Distributed System Development with ScalaLoci

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

Flink

```
class TaskManagerGateway {
    def disconnectFromJobManager(instanceId: InstanceID, cause: Exception,
                                 mgr: ActorRef) = {
      mgr ! decorateMessage(Disconnect(instanceId, cause))
    }

    def stopCluster(applicationStatus: ApplicationStatus, message: String,
                    mgr: ActorRef) = {
      mgr ! decorateMessage(StopCluster(applicationStatus, message))
    }

    def requestStackTrace(mgr: ActorRef) = {
      (mgr ? decorateMessage(SendStackTrace)).mapTo[StackTrace]
    }

    def submitTask(tdd: TaskDeploymentDescriptor, mgr: ActorRef) = {
      (mgr ? decorateMessage(SubmitTask(tdd))).mapTo[Acknowledge]
    }

    def stopTask(executionAttemptID: ExecutionAttemptID, mgr: ActorRef) = {
      (mgr ? decorateMessage(StopTask(executionAttemptID))).mapTo[Acknowledge]
    }

    def cancelTask(executionAttemptID: ExecutionAttemptID, mgr: ActorRef) = {
      (mgr ? decorateMessage(CancelTask(executionAttemptID))).mapTo[Acknowledge]
    }

    def updatePartitions(executionAttemptID: ExecutionAttemptID,
      partitionInfos: Iterable[PartitionInfo], mgr: ActorRef) = {
      (mgr ? decorateMessage(UpdateTaskMultiplePartitionInfos(executionAttemptID, partitionInfos)))
        .mapTo[Acknowledge]
    }

    def failPartition(executionAttemptID: ExecutionAttemptID, mgr: ActorRef) = {
      mgr ! decorateMessage(FailIntermediateResultPartitions(executionAttemptID))
    }

    def notifyCheckpointComplete(executionAttemptID: ExecutionAttemptID,
      jobId: JobID, checkpointId: long, timestamp: long, mgr: ActorRef) = {
      mgr ! decorateMessage(NotifyCheckpointComplete(jobId, executionAttemptID, checkpointId,
        timestamp))
    }

    def triggerCheckpoint(executionAttemptID: ExecutionAttemptID, jobId: JobID,
        checkpointId: long, timestamp: long, checkpointOptions: CheckpointOptions,
        mgr: ActorRef) = {
      mgr ! decorateMessage(TriggerCheckpoint(jobId, executionAttemptID, checkpointId, timestamp,
        checkpointOptions))
    }

    def requestTaskManagerLog(logTypeRequest: LogTypeRequest, mgr: ActorRef) = {
      (mgr ? decorateMessage(RequestTaskManagerLog(logTypeRequest))).mapTo[BlobKey]
    }
  }
```

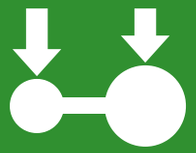
```
class TaskManager extends Actor {
    def receiveMessage = {
      case SendStackTrace => sendStackTrace() foreach { message =>
        sender ! decorateMessage(message)
      }

      case Disconnect(instanceIdToDisconnect, cause) =>
        if (instanceIdToDisconnect.equals(instanceID)) {
          handleJobManagerDisconnect("JobManager requested disconnect: " +
            cause.getMessage())
          triggerTaskManagerRegistration()
        } else {
          log.debug("Received disconnect message for wrong instance id " +
            instanceIdToDisconnect)
        }

      case StopCluster(applicationStatus, message) =>
        log.info(s"Stopping TaskManager with final application status " +
          s"$applicationStatus and diagnostics: $message")
        shutdown()

      case RequestTaskManagerLog(requestType) =>
        blobService match {
          case Some(_) =>
            handleRequestTaskManagerLog(requestType, currentJobManager.get) match {
              case Left(message) => sender ! message
              case Right(message) => sender ! message
            }
          case None =>
            sender ! akka.actor.Status.Failure(new IOException(
              "BlobService not available. Cannot upload TaskManager logs."))
        }

      case UpdateTaskMultiplePartitionInfos(executionID, partitionInfos) =>
        sender ! decorateMessage(updateTaskInputPartitions(executionID, partitionInfos))

      case FailIntermediateResultPartitions(executionID) =>
        log.info(s"Discarding the results produced by task execution $executionID")
        try {
          network.getResultPartitionManager.releasePartitionsProducedBy(executionID)
        } catch {
          case t: Throwable => killTaskManagerFatal(
            "Fatal leak: Unable to release intermediate result partition data", t)
        }

      case SubmitTask(tdd) =>
        sender ! decorateMessage(submitTask(tdd))

      case StopTask(executionID) =>
        val task = runningTasks.get(executionID)
        if (task != null) {
          try {
            task.stopExecution()
            sender ! decorateMessage(Acknowledge.get())
          } catch {
            case t: Throwable =>
              sender ! decorateMessage(Status.Failure(t))
          }
        } else {
          log.debug(s"Cannot find task to stop for execution $executionID)")
          sender ! decorateMessage(Acknowledge.get())
        }

      case CancelTask(executionID) =>
        val task = runningTasks.get(executionID)
        if (task != null) {
          task.cancelExecution()
          sender ! decorateMessage(Acknowledge.get())
        } else {
          log.debug(s"Cannot find task to cancel for execution $executionID)")
          sender ! decorateMessage(Acknowledge.get())
        }

      case TriggerCheckpoint(jobId, taskExecutionId, checkpointId, timestamp,
        checkpointOptions) =>
        log.debug(s"Receiver TriggerCheckpoint $checkpointId@$timestamp " +
          s"for $taskExecutionId.")

        val task = runningTasks.get(taskExecutionId)
        if (task != null) {
          task.triggerCheckpointBarrier(checkpointId, timestamp, checkpointOptions)
        } else {
          log.debug(s"TaskManager received a checkpoint request " +
            s"for unknown task $taskExecutionId.")
        }

      case NotifyCheckpointComplete(jobId, taskExecutionId, checkpointId, timestamp) =>
        log.debug(s"Receiver ConfirmCheckpoint $checkpointId@$timestamp " +
          s"for $taskExecutionId.")

        val task = runningTasks.get(taskExecutionId)
        if (task != null) {
          task.notifyCheckpointComplete(checkpointId)
        } else {
          log.debug(s"TaskManager received a checkpoint confirmation " +
            s"for unknown task $taskExecutionId.")
        }
    }
  }
```

Multitier Languages

Single Compilation Unit

ScalaLoci

Generic Distributed Architectures



Placement Types

Multitier Event Streams

Placement Types

```
trait Registry extends Peer
trait Node extends Peer
```

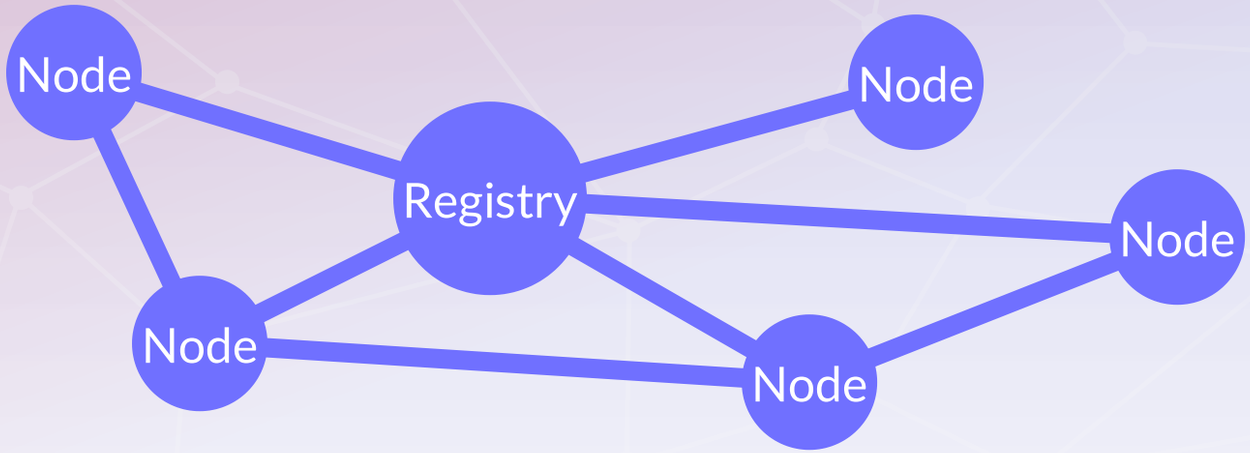
Peers

```
val message: Event[String] on Registry
  = placed { getMessageStream() }
```

Placement Types

Architecture

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



Architecture Specification through Peer Types

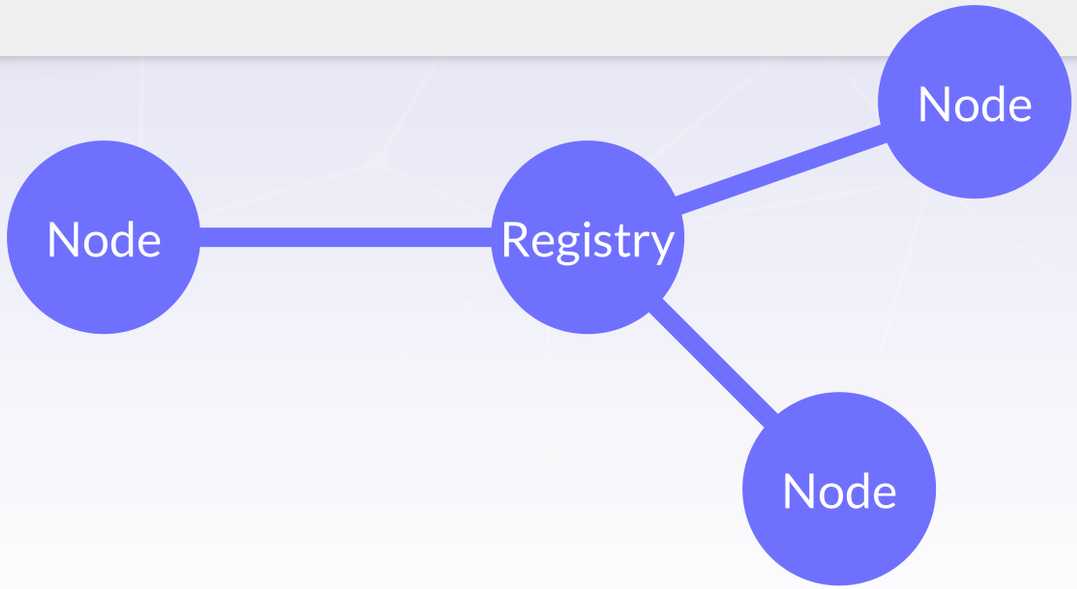
Architecture-Based Remote Access

Remote Access

```
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]
}
```

Aggregation

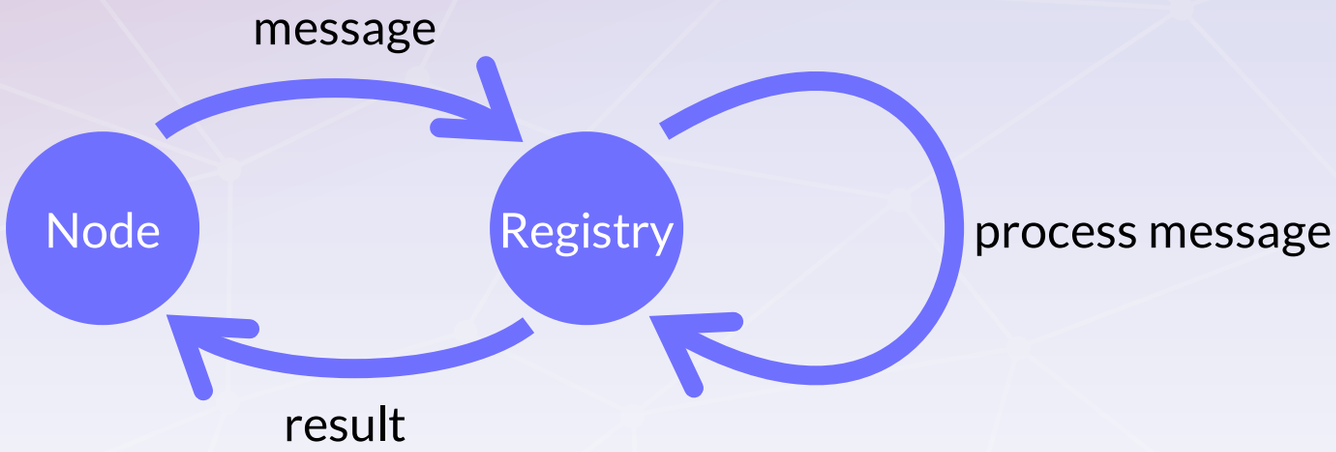
```
trait Registry extends Peer { type Tie <: Multiple[Node] }
trait Node extends Peer { type Tie <: Single[Registry] }

val message: Event[String] on Node

placed[Registry] {
  message.asLocalFromAll: Map[Remote[Node], Event[String]]
}
```



Communication

Data Flow

```
val message = Event[String]()
val result = message map processMessage
val ui = new UI(result)
```

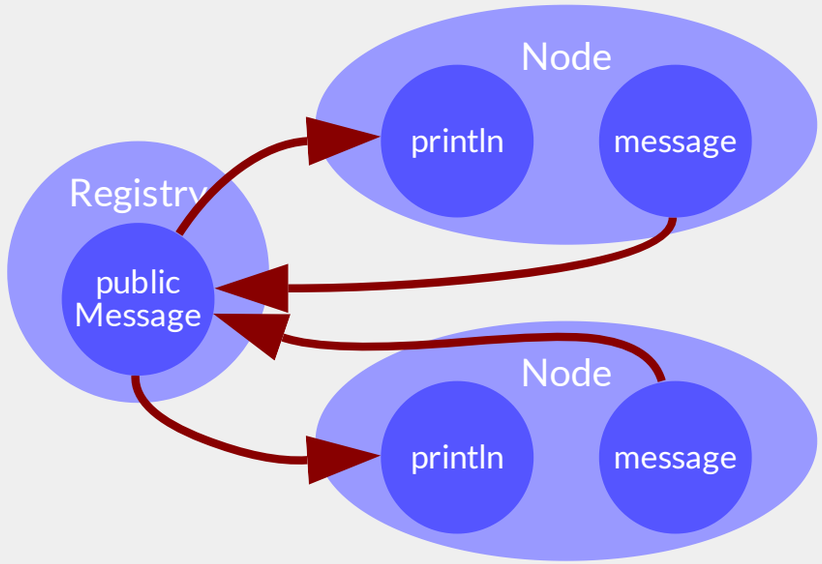


Distributed Data Flow

```
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) }
```

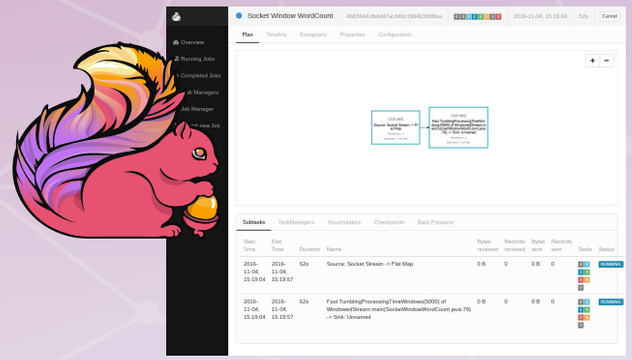
Complete Distributed Chat

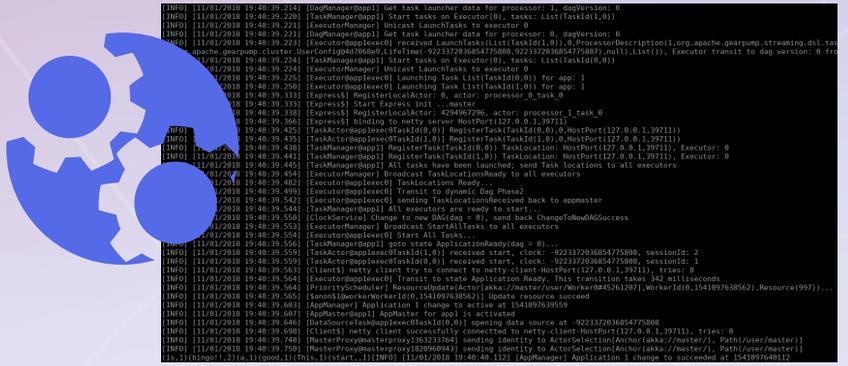
```
@multitier object Chat {
  trait Registry extends Peer { type Tie <: Multiple[Node] }
  trait Node extends Peer { type Tie <: Single[Registry] }

  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)
  }
}
```



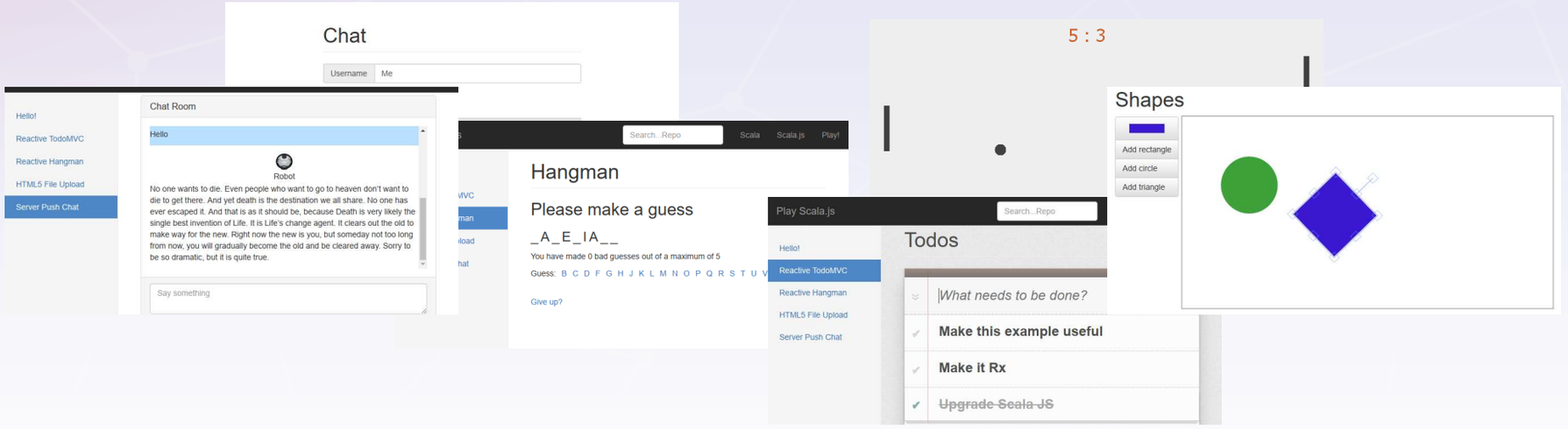
Evaluation

Two **existing** systems





Case studies: **22 variants**



Porting to Distribution

Local

```
val ballSize = 20
val maxX = 800
val maxY = 400
val leftPos = 30
val rightPos = 770
val initPosition = Point(400, 200)
val initSpeed = Point(10, 8)

val ball: Signal[Point] = tick.fold(initPosition) {
  (ball, _) => ball + speed.get }

val areas = {
  val racketY = Seq(
    Signal { UI.mousePosition().y },
    Signal { ball().y })
  val leftRacket = Racket(leftRacketPos, racketY(0))
  val rightRacket = Racket(rightRacketPos, racketY(1))
  val rackets = List(leftRacket, rightRacket)
  Signal { rackets map { _.area() } } }

val leftWall = ball.changed && { _.x < 0 }
val rightWall = ball.changed && { _.x > maxX }

val xBounce = {
  val ballInRacket = Signal { areas() exists { _ contains ball() } }
  val collisionRacket = ballInRacket changedTo true
  leftWall || rightWall || collisionRacket }
val yBounce = ball.changed &&
  { ball => ball.y < 0 || ball.y > maxY }

val speed = {
  val x = xBounce toggle (initSpeed.x, -initSpeed.x)
  val y = yBounce toggle (initSpeed.y, -initSpeed.y)
  Signal { Point(x(), y()) } }

val score = {
  val leftPoints = rightWall.iterate(0) { _ + 1 }
  val rightPoints = leftWall.iterate(0) { _ + 1 }
  Signal { leftPoints() + " : " + rightPoints() } }

val ui = new UI(areas, ball, score)
```

ScalaLoci

```
trait Server extends ServerPeer[Client]
trait Client extends ClientPeer[Server]

val ballSize = 20
val maxX = 800
val maxY = 400
val leftPos = 30
val rightPos = 770
val initPosition = Point(400, 200)
val initSpeed = Point(10, 8)

val clientMouseY = placed[Client] {
  Signal { UI.mousePosition().y } }

val isPlaying = placed[Server].local {
  Signal { remote[Client].connected().size > 2 } }

val ball: Signal[Point] on Server = placed {
  tick.fold(initPosition) { (ball, _) =>
    if (isPlaying.get) ball + speed.get else pos } }

val players = placed[Server].local { Signal {
  remote[Client].connected() match {
    case left :: right :: _ => Seq(Some(left), Some(right))
    case _ => Seq(None, None) } } }

val areas = placed[Server] {
  val racketY = asLocalized(players() map { _ map {
    client => (clientMouseY from client).asLocal() } getOrElse
    initPosition.y } }
  val leftRacket = Racket(leftPos, Signal { racketY()(0) })
  val rightRacket = Racket(rightPos, Signal { racketY()(1) })
  val rackets = List(leftRacket, rightRacket)
  Signal { rackets map { _.area() } } }

val leftWall = placed[Server].local { ball.changed && { _.x < 0 } }
val rightWall = placed[Server].local { ball.changed && { _.x > maxX } }

val xBounce = placed[Server].local {
  val ballInRacket = Signal { areas() exists { _ contains ball() } }
  val collisionRacket = ballInRacket changedTo true
  leftWall || rightWall || collisionRacket }
val yBounce = placed[Server].local { ball.changed &&
  { ball => ball.y < 0 || ball.y > maxY } }

val speed = placed[Server].local {
  val x = xBounce toggle (initSpeed.x, - initSpeed.x)
  val y = yBounce toggle (initSpeed.y, - initSpeed.y)
  Signal { Point(x(), y()) } }

val score = placed[Server] {
  val leftPoints = rightWall.iterate(0) { _ + 1 }
  val rightPoints = leftWall.iterate(0) { _ + 1 }
  Signal { leftPoints() + " : " + rightPoints() } }

val ui = placed[Client] {
  new UI(areas.asLocal, ball.asLocal, score.asLocal) }
```

Akka

```
val ballSize = 20
val maxX = 800
val maxY = 400
val leftPos = 30
val rightPos = 770
val initPosition = Point(400, 200)
val initSpeed = Point(10, 8)

class Server extends Actor {
  def receive = addPlayer orElse mouseYChanged

  val clients = Var(Seq.empty[ActorRef])

  val mousePositions = Var(Map.empty[ActorRef, Int])

  def mouseYChanged: Receive = { case MouseYChanged(y) =>
    mousePositions transform { _ + (sender -> y) } }

  val isPlaying = Signal { clients().size >= 2 }


  val ball: Signal[Point] =
    tick.fold(initPosition) { (ball, _) =>
      if (isPlaying.get) ball + speed.get else ball }

  def addPlayer: Receive = { case AddPlayer =>
    clients transform { _ + sender } }

  val players = Signal {
    clients() match {
      case left :: right :: _ => Seq(Some(left), Some(right))
      case _ => Seq(None, None) } }

  val areas = {
    val racketY = Signal {
      players() map {
        flatMap { mousePositions() get _ } getOrElse initPosition.y } }
    val leftRacket = new Racket(leftRacketPos, Signal { racketY()(0) })
    val rightRacket = new Racket(rightRacketPos, Signal { racketY()(1) })
    val rackets = List(leftRacket, rightRacket)
    Signal { rackets map { _.area() } } }

  val leftWall = ball.changed && { _.x < 0 }
  val rightWall = ball.changed && { _.x > maxX }

  val xBounce = {
    val ballInRacket = Signal { areas() exists {  contains ball() } }
    val collisionRacket = ballInRacket changedTo true
    leftWall || rightWall || collisionRacket }
  val yBounce = ball.changed &&
    { ball => ball.y < 0 || ball.y > maxY }

  val speed = {
    val x = xBounce toggle (Signal { initSpeed.x }, Signal { -initSpeed.x })
    val y = yBounce toggle (Signal { initSpeed.y }, Signal { -initSpeed.y })
    Signal { Point(x(), y()) } }

  val score = {
    val leftPlayerPoints = rightWall.iterate(0) { _ + 1 }
    val rightPlayerPoints = leftWall.iterate(0) { _ + 1 }
    Signal { leftPlayerPoints() + " : " + rightPlayerPoints() } }

  areas observe { areas => clients.now foreach {  ! UpdateAreas(areas) } }
  ball observe { ball => clients.now foreach {  ! UpdateBall(ball) } }
  score observe { score => clients.now foreach { _ ! UpdateScore(score) } }

  clients observe {  foreach { client =>
    client ! UpdateAreas(areas.now)
    client ! UpdateBall(ball.now)
    client ! UpdateScore(score.now) } }
}

abstract class Client(server: ActorSelection) extends Actor {
  val areas = Var(List.empty[Area])
  val ball = Var(Point(0, 0))
  val score = Var("0 : 0")

  mousePosition observe { pos =>
    server ! MouseYChanged(pos.y) }

  val ui = new UI(areas, ball, score)

  def receive = {
    case UpdateAreas(areas) => this.areas set areas
    case UpdateBall(ball) => this.ball set ball
    case UpdateScore(score) => this.score set score }

  server ! AddPlayer
}
```

RMI

```
val ballSize = 20
val maxX = 800
val maxY = 400
val leftPos = 30
val rightPos = 770
val initPosition = Point(400, 200)
val initSpeed = Point(10, 8)

@remote trait Server {
  def addPlayer(client: Client): Unit
  def mouseYChanged(client: Client, y: Int): Unit }

class ServerImpl extends Server {
  val clients = Var(Seq.empty[Client])

  val mousePositions = Var(Map.empty[Client, Int])

  def mouseYChanged(client: Client, y: Int) = synchronized {
    mousePositions() = mousePositions.get + (client -> y) }

  val isPlaying = Signal { clients().size >= 2 }


  val ball: Signal[Point] =
    tick.fold(initPosition) { (ball, _) =>
      if (isPlaying.get) ball + speed.get else ball }

  def addPlayer(client: Client) = synchronized {
    clients transform { _ + client } }

  val players = Signal {
    clients() match {
      case left :: right :: _ => Seq(Some(left), Some(right))
      case _ => Seq(None, None) } }

  val areas = {
    val racketY = Signal {
      players() map {
        flatMap { mousePositions() get _ } getOrElse initPosition.y } }
    val leftRacket = new Racket(leftRacketPos, Signal { racketY()(0) })
    val rightRacket = new Racket(rightRacketPos, Signal { racketY()(1) })
    val rackets = List(leftRacket, rightRacket)
    Signal { rackets map { _.area() } } }

  val leftWall = ball.changed && { _.x < 0 }
  val rightWall = ball.changed && { _.x > maxX }

  val xBounce = {
    val ballInRacket = Signal { areas() exists {  contains ball() } }
    val collisionRacket = ballInRacket changedTo true
    leftWall || rightWall || collisionRacket }
  val yBounce = ball.changed &&
    { ball => ball.y < 0 || ball.y > maxY }

  val speed = {
    val x = xBounce toggle (initSpeed.x, -initSpeed.x)
    val y = yBounce toggle (initSpeed.y, -initSpeed.y)
    Signal { Point(x(), y()) } }

  val score = {
    val leftPoints = rightWall.iterate(0) { _ + 1 }
    val rightPoints = leftWall.iterate(0) { _ + 1 }
    Signal { leftPoints() + " : " + rightPoints() } }

  areas observe { updateAreasClients(clients.get, _) }
  ball observe { updateBallClients(clients.get, _) }
  score observe { updateScoreClients(clients.get, _) }

  clients observe { clients =>
    updateAreasClients(clients, areas.get)
    updateBallClients(clients, ball.get)
    updateScoreClients(clients, score.get) }

  def updateAreasClients(clients: Seq[Client], areas: List[Area]) =
    clients foreach {  updateAreas areas }
  def updateBallClients(clients: Seq[Client], ball: Point) =
    clients foreach {  updateBall ball }
  def updateScoreClients(clients: Seq[Client], score: String) =
    clients foreach { _ updateScore score }
}

@remote trait Client {
  def updateAreas(areas: List[Area]): Unit
  def updateBall(ball: Point): Unit
  def updateScore(score: String): Unit }

class ClientImpl(server: Server) extends Client {
  val self = makeStub[Client](this)

  val areas = Var(List.empty[Area])
  val ball = Var(Point(0, 0))
  val score = Var("0 : 0")

  UI.mousePosition observe { pos =>
    server mouseYChanged (self, pos.y) }

  val ui = new UI(areas, ball, score)

  def updateAreas(areas: List[Area]) = synchronized { this.areas() = areas }
  def updateBall(ball: Point) = synchronized { this.ball() = ball }
  def updateScore(score: String) = synchronized { this.score() = score }

  server addPlayer self
}
```

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

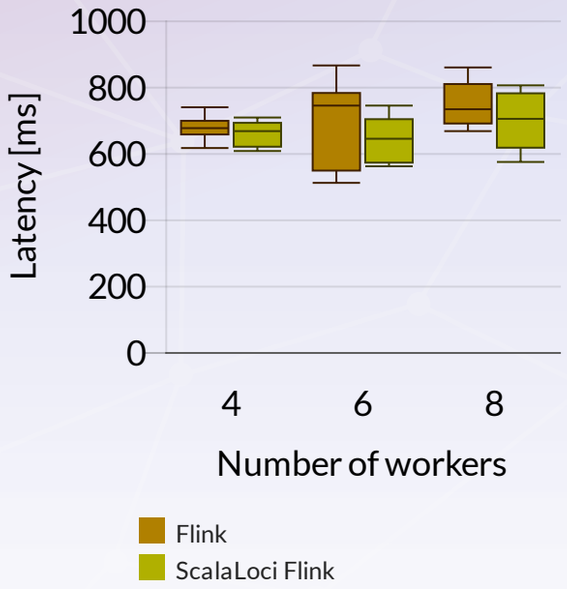
Eliminated **23** non-exhaustive pattern matches and **8** type casts

Apache Flink Stream Processor

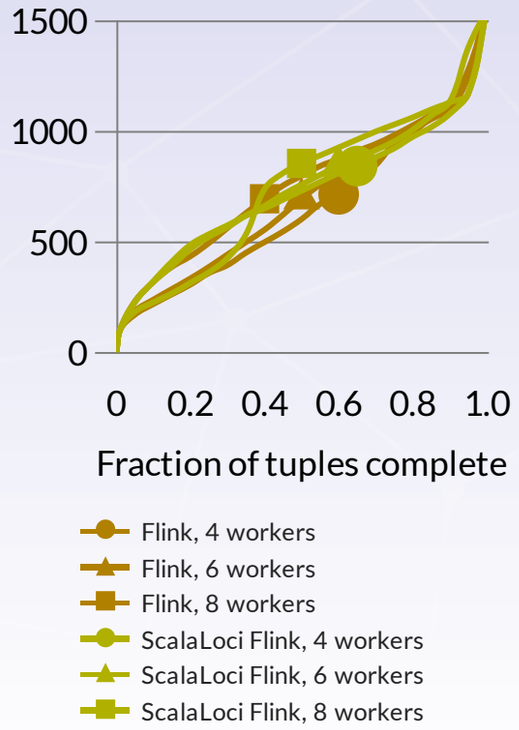
Cloud Deployment **Amazon EC2**

Yahoo Streaming Benchmark

Latency



Cumulative Distribution



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 differe
react to events, like user input, and in turn produce new events for the other modules. Se
nents requires time-consuming integration. Manual implementation of communication forces programmers
to deal with low-level details. The combination of the two results in obscure distributed
among multiple modules, hindering reasoning about the system as a whole.

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

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

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

ACM Reference Format:
Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with
ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 129 (November 2018), 30 pages. https://doi.org/10.
1145/3276499

Fault Tolerance

Dynamic Topologies

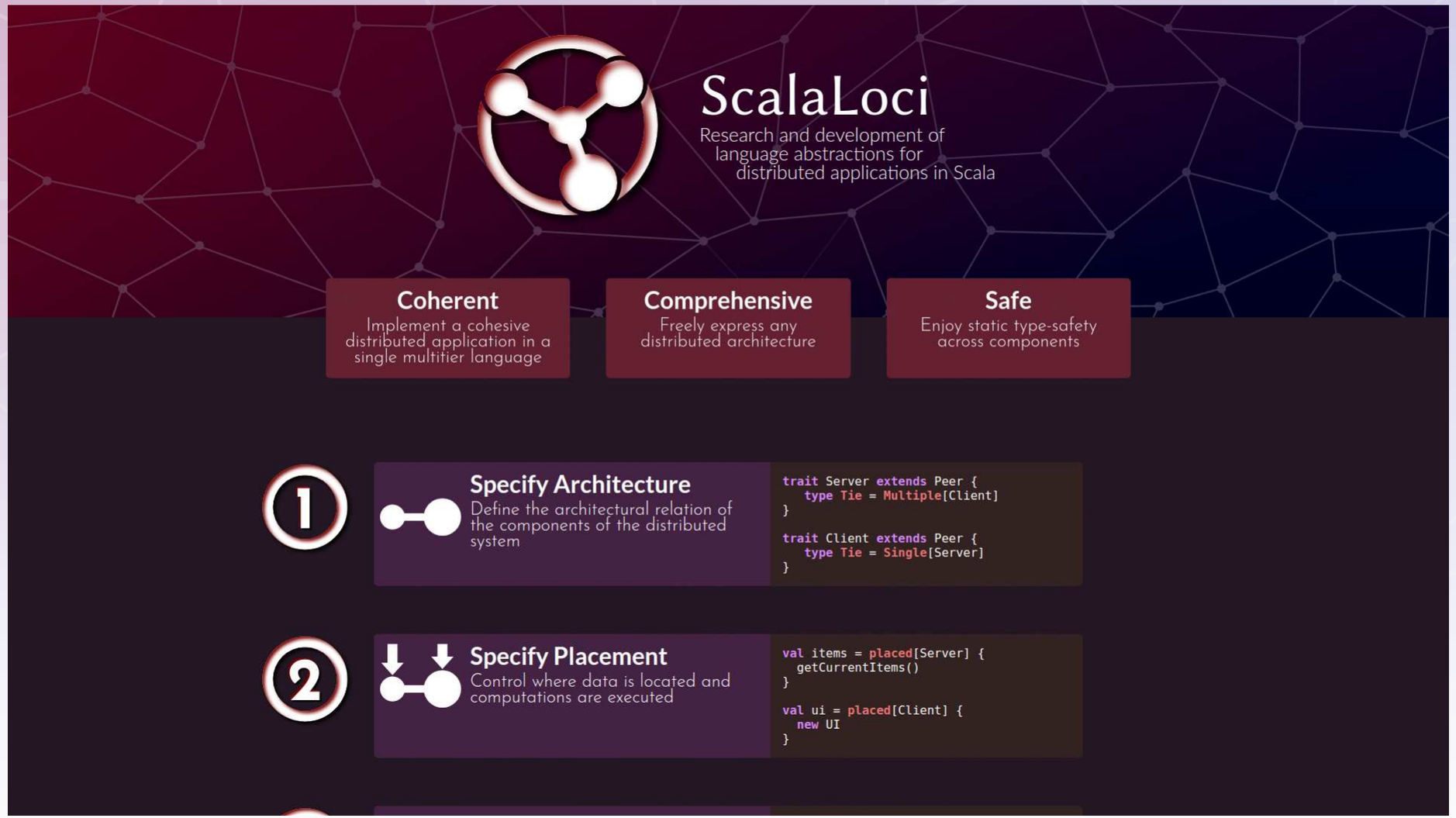
Design Metrics

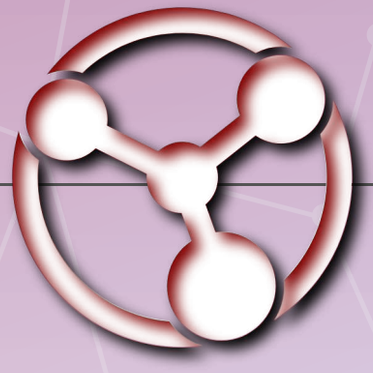
Microbenchmarks

Multiple Backends

Formalization

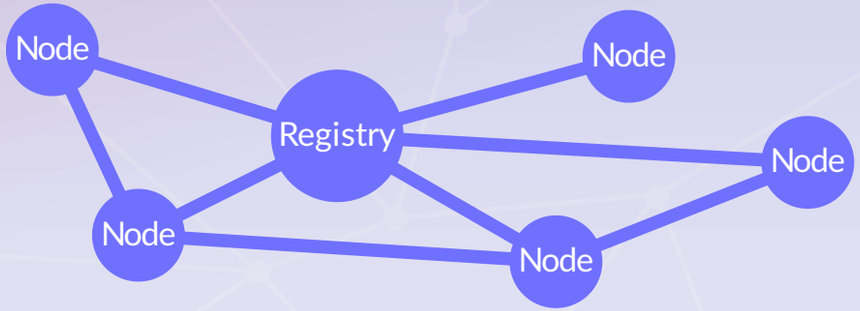
scala-loci.github.io





ScalaLoci

- Generic Distributed Architectures
- Placement Types
- Multitier Event Streams




```
Value on Peer
```