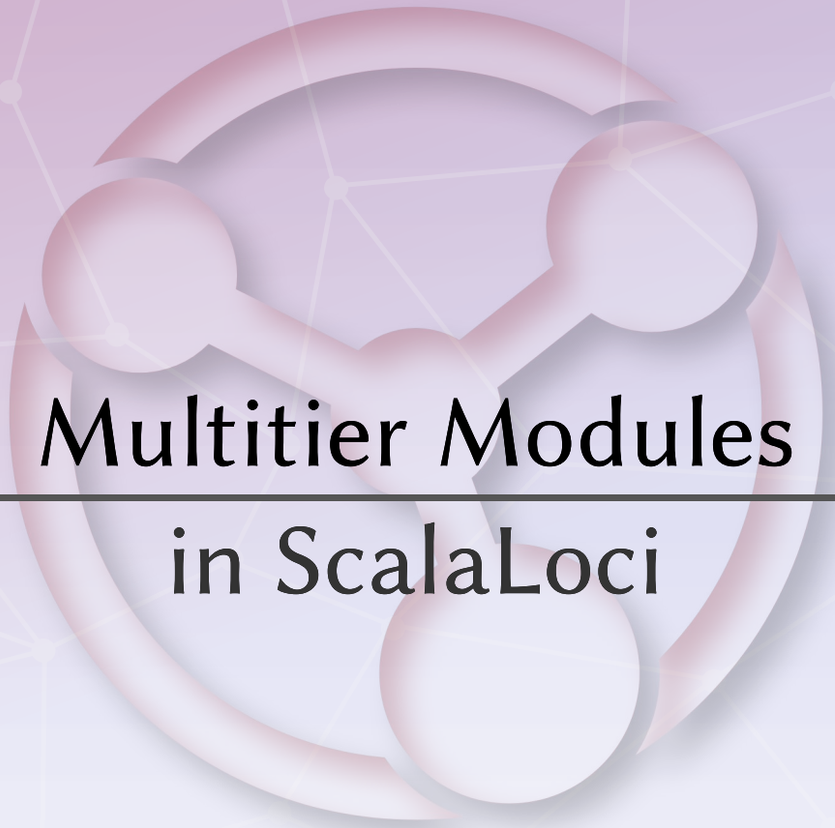


Multitier Modules in ScalaLoci

Pascal Weisenburger, Guido Salvaneschi
TU Darmstadt, Germany

Programming Distributed Systems

Developing distributed systems is *hard*

- Consistency
- Replication
- Fault Tolerance
- Distributed functionalities and communication

Flink

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

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

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

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

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

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

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

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

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

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

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

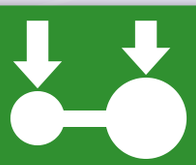
```
class TaskManager extends Actor {
    def receive = {
      case SendStackTrace => sendStackTrace() foreach { message =>
        sender ! decorateMessage(message)
      }

      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

Placement Types

```
@peer type Master
@peer type Worker
```

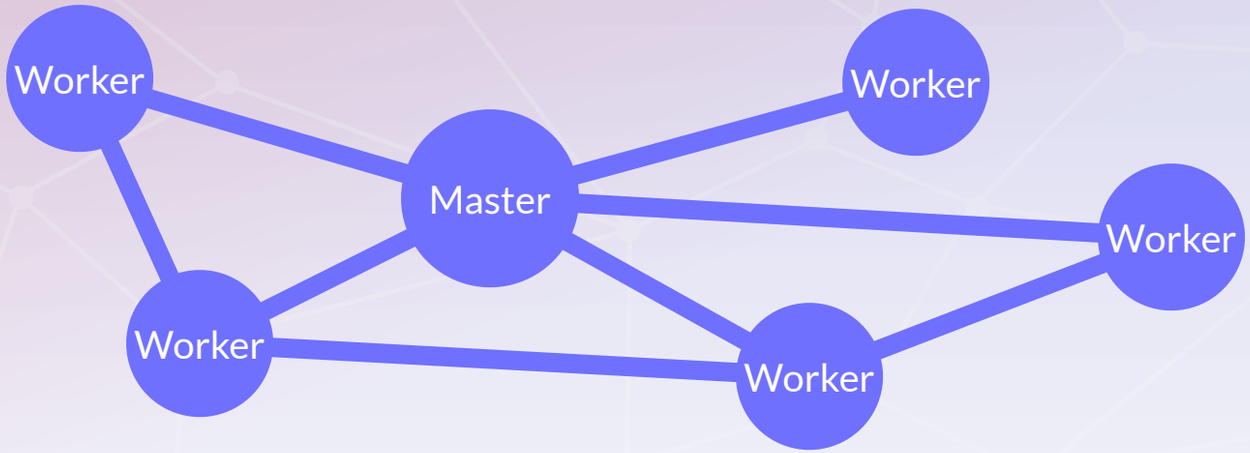
Peers

```
val tasks: List[Task] on Master
  = placed { getTaskList() }
```

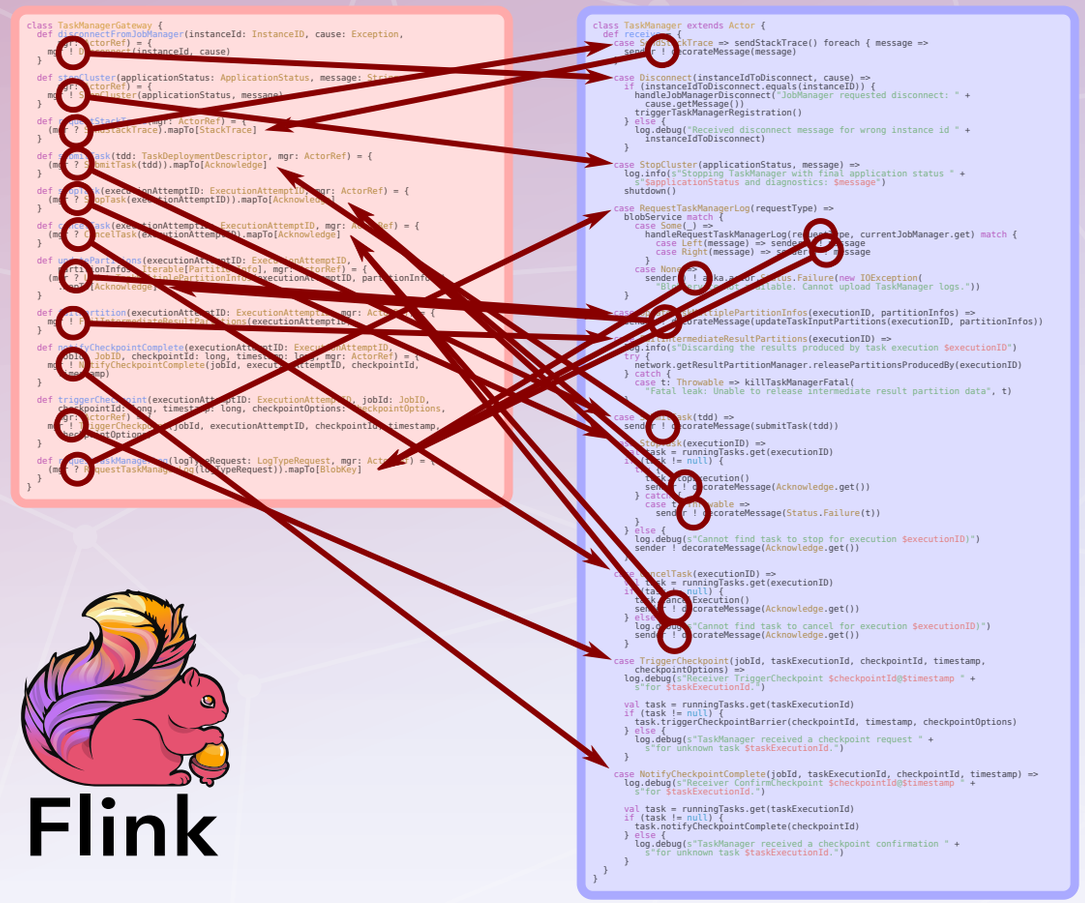
Placement Types

Architecture

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



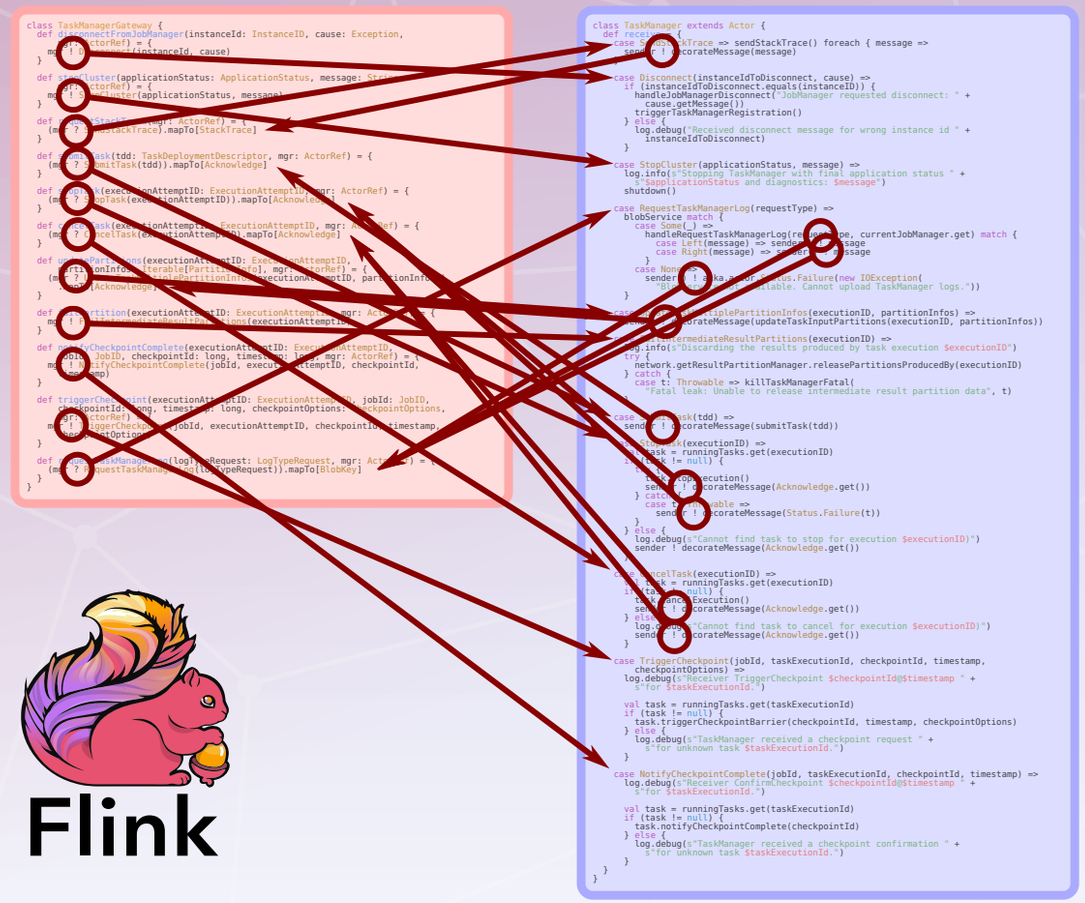
Architecture Specification through Peer Types



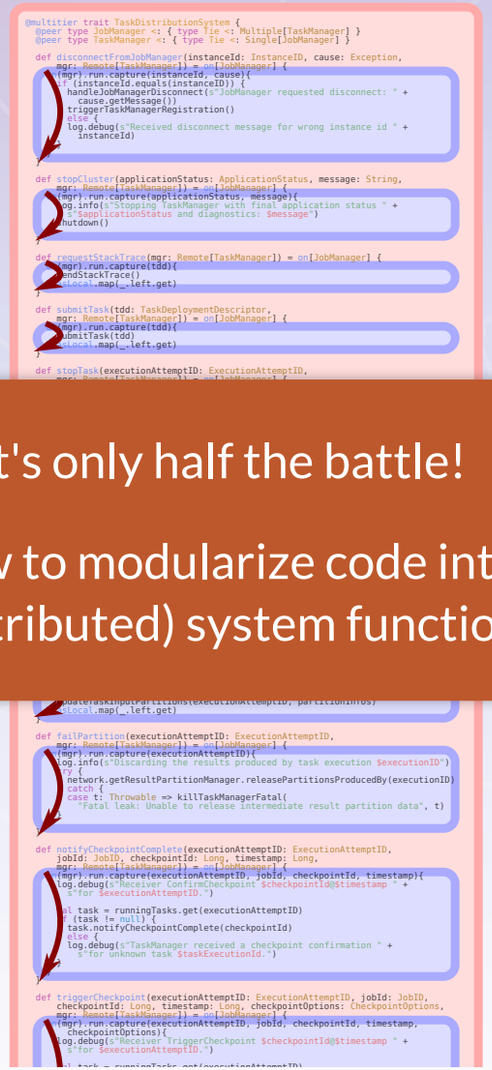
Flink

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

✗ Crosscutting functionality separated among compilation units



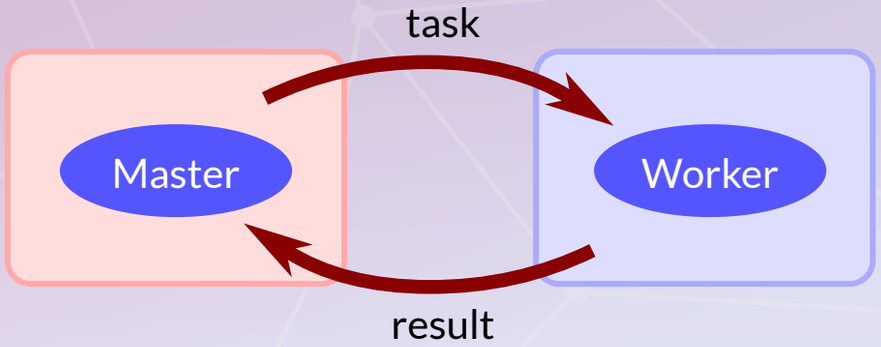
✓ Developers are **not** forced to modularize **along network boundaries**



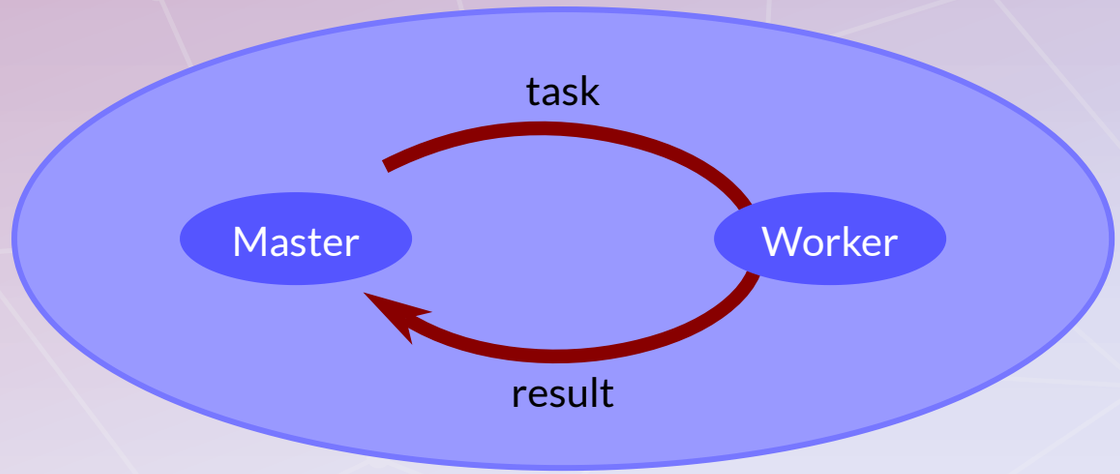
😕 That's only half the battle!

How to modularize code into (distributed) system functionalities?

Distributed Functionalities



Distributed Multitier Functionalities



Multitier Modules

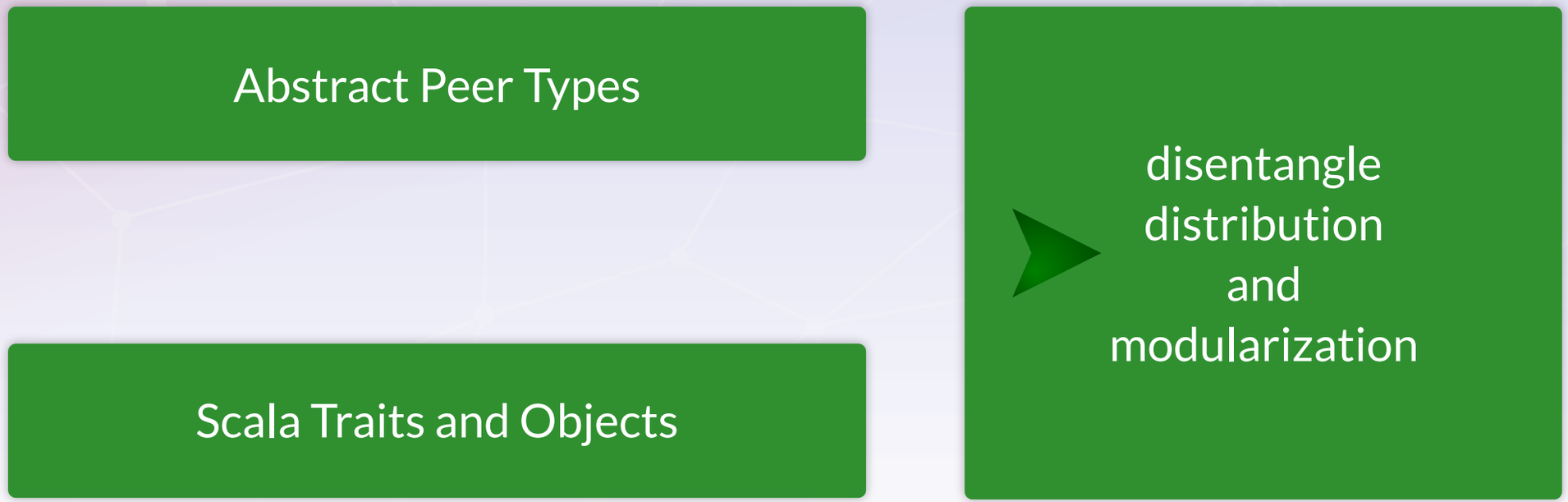
- Handle large code bases
- Modularize distributed system functionalities
- Compose subsystems

Abstract Peer Types

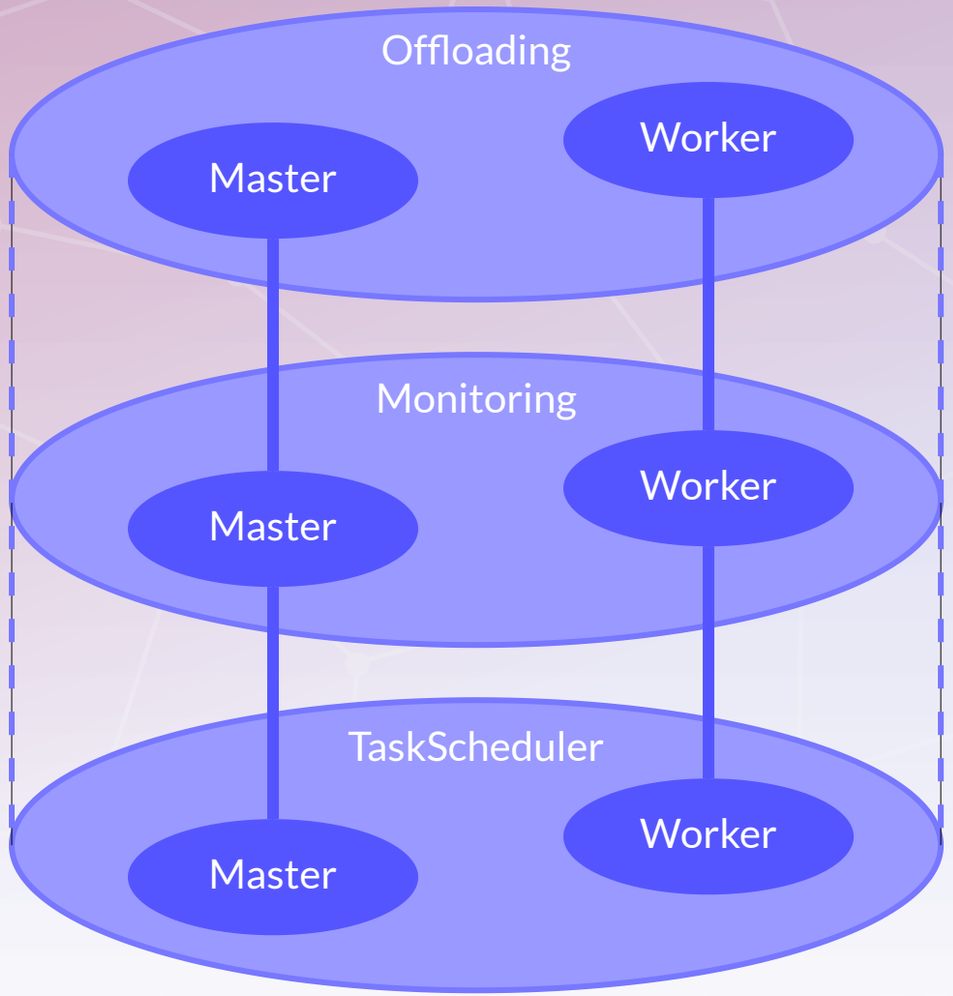
disentangle distribution and modularization

Abstract Peer Types

- Define multitier modules on abstract peer types
- Compose functionality of different modules by identifying abstract peer types

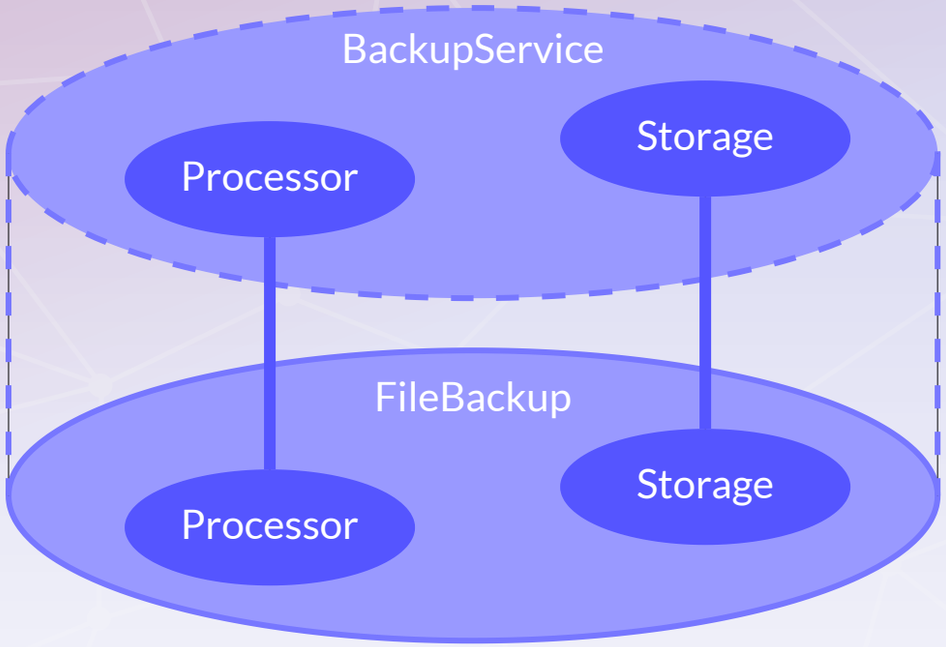


Stacking Multitier Modules




```
@multitier trait Offloading[T] {
  @peer type Master <: { type Tie <: Multiple[Worker] }
  @peer type Worker <: { type Tie <: Single[Master] }
  def run(task: Task[T]): Future[T] on Master =
    placed { (remote(selectWorker()) call execute(task)).asLocal }
  private def execute(task: Task[T]): T on Worker =
    placed { task.process() }
}

@multitier trait Monitoring {
  @peer type Master <: { type Tie <: Multiple[Worker] }
  @peer type Worker <: { type Tie <: Single[Master] }
  def monitoredTimedOut(monitored: Remote[Worker]): Unit on Master
}

@multitier trait TaskScheduler[T] extends
  Offloading[T] with
  Monitoring
```

Abstract Multitier Modules



Interfaces for Subsystems

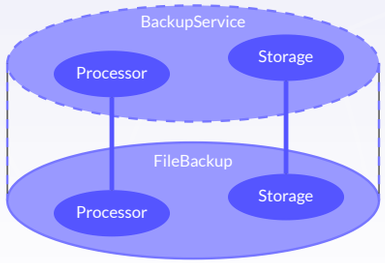
```
@multitier trait BackupService {
  @peer type Processor <: { type Tie <: Single[Storage] }
  @peer type Storage <: { type Tie <: Single[Processor] }

  def store(id: Long, data: Data): Unit on Processor
  def load(id: Long): Future[Data] on Processor
}
```

upper bound allows for refinement

Subsystem Architecture

Placed Methods

Modularization Across Peers

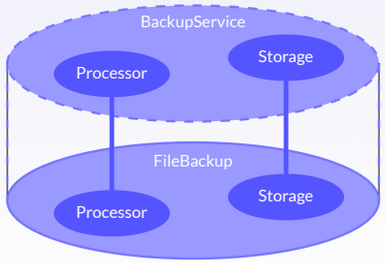


Implementations for Subsystems

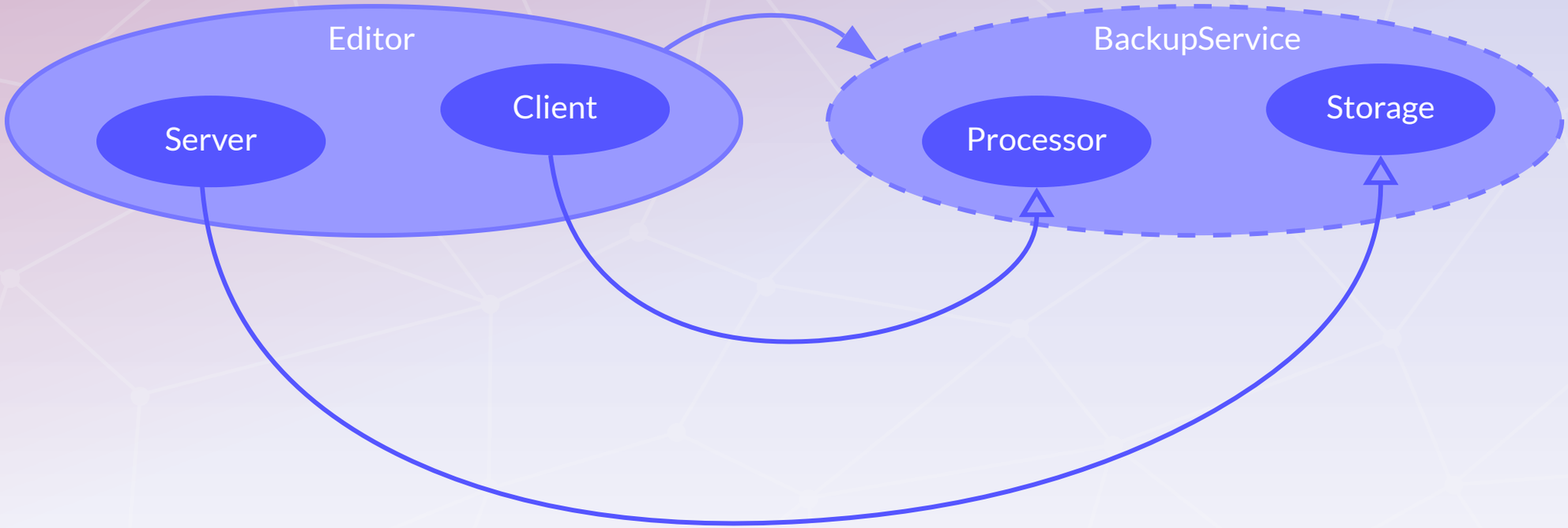
```
@multitier trait FileBackup extends BackupService {
  def store(id: Long, data: Data): Unit on Processor =
    placed { remote call write(id, data) }
  def load(id: Long): Future[Data] on Processor =
    placed { (remote call read(id)).asLocal }

  private def write(id: Long, data: Data): Unit on Storage =
    placed { writeToFile(data, s"/storage/$id") }
  private def read(id: Long): Data on Storage =
    placed { readFromFile[Data](s"/storage/$id") }
}
```

Implementation for Abstract Methods

Encapsulation



References to Multitier Modules

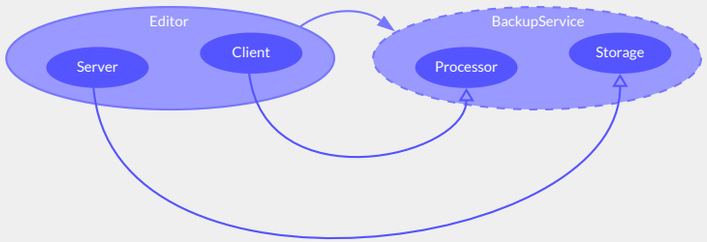


Composing Multitier Modules by References

```
@multitier trait Editor {

  val backup: BackupService

  @peer type Client <: backup.Processor {
    type Tie <: Single[Server] with Single[backup.Storage] }
  @peer type Server <: backup.Storage {
    type Tie <: Single[Client] with Single[backup.Processor] }
}

@multitier object editor extends Editor {
  @multitier object backup extends FileBackup
}
```

abstract module reference

peer refinement



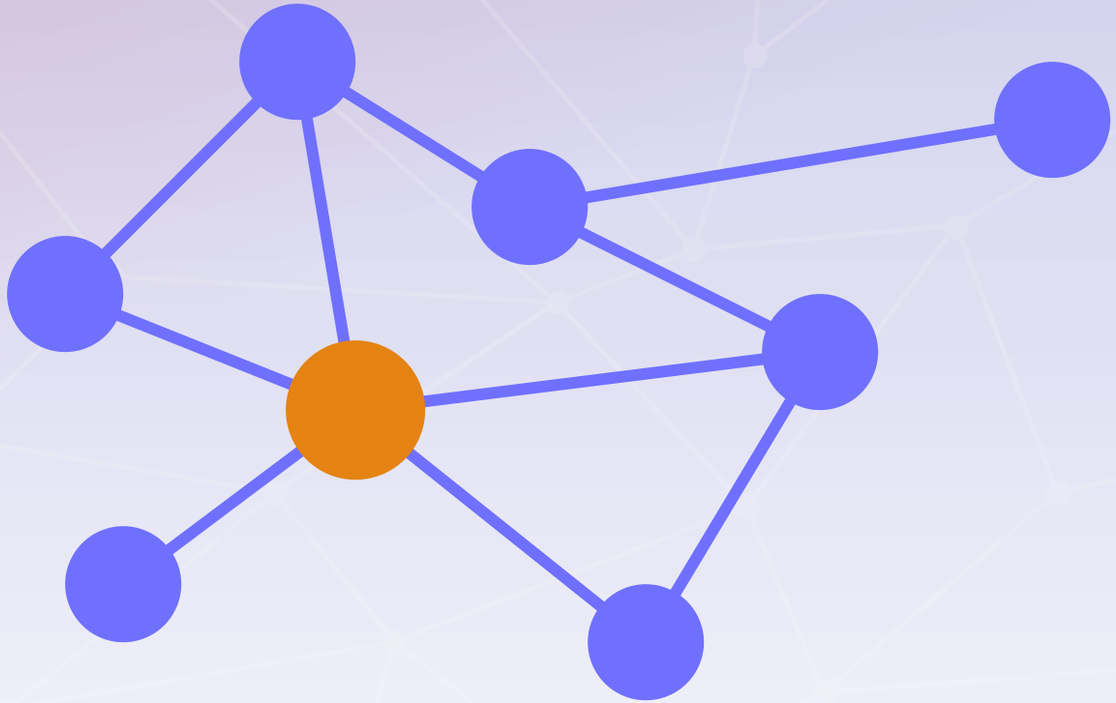
Module Composition

Peer Composition

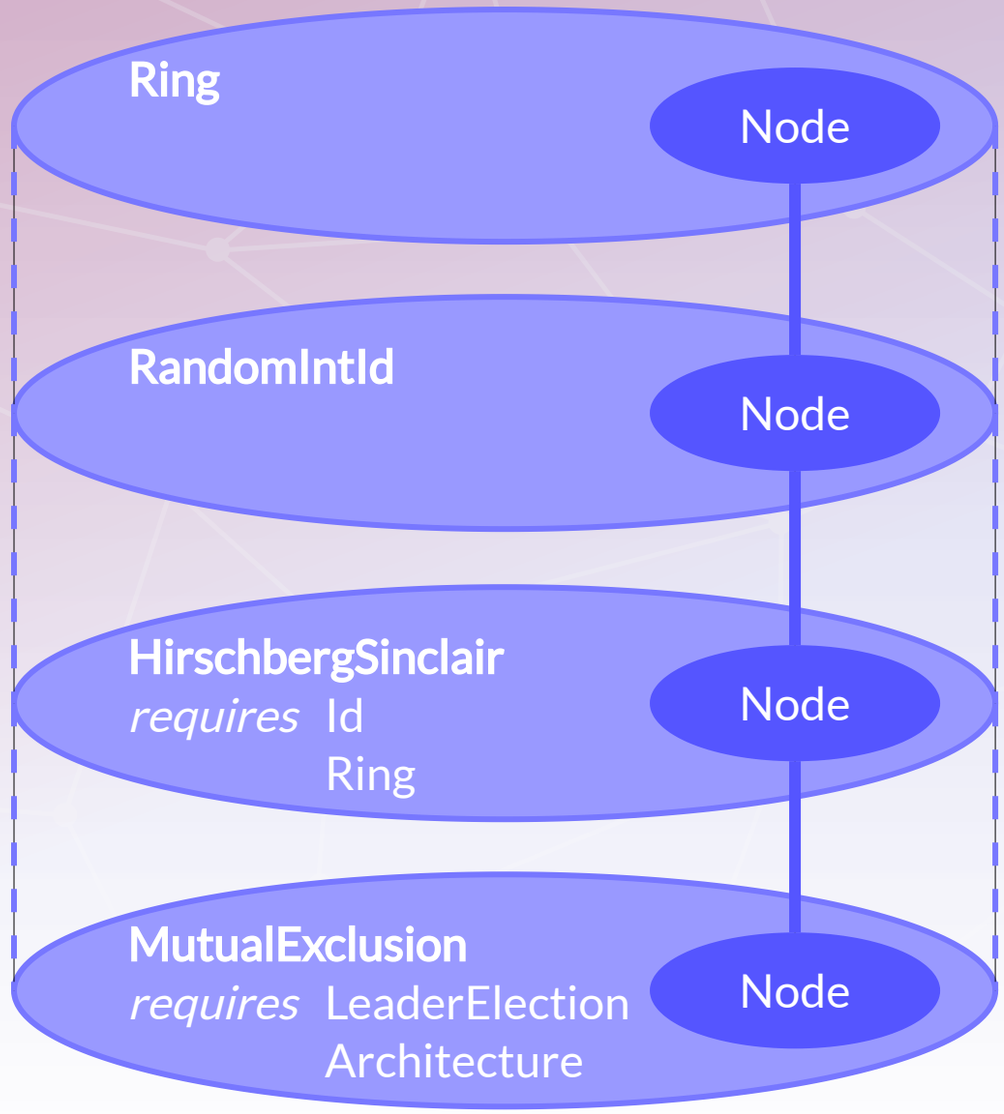
Instantiation

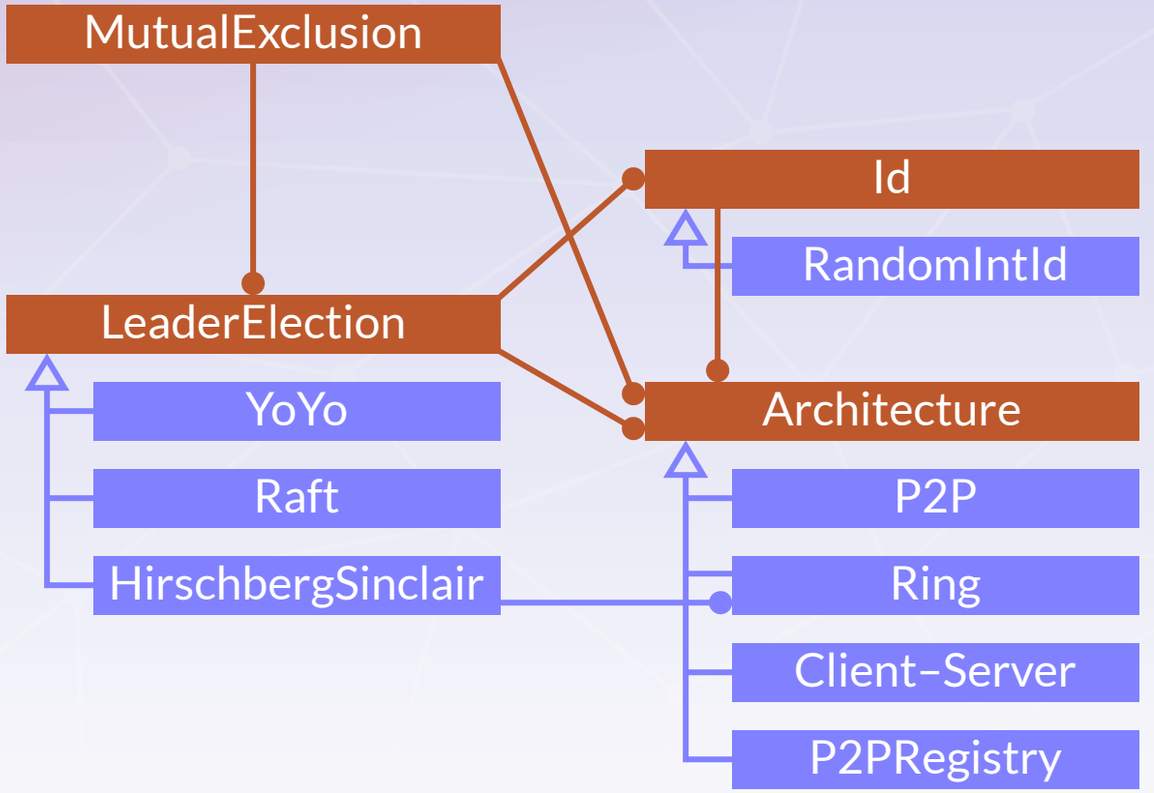
Distributed Mutual Exclusion Algorithm

- Nodes elect a leader
- Followers acquire locks
- Leader grants or denies the lock



Mixing Constrained Modules



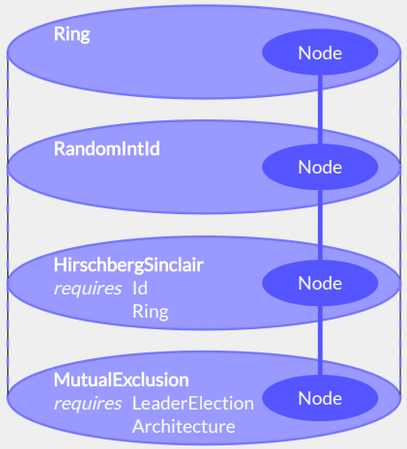


Leader Election Case Study

```
@multitier trait MutualExclusion[T] {
    this: Architecture with LeaderElection[T] =>
  def lock(id: T): Boolean on Node = { ... }
  def unlock(id: Id): Unit on Node = { ... }
}

@multitier trait LeaderElection[T] {
    this: Architecture with Id[T] =>
  def electLeader(): Unit on Node
  def electedAsLeader(): Unit on Node
}

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

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

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



```
@multitier trait TaskDistributionSystem extends CheckpointResponder with KvStateRegistryListener with PartitionProducerStateChecker with ResultPartitionConsumableNotifier with TaskManagerGateway with TaskManagerAction
```

```
@multitier trait TaskManagerActions {
  @peer type TaskManager <: { type Tie <: Single[TaskManager] }
  def notifyFinalState(executionAttemptID: ExecutionAttemptID) =
    on[TaskManager] {
      mgr: Remote[TaskManager] =>
        mgr.run.capture(executionAttemptID) {
          unregisterTaskAndNotifyFinalState(executionAttemptID)
        }
    }

  def notifyFatalError(message: String, cause: Throwable) = on[TaskManager] {
    mgr: Remote[TaskManager] =>
      mgr.run.capture(message, cause) {
        killTaskManagerFatal(message, cause)
      }
  }

  def failTask(executionAttemptID: ExecutionAttemptID,
               cause: Throwable) = on[TaskManager] {
    mgr: Remote[TaskManager] =>
      mgr.run.capture(executionAttemptID, cause) {
        val task = runningTasks.get(executionAttemptID)
        if (task != null) {
          task.failExternally(cause)
        } else {
          log.debug(s"Cannot find task to fail for execution $executionAttemptID)")
        }
      }
  }

  def updateTaskExecutionState(
      taskExecutionState: TaskExecutionState) = on[TaskManager] {
    mgr: Remote[TaskManager] =>
      mgr.run.capture(taskExecutionState) {
        currentJobManager foreach { jobManager =>
          val futureResponse = (jobManager ?
            decorateMessage(UpdateTaskExecutionState(taskExecutionState)))(
            askTimeout)
          futureResponse.mapTo[Boolean].onComplete {
            case scala.util.Success(result) =>
              if (!result) {
                self ! decorateMessage(
                  FailTask(
                    taskExecutionState.getID,
                    new Exception("Task has been cancelled on the JobManager."))
                )
              }
            case scala.util.Failure(t) =>
              self ! decorateMessage(FailTask(
                taskExecutionState.getID,
                new Exception("Failed to send ExecutionStateChange notification " +
                  "to JobManager", t))
              )
          }(context.dispatcher)
        }
      }
  }
}
```

```
@multitier trait CheckpointResponder {
  @peer type JobManager <: { type Tie <: Multiple[TaskManager] }
  @peer type TaskManager <: { type Tie <: Single[JobManager] }

  def acknowledgeCheckpoint(jobID: JobID, executionAttemptID: ExecutionAttemptID,
      checkpointId: Long, checkpointMetrics: CheckpointMetrics,
      checkpointStateHandles: SubtaskState) = on[TaskManager] {
    on[JobManager].run.capture(
        jobID, executionAttemptID, checkpointId,
        checkpointMetrics, checkpointStateHandles) {
      currentJobs.get(jobID) match {
        case Some((graph, _)) =>
          val checkpointCoordinator = graph.getCheckpointCoordinator()
          if (checkpointCoordinator != null)
            future {
              try if (!checkpointCoordinator.receiveAcknowledgeMessage(
                    AcknowledgeCheckpoint(jobID, executionAttemptID,
                      checkpointId,checkpointMetrics, checkpointStateHandles)))
                  log.info("Received message for non-existing checkpoint " +
                    checkpointId)
              catch {
                case t: Throwable => log.error("Error in CheckpointCoordinator " +
                  "while processing acknowledge message", t)
              }
            }(context.dispatcher)
          else log.error(
            s"Received AcknowledgeCheckpoint message for job $jobID with no " +
            "CheckpointCoordinator")
        case None =>
          log.error(s"Received AcknowledgeCheckpoint for unavailable job $jobID")
      }
    }
  }

  def declineCheckpoint(jobID: JobID, executionAttemptID: ExecutionAttemptID,
      checkpointId: Long, reason: Throwable) = on[TaskManager] {
    on[JobManager].run.capture(
        jobID, executionAttemptID, checkpointId, reason) {
      currentJobs.get(jobID) match {
        case Some((graph, _)) =>
          val checkpointCoordinator = graph.getCheckpointCoordinator()
          if (checkpointCoordinator != null)
            future {
              try checkpointCoordinator.receiveDeclineMessage(DeclineCheckpoint(
                  jobID, executionAttemptID, checkpointId, reason))
              catch {
                case t: Throwable => log.error("Error in CheckpointCoordinator " +
                  "while processing decline message", t)
              }
            }(context.dispatcher)
          else log.error("Received DeclineCheckpoint message " +
            s"for job $jobID with no CheckpointCoordinator")
        case None =>
          log.error(s"Received DeclineCheckpoint for unavailable job $jobID")
      }
    }
  }
}
```

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

  def disconnectFromJobManager(instanceId: InstanceID, cause: Exception,
      mgr: Remote[TaskManager]) = on[JobManager] {
    (mgr).run.capture(instanceId, cause) {
      if (instanceId.equals(instanceID)) {
        handleJobManagerDisconnect(s"JobManager requested disconnect: " +
          cause.getMessage())
        triggerTaskManagerRegistration()
      } else {
        log.debug(s"Received disconnect message for wrong instance id " +
          instanceId)
      }
    }
  }

  def stopCluster(applicationStatus: ApplicationStatus, message: String,
      mgr: Remote[TaskManager]) = on[JobManager] {
    (mgr).run.capture(applicationStatus, message) {
      log.info(s"Stopping taskManager with final application status " +
        s"$applicationStatus and diagnostics: $message")
      shutdown()
    }
  }

  def requestStackTrace(mgr: Remote[TaskManager]) = on[JobManager] {
    (mgr).run.capture(tdd) {
      sendStackTrace()
    }.asLocal.map(_.left.get)
  }

  def submitTask(tdd: TaskDeploymentDescriptor,
      mgr: Remote[TaskManager]) = on[JobManager] {
    (mgr).run.capture(tdd) {
      submitTask(tdd)
    }.asLocal.map(_.left.get)
  }

  def stopTask(executionAttemptID: ExecutionAttemptID,
      mgr: Remote[TaskManager]) = on[JobManager] {
    (mgr).run.capture(executionAttemptID) {
      val task = runningTasks.get(executionAttemptID)
      if (task != null) {
        try {
          task.stopExecution()
          Left(Acknowledge.get())
        } catch {
          case t: Throwable =>
            Right(Status.Failure(t))
        }
      } else {
        log.debug(s"Cannot find task to stop for execution $executionAttemptID)")
        Left(Acknowledge.get())
      }
    }.asLocal.map(_.left.get)
  }

  def cancelTask(executionAttemptID: ExecutionAttemptID,
      mgr: Remote[TaskManager]) = on[JobManager] {
    (mgr).run.capture(executionAttemptID) {
      val task = runningTasks.get(executionAttemptID)
      if (task != null) {
        task.cancelExecution()
        Acknowledge.get()
      } else {
        log.debug(s"Cannot find task to cancel for execution $executionAttemptID")
        Acknowledge.get()
      }
    }.asLocal
  }

  def updatePartitions(
      executionAttemptID: ExecutionAttemptID,
      partitionInfos: java.lang.Iterable[PartitionInfo],
      mgr: Remote[TaskManager]) = on[JobManager] {
    (mgr).run.capture(executionAttemptID, partitionInfos) {
      updateTaskInputPartitions(executionAttemptID, partitionInfos)
    }.asLocal.map(_.left.get)
  }

  def failPartition(executionAttemptID: ExecutionAttemptID,
      mgr: Remote[TaskManager]) = on[JobManager] {
    (mgr).run.capture(executionAttemptID) {
      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)
      }
    }
  }

  def notifyCheckpointComplete(executionAttemptID: ExecutionAttemptID,
      jobId: JobID, checkpointId: Long, timestamp: Long,
      mgr: Remote[TaskManager]) = on[JobManager] {
    (mgr).run.capture(executionAttemptID, jobId, checkpointId, timestamp) {
      log.debug(s"Receiver ConfirmCheckpoint $checkpointId@$timestamp " +
        s"for $executionAttemptID.")

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

  def triggerCheckpoint(executionAttemptID: ExecutionAttemptID, jobId: JobID,
      checkpointId: Long, timestamp: Long, checkpointOptions: CheckpointOptions,
      mgr: Remote[TaskManager]) = on[JobManager] {
    (mgr).run.capture(executionAttemptID, jobId, checkpointId, timestamp,
        checkpointOptions) {
      log.debug(s"Receiver TriggerCheckpoint $checkpointId@$timestamp " +
        s"for $executionAttemptID.")

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

  def requestTaskManagerLog(logTypeRequest: LogTypeRequest,
      mgr: Remote[TaskManager]) = on[JobManager] {
    (mgr).run.capture(logTypeRequest) {
      blobService match {
        case Some(_) =>
          handleRequestTaskManagerLog(logTypeRequest, currentJobManager.get)
        case None =>
          Right(akka.actor.Status.Failure(new IOException(
            "BlobService not available. Cannot upload TaskManager logs.")))
      }
    }.asLocal.map(_.left.get)
  }
}
```

```
@multitier trait KvStateRegistryListener {
  @peer type JobManager <: { type Tie <: Multiple[TaskManager] }
  @peer type TaskManager <: { type Tie <: Single[JobManager] }

  def notifyKvStateRegistered(jobId: JobID, jobVertexId: JobVertexID,
      keyGroupRange: KeyGroupRange, registrationName: String,
      kvStateId: KvStateID) = on[TaskManager] {
    (jobManager).run.capture(
        jobId, jobVertexId, keyGroupRange, registrationName,
        kvStateId, kvStateServerAddress) {
      currentJobs.get(jobId) match {
        case Some((graph, _)) =>
          try {
            log.debug(s"Key value state registered for job $jobId " +
              s"under name $registrationName.")
            graph.getKvStateLocationRegistry.notifyKvStateRegistered(
              jobVertexId, keyGroupRange, registrationName,
              kvStateId, kvStateServerAddress)
          } catch {
            case t: Throwable => log.error(
              "Failed to notify KvStateRegistry about registration.")
          }
        case None =>
          log.error("Received state registration for unavailable job.")
      }
    }
  }

  def notifyKvStateUnregistered(jobId: JobID, jobVertexId: JobVertexID,
      keyGroupRange: KeyGroupRange,
      registrationName: String) = on[TaskManager] {
    (jobManager).run.capture(
        jobId, jobVertexId, keyGroupRange, registrationName) {
      currentJobs.get(jobId) match {
        case Some((graph, _)) =>
          try graph.getKvStateLocationRegistry.notifyKvStateUnregistered(
              jobVertexId, keyGroupRange, registrationName)
          catch {
            case t: Throwable => log.error(
              s"Failed to notify KvStateRegistry about registration.")
          }
        case None =>
          log.error("Received state unregistration for unavailable job.")
      }
    }
  }
}
```

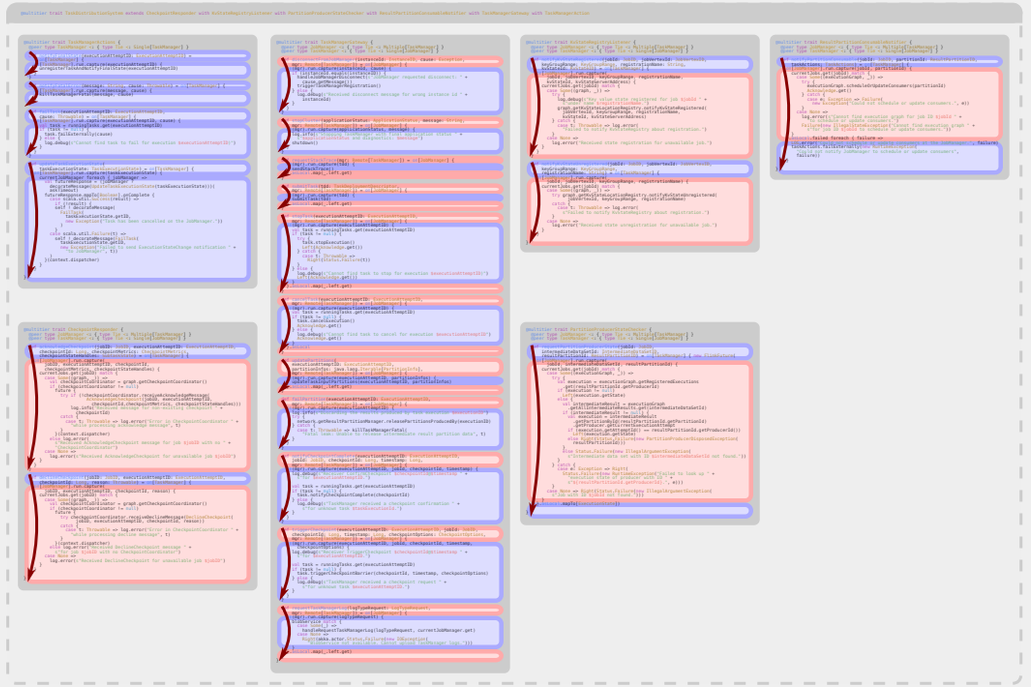
```
@multitier trait PartitionProducerStateChecker {
  @peer type JobManager <: { type Tie <: Multiple[TaskManager] }
  @peer type TaskManager <: { type Tie <: Single[JobManager] }

  def requestPartitionProducerState(jobId: JobID,
      intermediateDataSetId: IntermediateDataSetID,
      resultPartitionId: ResultPartitionID) = on[TaskManager] { new FlinkFuture(
    (jobManager).run.capture(
        jobId, intermediateDataSetId, resultPartitionId) {
      currentJobs.get(jobId) match {
        case Some((executionGraph, _)) =>
          try {
            val execution = executionGraph.getRegisteredExecutions
              .get(resultPartitionId.getProducerId)
            if (execution != null)
              Left(execution.getState)
            else {
              val intermediateResult = executionGraph
                .getAllIntermediateResults.get(intermediateDataSetId)
              if (intermediateResult != null) {
                val execution = intermediateResult
                  .getPartitionById(resultPartitionId.getPartitionId)
                  .getProducer.getCurrentExecutionAttempt
                if (execution.getAttemptId() == resultPartitionId.getProducerId())
                  Left(execution.getState)
                else Right(Status.Failure(new PartitionProducerDisposedException(
                  resultPartitionId)))
              }
              else Status.Failure(new IllegalArgumentException(
                s"Intermediate data set with ID $intermediateDataSetId not found."))
            }
          } catch {
            case e: Exception => Right(
              Status.Failure(new RuntimeException("Failed to look up " +
                "execution state of producer with ID " +
                s"${resultPartitionId.getProducerId}.", e)))
          }
        case None => Right(Status.Failure(new IllegalArgumentException(
          s"Job with ID $jobId not found.")))
      }
    }.asLocal.mapTo[ExecutionState])
  }
}
```

```
@multitier trait ResultPartitionConsumableNotifier {
  @peer type JobManager <: { type Tie <: Multiple[TaskManager] }
  @peer type TaskManager <: { type Tie <: Single[JobManager] }

  def notifyPartitionConsumable(jobId: JobID, partitionId: ResultPartitionID,
      taskActions: TaskActions) = on[TaskManager] {
    on[JobManager].run.capture(jobId, partitionId) {
      currentJobs.get(jobId) match {
        case Some((executionGraph, _)) =>
          try {
            executionGraph.scheduleOrUpdateConsumers(partitionId)
            Acknowledge.get()
          } catch {
            case e: Exception => Failure(
              new Exception("Could not schedule or update consumers.", e))
          }
        case None =>
          log.error(s"Cannot find execution graph for job ID $jobId " +
            "to schedule or update consumers.")
          Failure(new IllegalStateException("Cannot find execution graph " +
            s"for job ID $jobId to schedule or update consumers."))
      }
    }.asLocal.failed foreach { failure =>
      LOG.error("Could not schedule or update consumers at the JobManager.", failure)
      taskActions.failExternally(new RuntimeException(
        "Could not notify JobManager to schedule or update consumers",
        failure))
    }
  }
}
```

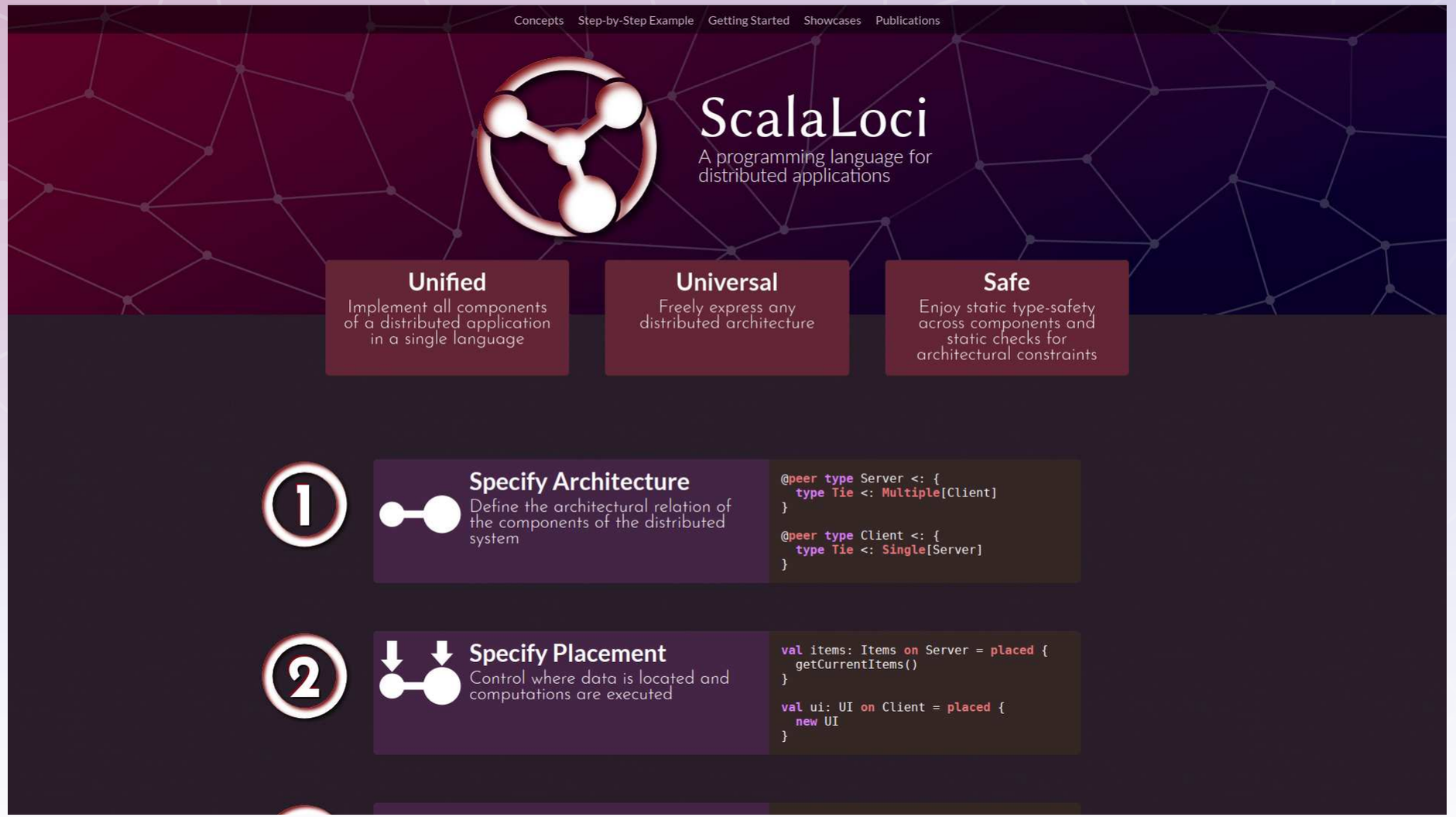
Flink Case Study

```
@multitier object TaskManager {
  @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()
  }
  ...
}

@multitier object TaskManagerActions { ... }
@multitier object CheckpointResponder { ... }
@multitier object ResultPartitionConsumableNotifier { ... }
@multitier object PartitionProducerStateChecker { ... }
@multitier object KvStateRegistryListener { ... }
```

```
@multitier object TaskDistributionSystem extends
  TaskManager with
  TaskManagerActions with
  CheckpointResponder with
  ResultPartitionConsumableNotifier with
  PartitionProducerStateChecker with
  KvStateRegistryListener
```



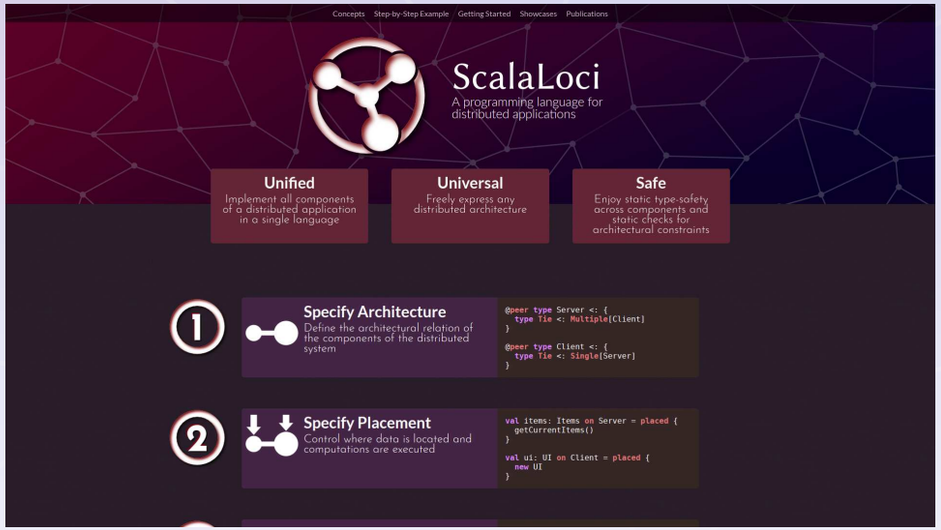
scala-loci.github.io



ScalaLoci: scala-loci.github.io

Multitier Modules

Abstract Peer Types



- Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 129.
- Pascal Weisenburger and Guido Salvaneschi. 2019. Multitier Modules. In *Proceedings of the 33rd European Conference on Object-Oriented Programming*, ECOOP.