Distributed System Development with ScalaLoci

Pascal Weisenburger, Mirko Köhler, Guido Salvaneschi
TU Darmstadt, Germany
Multitier Languages

Single Compilation Unit
ScalaLoci

Generic Distributed Architectures

Placement Types

Multitier Event Streams
trait Registry extends Peer
trait Node extends Peer

val message: Event[String] on Registry
  = placed { getMessageStream() }
trait Registry extends Peer { type Tie <: Multiple[Node] }
trait Node extends Peer { type Tie <: Single[Registry] with Multiple[Node] }
trait Registry extends Peer { type Tie <: Single[Node] }

trait Node extends Peer { type Tie <: Single[Registry] }

val message: Event[String] on Node

placed[Registry] {
  message.asLocal: Event[String]
}

Node -- Registry
trait Registry extends Peer { type Tie <: Multiple[Node] }
trait Node extends Peer { type Tie <: Single[Registry] }

val message: Event[String] on Node

placed[Registry] {
  message.asLocalFromAll: Map[Remote[Node], Event[String]]
}
Communication
val message = Event[String]()
val result = message map processMessage
val ui = new UI(result)
val message: Event[String] on Node = placed[Node] { Event[String](){ } }
val result = placed[Registry] { message.asLocal map processMessage }
val ui = placed[Node] { new UI(result.asLocal) }
@multitier object Chat {
  trait Registry extends Peer { type Tie <: Multiple[Node] }
  trait Node extends Peer { type Tie <: Single[Registry] }

  val message = placed[Node] { Event[String]() }

  val publicMessage = placed[Registry] {
    message.asLocalFromAllSeq map { case (_, msg) => msg }
  }

  placed[Node].main {
    publicMessage.asLocal.observe println
    for (line <- io.Source.stdin.getLines)
      message.fire(line)
  }
}
Evaluation

Two existing systems

Case studies: 22 variants
### Porting to Distribution

<table>
<thead>
<tr>
<th>Local</th>
<th>ScalaLoci</th>
<th>Akka</th>
<th>RMI</th>
</tr>
</thead>
</table>

- **multi-user support**
- **distribution**

---

**Porting to Distribution**

- **ScalaLoci**
  - Multi-user support
  - Distribution

- **Akka**
  - Multi-user support
  - Distribution

- **RMI**
  - Multi-user support
  - Distribution
Gearpump Real-Time Streaming Engine

components with placement types

no need to manually maintain architecture

trait MasterProxy extends Peer { type Tie <: Multiple[Master] with Multiple[Worker] }
trait Worker extends Peer { type Tie <: Single[MasterProxy] with Optional[Master] }
trait Master extends Peer { type Tie <: Multiple[MasterProxy] with Multiple[Worker] }
Multiple modules
- CheckpointResponder
- KvStateRegistryListener
- PartitionProducerStateChecker
- ResultPartitionConsumableNotifier
- TaskManager
- TaskManagerActions

Eliminated 23 non-exhaustive pattern matches and 8 type casts
Apache Flink Stream Processor

Cloud Deployment Amazon EC2
Yahoo Streaming Benchmark

Latency vs. Number of workers

Cumulative Distribution vs. Fraction of tuples complete

- Flink
- ScalaLoci Flink

- Flink, 4 workers
- Flink, 6 workers
- Flink, 8 workers
- ScalaLoci Flink, 4 workers
- ScalaLoci Flink, 6 workers
- ScalaLoci Flink, 8 workers
Distributed System Development with SCALALOCI

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

Distributed applications are traditionally developed as separate modules, often in different components, and require time-consuming integration. Manual implementation of communication protocols forces programmers to deal with low-level details. The combination of the two results in obscure distributed systems that are hard to reason about among multiple modules, hindering reasoning about the system as a whole.

The SCALALOCI distributed programming language addresses these issues with a cohesive approach, providing a common framework that enables reasoning about distributed data flows, supporting multiple software architectures via dedicated language features and abstracting over low-level communication details and placement types. As we show, SCALALOCI simplifies developing distributed systems, reduces error-prone development, and favors early detection of bugs.

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

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

ACM Reference Format:
ScalaLoci
Research and development of language abstractions for distributed applications in Scala

Coherent
Implement a cohesive distributed application in a single multifiler language

Comprehensive
Freely express any distributed architecture

Safe
Enjoy static type-safety across components

Specify Architecture
Define the architectural relation of the components of the distributed system

trait Server extends Peer {
  type Tle = Multiple[Client]
}

trait Client extends Peer {
  type Tle = Single[Server]
}

Specify Placement
Control where data is located and computations are executed

val items = placed[Server] {
  getCurrentItems()
}

val ui = placed[Client] {
  new UI
}
ScalaLoci

Generic Distributed Architectures

Placement Types

Multitier Event Streams

Value on Peer

message

result

process message