remote or E-AsLocal, T-AsLocalFrom steps with E-Context, E-RemoteFrom or E-AsLocalFrom, T-Block steps with E-Context or E-Block, T-BlockFrom steps with E-Context or E-BlockFrom, T-Signal steps with E-Signal, T-SourceVar steps with E-Context or E-SourceVar, T-Now steps with E-Context or E-Now and T-Set steps with E-Context or E-Set. □

Theorem 2 (Preservation on t-terms). *If $\Psi;\Delta;\Gamma;P \vdash t : T$ and $\Psi;\Delta;\Gamma \vdash \rho$ and $\vartheta{:}\,P \rhd t;\rho \xrightarrow{\vartheta'} t';\rho'$ with $\vartheta \in \mathcal{I}^{[P]}$, then $\Psi';\Delta;\Gamma;P \vdash t' : T$ and $\Psi';\Delta;\Gamma \vdash \rho'$ for some $\Psi' \supseteq \Psi$.*

Proof. By induction on the typing derivation using the aggregation lemma in the case T-AsLocal, the Var allocation property in the case T-SourceVar, the signal allocation property in the case T-Signal, the retrieval property in the case T-Now and the update property in the case T-Set. □

Based on progress and preservation for terms t, we prove type soundness for whole programs s:

Theorem 3 (Progress on s-terms). *Suppose s is a closed, well-typed term (that is, $\Psi; \varnothing \vdash s$ for some Ψ). Then either s is a value or else, for any reactive system ρ such that $\Psi; \varnothing; \varnothing \vdash \rho$, there is some term s', some ϑ and some reactive system ρ' with $s; \rho \xrightarrow{\vartheta}\!\!\!\!\rightarrow s'; \rho'$.*

Proof. By case analysis on the typing derivation using the progress theorem for t-terms. □

Theorem 4 (Preservation on s-terms). *If $\Psi; \Delta \vdash s$ and $\Psi; \Delta; \varnothing \vdash \rho$ and $s; \rho \xrightarrow{\vartheta}\!\!\!\!\rightarrow s'; \rho'$, then $\Psi'; \Delta \vdash s'$ and $\Psi'; \Delta; \varnothing \vdash \rho'$ for some $\Psi' \supseteq \Psi$.*

Proof. By case analysis on the typing derivation using the preservation theorem for t-terms. □

6.5 Placement Soundness

We prove placement soundness for the core calculus. We show that we can statically reason about the peer on which code is executed, i.e., that the peer context P in which a term t is type-checked matches the type P of the peer instances ϑ on which t is evaluated. The type system is sound for a placement if the code placed on a peer P is executed on peer instances of peer type P:

Theorem 5 (Static Placement on t-terms). *If $\Psi; \Delta; \Gamma; P \vdash t : T$ and $\vartheta : P \triangleright t; \rho \xrightarrow{\vartheta'} t'; \rho'$ with $\vartheta \subseteq \mathcal{I}^{[P]}$, then for every subterm t_i with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ and $\vartheta_i : P'_i \triangleright t_i; \rho_i \xrightarrow{\vartheta''} t'_i; \rho'_i$ holds $\vartheta_i \subseteq \mathcal{I}^{[P'_i]}$ and $P_i = P'_i$.*

Proof. By case analysis on the typing derivation for terms t. □

Theorem 6 (Static Placement on s-terms). *If $\Psi; \Delta \vdash s$ and $s; \rho \xrightarrow{\vartheta}\!\!\!\!\rightarrow s'; \rho'$, then for every subterm t_i with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ and $\vartheta_i : P'_i \triangleright t_i; \rho_i \xrightarrow{\vartheta''} t'_i; \rho'_i$ holds $\vartheta_i \subseteq \mathcal{I}^{[P'_i]}$ and $P_i = P'_i$.*

Proof. By case analysis on the typing derivation for terms s. □

Further, we prove that remote access is explicit, i.e., it is not possible to compose expressions placed on different peers without explicitly using asLocal:

Theorem 7 (Explicit Remote Access). *If $\Psi; \Delta; \Gamma; P \vdash t : T$, then every subterm t_i of t with $\Psi_i; \Delta_i; \Gamma_i; P_i \vdash t_i : T_i$ is either an explicitly accessed remote term (that is, t_r in one of* asLocal $t_r : S$, asLocal $t_r : S$ from t_f, asLocal block $x : T = t_x$ in $t_r : S$ *or* asLocal block $x : T = t_x$ in t_r from $t_f : S$*) or $P = P_i$.*

Proof. By case analysis on the typing derivation of terms t. □

7 IMPLEMENTATION

The implementation of ScalaLoci primarily entails (1) the type-level encoding of placement types and ties into Scala to type-check remote access and statically determine the type of a remote access based on the ties between peers, and (2) the macro-driven code separation. Overall ScalaLoci amounts to ~ 7 K LOC in Scala plus ~ 2 K LOC to integrate different network protocols and reactive systems. An interesting challenge is the encoding of placement types into the Scala type system with the goal of preserving compatibility with plain Scala, i.e., every ScalaLoci program is a syntactically correct and well-typed Scala program. Thanks to our approach based on macro expansion, existing libraries and tooling can be seamlessly used with ScalaLoci and we impose no restrictions on the Scala code a developer can place on any peer.

Type Checking. In our approach, we first use the standard Scala type checker to enforce correct placement and remote access. Placement types can even be inferred in cases where the standard Scala type inference applies. We encode the ties between peers at the type level in Scala (i.e., `type Tie = ...`). We (i) check that remote access using `asLocal` is permitted between peers and

(ii) statically select an aggregation scheme for remote access based on the tie (Section 2.4.1) using Scala's type-level programming, which builds on implicit value resolution.

Further, we ensure statically that there is a serialization method for the type of every value that is transmitted remotely. We use the *Concept* pattern, an encoding of type classes using Scala implicits [Oliveira et al. 2010]: The compiler checks that a `Serializable[T]` type class instance is available for every type *T* that needs to be serializable.

Code Separation into Tiers. After type checking, we use Scala annotation macros [Burmako 2013] (i.e., the enclosing class, trait or object is annotated with `@multitier`) to separate the type-checked program into peer-specific parts and replace remote accesses with calls into the underlying communication backend, auto-generating the communication boilerplate. The macro expansion gets the untyped abstract syntax tree (AST) of the Scala code to be expanded as input. After invoking the Scala type checker, we generate code for each peer by transforming the type-checked AST. The only correctness property which cannot be checked by the Scala type system and which is performed during macro expansion is checking that remote blocks do not capture values implicitly (i.e., they need to be listed explicitly in the `capture` clause, cf. Section 2.5). For this property, we perform static analysis on the AST. The macro aborts compilation and produces an appropriate compiler error if implicit captures are found.

The macro expansion generates a new class for every peer, which contains the abstractions which are placed on the respective peer given by their placement types, thereby effectively erasing placement types from the generated code. ScalaLoci supports both the compilation to Java bytecode and to JavaScript via Scala.js [Doeraene 2013].

Context Propagation and Compiler Plugin. To carry the current *peer context*, i.e., the peer to which an expression belongs, `placed` and `remote` code blocks implicitly define a value in their scope, which is used to lookup the peer context from the implicit scope. A placed expression can be written as a function `placed[P] { implicit! => e}`, where the implicit argument – "`!`" here – is not directly used by the programmer. The use of implicit arguments as a means to propagate context is a known pattern adopted by many Scala frameworks, e.g., Play [2012], ScalaSTM [2014] or Slick [2014]. Scala extensions to support a syntactically light solution have already been discussed on the scala-debate mailing list [Li 2014] and are available in the Dotty compiler as *implicit function types* [Odersky et al. 2017]. For ScalaLoci, we developed a compiler plugin to omit the implicit argument when declaring the function, e.g., the developer can simply write `placed[P] { e }`. The plugin is not specific to ScalaLoci and, for compatibility with plain Scala, it is not mandatory for using our system.

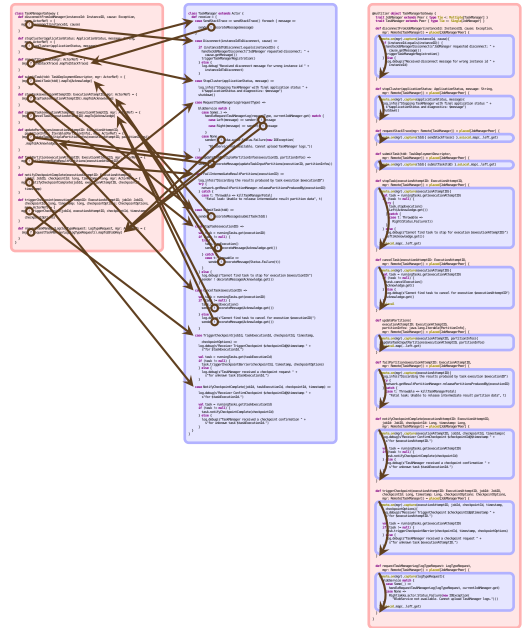
8 EVALUATION

The main hypothesis of ScalaLoci's design is that its multitier reactive abstractions reduce the complexity of implementing a distributed system at negligible cost. We validate this hypothesis with open-source case studies and side-by-side comparisons of alternative designs of applications belonging to different domains (e.g., big data processing, real-time streaming, games, collaborative editing, instant messaging), covering both the compilation of Scala to Java bytecode and to JavaScript via Scala.js. We conduct performance benchmarks applying ScalaLoci in a real-world setting and microbenchmarks to isolate the performance impact of the provided abstractions.

8.1 Case Studies

To evaluate the applicability of ScalaLoci to existing real-word software, we ported several open source applications. Our ports are *not* simplified versions. We reimplemented components of the existing software in ScalaLoci to achieve a functionally equivalent system.

Apache Flink. We reimplemented the *task distribution system* of the Apache Flink stream processing framework [Carbone et al. 2015] in ScalaLoci, which provides Flink's core task scheduling and deployment logic. The task distribution system is based on Akka actors and consists of 23 remote procedures in six *gateways* (an API that encapsulates sending actor messages into asynchronous RPCs) amounting to ~ 500 highly complex Scala LOC. 19 out of the 23 RPCs are processed in a different compilation unit within another package, impeding to correlate sent messages with the remote computations they trigger. The ScalaLoci version replaces the sending and receiving operations between actors – used in the gateway implementation to implement RPCs – with remote blocks as high-level communication abstraction. Cross-peer control and data flow is explicit, thus much easier to track.



(a) Original Flink. (b) ScalaLoci.

Fig. 5. Communication for two actors in Flink.

Figure 5 provides an overview of the communication between the JobManager and a TaskManager. Figure 5a shows the code of the TaskManagerGateway (red box on the left) used by the JobManager actor and its communication (arrows) with the TaskManager actor (blue box on the right). Figure 5b shows the ScalaLoci implementation of the JobManager peer (red) and the TaskManager peer (blue). Flink's actor-based approach intertwines data flow between components with send and receive operations. Overall, the data flow is hard to track (Figure 5a). Yet, data flow between both components is quite *regular*, with the JobManager triggering remote computations on the TaskManager and the TaskManager returning a result. This *regularity* is directly captured using ScalaLoci's remote blocks (Figure 5b).

Flink communication is unsafe, with actor messages having type `Any` requiring downcasting or non-exhaustive pattern matching. Crucially, the compiler cannot enforce that a message is even handled to produce a remote result. In the ScalaLoci version, we were able to eliminate 23 unsafe pattern matches and 8 type casts.

Apache Gearpump. Apache Gearpump [Zhong et al. 2014] is a real-time streaming engine. In Gearpump, *Master* actors allocate processing tasks to *Worker* actors and collect results. A *MasterProxy* actor assigns Workers to Masters. We ported the assignment logic (~ 100 LOC) to ScalaLoci, replacing actor communication with multitier reactive abstractions. The MasterProxy message loop mixes largely unrelated tasks, e.g., assigning Workers to a Master and monitoring the Master for termination, which are captured by separated multitier streams in the ScalaLoci version. We also removed imperative state changes on the Master – managing the list of currently connected Workers – and on the Worker – keeping track of the currently connected Master. In ScalaLoci, the list of connected remote instances is automatically handled by the runtime. Finally, the constraint that a Master can connect to multiple Workers and a Worker can connect to at most one Master is enforced at compile time in the ScalaLoci version with in-language architecture definitions.

Play Scala.js Application. We ported the *Play Framework with Scala.js Showcase* [Puripunpinyo 2014], an open source demonstrator of Scala.js combined with the Play Framework [2007], to

ScalaLoci. It implements several components amounting to ~ 1 500 LOC including instant messaging and a reactive Todo list [Li 2013] based on TodoMVC [2011]. The ScalaLoci version is feature-equivalent to the original one, except for the Todo list where the updates made by a client are automatically propagated to all other clients using ScalaLoci's multitier reactives. In contrast, the original version requires to reload the page to propagate changes. We were able to reuse 70 % LOC just by combining highly coupled code for handling client–server interaction into a multitier object and adding placement annotations. Send and receive operations for transferring values are implemented inside different modules in the original baseline. Communication on the server side is handled by a *controller*, which defines *actions*, i.e., a callback mechanism to handle HTTP client requests and create a response or setup communication channels over WebSocket. Clients send requests using Ajax and handle responses using callbacks. With ScalaLoci, all communication code is automatically generated. The communication boilerplate (e.g., sending messages with the WebSocket API, callbacks, or Play's message serialization to JSON) is reduced by 63 %. The only boilerplate code left is needed for integration with the *Play* framework.

Table 1. Code metrics.

LOC Lines of Code **CB** Callbacks **ISU** Imperative State Updates
CRC Cross-host Composition **RA** Remote Access/Reads/Writes

Case Study Communication / Computation	**LOC**	**CB**	**ISU**	**CRC**	**RA**
Pong					
(local) / observer	355	17	10	0	0
(local) / reactive	327	8	0	0	0
RMI / observer	460	24	14	0	5
RMI / reactive	431	13	6	0	5
Akka / observer	440	24	13	0	5
Akka / reactive	413	18	5	0	5
ScalaLoci / observer	426	22	13	0	7
ScalaLoci / reactive	369	8	0	4	4
Shapes					
WebSocket / observer (JS) [a]	483	11	10	0	9
WebSocket / observer [b]	474	11	10	0	7
WebSocket / reactive [b]	462	9	5	0	3
Akka / observer [b]	478	13	9	0	7
Akka / reactive [b]	424	11	4	0	3
ScalaLoci / observer [b]	350	9	7	2	9
ScalaLoci / reactive [b]	345	2	0	6	6
P2P Chat					
WebRTC / observer (JS) [a]	776	22	25	0	7
WebRTC / observer [b]	824	24	25	0	7
WebRTC / reactive [b]	820	14	10	0	7
Akka / observer [b]	772	25	24	0	7
Akka / reactive [b]	771	15	9	0	7
ScalaLoci / observer [b]	637	21	19	4	8
ScalaLoci / reactive [b]	593	4	4	7	7

[a] Uses handwritten JavaScript for the client-side, Scala for the server side.
[b] All code is in Scala or ScalaLoci. The client is compiled to JavaScript via Scala.js.

8.2 Variants Analysis

To evaluate the design of the applications that use ScalaLoci, we compare different variants of the same software. We reimplemented every variant from scratch to provide a fair comparison when exchanging different aspects of the implementation, i.e., the communication mechanism and the event processing strategy. For each of the 22 variants (each line in Table 1) we report both aspects. For example, *Akka / observer* (Line 5) adopts Akka actors for the communication but no RP for local event propagation. The *local* variant is a non-distributed, purely local baseline. The *ScalaLoci* variants use multitier abstractions. All other variants use manually written communication code between distributed components, i.e., Akka actors, Java Remote Method Invocation (RMI) for JVM applications or native web browser APIs (WebSocket or WebRTC) for JS applications. The variants marked with *JS* use handcrafted JavaScript for the browser side. The other variants are compiled from Scala.

Pong implements the arcade *Pong* game. We extended a local implementation (the user plays against the computer) to a distributed multiplayer implementation. The latter adopts a client–server model. Both the server and the clients run on the JVM. **Shapes** is a collaborative drawing web application, where clients connect to a central server. **P2P Chat** is the P2P web chat application introduced in the initial running example, which supports multiple one-to-one chat sessions. Peers communicate directly in a P2P fashion after discovery via a registry. In the latter two cases, the server and the registry run on the JVM while clients and peers run in the web browser.

Programming Experience. To give an intuition of the experience of using ScalaLoci, Figure 6 shows a side-by-side comparison of the four different reactive *Pong* variants. We exclude the GUI from the code excerpts and from the following discussion as it is the same for all variants. We highlight the additional code needed for the distributed versions in Figure 6b, 6c and 6d compared to the local baseline in Figure 6a. We use yellow for code added to enable the multiplayer game. Orange indicates code added for distribution.

(a) Local. (b) ScalaLoci. (c) Akka. (d) RMI.

Fig. 6. Pong variants.

Transforming the local variant of *Pong* into a distributed application is straightforward. In the ScalaLoci variant (Figure 6b), 8 LOC, which implement multiplayer, are the majority of the additions. As much as 91 % LOC (excluding the GUI) could be reused from the local version just by adding placement to assign values to either the client or the server. The reused code amounts to 72 % LOC of the ScalaLoci version. In the Akka and RMI versions (Figure 6c and 6d), we could also reuse 91 % LOC of the local baseline, which is 53 % LOC of the Akka version and 46 % LOC for RMI. The explanation is that the Akka and RMI versions lack multitier reactives, thus data flows cannot be directly defined across client and server boundaries. Instead, message-passing and callbacks or imperative remote calls are needed to explicitly propagate changes across hosts – which constitutes ~ 60 % of the added code (orange) – tangling the application logic with connection management and data propagation.

Design – Code Metrics and Safety. Table 1 compares code metrics for all variants (rows). Not surprisingly, ScalaLoci requires less code. Multitier reactives further reduce the code size and the amount of callbacks (*Callbacks* column). We use the number of callbacks as a measure of design quality, since replacing callbacks – which are not composable – by composable reactives removes manual imperative state updates (*Imperative State Updates* column) and improves extensibility [Meyerovich et al. 2009]. Reactives exhibit better composability properties compared to callbacks due to the inversion of control problem [Maier et al. 2010]. To completely eliminate callbacks, in RP, libraries need to support reactive abstractions, e.g., a text input widget in a graphic library should expose a signal holding the current text. Since most libraries adopt callbacks at their boundaries, some callbacks are still necessary even if the application logic is based on RP.

In ScalaLoci, data flows on different hosts can be seamlessly composed (*Cross-host Composition* column), which is not possible in approaches where data flow across hosts is interrupted, e.g., by RPC or message sending. The applications introduced in Section 3 compose data flows across hosts. For instance, the token ring example (Listing 5, Line 17) composes the local events `recv` and `sendToken` and the remote event `sent` seamlessly inside the single expression `(sent.asLocal \ recv) || sendToken`.

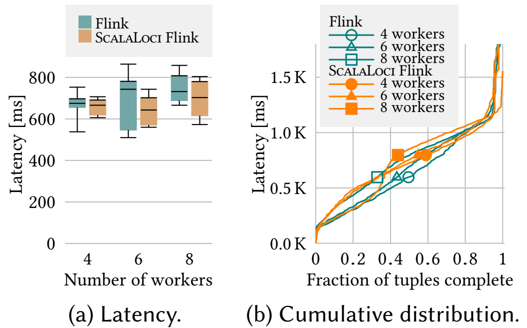
Statically typed access to remote values in ScalaLoci ensures that they are serializable, opposed to Java RMI serialization, which can fail at run time. Such remote accesses (*Remote Access* column) involve potentially problematic serialization in non-ScalaLoci cases. For example, the variants using manually written communication code explicitly invoke methods to convert between in-memory and serialized representation, e.g., `JSON.stringify(v)` and `JSON.parse(v)` in JavaScript code or `write(v)` and `read[T](v)` in Scala code. Also, remote access is achieved with explicit sending and receiving operations. Since the compiler cannot enforce that the types on both sides match, it cannot prevent that values are manipulated inconsistently, resulting in run time errors. In ScalaLoci, instead, a shared value is accessed locally as `v` and remotely as `v.asLocal`. Hence the compiler can statically check that `v`'s type is consistent for local and remote accesses.

These results suggest that there exist quantifiable differences in design regarding the number of used imperative callbacks compared to reactive abstractions and regarding abstractions for composing local and remote values, which we interpret as ScalaLoci enabling improvement in software design, fostering more concise and composable code, and reducing the number of callbacks and imperative state updates. Also, applications in ScalaLoci are safer than their counterparts, e.g., due to reduced risk of run time type mismatches thanks to static type-checking across peers.

8.3 Performance

We evaluate ScalaLoci performance in two ways. First, we consider the impact of ScalaLoci on a real-world system running in the cloud compared to the original implementation with Akka actors. Second, we compare remote change propagation cost in ScalaLoci among common alternatives.

System Benchmarks. We measure the performance of Apache Flink compared to the ScalaLoci reimplementation presented in Section 8.1. Both systems are functionally equivalent and only differ for the language abstractions covered by ScalaLoci (i.e., architecture definition, placement and remote communication). We use the Yahoo Streaming Benchmark [Chintapalli et al. 2016] on the Amazon EC2 Cloud (2,3 GHz Intel Xeon E5-2686, 8 GiB instances) with 8 servers for data generation, one server as sink, one Apache Zookeeper server for orchestration, one JobManager master server, and 4 to 8 TaskManager worker nodes. Events are marked with timestamps at generation time. The query groups events into 10 s windows. Each data source generates 20 K events/s. The Yahoo Streaming Benchmark measures latency. Figure 7a shows latency average and variance between the two versions. Figure 7b shows the empirical cumulative distribution: The latency for completing the processing of a given fraction of events. We consider the difference between the two versions negligible (in some cases, ScalaLoci is even faster) and only due to variability in system measurements. In conclusion, at the system level, there is no observable performance penalty when using ScalaLoci.

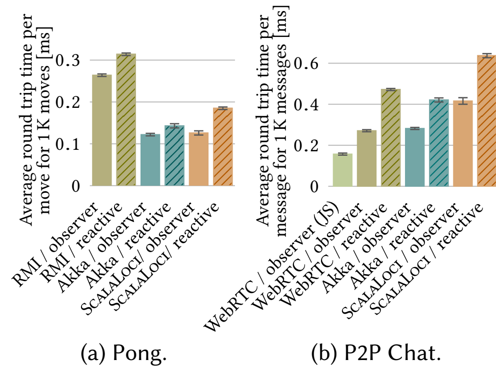


(a) Latency. (b) Cumulative distribution.

Fig. 7. System Benchmarks.

Microbenchmarks. Our microbenchmarks are based on the *Pong* and *P2P Chat* applications from Section 8.2. *Pong* runs on the Java Virtual Machine (JVM) and *P2P Chat* runs on a JavaScript engine in a web browser. In both settings, we microbenchmark the performance impact of ScalaLoci's

abstractions. We do not microbenchmark different virtual machines (i.e., running the same application on a JavaScript and a Java virtual machine) because the differences would be due to the execution environment rather than to our implementation. To specifically only measure ScalaLoci's overhead, all peers run on the same machine. Remote communication is piped through the network stack locally and its latency is negligible. The setup is an Intel Core i7-5600U, 2.6–3.2 GHz, 8 GiB. The benchmarks initiate a value change which propagates to a remote peer where it triggers a computation. The result is propagated back to the initiating peer. We measure the average time over 1 K round trips. Figure 8 shows the mean over 200 iterations and 99 % confidence intervals.



(a) Pong. (b) P2P Chat.

Fig. 8. Microbenchmarks.

For *Pong*, ScalaLoci even outperforms the RMI variants (Figure 8a). We attribute this result to RMI's blocking remote call approach causing the application to block upon sending a changed value before propagating the next change. The performance of the ScalaLoci variants is comparable to the Akka variants. Using our reactive runtime system incurs a small overhead of ~ 0.02–0.05 ms as compared to handcrafted value propagation for both the RMI and ScalaLoci variants.

For the *P2P Chat* benchmark (Figure 8b), the observer variant with handwritten JavaScript for the client is the fastest. Akka, ScalaLoci, and the compilation to JavaScript show an overhead partially due to the Scala.js translation. Akka.js on the client-side has a comparable performance to plain Scala.js. This is mostly because remote messages in Akka.js are sent using the same WebRTC browser API as in the *WebRTC* cases, resulting in the same amount of overhead. The overhead amounts to ~ 0.1 ms for using Scala.js (bar labeled *WebRTC / observer (JS)* compared to *WebRTC / observer* in the graph), ~ 0.15 ms for ScalaLoci compiled to JavaScript (bars labeled *ScalaLoci* compared to *WebRTC* in the graph) and ~ 0.15–0.35 ms for the reactive runtime compiled to JavaScript (bars labeled *reactive* compared to *observer* in the graph) as reactive framework. All measurements are under 1 ms – distinctly lower than network latency for Internet applications, which is in line with the results of the system benchmarks where the overhead of ScalaLoci is negligible for distributed applications.

9 RELATED WORK

First, we discuss research that influenced our work in terms of techniques. Then we consider related work on *multitier languages*, *FRP languages* and languages that combine both.

Unifying architectural definition and implementation in one language has been pioneered by ArchJava [Aldrich et al. 2002]. ScalaLoci peers and ties are similar to ArchJava components and connections. Subsequent work [Aldrich et al. 2003] allows setting up ArchJava connections over a network and hooking into the type-checking process to give additional type safety guarantees (e.g., that values implement the Java `Serializable` interface). ArchJava, however, does not address the composition of values with different placement in the same compilation unit using placement types nor supports abstractions for data flows spanning over multiple distributed components. Partitioned Global Address Space Languages (PGAS) such as X10 [De Wael et al. 2015] provide a programming model for high-performance parallel execution. PGAS languages define a globally shared address space aiming at a goal similar to multitier languages – reduce boundaries among hosts. The scope of HPC, however, is very diverse – we focus on generic distributed systems based

on reactive communication abstractions. Chandra et al. [2008] extend X10 with dependent *placed types*, one for each processing location. In contrast, ScalaLoci abstracts over peer instances of the same type, e.g., to refer uniformly to all peers in a P2P architecture. The actor model, introduced by Hewitt et al. [1973], encapsulates state into concurrent units that exchange asynchronous messages. The resulting decoupling enables scalability and fault tolerance. ScalaLoci adheres to such properties and provides developers with higher level abstractions, i.e., multitier reactives, for both communication and fault tolerance. Akka [2009a] and Akka Typed [2015] actors do not provide high-level communication abstractions like reactives nor support multitier programming. For this reason, in Akka and Akka Typed, programmers need to reason in terms of message-passing not in terms of distributed (placed) reactives. Vats in the E programming language [Miller et al. 2005] are similar to ScalaLoci's peers, exchanging messages asynchronously between objects in (distributed) vats or placed values on peers, respectively. In contrast to ScalaLoci, E does not provide architecture specification or a reactive communication mechanism.

Multitier Languages. Multitier languages emerge in the web context to remove the separation between client and server code, either by compiling the client side to JavaScript or by adopting JavaScript for the server, too. Hop [Serrano et al. 2006] and Hop.js [Serrano and Prunet 2016] are dynamically typed languages that follow a traditional client–server communication scheme with asynchronous callbacks. They do not provide reactives for distributed data flow nor ensure static guarantees for the behavior of the distributed system. Remote blocks are similar to Hop's anonymous services and prevent accidental capture, similar to spores [Miller et al. 2014]. In Links [Cooper et al. 2007] and Opa [Rajchenbach-Teller and Sinot 2010], functions are annotated to specify either client- or server-side execution. In Opa, the server can push messages to clients over channels, which are similar to event streams but cannot be composed like RP abstractions. Links' server is stateless for scalability reasons – limiting the spectrum of the supported architectures. In StiP.js [Philips et al. 2014], annotations assign code fragments to the client or the server. Slicing detects the dependencies between each fragment and the rest of the program. In contrast, in ScalaLoci, developers specify placement *in types*, enabling architectural reasoning. All approaches above focus on the web, contrarily to our goal of supporting other architectures. An exception is ML5 [Murphy et al. 2008], a multitier language for generic software architectures: *Possible worlds*, as known from modal logic, address the purpose of placing computations and, similar to ScalaLoci, are part of the type. ML5, however, does not provide RP abstractions.

(Functional) Reactive Programming. FRP was originally proposed by Elliott and Hudak [1997] to declaratively program visual animations. Formally modeling continuous time led to a denotational semantics where time-changing variables are functions from time to values [Nilsson et al. 2002]. FRP has since been applied to a number of fields including robotics [Hudak et al. 2003], network switches [Foster et al. 2011] and wireless sensor networks [Newton et al. 2007]. User interfaces have become a largely popular application field for RP. Flapjax [Meyerovich et al. 2009] pioneered using RP in JavaScript web clients. Elm [Czaplicki and Chong 2013], a functional language akin to Haskell that compiles to JavaScript, adopts a similar approach. Flapjax provides behaviors and event streams, while Elm uses only signals to model both time-varying values and events. Microsoft Reactive Extensions (Rx) [Meijer 2010] offer abstractions for event streams. Rx is available for both Java (RxJava) and JavaScript (RxJS), but as separate implementations, i.e., reactive dependencies cannot cross the boundaries among different hosts. Recently, Ramson and Hirschfeld [2017] proposed *active expressions* as a primitive for change detection, upon which different kinds of RP can be built. Other recent research directions in RP include debugging [Banken et al. 2018; Perez and Nilsson 2017; Salvaneschi and Mezini 2016], thread-safe concurrency [Drechsler et al. 2018], and application to new domains such as IoT and edge computing [Calus et al. 2017] and autonomous vehicles [Finkbeiner

et al. 2017]. In the distributed setting, the DREAM reactive middleware [Margara and Salvaneschi 2014, 2018] allows selecting different levels of consistency guarantees for distributed RP. CRDTs have been used to handle state replication for signals shared among hosts [Myter et al. 2016]. Mogk et al. [2018] extend the RP language REScala [Salvaneschi et al. 2014b] with distributed fault handling based on state replication via CRDTs and state snapshotting. None of the works discussed above support multitier programming.

A line of work similar to RP concerns streams, e.g., to support different propagation strategies via object algebras [Biboudis et al. 2015], or to improve performance via staging [Kiselyov et al. 2017]. Akka Streams [2014] can send messages to actors and actors can queue messages into streams, but remote communication occurs via message-passing among actors. Distributed Akka streams are currently in experimental stage. Yet, there is no explicit notion of placement or of architecture specification.

Multitier Functional Reactive Languages. Ur/Web [Chlipala 2015] supports FRP user interfaces on the client. However, it does not directly support data flows over multiple hosts. Communication from the server to the client is achieved by message-passing channels and from client to server by RPC. Eliom [Radanne et al. 2016], the multitier language used in the Ocsigen [Balat 2006] project, provides signals and events – both can propagate values remotely. Contrarily to our approach, however, it only supports client–server web applications. Hiphop [Berry et al. 2011] extends Hop with a event system for client–server communication (only), which is based on the synchronous data flow model introduced by *synchronous data flow languages*, e.g., Esterel [Berry and Gonthier 1992] or Lustre [Halbwachs et al. 1991]. Hiphop's approach focuses on providing strong guarantees on memory and time bounds by restricting the expressivity of the language rather than providing high-level abstractions for distribution. AmbientTalk/R [Carreton et al. 2010] targets mobile applications with loosely coupled devices and no stable data flow. It provides reactives on top of a publish-subscribe middleware. Contrarily to ScalaLoci, it does not support multitier programming via placement types. Scala Multi-Tier FRP [Reynders et al. 2014] achieves multitier reactivity via behaviors and events accessible across tiers. Similar to ScalaLoci, client-side and server-side reactives belong to different types. Scala Multi-Tier FRP only supports client–server web applications: Other architectures like P2P (used in our P2P Chat example) or more complex ones with masters, workers, proxies etc. (e.g., the Flink and Gearpump case studies implemented in our evaluation) are not supported.

10 CONCLUSION

We presented ScalaLoci, a multitier reactive language with statically typed specification of data flows spanning over multiple hosts. ScalaLoci provides language abstractions to specify value distribution to different hosts via explicit placement and seamlessly compose distributed reactives. The evaluation on case studies and third party applications shows higher design quality at the cost of negligible performance overhead. We are currently investigating means to modularly compose multitier applications reconciling the tension between multitier programming (condensing code from different peers into the same compilation unit) and the need for encapsulation and strong interfaces (separating different concerns into different modules).

ACKNOWLEDGMENTS

This work has been supported by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 – MAKI, subproject C2, by the European Research Council, grant No. 321217, and by the DFG project SA 2918/2-1.

REFERENCES

Akka. 2009a. http://akka.io/. Accessed 2018-04-16.

Akka. 2009b. Message Delivery Reliability. http://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html. Accessed 2018-04-16.

Akka. 2009c. Supervision and Monitoring. http://doc.akka.io/docs/akka/current/scala/general/supervision.html. Accessed 2018-04-16.

Akka. 2009d. Watching Remote Actors. http://doc.akka.io/docs/akka/current/remoting.html#watching-remote-actors. Accessed 2018-04-16.

Akka. 2009e. What Lifecycle Monitoring Means. http://doc.akka.io/docs/akka/current/general/supervision.html#what-lifecycle-monitoring-means. Accessed 2018-04-16.

Akka HTTP. 2016. http://doc.akka.io/docs/akka-http/current/. Accessed 2018-04-16.

Akka Streams. 2014. http://doc.akka.io/docs/akka/current/stream/. Accessed 2018-04-16.

Akka Typed. 2015. http://doc.akka.io/docs/akka/current/typed/. Accessed 2018-04-16.

Jonathan Aldrich, Craig Chambers, and David Notkin. 2002. ArchJava: Connecting Software Architecture to Implementation. In *ICSE '02*. ACM, New York, NY, USA.

Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. 2003. Language Support for Connector Abstractions. In *ECOOP*. Springer, Berlin, Heidelberg.

Apache Storm. 2011. http://storm.apache.org/. Accessed 2018-04-16.

Vincent Balat. 2006. Ocsigen: Typing Web Interaction with Objective Caml. In *ML '06*. ACM, New York, NY, USA.

Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging Data Flows in Reactive Programs. In *ICSE '18*. ACM, New York, NY, USA.

Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19, 2 (1992).

Gérard Berry, Cyprien Nicolas, and Manuel Serrano. 2011. HipHop: A Synchronous Reactive Extension for Hop. In *PLASTIC '11*. ACM, New York, NY, USA.

Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. 2015. Streams a la carte: Extensible Pipelines with Object Algebras. In *ECOOP '15*, Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany.

Eugene Burmako. 2013. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *SCALA '13*. ACM, New York, NY, USA, Article 3.

Ben Calus, Bob Reynders, Dominique Devriese, Job Noorman, and Frank Piessens. 2017. FRP IoT Modules As a Scala DSL. In *REBLS '17*. ACM, New York, NY, USA.

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 Engineering Bulletin* 38 (2015).

Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. 2010. Loosely-coupled Distributed Reactive Programming in Mobile Ad Hoc Networks. In *TOOLS '10*. Springer-Verlag, Berlin, Heidelberg.

Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. 2001. Design and Evaluation of a Wide-area Event Notification Service. *ACM Transactions on Computer Systems* 19, 3 (2001).

Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair Reactive Programming. In *POPL '14*. ACM, New York, NY, USA.

Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. 2008. Type Inference for Locality Analysis of Distributed Data Structures. In *PPoPP '08*. ACM, New York, NY, USA.

S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *IPDPSW '16*.

Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *POPL '15*. ACM, New York, NY, USA.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects (FMCO'06)*. Springer-Verlag, Berlin, Heidelberg.

Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *ESOP '06*.

Coq Development Team. 2016. The Coq Proof Assistant, version 8.6.0. http://coq.inria.fr/.

Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *PLDI '13*. ACM, New York, NY, USA.

Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. 2015. Partitioned Global Address Space Languages. *Comput. Surveys* 47, 4, Article 62 (2015).

Sébastien Doeraene. 2013. *Scala.js: Type-Directed Interoperability with Dynamically Typed Languages*. Technical Report. EPFL.

Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. Thread-Safe Reactive Programming. *Proceedings of the ACM on Programming Languages* OOPSLA '18 (2018).

Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA.

Jonathan Edwards. 2009. Coherent Reaction. In *OOPSLA '09*. ACM, New York, NY, USA.

Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *ICFP '97*. ACM, New York, NY, USA.

Conal M. Elliott. 2009. Push-Pull Functional Reactive Programming. In *Haskell '09*. ACM, New York, NY, USA.

Erlang. 1987a. Academic and Historical Questions. http://erlang.org/faq/academic.html. Accessed 2018-04-16.

Erlang. 1987b. Supervision Principles. http://erlang.org/documentation/doc-9.3/doc/design_principles/sup_princ.html. Accessed 2018-04-16.

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

Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2017. Vehicle Platooning Simulations with Functional Reactive Programming. In *SCAV '17*. ACM, New York, NY, USA.

Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. 2007. Tasks: Language Support for Event-driven Programming. In *PEPM '07*. ACM, New York, NY, USA.

Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: a network programming language. In *ICFP '11*. ACM, New York, NY, USA.

Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language Lustre. *Proc. IEEE* 79, 9 (1991).

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI '73*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Summer School on Advanced Functional Programming 2002, Oxford University (Lecture Notes in Computer Science)*, Vol. 2638. Springer-Verlag.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *POPL '17*. ACM, New York, NY, USA.

Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. 2012. Higher-order Functional Reactive Programming in Bounded Space. In *POPL '12*. ACM, New York, NY, USA.

Haoyi Li. 2013. TodoMVC application written in Scala.js. http://github.com/lihaoyi/workbench-example-app. Accessed 2018-04-16.

Haoyi Li. 2014. Method arguments that introduce new identifiers into scope? Or: language support for enums and contextual DSLs? http://groups.google.com/forum/#!topic/scala-debate/f4CLmYShX6Q. Accessed 2018-04-16.

Ingo Maier and Martin Odersky. 2013. Higher-Order Reactive Programming with Incremental Lists. In *ECOOP '13*. Springer-Verlag, Berlin, Heidelberg.

Ingo Maier, Tiark Rompf, and Martin Odersky. 2010. *Deprecating the Observer Pattern.* Technical Report.

Alessandro Margara and Guido Salvaneschi. 2014. We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees. In *DEBS '14*. ACM, New York, NY, USA.

A. Margara and G. Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering* 44, 7 (July 2018).

René Meier and Vinny Cahill. 2005. Taxonomy of Distributed Event-Based Programming Systems. *Comput. J.* 48, 5 (2005).

Erik Meijer. 2010. Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues. In *CUFP '10*. ACM, New York, NY, USA, Article 11.

Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A programming language for Ajax applications. In *OOPSLA '09*. ACM, New York, NY, USA.

Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *ECOOP '14*. Springer-Verlag New York, Inc., New York, NY, USA.

Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. 2005. Concurrency Among Strangers. In *TGC '05*, Rocco De Nicola and Davide Sangiorgi (Eds.). Springer, Berlin, Heidelberg.

Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault-tolerant Distributed Reactive Programming. In *ECOOP '18 (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

Tom Murphy, VII., Karl Crary, and Robert Harper. 2008. Type-safe Distributed Programming with ML5. In *TGC '07*. Springer-Verlag, Berlin, Heidelberg.

Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. 2016. I Now Pronounce You Reactive and Consistent: Handling Distributed and Replicated State in Reactive Programming. In *REBLS '16*. ACM, New York, NY, USA.

Matthias Neubauer and Peter Thiemann. 2005. From Sequential Programs to Multi-Tier Applications by Program Transformation. In *POPL '05*. ACM, New York, NY, USA.

R. Newton, G. Morrisett, and M. Welsh. 2007. The Regiment Macroprogramming System. In *IPSN '07*.

Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional reactive programming, continued. In *Haskell '02*. ACM, New York, NY, USA.

Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2017. Simplicitly: Foundations and Applications of Implicit Function Types. *Proceedings of the ACM on Programming Languages* 2, POPL '17, Article 42 (2017).

Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes As Objects and Implicits. In *OOPSLA '10*. ACM, New York, NY, USA.

Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proceedings of the ACM on Programming Languages* 1, ICFP '17, Article 2 (2017).

Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. 2014. Towards Tierless Web Development Without Tierless Languages. In *Onward! 2014*. ACM, New York, NY, USA.

Peter R. Pietzuch and Jean Bacon. 2002. Hermes: A Distributed Event-Based Middleware Architecture. In *ICDCSW '02*. IEEE Computer Society, Washington, DC, USA.

Play. 2012. Documentation Play Framework 2.0. Actions, Controllers and Results. Building an Action. http://playframework.com/documentation/2.0/ScalaActions#Building-an-Action. Accessed 2018-04-16.

Play Framework. 2007. http://playframework.com/. Accessed 2018-04-16.

Hussachai Puripunpinyo. 2014. Play Framework with Scala.js Showcase. http://github.com/hussachai/play-scalajs-showcase/. Accessed 2018-04-16.

Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A core ML language for Tierless Web Programming. In *APLAS '16*. Springer, Hanoi, Vietnam.

David Rajchenbach-Teller and Franois-Régis Sinot. 2010. Opa: Language support for a sane, safe and secure web. In *OWASP AppSec Research*.

Stefan Ramson and Robert Hirschfeld. 2017. Active Expressions: Basic Building Blocks for Reactive Programming. *CoRR* (2017).

Reactive Streams. 2014. http://www.reactive-streams.org/. Accessed 2018-04-16.

Bob Reynders, Dominique Devriese, and Frank Piessens. 2014. Multi-Tier Functional Reactive Programming for the Web. In *Onward! 2014*. ACM, New York, NY, USA.

Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. 2014a. An Empirical Study on Program Comprehension with Reactive Programming. In *FSE 2014*. ACM, New York, NY, USA.

Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014b. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *MODULARITY '14*. ACM, New York, NY, USA.

Guido Salvaneschi and Mira Mezini. 2014. Towards Reactive Programming for Object-Oriented Applications. In *Transactions on Aspect-Oriented Software Development XI (Lecture Notes in Computer Science)*, Vol. 8400. Springer, Berlin, Heidelberg.

Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *ICSE '16*. ACM, New York, NY, USA.

ScalaSTM. 2014. Quick Start. Wrap your code in atomic. http://nbronson.github.io/scala-stm/quick_start.html#atomic. Accessed 2018-04-16.

Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop: A Language for Programming the Web 2.0. In *Companion to OOPSLA '06*. ACM.

Manuel Serrano and Vincent Prunet. 2016. A Glimpse of Hopjs. In *ICFP '16*. ACM, New York, NY, USA.

Slick. 2014. Documentation Slick 2.0. Connections/Transactions. Implicit Session. http://slick.typesafe.com/doc/2.0.0/connection.html#implicit-session. Accessed 2018-04-16.

Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. 1994. Separating Data and Control Transfer in Distributed Operating Systems. In *ASPLOS VI*. ACM, New York, NY, USA.

TodoMVC. 2011. http://todomvc.com/. Accessed 2018-04-16.

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994).

Sean Zhong, Kam Kasravi, Huafeng Wang, Manu Zhang, and Weihua Jiang. 2014. GearPump – Real-Time Streaming Engine Using Akka. (2014).