Technology know-how:

Multitier Programming and ScalaLoci Analysis

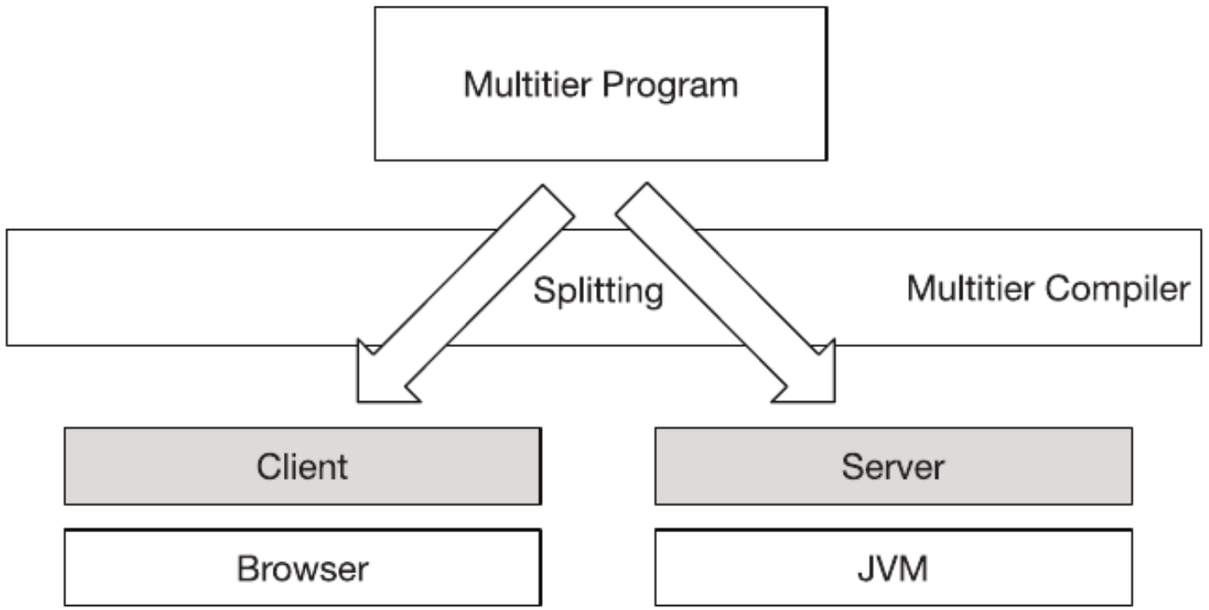
Project’s objectives

- Analyze the multitier programming paradigm
- Analyze the multitier language ScalaLoci
- Provide a comparison between multitier and traditional programming

Multitier Programming

A programming paradigm for distributed systems based on the multitier architecture model

- single code base, single programming language
- code generated at run time or results from compilation
- placement strategy technique
  - automatic allocation
  - explicit allocation
- integrated communication support



Advantages

- *Higher Abstraction Level:* eliminates challenges like network communication and API design
- *Formal Reasoning:* forces the designer to consider the system as a whole
- *Code Maintenance:* easier to migrate features between tiers and platforms
- *Program Comprehension:* streamlines understanding of data flow across hosts

Limitations

- *Generic Distributed Systems:* focus on client-server architecture, lack of abstractions for complex systems
- *Consistency Levels:* single level of consistency, limitate flexibility
- *Programming in the Large:* scalability challenges

ScalaLoci

A distributed programming language that extends Scala with multitier abstractions

- abstraction over low-level network communication details
- abstraction over data conversion between tiers
- support for generic distributed systems (not only client-server)

Language constructs:

- *Peers and ties:* respectively represent the components and the connections between them
- *Placement types:* to specify the tier of a value or a computation
- *Multitier reactives:* as communication model (REScala)
- *Multitier modules:* for scalability and reuse of code

Peers and Ties

Client-Server

```
@peer type Server <: { type Tie <: Multiple[Client] }
@peer type Client <: { type Tie <: Single[Server] }
```

Peer-To-Peer

```
@peer type Peer <: { type Tie <: Multiple[Peer] }
```

Placement Types

Placed data

```
val id: Int on Peer = UUID.randomUUID().hashCode()
```

Access placed data

```
val connectedNodesIds = id.asLocalFromAll.observe(m => m.foreach { case
(remote, event) => event observe println })
```

Multitier Reactives

ScalaLoci reactive abstractions: `Events` and `Signals`.

- Events represent discrete changes
- Signals represent continuous time-changing values

```
val messageSent: Evt[String] on Client = Evt[String]()

on[Client] { messageSent.fire("new message") }

on[Server] { message.asLocal observe println }
```

```
val time = on[Server] { Var(0L) }

on[Client] { val display = Signal { format().format(new Date(time.asLocal())) } }
```

Multitier Modules

Encapsulation

```
@multitier trait ChatApp {
  @peer type Client <: { type Tie <: Single[Server] }
  @peer type Server <: { type Tie <: Single[Client] }
  val messageService: MessageService
}
```

```
@multitier trait MessageService {
  @peer type UserInterface <: { type Tie <: Single[Database] }
  @peer type Database <: { type Tie <: Single[UserInterface] }

  def sendMessage(message: String): Unit on UserInterface = placed {
    remote call storeMessage(message)
  }
  def fetchMessages(): Future[List[String]] on UserInterface = placed {
    (remote call retrieveMessages()).asLocal
  }

  private def storeMessage(message: String): Unit on Database = // storing logics
  private def retrieveMessages(): List[String] on Database = // retrieving logics
}
```

Multitier Modules

Abstraction over placement

```
@multitier trait ChatApp {
  @peer type Client <: messageHandler.UserInterface {
  type Tie <: Single[Server] }
  @peer type Server <: messageHandler.Database { type Tie <: Single[Client] }
  val messageHandler: MessageService
}
```

Module composition

```
@multitier trait Monitoring {
  @peer type Monitor <: { type Tie <: Multiple[Monitored] }
  @peer type Monitored <: { type Tie <: Single[Monitor] }
}
  @multitier class MonitoredClass extends Monitoring {
  @peer type SupervisorNode <: Monitor {
  type Tie <: Multiple[Node] with Multiple[Monitored] }
  @peer type Node <: Monitored { type Tie <: Single[SupervisorNode] }
}
```

Case Studies

Authentication Service

- Client-Server architecture
- Funtionalities:
  - *Register* a new user: receives a username and a password and returns a token if the username is not already taken
  - *Login* a user: receives a username and a password and returns a token if the credentials are valid
  - Provide *access to a secure endpoint* that requires authentication

Traditional approach: Akka Http, Spray Json

Multitier approach: ScalaLoci

Traditional approach

```scala
trait JsonSupport {
  implicit val userFormat: RootJsonFormat[User] = jsonFormat2
  implicit val tokenFormat: RootJsonFormat[Token] = jsonForma
}

object AuthServer extends App with JsonSupport {
  implicit val system: ActorSystem[Unit] = ActorSystem(Server
  var users: Map[String, User] = Map.empty
  var tokens: Map[String, String] = Map.empty

   val routes: Route =
    path("register") {
      post {
        entity(as[User]) { newUser =>
          if (users.contains(newUser.username)) {
            complete(StatusCodes.BadRequest, "Username alread
          } else {
            // add user and generate token
            complete(StatusCodes.Created, token)
          }
        }
      }
    }

    val binding = Http().newServerAt("localhost", 8080).bind(
}
```

```scala
trait JsonSupport {
  implicit val userFormat: RootJsonFormat[User] = j
  implicit val tokenFormat: RootJsonFormat[Token] =
}

object AuthClient extends App with JsonSupport {
  implicit val system: ActorSystem[Unit] = ActorSys
  val baseUrl = "http://localhost:8080"

    def register(user: User): Future[Unit] = {
    Http()
      .singleRequest(
        HttpRequest(
          method = HttpMethods.POST,
          uri = s"$baseUrl/register",
          entity = HttpEntity(ContentTypes.`applica
        )
      )
      .flatMap { response =>
        println(s"Received response: ${response.sta
        if (response.status.isSuccess()) {
          Unmarshal(response.entity)
            .to[Token]
            .map(token => println(s"Registration su
        } else {
          Unmarshal(response.entity).to[String].map
        }
      }
  }
}
```

ScalaLoci approach

```
@multitier object AuthService {
  @peer type Server <: { type Tie <: Single[Client] }
  @peer type Client <: { type Tie <: Single[Server] }

  private var users: Map[String, User] on Server = Map[String, User]()
  private var tokens: Map[String, String] on Server = Map[String, String]()
  private val _register: Evt[User] on Client = Evt[User]()

  on[Server] {
  _register.asLocal observe { case newUser: User =>
      if (users.contains(newUser.name)) {
        failure.fire("Username already exists!")
      } else {
        // add user and generate token
        success.fire("Created", token)
      }
    }
  }

  def register(user: User): Unit on Client =
    _register.fire(user)
}

object AuthServiceServer extends App {
  multitier start new Instance[AuthService.Server]
  (listen[AuthService.Client](TCP(43053)))
}

object AuthServiceClient extends App {
  multitier start new Instance[AuthService.Client](
    connect[AuthService.Server](TCP("localhost", 43053))
  )
}
```

Word Counter

- Master-Worker architecture
- Functionalities:
  - Master
    - provides PDF files for workers to process
    - collects the results from the workers
    - ranks the top 10 words
  - Workers:
    - process these PDFs
    - count the number of occurrences of each word
    - send the counts back to the master

Traditional approach: Akka Http, Spray Json, Pdf Box

Multitier approach: ScalaLoci, Pdf Box

Traditional approach

```
object Master extends JsonFormats {
  private val wordRanker = new WordRanker()
  private val pdfQueue: mutable.Queue[String] = mutable.Queue

  def main(args: Array[String]): Unit = {
    if (args.length != 1) {
      println("Usage: Master <directory-path>")
      System.exit(1)
    }

    val directoryPath = args(0)
    pdfQueue ++= listPdfFiles(directoryPath)

    implicit val system: ActorSystem[_] = ActorSystem(Behavi

     val route = handleExceptions(myExceptionHandler) {
     path("task") {
       get {
         println("Received task request")
         if (pdfQueue.nonEmpty) {
           val pdfPath = pdfQueue.dequeue()
           complete(ProcessPdf(pdfPath))
         } else {
           complete(StatusCodes.NoContent) // No more PDFs t
         }
       }
     }
   }
  }
}
```

```
object Worker extends JsonFormats {
  def startWorker(): Unit = {
    implicit val system: ActorSystem[_] = ActorSyst

    val wordCounter = new WordCounter()

    val responseFuture = Http().singleRequest(HttpR

    responseFuture.flatMap { response =>
      if (response.status == StatusCodes.NoContent)
        println("No more tasks available.")
        system.terminate()
        Future.failed(new Exception("No tasks avail
      } else {
        Unmarshal(response.entity).to[ProcessPdf]
      }
    }.foreach { task =>
      println(s"Processing PDF ${task.path}")
      val wordCounts = wordCounter.countWordsFromPd

      val jsonData = s"{\"wordCounts\": ${wordCount
            Http()
        .singleRequest(
          HttpRequest(
            method = HttpMethods.POST,
            uri = "http://localhost:8080/collect",
            entity = HttpEntity(ContentTypes.`appli
          )
        )
    }
  }
}
```

ScalaLoci approach

```
case class PdfTask(pdfPath: String) {
  def exec: Map[String, Int] = // execution logics
}
```

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

  private val pdfTaskStream: Local[Evt[PdfTask]] on Master = Evt[PdfTask]()

  private val deployedTask = on[Master] sbj { worker: Remote[Worker] =>
    Signal(assocs().get(worker))
  }

  private val pdfTaskResult = on[Worker] {
    deployedTask.asLocal.changed collect { case Some(task) =>
      task.exec
    }
  }

  private val aggregatedResults: Signal[Map[String, Int]] on Master = {
    pdfTaskResult.asLocalFromAllSeq.fold(Map.empty[String, Int]) { case (acc, (_, result)) =>
      mergeWordCounts(acc, result)
    }
  }

    def main(): Local[Unit] on Master =
    aggregatedResults.changed observe { result =>
      val topWords = result.toSeq.sortBy(-_._2).take(10)
      println(s"Top 10 Words:\n${topWords.mkString("\n")}")
    }
}
```

Thank you for your attention!