Multitier Modules

Pascal Weisenburger
Technische Universität Darmstadt, Germany
weisenburger@cs.tu-darmstadt.de

Guido Salvaneschi
Technische Universität Darmstadt, Germany
salvaneschi@cs.tu-darmstadt.de

Abstract

Multitier programming languages address the complexity of developing distributed systems abstracting over low level implementation details such as data representation, serialization and network protocols. Since the functionalities of different peers can be defined in the same compilation unit, multitier languages do not force developers to modularize software along network boundaries. Unfortunately, combining the code for all tiers into the same compilation unit poses a scalability challenge or forces developers to resort to traditional modularization abstractions that are agnostic to the multitier nature of the language.

In this paper, we address this issue with a module system for multitier languages. Our module system supports encapsulating each (cross-peer) functionality and defining it over abstract peer types. As a result, we disentangle modularization and distribution and we enable the definition of a distributed system as a composition of multitier modules, each representing a subsystem. Our case studies on distributed algorithms, distributed data structures, as well as on the Apache Flink task distribution system, show that multitier modules allow the definition of reusable (abstract) patterns of interaction in distributed software and enable separating the modularization and distribution concerns, properly separating functionalities in distributed systems.

2012 ACM Subject Classification Computing methodologies → Distributed programming languages; Software and its engineering → Modules / packages

Keywords and phrases Distributed Programming, Multitier Programming, Abstract Peer Types, Placement Types, Module Systems, Scala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.20

Funding This work has been supported by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 MAKI and 1119 CROSSING, by the DFG projects SA 2918/2-1 and SA 2918/3-1, by the Hessian LOEWE initiative within the Software-Factory 4.0 project, by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, and by the AWS Cloud Credits for Research program.

Acknowledgements We would like to thank Philipp Schulz for the implementation of the first prototype of this work and all reviewers of this paper for their comments and suggestions.

1 Introduction

Implementing distributed systems is notoriously hard because of a number of issues that naturally arise in this setting, such as consistency, fault tolerance, concurrency, mismatch among data formats, as well as mix of languages and execution platforms.

Multitier – or *tierless* – languages [36, 13, 26, 14, 41] address some of these problems. In *multitier* languages, peers (e.g., the client and the server in a Web application) are written in the same compilation unit. The compiler splits the code into a client unit and a server unit, adds the necessary communication code, performs the necessary translations (e.g., translating client code to JavaScript) and generates the deployable components. As

a result, developers need to use a single language, are not forced to worry about network communication, serialization, data formats and conversions and can focus on the application logic without breaking it down along network boundaries.

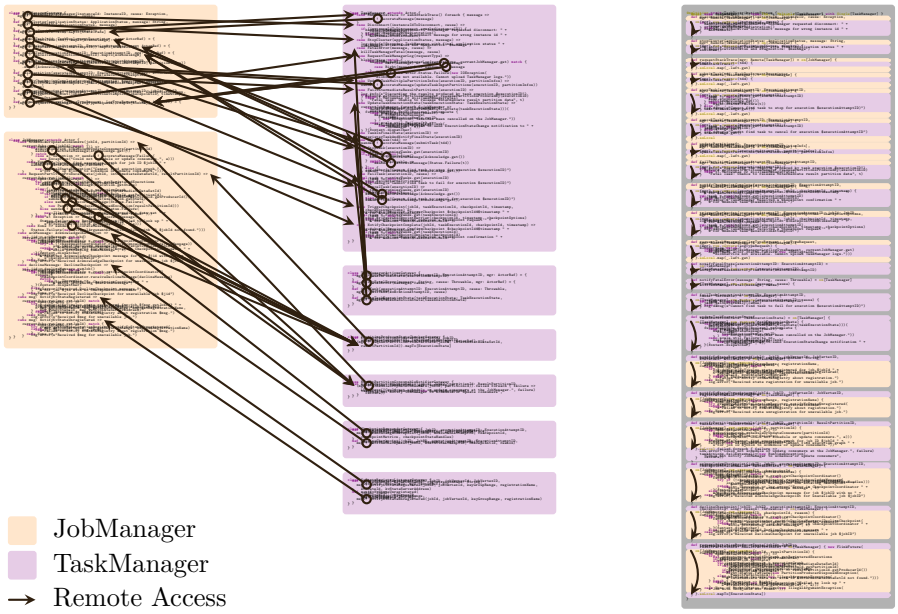
Lack of Modularization Abstractions. Scaling multitier code to large applications, however, is an open problem. Researchers have been focusing on small use cases that primarily aim to demonstrate the design of their language rather than investigating the development of complex applications that require sophisticated modularization and composition, or require abstracting over program locations and architecture. In use cases of limited size, client code and server code are nicely combined in a single compilation unit, but it is unclear what happens when one compilation unit is not enough.

As an example of this issue, Figure 1 provides an overview of the *task distribution system* in the Apache Flink stream processing framework [9]. It consists of the coordinator of the Flink instance, the *JobManager* and one or more *TaskManagers*, which execute computational tasks. Figure 1a shows the JobManager (light orange boxes, left), the TaskManager (dark violet boxes, right) and their communication (arrows). Every box is a class or an actor which is confined by network boundaries. Thus, cross-host data flow belonging to the same (distributed) functionality is scattered over multiple modules. Figure 1b shows an implementation of the same system in the ScalaLoci multitier language [48] (the figure is adapted from the same work). The data flow in the system is much more *regular*, due to the reorganization of the same code in a single unit, yet all the functionalities of the system are concentrated in a single large compilation unit with $\sim$400 LOC.

Unfortunately, adopting the traditional modularization mechanism supported by the base language (e.g., a Haskell module for a Haskell-based multitier language) is not sufficient because such modularization mechanism is not aware of multitier code and it is unclear what output code is produced after the compiler splits the code.

In summary, simplifying reasoning about distributed applications, abstracting over network communication and format conversions, and providing a single language to implement all components, multitier languages have the potential to significantly help programmers developing distributed systems. However, enabling multitier programming to scale to large code bases is still a research problem because of the lack of proper modularization mechanisms. In its current state, multitier programming does effectively defeat the *tyranny of the dominant decomposition* [45] for distributed systems, removing the need to modularize applications according to tiers and network communication. Yet, it does not offer an alternative modularization solution designed in synergy with multitier abstractions.

Contribution. In this paper, we propose LociMod, a novel *multitier module system* for ScalaLoci. Multitier modules encapsulate the interaction between distributed components of (sub)systems, allowing for (1) **decoupling modularization from distribution** and (2) **defining reusable patterns of interaction** that model the functionality of a (distributed) subsystem and that can be composed to build larger distributed systems. LociMod supports strong interfaces [22] to achieve encapsulation and information hiding, such that implementations can be easily exchanged. The main contribution of the work is to make peer types, which define the placement of a functionality in LociMod, abstract. This design choice enables the definition of abstract modules, which capture a fragment of a distributed system, can be further composed with other abstract modules, and can eventually be instantiated for a concrete software architecture.



(a) Original Flink.

(b) ScalaLoci reimplementation.

Figure 1 Flink task distribution system in ScalaLoci, adapted from [48].

Our case studies on distributed algorithms, distributed data structures, as well as on the Apache Flink task distribution system, show that LociMod multitier modules allow the definition of reusable (abstract) patterns of interaction in distributed software and enable separating the modularization and distribution concerns, properly separating functionalities in distributed systems. In summary, this paper makes the following contributions:

- We present LociMod, a novel module system for multitier languages, featuring multitier modules, which support strong interfaces and exchangeable implementations.
- We show that, thanks to LociMod abstract peer types, the interaction between multitier abstractions and modularization features results in a number of powerful abstractions to define and compose distributed systems, including multitier mixin composition and constrained modules.
- We provide an implementation of LociMod as an extension to ScalaLoci, a multitier language embedded into Scala. The implementation supports separate compilation and is publicly available.[1]
- We evaluate LociMod with case studies, including distributed algorithms, distributed data structures and the Apache Flink Big Data processing framework, demonstrating the composition properties of multitier modules and how they can capture (distributed) functionalities in complex systems.

The paper is structured as follows. Section 2 provides an overview of ScalaLoci and the important features of the Scala type system. Section 3 describes the design of multitier modules. Section 4 discusses the implementation. Section 5 presents the evaluation. Section 6 discusses the related work. Section 7 concludes.

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

2 Background

2.1 ScalaLoci

Since LociMod is an extension of ScalaLoci [48], we first provide an overview of ScalaLoci to the reader. ScalaLoci is a *general purpose* multitier language for distributed systems – unlike most multitier languages, which focus on the Web (i.e., client-server architecture) only. To support generic distributed architectures, ScalaLoci provides language abstractions for developers to freely define the different components – called *peers* – of the distributed system and their architectural relation. Peers are defined as *peer types*, which allow for specifying the placement of data and computations at the type level using *placement types*, enabling static reasoning about placement. A placement type `T on P`[2] represents a value of a traditional type `T` which is placed on a peer of peer type `P`.

Placed Values. Placed values of type `T on P` are initialized with `placed { e }` expressions. For example, the following string is placed on the `Master` peer:

```
val name: String on Master = placed { "the one and only master" }
```

The `name` value is accessible remotely from other peers. Accessing remote values requires the `asLocal` marker, creating a local representation of the remote value by transmitting it over the network:

```
val masterName: Future[String] on Worker = placed { name.asLocal }
```

Accessing `name` remotely from a `Worker` peer returns a value of type `Future[String]`. Futures – which are part of Scala's standard library – 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.

Remote accessibility of placed values can be regulated: Local placed values denoted by the type `Local[T] on P` specify values that can only be accessed locally from the same peer instance, i.e., remote access via `asLocal` is not possible:

```
val realName: Local[String] on Master = placed { "Rumpelstiltskin" }
```

For a value definition `val v: T on P = placed { e }`, the shorthand notation `val v = on[P] { e }` can be used, inferring the placement type `T on P`.

Placed Computations. Like placed values, placed computations are declared to have a placement type and defined using a `placed` expression:

```
def execute(task: Task[T]): T on Worker = placed { task.process() }
```

Invoking a remote computation is explicit using `remote call`. If the result of a remote computation is of interest to the local peer instance, it can be made available locally using `asLocal` (as described before):

```
val result: Future[T] on Master = placed { (remote call execute(new Task()).asLocal }
```

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

Architecture Specification. In LociMod, the architectural scheme of the distributed system is expressed using *ties*, which specify the kind of relation among peers. Ties are encoded as structural type refinements specifying the `Tie` type for peers. Ties to multiple peers are defined by declaring a compound type (e.g., `type Tie <: Single[Master] with Multiple[Worker]`). Remote access is only possible between tied peers.

For instance, an architecture with a single master that offloads computations to a single worker is defined by a `Master` peer and a `Worker` peer (specified through the `@peer` annotation on type members):

```
@peer type Master <: { type Tie <: Single[Worker] }
@peer type Worker <: { type Tie <: Single[Master] }
```

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

```
@peer type Master <: { type Tie <: Multiple[Worker] }
@peer type Worker <: { type Tie <: Single[Master] }
```

Section 5.1 presents a more systematic categorization of common distributed architectures and their encoding using peers and ties.

2.2 Scala Abstract Data Types and Path-Dependent Types

The LociMod module system leverages Scala's type system features, in particular abstract types and path-dependent types, which we quickly revise. In Scala, traits, classes and objects (i.e., singleton classes) define type members, which are either abstract (e.g., `type SomeType`) or define concrete type aliases (e.g., `type SomeType = Int`). Abstract type members can define lower and upper type bounds (e.g., `type SomeType >: LowerBound <: UpperBound`), which refine the type while keeping it abstract. Inherited abstract type members can be overridden. Such mechanism enables specializing the upper bound and generalizing the lower bound. In Section 3.2, we define the peers of the distributed system as abstract type members. Refining the upper bound enables specializing a peer as (a subtype of) another peer, enabling peer composition by combining super peers into a sub peer.

Scala types can be *dependent on an path* (of objects). For example, in the following code snippet, both objects `a` and `b` inherit the `SomeType` abstract type member defined in the `Module` trait:

```
trait Module { type SomeType }
object a extends Module
object b extends Module
```

The path-dependent type `a.SomeType` refers to `a`'s `SomeType` and the path-dependent type `b.SomeType` refers to `b`'s `SomeType`. The types depend on the objects `a` or `b`, respectively. They are distinct since their paths differ.

For instance, the following example defines a module `A` with an abstract type member `T`. Further, a module `B` defines an abstract type member `U`. The module `C` extends module `B` inheriting its abstract type member `U`. The module `C` also references an instance `a` of module

`A`. Module `C`'s `U` type is overridden and refined as a subtype of the `T` type defined in the `a` instance by specifying that the upper bound of `U` is the path-dependent type `a.T`:

```
1 trait A { type T }
2
3 trait B { type U }
4
5 trait C extends B {
6   type U <: a.T
7   val a: A
8 }
```

In Section 3.2.1, we use this mechanism to declare references to other modules from within a module and to refer to the peers (defined as abstract type members) in the referenced modules via their path-dependent types.

3 LociMod Multitier Modules

In this section, we describe the LociMod module system. The goal of this section is twofold. On the one hand, we present the design of multitier modules. On the other hand, we demonstrate a number of examples for multitier modules and their composition mechanisms.

We first introduce (concrete) *multitier modules* and show how they can be composed into larger applications, using *module references* – references to other multitier modules. Then we introduce modules with *abstract peer types* and show their composition through module references as well as another composition mechanism, *multitier mixing*. Next we show how such composition mechanism enables defining *constrained multitier modules*.

3.1 Multitier Modules

We embed LociMod into Scala, following the same approach of ScalaLoci, which is a Scala DSL. Scala traits represent modules – adopting Scala's design that unifies object and module systems [30]. Traits can contain abstract declarations and concrete definitions for both type and value members – thus serve as both module definitions and implementations – and Scala objects can be used to instantiate traits.

Module Definition. In LociMod, multitier modules are defined by a trait with the `@multitier` annotation. Multitier modules can define (i) values placed on different peers and (ii) the peers on which the values are placed – including constraints on the architectural relation between peers. This approach enables modularization across peers (not necessarily along peer boundaries) combining the benefits of multitier programming and modular development. To illustrate, consider an application that allows a user to edit documents offline but also offers the possibility to backup the data to an online storage:

```
1 @multitier trait Editor {
2   @peer type Client <: { type Tie <: Single[Server] }
3   @peer type Server <: { type Tie <: Single[Client] }
4
5   val backup: FileBackup
6 }
```

The `Editor` specifies a `Client` (Line 2) and a `Server` (Line 3). The client should be able to backup/restore documents to/from the server, e.g., the client can invoke a `backup.store` method to backup data. Thus, the module requires an instance of the `FileBackup` multitier module (Line 5) providing the backup service. Section 3.2 shows how the `Editor` and the `FileBackup` module can be composed.

Encapsulation with Multitier Modules. LociMod's multitier modules *encapsulate distributed (sub)systems* with a specified distributed architecture, enabling the creation of larger distributed applications by composition. The following code shows a possible implementation for the backup service subsystem using the file system to store backups:

```
1  @multitier trait FileBackup {
2    @peer type Processor <: { type Tie <: Single[Storage] }
3    @peer type Storage <: { type Tie <: Single[Processor] }
4
5    def store(id: Long, data: Data): Unit on Processor = placed { remote call write(id, data) }
6    def load(id: Long): Future[Data] on Processor = placed { (remote call read(id)).asLocal }
7
8    private def write(id: Long, data: Data): Unit on Storage = placed {
9      writeToFile(data, s"/storage/$id") }
10   private def read(id: Long): Data on Storage = placed {
11     readFromFile[Data](s"/storage/$id") }
12 }
```

The multitier `FileBackup` module specifies a `Processor` to compress data (of type `Data`) and a `Storage` peer to store and retrieve data associating them to an ID. The `store` (Line 5) and `load` (Line 6) methods can be called on the `Processor` peer, invoking `write` (Line 8) and `read` (Line 10) remotely on the `Storage` peer. The implementations of the `write` and `read` methods operate on files.

LociMod multitier modules support standard access modifiers for placed values (e.g., `private`, `protected` etc.), which are used as a technique to encapsulate module functionality. In the `FileBackup` module, the `write` and the `read` methods are declared private, so other modules that use `FileBackup` cannot directly access them. Overall, the `FileBackup` module encapsulates all the functionalities related to the backup service subsystem, including the communication between `Processor` and `Storage`.

As the last example demonstrates, multitier modules enable separating modularization and distribution concerns, allowing developers to organize applications based on logical units instead of network boundaries. A multitier module abstracts over potentially multiple components and the communication between them, specifying distribution by expressing the placement of a computation on a peer in its type. Both axes are traditionally intertwined by having to implement a component of the distributed system in a module (e.g., a class, an actor, etc.) leading to cross-host functionality being scattered over multiple modules.

Multitier Modules as Interfaces and Implementations. To decouple the code that uses a multitier module from the concrete implementation of such a module, LociMod supports modules to be used as interfaces and implementations. Multitier modules can be abstract, i.e., defining only abstract members, acting as module interfaces or they can define concrete implementations. For example, applications that require a backup service can be developed against the `BackupService` module interface, which declares a `store` and a `load` method:

```
1  @multitier trait BackupService {
2    @peer type Processor <: { type Tie <: Single[Storage] }
3    @peer type Storage <: { type Tie <: Single[Processor] }
4
5    def store(id: Long, data: Data): Unit on Processor
6    def load(id: Long): Future[Data] on Processor
7  }
```

LociMod adopts Scala's inheritance mechanism to express the relation between the multitier modules used as interfaces and their implementations. The `FileBackup` module presented

before is a possible implementation for the `BackupService` module interface, i.e., we can redefine `FileBackup` to let it implement `BackupService`:

```
1 @multitier trait FileBackup extends BackupService { ... }
```

The following example presents a different implementation for the `BackupService` module interface using a database backend (instead of a file system) as storage:

```
1  @multitier trait DatabaseBackup extends BackupService {
2    def store(id: Long, data: Data): Unit on Processor = placed { remote call insert(id, data) }
3    def load(id: Long): Future[Data] on Processor = placed { (remote call query(id)).asLocal }
4
5    private val db: AsyncContext = ...
6
7    private def insert(id: Long, data: Data): Unit on Storage = placed {
8      db.run(query[(Long, Data)].insert(lift(id -> data))) }
9    private def query(id: Long): Future[Data] on Storage = placed {
10     db.run(query[(Long, Data)].filter { _._1 == lift(id) }) map { _.head._2 } }
11 }
```

The implementations of the `store` and of the `load` methods invoke `insert` (Line 7) and `query` (Line 9) remotely, which insert the backup data into a database and retrieve the data from a database, respectively.[3]

Combining Multitier Modules. Thanks to the separation between module interfaces and module implementations, applications can be developed against the interface, remaining agnostic to the implementation details of a subsystem encapsulated in a multitier module. For example, the `Editor` presented before can be adapted to use a `BackupService` interface instead of the concrete `FileBackup` implementation (Line 5):

```
1 @multitier trait Editor {
2   @peer type Client <: { type Tie <: Single[Server] }
3   @peer type Server <: { type Tie <: Single[Client] }
4
5   val backup: BackupService
6 }
```

Finally, a multitier module can be instantiated by instantiating concrete implementations of the module interfaces it refers to. LociMod relies on the Scala approach of using an `object` to instantiate a module, i.e., declaring an object that extends a trait – or mixes together multiple traits – creates an instance of those traits. For example, the following code creates an `editor` instance of the `Editor` module by providing a concrete `DatabaseBackup` instance for the abstract `backup` value:

```
1 @multitier object editor extends Editor {
2   @multitier object backup extends DatabaseBackup
3 }
```

The multitier module instance of a `@multitier object` can be used to run different peers from (non-multitier) standard Scala code (e.g., the `Client` and the `Server` peer), where every peer instance only contains the values placed on the respective peer. Peer startup is presented in Section 3.4.

[3] The example uses the Quill `http://getquill.io` query language to access the database

3.2 Abstract Peer Types

In the previous section, we have shown how to encapsulate a subsystem within a multitier module and how to define a module interface such that multiple implementations are possible. LociMod modules allow for going further, enabling abstraction over placement using abstract peer types. Peer types are abstract type members of traits, i.e., they can be overridden in sub traits, specializing their type. As a consequence, LociMod multitier modules are parametric on peer types. For example, the `BackupService` module of the previous section defines an abstract `Processor` peer, but the `Processor` peer does not necessarily need to refer to a physical peer in the system. Instead, it denotes a *logical* place. When running the distributed system, a `Client` peer, for example, may adopt the `Processor` role, by specializing the `Client` peer to be a `Processor` peer.

Peer types are used to distinguish places only at the type level, i.e., the placement type `T on P` represents a run time value of type `T`. The peer type `P` is used to keep track of the value's placement, but a value of type `P` is never constructed at run time. Hence, `T on P` is essentially a "phantom type" [12] due to its parameter `P`.

The next two sections describe the interaction of abstract peer types with two composition mechanisms for multitier modules. We already encountered the first mechanism, module references. The other mechanism, multitier mixing, enables combining multitier modules directly. In both cases, the peers defined in a module can be specialized with the role of other modules' peers.

3.2.1 Peer Type Specialization with Module References

Since peer types are abstract, they can be specialized by narrowing their upper type bound, augmenting peers with different roles defined by other peers. Peers can subsume the roles of other peers – similar to subtyping on classes – enabling polymorphic usage of peers. Programmers can use this feature to augment peer types with roles defined by other peer types by establishing a subtyping relation between both peers. This mechanism enables developers to define reusable patterns of interaction among peers that can be specialized later to any of the existing peers of an application.

For example, the editor application that requires the backup service (Section 3.1) needs to specialize its `Client` peer to be a `Processor` peer and its `Server` peer to be a `Storage` peer for clients to be able to perform backups on the server:

```
1 @multitier trait Editor {
2   @peer type Client <: backup.Processor { type Tie <: Single[Server] with Single[backup.Storage] }
3   @peer type Server <: backup.Storage { type Tie <: Single[Client] with Single[backup.Processor] }
4
5   val backup: BackupService
6 }
```

We specify the `Client` peer to be a (subtype of the) `backup.Processor` peer (Line 2) and the `Server` peer to be a (subtype of the) `backup.Storage` peer (Line 3). Both `backup.Processor` and `backup.Storage` refer to the peer types defined on the `BackupService` instance referenced by `backup`. We can use such *module references* to refer to (path-dependent) peer types through a reference to the multitier module.

Since the subtyping relation `Server <: backup.Storage` specifies that a server *is a* storage peer, the backup functionality (i.e., all values and methods placed on the `Storage` peer) are also placed on the `Server` peer. Super peer definitions are locally available on sub peers,

making peers composable using subtyping. Abstract peer types specify such subtyping relation by declaring an upper type bound. When augmenting the server with the storage functionality using subtyping, the `Tie` type also has to be a subtype of the `backup.Storage` peer's `Tie` type. This type level encoding of the architectural relations among peers enables the Scala compiler to check that the combined architecture of the system complies to the architectural constraints of every subsystem.

Note that, for the current example, one may expect to unify the `Server` and the `Storage` peer, so they refer to the same peer, specifying type equality instead of a subtyping relation:

```
1 @peer type Server = backup.Storage { type Tie <: Single[Client] }
```

Since peer types, however, are never instantiated (they are used only as phantom types to keep track of placement at the type level) we can always keep peer types abstract, only specifying an upper type bound. Hence, it is sufficient to specialize `Server` to be a `backup.Storage`, keeping the `Server` peer abstract for potential further specialization.

3.2.2 Peer Type Specialization with Multitier Mixing

The previous section shows how peer types can be specialized when referring to modules through *module references*. This section presents a different composition mechanism based on composing traits – similar to *mixin composition* [4]. Since LociMod multitier modules can encapsulate distributed subsystems (Section 3.1), mixing multitier modules enables including the implementations of different subsystems into a single module.

LociMod separates modules from peers, i.e., mixing modules does not equate to unify the peers they define. Hence we need a way to coalesce different peers. We use (i) subtyping and (ii) overriding of abstract types as a mechanism to specify that a peer also comprises the placed values of (i) the super peers and (ii) the overridden peers, i.e., a peer subsumes the functionalities of its super peers (Section 3.2.1) and its overridden peers. Since peers are abstract type members, they can be overridden in sub modules. To demonstrate mixing of multitier modules we consider the case of two different functionalities.

First, we consider a computing scheme where a master offloads tasks to worker nodes:

```
1 @multitier trait MultipleMasterWorker[T] {
2   @peer type Master <: { type Tie <: Multiple[Worker] }
3   @peer type Worker <: { type Tie <: Single[Master] }
4
5   def run(task: Task[T]): Future[T] on Master = placed {
6     (remote(selectWorker()) call execute(task)).asLocal
7   }
8   private def execute(task: Task[T]): T on Worker = placed { task.process() }
9 }
```

The example defines a master that has a multiple tie to workers (Line 2) and a worker that has a single tie to a master (Line 3). The `run` method has the placed type `Future[T] on Master` (Line 5), placing `run` on the `Master` peer. Running a task remotely results in a `Future` [3] to account for processing time and network delays. The remote call to `execute` – to be executed on the worker – (Line 6) starts processing the task (Line 8). The remote result is transferred back to the master as `Future[T]` using `asLocal` (Line 6). A *single* worker instance in a pool of workers is selected for processing the task via the `selectWorker` method (Line 6, the implementation of `selectWorker` is omitted, for simplicity).

Second, we consider the case of monitoring, a functionality that is required in many distributed applications to react to possible failures [23]. In LociMod, a heartbeat mechanism can be defined across a `Monitored` and a `Monitor` peer in a multitier module:

```
1 @multitier trait Monitoring {
2   @peer type Monitor <: { type Tie <: Multiple[Monitored] }
3   @peer type Monitored <: { type Tie <: Single[Monitor] }
4
5   def monitoredTimedOut(monitored: Remote[Monitored]): Unit on Monitor
6
7   ...
8 }
```

The module defines the architecture with a single monitor and multiple monitored peers (Line 2 and 3). The `monitoredTimedOut` method (Line 5) is invoked by `Monitoring` implementations whenever a heartbeat was not received from a monitored peer instance for some time. We leave out the actual implementation of the monitoring logic for brevity.

To add monitoring to an application, such application has to be mixed with the `Monitoring` module. Mixing composition brings the members declared in all mixed-in modules into the local scope of the module that mixes in the other modules, i.e., all peer types of the mixed-in modules are in scope. However, the peer types of different modules define separate architectures, which can then be combined by specializing the peers of one module to the peers of other modules. For example, to add monitoring to the the `MultipleMasterWorker` functionality, `MultipleMasterWorker` needs to be mixed with `Monitoring` and the `Master` and `Worker` peers need to be overridden to be (subtypes of) `Monitor` and `Monitored` peers:

```
1 @multitier trait MonitoredMasterWorker[T] extends MultipleMasterWorker[T] with Monitoring {
2   @peer type Master <: Monitor { type Tie <: Multiple[Worker] with Multiple[Monitored] }
3   @peer type Worker <: Monitored { type Tie <: Single[Master] with Single[Monitor] }
4 }
```

Specializing peers of mixed modules follows the same approach as specializing peers accessible through module references (Section 3.2.1), i.e., `Master <: Monitor` specifies that a master is a monitor peer, augmenting the master with the monitor functionality. Also, for specialization using peers of mixed-in modules, the compiler checks that the combined architecture of the system complies to the architectural constraints of every subsystem.

3.2.3 Properties of Abstract Peer Types

LociMod abstract peer types share commonalities with both parametric polymorphism – considering type parameters as type members [29, 46] – like ML parameterized types [25] or Java generics [5], as well as subtyping in object-oriented languages. Similar to parametric polymorphism, abstract peer types allow parametric usage of peer types as shown for the `BackupService` module defining a `Storage` peer parameter. Distinctive from parametric polymorphism, however, with abstract peer types, peer parameters remain abstract, i.e., specializing peers does not unify peer types. Instead, similar to subtyping, specializing peers establishes an *is-a* relation.

Placement types `T on P` support suptyping between peers by being covariant in the type of the placed value and contravariant in the peer (i.e., the `on` type is defined as `type on[+T, -P]`), which allows values to be used in a context where a value of a super type placed on a sub peer is expected. This encoding is sound since a subtype can be used where a super type is expected and values placed on super peers are available on all sub peers. For example, we can extend

the `Editor` with a `WebClient`, which is a special kind of client (i.e., `WebClient <: Client`, Line 5) with a Web user interface (Line 8), and a `MobileClient` (i.e., Line 6):

```
1  @multitier trait Editor {
2    @peer type Server <: { type Tie <: Multiple[Client] }
3    ...
4    @peer type Client <: { type Tie <: Single[Server] }
5    @peer type WebClient <: Client { type Tie <: Single[Server] }
6    @peer type MobileClient <: Client { type Tie <: Single[Server] }
7
8    val webUI: UI on WebClient
9    val ui: UI on Client = placed { webUI }  // ✗ Error: `Client` not a subtype of `WebClient`
10 }
```

By using subtyping on peer types, not unifying the types, we are able to distinguish between the general `Client` peer, which can have different specializations (e.g., `WebClient` and `MobileClient`), i.e., every Web client is a client but not every client is a Web client. By keeping the types distinguishable, the `ui` binding (Line 9) is rejected by the compiler since it defines a value on the `Client` peer, i.e., the access to `webUI` inside the placed expression is computed on the `Client` peer. However, `webUI` is not available on `Client` since it is placed on `WebClient` and a client is not necessarily a Web client.

3.3 Constrained Multitier Modules

LociMod multitier modules not only allow abstraction over placement, but also the definition of *constrained multitier modules* that refer to other modules. This feature enables expressing constraints among the modules of a system, such as that one functionality is required to enable another. In LociMod, Scala's self-type annotations express such constraints, indicating which other module is required during mixin composition. To improve decoupling, constraints are often defined on module interfaces, such that multiple module implementations are possible.

Applications requiring constrained modules include distributed algorithms, discussed in more detail in the evaluation (Section 5.1). For example, a global locking scheme ensuring mutual exclusion for a shared resource can be implemented based on a central coordinator. Choosing a coordinator among connected peers requires a leader election algorithm. The `MutualExclusion` module declares a `lock` (Line 2) and `unlock` (Line 3) method for regulating access to a shared resource. `MutualExclusion` is constrained over `LeaderElection` since our locking scheme requires the leader election functionality:

```
1  @multitier trait MutualExclusion { this: LeaderElection =>
2    def lock(id: T): Boolean on Node
3    def unlock(id: Id): Unit on Node
4  }
```

Such requirement, expressed as a Scala self-type (Line 1), forces the developer to mix in a `LeaderElection` implementation to create instances of the `MutualExclusion` module.

A leader election algorithm can be defined by the following module interface:

```
1  @multitier trait LeaderElection[T] {
2    @peer type Node
3
4    def electLeader(): Unit on Node
5    def electedAsLeader(): Unit on Node
6  }
```

The module defines an `electLeader` method (Line 4) to initiate the leader election. The `electedAsLeader` method (Line 5) is called by `LeaderElection` module implementations on the peer instance that has been elected to be the leader.

All definitions of the `LeaderElection` module required by the self-type annotation are available in the local scope of the `MutualExclusion` module, which includes peer types and placed values. A self-type expresses a requirement but not a subtyping relation, i.e., we express the requirement on `LeaderElection` in the example as self-type since the `MutualExclusion` functionality requires leader election but is not a leader election module itself.

Multiple constraints can be expressed by using a compound type. For example, different peer instances often need to have unique identifiers to distinguish among them. Assuming an `Id` module provides such mechanism, a module which requires both the leader election and the identification functionality can specify both required modules as compound self-type `this: LeaderElection with Id`. Such requirement makes the definitions of both the `LeaderElection` and the `Id` module available in the module's local scope and forces the developer to mix in implementations for both modules.

Mixin composition is guaranteed by the compiler to conform to the self-type (which is the essence of the Scala *cake pattern*). Assuming a `YoYo` implementation of the `LeaderElection` interface which implements the Yo-Yo algorithm [39] (Section 5.1 presents different leader election implementations), the following code shows how a `MutualExclusion` instance can be created by mixing together `MutualExclusion` and `YoYo`:

```
1 @multitier object mutualExclusion extends MutualExclusion with YoYo
```

The `YoYo` implementation of the `LeaderElection` interface satisfies the `MutualExclusion` module's self-type constraint on the `LeaderElection` interface. Since mixing together `MutualExclusion` and `YoYo` fulfills all constraints and leaves no values abstract, the module can be instantiated.

3.4 Peer Startup

In the previous sections, we have shown how LociMod multitier *modules are instantiated*. To start up a distributed system, however, we also need to *start peers* defined in the modules. Different peer instances are typically started on different hosts and connect with each other over a network according to the architecture specification. As a consequence, an additional step is required to start the peers of (already instantiated) modules. For the master–worker example, the master and the worker peers are started as follows:

```
1  @multitier object masterWorker extends MultipleMasterWorker[Int]
2
3  object Master extends App {
4    multitier start new Instance[masterWorker.Master](
5      listen[masterWorker.Worker] { TCP(1099) })
6  }
7
8  object Worker extends App {
9    multitier start new Instance[masterWorker.Worker](
10     connect[masterWorker.Master] { TCP("localhost", 1099) })
11 }
```

We follow the idiomatic way of defining an executable Scala application, where an `object` extends `App` (Line 3 and 8). The object body is executed when the application starts. The code executed when staring a Scala application is standard (non-multitier) Scala, which, in our example, uses `multitier start Instance[...]` to start a peer of an instantiated multitier module. Line 1 instantiates a `MasterWorker` module using the `MultipleMasterWorker` implementation. Line 4 starts a `Master` peer of the module, which uses TCP to listen for connections from `Worker` peer instances. Line 9 starts a `Worker` peer of the module, which uses TCP to connect to a running `Master` peer instance.

4 Implementation

The implementation of LociMod required to modify ∼5 K LOCs of the ScalaLoci codebase. The ScalaLoci compilation process entails three main aspects [48]: (1) the type-level encoding of placement types into the Scala type system, (2) the compile-time macro-driven code separation of code belonging to different peers and (3) the injection of the communication code. The implementation of LociMod requires plugging into the steps above to introduce functionalities for module definition and composition as well as checks for architectural conformance. Both are discussed hereafter.

We preserve Scala's separate compilation because our implementation is based on Scala macros, which expand locally and cannot transform any other code than the annotated trait, class or object under expansion. Once modules are compiled, they are not recompiled unless their code or interfaces on which they depend change.

Macro Expansion. To enable distributed functionalities bundled in a multitier module (Section 3.1) to be executed on different machines (Section 3.4), our implementation separates multitier modules into peer-specific parts and replaces remote accesses with calls to the communication runtime, auto-generating the transmission boilerplate code. For the splitting, we rely on Scala annotation macros [8] (traits and objects are annotated with `@multitier`), transforming the type-checked abstract syntax tree[4] of the module. Placement types, specifying which values belong to which peer, have no direct semantic equivalent in plain Scala. The implementation splits multitier code based on placement types, thereby effectively erasing placement types from the generated code.

Listing 1 provides an intuition of how the macro expansion works, demonstrating module and peer composition as well as remote access. The LociMod code (Listing 1a) defines a module `A` with a peer `Peer` and a placed value `value`. Module `B` mixes in module `A` (Line 6), defines a reference to an instance of module `A` (Line 9), and accesses a remote value through the reference (Line 13).

In the expanded code (Listing 1b, simplified excerpt), placed values are annotated with `compileTimeOnly` (Line 3), which instructs the Scala compiler to issue an error in case such value is referenced in user code after macro expansion. The code generation creates `Marshallable` instances (Line 4) for network transmission of placed values and runtime identifiers for placed values (Line 5), modules (Line 16) and peers (Line 17) for dispatching remote accesses. The splitting process generates a `<placed values>` trait, which contains all placed values in the same order in which they appear in the multitier module to retain the initialization order. Values, however, are nulled (Line 9) and only initialized for the peer on which they are placed. Therefore, the splitting process generates an additional *peer trait* for every peer (Line 10), thus splitting multitier code into peer-specific components. Peer traits also handle local dispatching of remote requests, unmarshalling arguments and marshalling the return value (Line 14).

The example illustrates our module composition mechanisms. Mixing module `A` into module `B` results in the respective `<placed values>` and peer traits being mixed in (Line 25 and 34), using Scala mixin composition. For the `module` reference (Line 22, largely left out for brevity), both the generated module identifier (Line 22) and the dispatching logic for remote requests (Line 32) keep the path of the module reference (`"module"`) into account, to

[4] Annotation macros are expanded before type-checking but can explicitly invoke the type checker to obtain typed abstract syntax trees

Listing 1 Macro Expansion.

(a) LociMod user code.

```
1  @multitier trait A {
2    @peer type Peer
3    val value: Int on Peer
4  }
5
6  @multitier trait B extends A {
7    @peer type Peer <: module.Peer { type Tie <: Single[module.Peer] }
8
9    @multitier object module extends A {
10     val value: Int on Peer = placed { 42 }
11   }
12
13   val localValue: Local[Future[Int]] on Peer = placed { module.value.asLocal }
14 }
```

(b) Scala code after LociMod expansion.

```
1  trait A {
2    @peer type Peer
3    @compileTimeOnly("Remote access must be explicit.") val value: Int on Peer
4    @MarshallableInfo final val $loci$mar$A$0 = Marshallable[Int]
5    @PlacedValueInfo("value:scala.Int", null, $loci$mar$A$0) final val $loci$val$A$0 =
6      new PlacedValue[Unit, Unit, Future[Unit], Int, Int, Future[Int]](
7        Value.Signature("value:scala.Int", $loci$sig.path), true, null, $loci$mar$A$0)
8
9    trait `<placed values>` extends PlacedValues { val value: Int = null.asInstanceOf[Int] }
10   trait $loci$peer$Peer extends `<placed values>` {
11     def $loci$dispatch(req: MessageBuffer, sig: Value.Signature, ref: Value.Reference) =
12       if (sig.path.isEmpty) sig.name match {
13         case $loci$val$A$0.sig.name =>
14           Try(value) map { response => $loci$mar$A$0.marshal(response, ref) } ... } else ... }
15
16   lazy val $loci$sig = Module.Signature("A")
17   lazy val $loci$peer$sig$Peer = Peer.Signature("Peer", collection.immutable.Nil, $loci$sig)
18 }
19
20 trait B extends A {
21   @peer type Peer <: module.Peer { type Tie <: Single[module.Peer] }
22   object module extends A { lazy val $loci$sig = Module.Signature("B#module", "module") ... }
23   @compileTimeOnly("...") val remoteValue = null.asInstanceOf[Local[Future[Int]] on Peer]
24
25   trait `<placed values>` extends PlacedValues with super[A].`<placed values>` {
26     final lazy val module: B.this.module.`<placed values>` = $loci$multitier$module()
27     val remoteValue: Future[Int] = $loci$expr$B$0()
28     protected[this] def $loci$expr$B$0(): Future[Int] = null.asInstanceOf[Future[Int]]
29
30     def $loci$dispatch(req: MessageBuffer, sig: Value.Signature, ref: Value.Reference) =
31       if (sig.path.isEmpty) ... else sig.path.head match {
32         case "module" => module.$loci$dispatch(req, sig.copy(sig.name, sig.path.tail), ref) ... } }
33
34   trait $loci$peer$Peer extends `<placed values>` with super[A].$loci$peer$Peer {
35     protected[this] def $loci$multitier$module() = new B.this.module.$loci$peer$Peer { ... }
36     protected[this] def $loci$expr$B$0(): Future[Int] = SingleIntAccessor(RemoteValue)(
37       new RemoteRequest[Int from B.this.module.Peer, Future[Int], Peer, Single, Unit](
38         (), B.this.module.$loci$val$A$0, B.this.module.$loci$peer$sig$Peer, ...)).asLocal }
39 ... }
```

handle remote access to path-dependent modules. The `module` reference for the peer trait generated for module `B`'s `Peer` (Line 34) is instantiated to the peer trait generated for module `A`'s `Peer` (Line 35), so that values placed on module `B`'s `Peer` can access values placed on module `A`'s `Peer` since module `B` defines `Peer <: module.Peer`. Since peer types are used to guide the splitting and define the composition scheme of the synthesized peer traits, peer types themselves are never instantiated. Hence, they can be abstract.

Like `value` of module `A`, `localValue` of module `B` is nulled in the `<placed values>` trait (Line 27 and 28) and initialized in the generated peer trait (Line 36). Since `localValue` is defined local (i.e., not remotely accessible), no `Marshallable` instance or runtime identifier is generated for `localValue`. The remote access `module.value.asLocal` is expanded into a call to the communication backend with the remote value and remote peer identifiers as arguments (Lines 36–38).

As illustrated by the example, the code generation solely replaces the code of the annotated trait, class or object and only depends on the super traits and classes and the definitions in the multitier modules' body, thus retaining the same support for separate compilation offered by standard Scala traits, classes and objects.

Correctness Checks. Abstract peer types can be specialized, introducing further constraints on the architecture in which they are already involved. Our approach ensures that the architecture of the specialized peers does not violate the architectural constraints of the more general peers. Specifically, ties defined for a peer also need to be defined when specializing the peer, i.e., the tie of a peer needs to be a subtype of the ties of all super and overridden peers. It is, however, possible to refine a tie to make it more specific (i.e., a *multiple* tie is the most general from, whereas an *optional* tie is more specific and a *single* tie is the most specific form). For example, when specializing a `Server` peer with a `Multiple[Client]` tie to a `WebServer <: Server` peer, the `WebServer` also needs to specify the tie to the `Client`. It can specify the type as `Multiple[Client]` (like its super peer), but it can also specify a more specific tie, e.g., `Single[Client]`. Refining ties is sound since, if code placed on a peer is able to handle any number of connected remote instances (multiple tie), particularly, it can also handle the case when at most one instance is connected (optional or single tie) – but not the other way around.

5 Evaluation

The objective of the evaluation is to assess the design goals established in Section 3, answering the following research questions:

RQ1 Do multitier modules enable defining reusable patterns of interaction in distributed software?
RQ2 Do multitier modules enable separating the modularization and distribution concerns?

For RQ1, we first consider **distributed algorithms** as a case study. Distributed algorithms are a suitable case study because – as we explain soon – they depend on each other and on the underlying architecture. Yet, one wants to keep each algorithm modularized in a way that algorithms can be freely composed. Second, we show how **distributed data structures** can be implemented in LociMod. This case study requires to hide the internal behavior of the data structure from user code as well as to provide a design that does not depend on the specific system architecture. For RQ2, we evaluate the applicability of LociMod to existing real-word software. We reimplemented the *task distribution system* of the Apache Flink **distributed stream processing** framework introduced in Section 1 using multitier modules.

5.1 Distributed Algorithms

We present a case study on a distributed algorithm for mutual exclusion through global locking to access a shared resource. As global locking requires a leader election algorithm, we implement different election algorithms as reusable multitier modules. Also, leader

Listing 2 Mutual Exclusion.

```
1  @multitier trait MutualExclusion[T] { this: Architecture with LeaderElection[T] =>
2    private var locked: Option[T] localOn Node = placed { None }
3
4    def lock(id: T): Boolean on Node = placed {
5      if (state == Leader && locked.isEmpty) {
6        locked = Some(id)
7        true
8      }
9      else
10       false
11   }
12
13   def unlock(id: Id): Unit on Node = placed {
14     if(state == Leader && locked == Some(id))
15       locked = None
16   }
17 }
```

election algorithms assume different distributed architectures, which we represent as multitier modules, too. The implemented mechanism relies on a central coordinator (Listing 2). The `MutualExclusion` module is parameterized over the leader election algorithm using constrained multitier mixing by specifying a requirement on the `LeaderElection` interface (Line 1) abstracting over concrete leader election implementations. `LeaderElection` provides the `state` method (Line 5 and 14) indicating whether the local node is the elected leader. The `MutualExclusion` module defines the `lock` (Line 4) and the `unlock` (Line 13) methods, to acquire and release the lock.

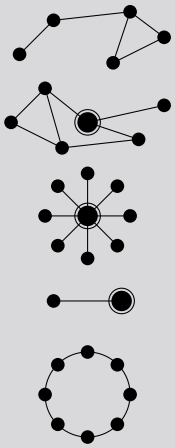
System Architectures. The `MutualExclusion` module (Listing 2) specifies a constraint on `Architecture` (Line 1) requiring any distributed architecture for the system abstracting over a concrete one. `Architecture` is the base trait for different distributed architectures expressed as reusable modules. Listing 3 shows the definitions for different architectures with their iconification on the right. The `Architecture` module defines the general `Node` peer and the constraint that peers of type `Node` are connected to an arbitrary number of other `Node` peers. The `P2P` module defines a `Peer` that can connect to arbitrary many other peers. Thus, the P2P is essentially the general architecture since nodes connecting in a P2P fashion do not impose any additional architectural constraints. The `P2PRegistry` module adds a central registry, to which peers can connect. The `MultiClientServer` module defines a client that is always connected to single server, while the server can handle multiple clients simultaneously. The `ClientServer` module specifies a server that always handles a single client instance. For the `Ring` module, we define a `Prev` and a `Next` peer. A `RingNode` itself is both a predecessor and a successor. All `Node` peers have a single tie to their predecessor and a single tie to their successor.

Leader Election. We present the `LeaderElection` interface for a generic leader election algorithm in LociMod. Since leader election differs depending on the network architecture, the interface defines a self-type constraint on `Architecture`, abstracting over the concrete network architecture constraining multitier mixing:

```
1 @multitier trait LeaderElection[T] { this: Architecture with Id[T] =>
2   def state: State on Node
3   def electLeader(): Unit on Node
4   def electedAsLeader(): Unit on Node
5 }
```

Listing 3 Distributed Architectures.

```
1  @multitier trait Architecture {
2    @peer type Node <: { type Tie <: Multiple[Node] }
3  }
4  @multitier trait P2P extends Architecture {
5    @peer type Peer <: Node { type Tie <: Multiple[Peer] }
6  }
7  @multitier trait P2PRegistry extends P2P {
8    @peer type Registry <: Node { type Tie <: Multiple[Peer] }
9    @peer type Peer <: Node { type Tie <: Optional[Registry] with Multiple[Peer] }
10 }
11 @multitier trait MultiClientServer extends Architecture {
12   @peer type Server <: Node { type Tie <: Multiple[Client] }
13   @peer type Client <: Node { type Tie <: Single[Server] with Single[Node] }
14 }
15 @multitier trait ClientServer extends MultiClientServer {
16   @peer type Server <: Node { type Tie <: Single[Client] }
17   @peer type Client <: Node { type Tie <: Single[Server] with Single[Node] }
18 }
19 @multitier trait Ring extends Architecture {
20   @peer type Node <: { type Tie <: Single[Prev] with Single[Next] }
21   @peer type Prev <: Node
22   @peer type Next <: Node
23   @peer type RingNode <: Prev with Next
24 }
```



Further, the interface abstracts over a mechanism for assigning IDs to nodes implemented by the `Id[T]` module, where `T` is the type of the IDs. The `Id` module interface defines a local `id` value on every node and requires an ordering relation for IDs:

```
1 @multitier abstract class Id[T: Ordering] { this: Architecture =>
2   val id: Local[T] on Node
3 }
```

The `LeaderElection` module defines a local variable `state` that captures the state of each peer (e.g., `Candidate`, `Leader` or `Follower`). The `electLeader` method is kept abstract to be implemented by a concrete implementation of the interface. After a peer instance has been elected to be the leader, implementations of `LeaderElection` call `electedAsLeader`. We consider three leader election algorithms:

Hirschberg-Sinclair Leader Election The Hirschberg-Sinclair algorithm [21] implements leader election for a ring topology. In every algorithm phase, each peer instance sends its ID to both of its neighbors in the ring. IDs circulate and each node compares the ID with its own. The peer with the greatest ID becomes the leader. The logic of the algorithm is encapsulated into the `HirschbergSinclair` module, which extends `LeaderElection`:

```
1 @multitier trait HirschbergSinclair[T] extends LeaderElection[T] { this: Ring with Id[T] =>
2   def electLeader() = placed[Node] { elect(0) }
3   private def elect(phase: Int) = placed[Node] { /* ... */ }
4   private def propagate(remoteId: T, hops: Int, direction: Direction) = placed[Node] { /* ... */ }
5 }
```

The module's self-type encodes that the algorithm is designed for ring networks (Line 1). When a new leader election is initiated by calling `electLeader` (Line 2), the `elect` method is invoked (Line 3). The `propagate` method passes the IDs of peer instances along the ring and compares them with the local ID.

Yo-Yo Leader Election The Yo-Yo algorithm [39] is a universal leader election protocol, i.e., it is independent of the network architecture. For this reason, the self-type constraint of the `YoYo` implementation is simply `Architecture with Id[T]`. In the Yo-Yo algorithm,

each node exchanges its ID with all neighbors, progressively pruning subgraphs where there is no lower ID. The node with the lower ID becomes the leader.

Raft Leader Election The Raft consensus algorithm [32] elects a leader by making use of randomized timeouts. Once a leader is elected, it maintains its leadership by sending heartbeat messages to all peer instances. If instances do not receive a heartbeat message from the current leader for a certain amount of time, they initiate a new election.

Instantiating Global Locking. The following code instantiates a `MutualExclusion` module using the Hirschberg-Sinclair leader election algorithm for a ring architecture:

```
1 @multitier object locking extends
2   MutualExclusion[Int] with HirschbergSinclair[Int] with Ring with RandomIntId
```

The example mixes in a module implementation for every module over which other modules are parameterized, i.e., `MutualExclusion` is parameterized over `LeaderElection`, which is instantiated to `HirschbergSinclair`. `HirschbergSinclair` requires a `Ring` architecture and an `Id` implementation, which is instantiated to a `RandomIntId` module (whose implementation is left out for brevity). The following code, instead, instantiates a `MutualExclusion` module using the Yo-Yo leader election algorithm for a P2P architecture:

```
1 @multitier object locking extends
2   MutualExclusion[Int] with YoYo[Int] with P2P with RandomIntId
```

Summary. The case study demonstrates how module implementations for concrete architectures and leader election algorithms can be composed into a module providing global locking and made reusable. Since modules encapsulate a functionality within a well-defined interface, leader election algorithms can be easily exchanged. Our approach allows for simply mixing different cross-peer functionality together without changing any multitier code that is encapsulated into the modules (RQ1).

5.2 Distributed Data Structures

This section demonstrates how distributed data structures can be implemented in LociMod. First, we reimplement non-multitier conflict-free replicated data types (CRDTs) as multitier modules in LociMod. Second, we compare to an existing multitier cache originally implemented in Eliom [36].

Conflict-Free Replicated Data Types. Conflict-free replicated data types (CRDT) [42, 43] offer eventual consistency across replicated components for specific data structures, avoiding conflicting updates by design. With CRDTs, updates to shared data are sent asynchronously to the replicas and *eventually* affect all copies. Such *eventually consistent* model [47] provides better performance (no synchronization is required) and higher availability (each replica has a local copy which is ready to use). We reimplemented several CRDTs, publicly available in Scala[5], in LociMod (Table 1).

We discuss the representative case of the `GSet`. G-Sets (grow-only sets) are sets which only support adding elements; elements cannot be removed. A *merge* operation computes the union of two G-Sets. Listing 4a1 shows the G-Set in Scala. `GSet` defines a set `content` (Line 3) and a method to check if an element is in the set (Line 5). Adding an element

[5] `http://github.com/lihaoyi/crdt`

Table 1 Common Conflict-Free Replicated Data Types.

CRDT	Description	Lines of Code		Remote Accesses
		Scala *local*	LociMod *distrib.*	
G-Counter	Grow-only counter. Only supports incrementing.	14	15	1
PN-Counter	Positive-negative counter. Supports incrementing and decrementing.	13	14	1
LWW-Register	Last-write-wins register. Supports reading and writing a single value.	10	11	1
MV-Register	Multi-value register. Supports writing a single value. Reading may return a set of multiple values that were written concurrently.	12	13	1
G-Set	Grow-only set. Only supports addition.	7	9	1
2P-Set	Two-phase set. Supports addition and removal. Removed elements cannot be added again.	13	17	2
LWW-Element-Set	Last-write-wins set. Supports addition and removal. Associates each added and removed element to a time stamp.	15	19	2
PN-Set	Positive-negative set. Supports addition and removal. Associates a counter to each element, incrementing/decrementing the counter upon addition/removal.	12	16	2
OR-Set	Observed-removed set. Supports addition and removal. Associates a set of added and of removed (unique) tags to each element. Adding inserts a new tag to the added tags. Removing moves all tags associated to an element to the set of removed tags.	15	18	2

inserts it into the local `content` set (Line 7). Listing 4b presents a multitier module for a multitier G-Set. The implementations are largely similar despite that the LociMod version is distributed and the Scala version is not. The Scala CRDTs are only local. Distributed data replication has to be implemented by the developer (Listing 4a2).

In the LociMod variant, the peer type of the interacting nodes is abstract, hence it is valid for any distributed architecture. The LociMod multitier module can be instantiated by applications for their architecture:

```
1  @multitier trait EventualConsistencyApp {
2    @peer type Server <: ints.Node with strings.Node {
3      type Tie <: Single[Client] with Single[ints.Node] with Single[strings.Node] }
4    @peer type Client <: ints.Node with strings.Node {
5      type Tie <: Single[Server] with Single[ints.Node] with Single[strings.Node] }
6
7    @multitier object ints extends GSet[Int]
8    @multitier object strings extends GSet[String]
9
10   on[Server] { ints.add(42) }
11   on[Client] { strings.add("forty-two") }
12 }
```

The example defines a `GSet[Int]` (Line 7) and a `GSet[String]` (Line 8) instance. The `Server` and a `Client` peer are also `ints.Node` and `strings.Node` peers (Line 2 and 4) and are tied to other `ints.Node` and `strings.Node` peers (Line 3 and 5). Thus, both the server (Line 10) and the client (Line 11) can use the multitier module references `ints` and `strings` to add elements to both sets, which (eventually) updates the sets on the connected nodes. The plain Scala version, in contrast, does not offer abstraction over placement.

Listing 4 Conflict-Free Replicated Grow-Only Set.

(a) Scala implementation.

(a1) Traditional G-Set implementation.

```
1  class GSet[T] {
2    val content =
3      mutable.Set.empty[T]
4    def contains(v: T) =
5      content.contains(v)
6    def add(v: T) =
7      content += v
8
9    def merge(other: GSet[T]) =
10     content ++= other.content
11 }
```

(a2) Example of user code for distribution.

```
1  trait Host[T] {
2    val set = new GSet[T]
3    def add(v: T) = {
4      set.add(v)
5      send(set.content) }
6    def receive(content: T) =
7      set.merge(content)
8  }
```

(b) LociMod implementation.

```
1  @multitier trait GSet[T] extends Architecture {
2    val content = on[Node] {
3      mutable.Set.empty[T] }
4    def contains(v: T) = on[Node] {
5      content.contains(v) }
6    def add(v: T) = on[Node] {
7      content += v
8      remote call merge(content.toSet) }
9    private def merge(content: Set[T]) = on[Node] {
10     this.content ++= content }
11 }
```

In addition to be more concise, the LociMod version exhibits a better design thanks to the combination of multitier programming and modules. In plain Scala, the actual replication of added elements by propagating them to remote nodes is mingled with the user code: The Scala versions of all CRDTs transfer updated values explicitly to merge them on the replicas, i.e., `merge` needs to be public to user code. The *Remote Accesses* column in Table 1 counts the methods that mix replication logic and user code.

Listing 4a2 shows the user code adding an element to the local G-Set (Line 4), sending the content to a remote host (Line 5), receiving the content remotely (Line 6) and merging it into the remote G-Set (Line 7). In contrast, adding an element to the LociMod `GSet` (Listing 4b) directly merges the updated set into all connected remote nodes (Line 8 and 10). The multitier module implicitly performs remote communication between different peers (Line 8), encapsulating the remote access to the replicas, i.e., `merge` is private.

Distributed Caching. We implement a cache that is shared between a client–server architecture. It is modeled after Eliom's multitier cache holding the values already computed on a server [36]. The following code presents the cache using LociMod multitier modules:

```
1  @multitier trait Cache[K, V] extends MultiClientServer {
2    private val table = on[Node] { mutable.Map.empty[K, V] }
3
4    on[Client] {
5      table.asLocal foreach { serverTable =>
6        table ++= serverTable } }
7
8    def add(key: K, value: V) = on[Node] { table += key -> value }
9  }
```

The `Cache` module is implemented for a client–server architecture (Line 1). The `table` map (Line 2) is placed on every `Node`, i.e., on the client and the server peer. The `add` method adds an entry to the map (Line 8). As soon as the client instance starts, the client populates its local map with the content of the server's map (Line 6).

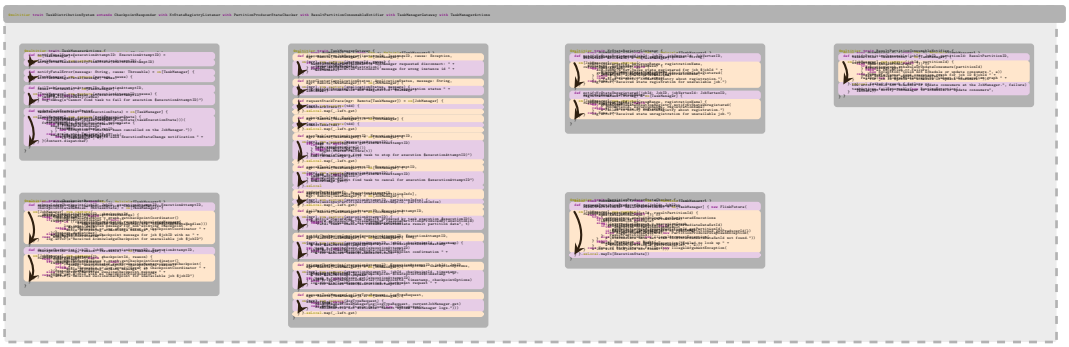


Figure 2 Example communication in Flink using LociMod multitier modules.

LociMod's multitier model is more expressive than Eliom's as it allows the definition of arbitrary peers through placement types. Placement types enable abstraction over placement, as opposed to Eliom, which only supports two fixed predefined places (server and client). LociMod supports Eliom's client–server model (Line 1) as a special case. Thanks to LociMod's abstract peer types, the `Cache` module can also be used for other architectures. For example, we can enhance the `Peer` and `Registry` peers of a P2P architecture with the roles of the client and the server of the `Cache` module by mixing `Cache` and `P2PRegistry` and composing both architectures:

```
1 @multitier trait P2PCache[K, V] extends Cache[K, V] with P2PRegistry {
2   @peer type Registry <: Server { type Tie <: Multiple[Peer] with Multiple[Client] }
3   @peer type Peer <: Client { type Tie <: Single[Registry] with Single[Server] with Multiple[Peer] }
4 }
```

Summary. The case studies demonstrate that, thanks to the multitier module system, distributed data structures can be expressed as reusable modules that can be instantiated for different architectures encapsulating all functionalities needed for the implementation of the data structure (RQ1).

5.3 Apache Flink

The *task distribution system* of the Apache Flink stream processing framework [9], provides Flink's core task scheduling and deployment logic. It is based on Akka actors and consists of six *gateways* (an API that encapsulates sending and receiving actor messages) amounting to ∼500 highly complex Scala LOC. Gateways wrap method arguments into messages, sending the message and (potentially) receiving a different message carrying a result.

With the current Flink design, however, code fragments that are executed on different distributed components (i.e., for sending and receiving a message), inevitably belong to different actors. The functionalities that conceptually belong to a single gateway are scattered over multiple files in the Flink implementation, breaking modularization. The messages sent by the actors in every gateway are hard to follow for developers because matching sending and receiving operations are completely separated in the code. 19 out of the 23 sent messages are processed in a different compilation unit within another package, hindering the correlation of messages with the remote computations they trigger.

Listing 5 Remote communication in Flink.

(a) Original Flink implementation.

(a1) Message definition.

```
package flink.runtime

case class SubmitTask(td: TaskDeployment)
```

(a2) Calling side.

```
package flink.runtime.job

case class SubmitTask(td: TaskDeployment)

class TaskManagerGateway {
  def submitTask(
      td: TaskDeployment,
      mgr: ActorRef) =
    (mgr ? SubmitTask(td))
     .mapTo[Acknowledge]
}
```

(a3) Responding side.

```
package flink.runtime.task

class TaskManager extends Actor {
  // standard Akka message loop
  def receive = {
    case SubmitTask(td) =>
      val task = new Task(td)
      task.start()
      sender ! Acknowledge()
} }
```

(b) Refactored LociMod implementation.

```
package flink.runtime.multitier

@multitier object TaskManagerGateway {
  @peer type JobManager <: {
    type Tie <: Multiple[TaskManager] }
  @peer type TaskManager <: {
    type Tie <: Single[JobManager] }

  def submitTask(
      td: TaskDeployment, tm: Remote[TaskManager]) =
    on[JobManager] {
      (remote(tm) call startTask(td)).asLocal
    }
  def startTask(td: TaskDeployment) =
    on[TaskManager] {
      val task = new Task(td)
      task.start()
      Acknowledge()
    }
}
```

We reimplemented the task distribution system using multitier modules, to cover the complete cross-peer functionalities that belong to each gateway. The resulting modules are (1) the `TaskManagerGateway` to control task execution, (2) the `TaskManagerActions` to notify of task state changes, (3) the `CheckpointResponder` to acknowledge checkpoints, (4) the `KvStateRegistryListener` to notify key-value store changes, (5) the `PartitionProducerStateChecker` to check of the state of producers and of result partitions and (6) the `ResultPartitionConsumableNotifier` to notify of available partitions. Since the different cross-peer functionalities of the task distribution system are cleanly separated into different modules, the complete `TaskDistributionSystem` application is simply the composition of the modules 1–6 that implement each subsystem:

```
1 @multitier trait TaskDistributionSystem extends
2     CheckpointResponder with KvStateRegistryListener with PartitionProducerStateChecker with
3     ResultPartitionConsumableNotifier with TaskManagerGateway with TaskManagerActions {
4   @peer type JobManager <: { type Tie <: Multiple[TaskManager] }
5   @peer type TaskManager <: { type Tie <: Single[JobManager] with Single[TaskManager] }
6 }
```

We mix together the subsystem modules (Line 2 and 3) and specify the architecture of the complete task distribution system (Line 4 and 5). As all subsystems share the same architecture, it is not necessary to specify the architecture in the `TaskDistributionSystem` module (as we did in the example code). Instead, it suffices to specify the architecture in the mixed-in modules.

Compared to Figure 1b, which merges the functionalities of all subsystems into a single compilation unit, the LociMod version using multitier modules encapsulates each functionality into a separate module. Figure 2 shows the `TaskDistributionSystem` module (background),

composed by mixing together the subsystem modules (foreground). The multitier modules contain code for the `JobManager` and the `TaskManager` peer. Arrows represent cross-peer data flow, which is encapsulated within modules and is not split over different modules. Importantly, even modules that place all computations on the same peer (e.g., the module containing only dark violet boxes) define remote accesses (arrows), i.e., different instances of the same peer type (e.g., the dark violet peer) communicate with each other.

It is instructive to look into the details of one of the modules. Listing 5 shows an excerpt of the – extensively simplified – `TaskManagerGateway` functionality for Flink (left) and its reimplementation in LociMod (right) side-by-side, focusing only on a single remote access of a single gateway. The example concerns the communication between the `TaskManagerGateway` used by the `JobManager` and the `TaskManager` – specifically, the job manager's submission of tasks to task managers. In the actor-based version (Listing 5a), this functionality is scattered over different modules hindering correlating sent messages (Listing 5a2, Line 9) with the remote computations they trigger (Listing 5a3, Line 7–9) by pattern-matching on the received message (Listing 5a3, Line 6). The LociMod version (Listing 5b) uses an intra-module cross-peer remote call (Line 12), explicitly stating the method for the remote computation (Line 16–18). Hence, in LociMod, there is no splitting over different actors as in the Flink version, thus keeping related functionalities inside the same module. The `TaskManagerGateway` multitier module contains a functionality that is executed on both the `JobManager` and the `TaskManager` peer. Further, the message loop of the `TaskManager` actor of Flink (Listing 5a3), does not only handle the messages belonging to the `TaskManager-Gateway` (shown in the code excerpt). The loop also needs to handle messages belonging to the other gateways – which execute parts of their functionality on the `TaskManager` – since modularization is imposed by the remote communication boundaries of an actor.

Summary. In summary, in the case study, the multitier module system enables decoupling of modularization and distribution as LociMod multitier modules capture cross-network functionalities expressed by Flink gateways without being constrained to modularization along network boundaries (RQ2).

6 Related Work

There is a long history of research concerned with proper software modularization mechanisms [33]. We organize related work as follows. First we discuss multitier languages. Second, we present recent advances in module systems. Third, we discuss approaches that partially combine the two solutions. Finally, we provide an overview of related research areas, including languages for distributed systems and component-based software development.

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 [40] and Hop.js [41] are dynamically typed languages that follow a traditional client–server communication scheme with asynchronous callbacks. They do not ensure static guarantees for the behavior of the distributed system. In Links [14] and Opa [37], functions are annotated to specify either client- or server-side execution. Both languages also follow the client–server model and feature a static type system. Links' server is stateless for scalability reasons – limiting the spectrum of the supported domains. In StiP.js [34], 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 LociMod, developers specify placement *in types*, enabling architectural reasoning.

Ur/Web [13], a multitier language for the Web, supports the standard ML module system. By requiring whole-program optimizations to slice the program into client and server parts, Ur/Web modules do not support separate compilation. The Eliom module system [35, 36] is also based on ML modules. It supports *mixed modules*—in Eliom terminology—which can contain declarations for both the server and the client and are similar to LociMod multitier modules that can also contain declarations for different peers (Section 3). Like LociMod modules, Eliom modules feature separate compilation. Due to the restriction to client–server applications, Eliom lacks language abstractions for architectural specifications and distributed system with multiple peers. More interestingly, Eliom modules do not support abstract peer types, hence it is not possible to specify the module functionalities over abstract peers and use such module to specialize the peers in another application (Section 3.2). For the same reason module composition does not support combining different architectures. All approaches above focus on the Web, contrarily to our goal of supporting other architectures. An exception is ML5 [26], a multitier language for generic software architectures: *Possible worlds*, as known from modal logic, address the purpose of placing computations and, similar to LociMod, are part of the type. ML5, however, does not support architecture specification, i.e., it does not allow for expressing different architectures in the language and was anyway applied only to the Web setting so far.

Module Systems. Rossberg and Dreyer design MixML, a module system that supports higher-order modules and modules as first-class values and combines ML modules' hierarchical composition with mixin's recursive linking of separately compiled components [38]. There are some commonalities in the way LociMod uses Scala traits as module interfaces – similar to ML signatures – and objects as module instances – similar to ML structures. Further, traits also support separate compilation. MixML signatures, like standard ML signatures, are structural types. In contrast, mixin composition in Scala operates on traits [15], which are nominal. LociMod, being a Scala embedding, inherits the modularization approach of using traits from Scala, but decouples it from distribution concerns.

Implicit resolution enables retroactive extensibility in the style of Hakell's type classes using the *concept pattern* [31]. The Genus programming language provides modularization abstractions that support generic programming and retroactive extension in the OO setting in a way similar to the concept pattern [49, 50]. Type classes do not support different instances for the same type and the concept pattern's encoding for type classes also requires unambiguous instances (or requires manual disambiguation otherwise). In contrast to Haskell type classes and similar to Genus, LociMod's approach to modularization using Scala traits as modules enables different implementations of the same trait.

Family polymorphism explores definition and composition of module hierarchies. The J& language supports hierarchical composability for nested classes in a mixin fashion [27, 28]. Nested classes are also supported by Newspeak, a dynamically typed object-oriented language [6]. Virtual classes [18] enable large-scale program composition through family polymorphism. Dependent classes [20] generalize virtual classes to multiple dispatching (i.e., class constructors are dynamically dispatched and are multimethods). Mixin composition is supported directly in Scala and it is used in LociMod as a way to compose multitier modules (Section 3.2.2). Virtual classes can be encoded in Scala [30], opening an interesting research direction that investigates multitier modules and family polymorphism.

Programming Languages and Calculi for Distributed Systems. Partitioned Global Address Space languages (PGAS), such as X10 [17], support high-performance parallel execution. These languages define a globally shared address space aiming to reduce boundaries among

hosts, similar to multitier languages. In X10, dependent *placed types* [11] identify processing locations ensuring that objects do not cross the boundaries of locations. Instead, our approach abstracts over peer instances (of the same type), to refer uniformly to all similar peers.

Several formal calculi model distributed systems and abstract, to various degrees, over placement and remote communication. The Ambient calculus [10] models concurrent systems with both mobile devices and mobile computation. In Ambient, it is possible to define named, bounded places where computations occur. Ambients can be moved to other places and are nested to model administrative domains and their access control. The Join calculus [19] defines processes that communicate by asynchronous message passing over channels in a way that models an implementation without expensive global consensus. However, we are not aware of higher level modularization abstractions built on top of the Join calculus. CPL [7] is a core calculus for combining services in the cloud computing environment. CPL is event-based and provides combinators that allow safe composition of cloud applications. Similar to LociMod, such combinators are generic with respect to placement and can be parameterized and constrained over other combinators. However, in CPL, there is no notion of architectural specification that can be used to check that multitier module composition adheres to the desired architecture.

Software Architectures and Component-based Software Development. ArchJava [1] unifies architectural definition and implementation in one language. Multitier modules and ties are similar to ArchJava components and connections. Also, similar to LociMod, ArchJava supports connections over a network and gives additional type safety guarantees compared to pure Java, e.g., that values sent over a network are serializable [2]. Different from LociMod modules, which can be parametric and can be mixed in, ArchJava's components can be only composed through connections.

Architecture description languages (ADL) are DSLs designed to support architecture evolution. ADLs define components, connectors, architectural invariants and a mapping of architectural models to an implementation infrastructure [24]. Influenced by ADLs and object-oriented design, component models [16] provide techniques and technologies to build software systems starting from units of composition with contractually specified interfaces and dependencies which can be deployed independently [44]. Component-based development (CBD) aims at separating different concerns throughout the whole software system, defining component interfaces for interaction with other components and mechanisms for composing components, which is similar to LociMod's approach of separating different functionalities into modules providing strong interfaces to other modules. The encapsulated functionalities in LociMod modules, however, can be distributed themselves, whereas CBD in a distributed setting usually models the different components of the distributed system as separate components, forcing developers to modularize along network boundaries.

7 Conclusion

Current multitier languages lack abstractions to properly modularize large code bases. In this paper, we presented LociMod, a *multitier module system* for ScalaLoci that allows developers to modularize multitier code, enabling encapsulation and code reuse. Thanks to abstract peer types, LociMod multitier modules capture abstract patterns of interaction among components in the system, enabling their composition and the definition of module constraints.

Our evaluation on distributed algorithms, distributed data structures, and the Apache Flink big data processing framework, shows that the LociMod's multitier module system is effective in properly modularizing multitier code.

References

1. Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, New York, NY, USA, 2002. ACM.
2. Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language support for connector abstractions. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, ECOOP '03, Berlin, Heidelberg, 2003. Springer.
3. Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Notices*, 12(8), 1977.
4. Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, OOPSLA/ECOOP '90, New York, NY, USA, 1990. ACM.
5. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, New York, NY, USA, 1998. ACM.
6. Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP '10, Berlin, Heidelberg, 2010. Springer-Verlag.
7. Oliver Bračevac, Sebastian Erdweg, Guido Salvaneschi, and Mira Mezini. CPL: A core language for cloud computing. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY '16, New York, NY, USA, 2016. ACM.
8. Eugene Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, New York, NY, USA, 2013. ACM.
9. Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38, 2015.
10. Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1), 2000.
11. Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, New York, NY, USA, 2008. ACM.
12. James Cheney and Ralf Hinze. Phantom types. Technical report, Cornell University, 2003.
13. Adam Chlipala. Ur/Web: A simple model for programming the web. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, New York, NY, USA, 2015. ACM.
14. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO '06, Berlin, Heidelberg, 2007. Springer-Verlag.
15. Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for scala type checking. In *Proceedings of the 31st International Conference on Mathematical Foundations of Computer Science*, MFCS '06, Berlin, Heidelberg, 2006. Springer-Verlag.
16. Ivica Crnkovic, Severine Sentilles, Vulgarakis Aneta, and Michel R. V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5), September 2011.
17. Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Computing Surveys*, 47(4), May 2015.
18. Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, New York, NY, USA, 2006. ACM.

19 Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, New York, NY, USA, 1996. ACM.

20 Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, New York, NY, USA, 2007. ACM.

21 D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11), 1980.

22 Barbara Liskov. Keynote address – data abstraction and hierarchy. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, New York, NY, USA, 1987. ACM.

23 Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

24 Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, New York, NY, USA, 1999. ACM.

25 R. Milner, L. Morris, and M. Newey. A logic for computable functions with reflexive and polymorphic types. In *Proceedings of the Conference on Proving and Improving Programs*, Arc-et-Senans, 1975.

26 Tom Murphy, VII., Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing*, TGC '07, Berlin, Heidelberg, 2008. Springer-Verlag.

27 Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, New York, NY, USA, 2004. ACM.

28 Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, New York, NY, USA, 2006. ACM.

29 Martin Odersky, Guillaume Martres, and Dmitry Petrashko. Implementing higher-kinded types in dotty. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala*, SCALA '16, New York, NY, USA, 2016. ACM.

30 Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, New York, NY, USA, 2005. ACM.

31 Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, New York, NY, USA, 2010. ACM.

32 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC '14, Berkeley, CA, USA, 2014. USENIX Association.

33 D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), December 1972.

34 Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. Towards tierless web development without tierless languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, New York, NY, USA, 2014. ACM.

35 Gabriel Radanne and Jérôme Vouillon. Tierless modules. 2017.

36 Gabriel Radanne and Jérôme Vouillon. Tierless web programming in the large. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

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

38 Andreas Rossberg and Derek Dreyer. Mixin' up the ML module system. *ACM Transactions on Programming Languages and Systems*, 35(1), April 2013.

39 Nicola Santoro. *Design and analysis of distributed algorithms*, volume 56. John Wiley & Sons, 2006.

40 Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: A language for programming the web 2.0. In *Companion to the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Companion to OOPSLA '06, New York, NY, USA, 2006. ACM.

41 Manuel Serrano and Vincent Prunet. A glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP '16, New York, NY, USA, 2016. ACM.

42 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt; INRIA, January 2011.

43 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS '11, Berlin, Heidelberg, 2011. Springer-Verlag.

44 Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

45 Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, New York, NY, USA, 1999. ACM.

46 Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of the 11th European Conference on Object-oriented Programming*, ECOOP '97, Berlin, Heidelberg, 1997. Springer-Verlag.

47 Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1), January 2009.

48 Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with ScalaLoci. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA '18), 2018.

49 Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. Lightweight, flexible object-oriented generics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, New York, NY, USA, 2015. ACM.

50 Yizhou Zhang and Andrew C. Myers. Familia: Unifying interfaces, type classes, and family polymorphism. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017.