Multitier Programming and ScalaLoci Analysis

Linda Vitali
linda.vitali2@studio.unibo.it

August 2023

Contents

1. Introduction

The problems of distributed systems are well-known and have been studied for decades. The main challenges are related to the fact that distributed systems are inherently concurrent, asynchronous and that they are composed of multiple components that communicate over a network. Using a multitier architecture is advantageous for distributed systems, as it allows for separating the different components of the system and for having a clear separation of concerns. For example, a 3-tier (or 3-layer) application is organized into three major parts usually presentation, application processing, and data management residing in different network locations. By organizing a system into tiers, the functionality that is encapsulated into one of the tiers can be modified independently, instead of redesigning the entire application.

However, writing multitier applications is still a challenging task as it usually requires to use different programming languages for different tiers,to make the tiers communicate with each other, and it usually makes the programmer consider one tier at a time, instead of the system as a whole.

Multitier languages' purpose is to simplify the development of distributed applications by bringing the development of code closer to programming single-host applications.

The project's purpose is to analyze and illustrate how multitier languages (ScalaLoci[1] in particular) work and how they can be used to solve some of the aforementioned problems.

Examples of distributed systems written in ScalaLoci will be provided and analyzed. A comparison with the same systems written in a "standard" programming language will be provided to highlight the advantages of using ScalaLoci.

[1] https://scala-loci.github.io/

2. Multitier Programming

Multitier programming (MT) is a programming paradigm for distributed systems that is based on the multitier architecture model, which consists of physically separating the application into tiers. Traditionally, tiers are developed using different programming languages e.g. Javascript for the client, Java for the server, and SQL for the database.

Multitier languages, in contrast, allow writing the functionalities that belong to different tiers within a single code base and a single programming language. The code for different tiers is then either generated at run time or results from a compilation process that splits the codebase into components that have to be deployed on different tiers. This is possible by leveraging user annotations, static analysis, types, etc. A simple schema of the compilation process is shown in Figure 1.

The first field of application of multitier programming was web development, given that web applications are inherently distributed having the client and the server running on networked systems and the fact that are sometimes developed using different programming languages and as multiple programs. Different multitier languages has been developed for this purpose, some of the most famouse are Hop[2], Links[3], GWT[4], and many others.

Even though nowadays projects like Node.js allow programmers to write both the client and the server in Javascript, multitier programming is still relevant for web development as the client and the server are nonetheless developed as two separate programs and executed on different machines making it challenging to ensure the semantic coherence of the distributed execution.

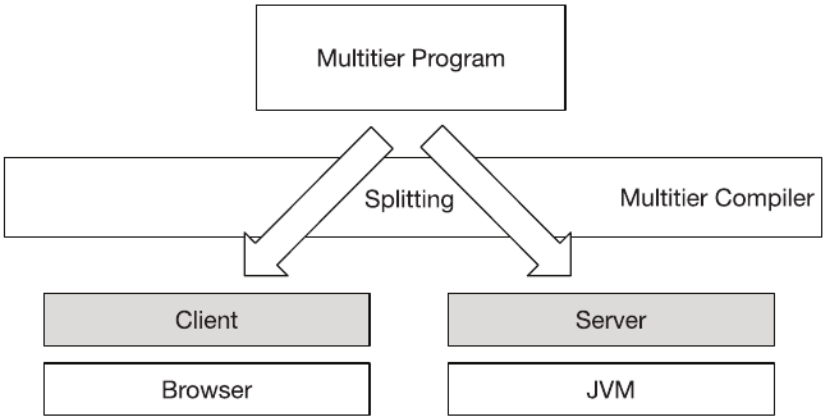

Figure 1: This image

[2]http://hop.inria.fr/
[3]https://links-lang.org/
[4]https://www.gwtproject.org/

2.1. Multitier languages characteristics

One of the core characteristics of multitier languages is that they utilize a "placement strategy" to allocate data and tasks to the various hosts in a distributed system and language primitives are usually provided to allow computation to change the location of execution. This allocation can either be done automatically or explicitly determined by the developer that might use location tags to guide specific placements. In the case of automatic placement, the language runtime is responsible for deciding where to execute the code and some segments are pre-assigned to particular areas - for instance browser DOM interactions should occur client-side.

Multitier languages offer unique abstractions designed to streamline remote communication, but these vary across languages. They either:

- Allow one-way communication, from either the client to the server or vice versa.
- Facilitate bidirectional communication, often necessitating client initiation.

These languages have diverse stands on:

- Making remote communication evident (thus highlighting performance consequences) or making it unseen to the developer.
- Embedding remote communication features directly within the language or offering them via the accompanying standard library.

Once the method is chosen, the language must rely on a communication system. Some of the most common are:

- **Remote Procedures:** This is the most common, they mimic local functions and offer a network layer abstraction. They can either be seamlessly integrated or demand a unique remote invocation syntax.
- **Message Passing:** This method aligns with the communication patterns of foundational network protocols, transferring messages between hosts.
- **Publish-Subscribe:** This allows system levels to follow specific topics and get updates from others based on those topics.
- **Reactive Programming:** Defines inter-tier data flows via events or mutable values that, when altered, auto-update the corresponding reactive values on distant tiers.
- **Shared State:** Updates to a communal data structure on one tier become available to the others accessing that data structure.

2.2. Advantages of Multitier Programming

Multitier programming(MT) offers several advantages over traditional distributed systems development, some of the most relevant are:

- **Higher Abstraction Level:** MT programming abstracts low-level distributed system details, simplifying software development. It eliminates challenges like network communication, serialization and data conversions. There's no need for inter-tier API design (for example REST API).
- **Improved Software Design:** In distributed apps, boundaries between functionalities and hosts can blur. MT programming avoids splitting functionalities between different tiers (with the consequent modularity compromise). It promotes single-time development and placement where needed.
- **Formal Reasoning:** MT languages present distributed apps comprehensively. They highlight typically implicit distributed systems aspects, forcing the designer to consider them and to take into account the system as a whole and not every component separately.
- **Code Maintenance:** Altering software is more straightforward with MT programming. Features can shift between tiers without entire rewrites. Migrating across platforms, like from desktop to Web, is made simpler by changing compilation targets.
- **Program Comprehension:** MT programming facilitates understanding data flow across hosts. Continuous data flow for specific functions is maintained, making development and debugging simpler.

2.3. Limitations of Multitier programming

Multitier programming is still a relatively new paradigm and subject to ongoing research, so there are still some limitations to be addressed. Some of the most relevant are:

- **Generic Distributed Systems:**
  - Most MT languages focus primarily on the client–server architecture, specifically for web applications. They do not cater to more intricate architectures of distributed systems which may have various component types like different client types, coordinators, backup nodes, etc.
  - Existing MT languages primarily address a single level of consistency, often determined by the communication system, thus limiting the flexibility for developers who might require different consistency levels for performance or safety.
  - There's an absence of dedicated abstractions in MT languages for building fault-tolerant systems, such as actor supervision trees.
- **Failures:** While hosts in distributed settings, including the Web, can unexpectedly fail or disconnect, the response of MT languages to these challenges varies.

Some languages offer non-blocking remote communication, and others have specialized mechanisms to detect and handle disconnections or failures.

- **Programming in the Large:**
  - Many MT languages lack dedicated abstractions for modularization, which poses scalability challenges for larger codebases and potentially compromises collaborative development and maintainability. A critical challenges is the need for modular compilation processes due to tier-specific code separation.
- **Controlled Experiments:** There's a lack of empirical studies and controlled experiments targeting MT programming, which makes it challenging to quantify aspects such as development time, program comprehension, and overall efficacy.

These challenges highlight areas that need further research and development to fully leverage the potential of MT programming.

ScalaLoci tries to address some of those issues, its approach will be discussed in more detail in the following sections.

3. ScalaLoci Analysis

3.1. Overview

Multitier programming languages are often integrated into general-purpose languages by extending them with support for distribution. This is the case of ScalaLoci, a distributed programming language that extends Scala with multitier abstractions.

ScalaLoci addresses the previously discussed problems of distributed systems - like the need to use different languages for different and separated modules and the manual implementation of communication that forces programmers to deal with low-level details - and provides a solution to some of the issues that othe MT languages have.

ScalaLoci allows the programmer to abstract over low-level communication details and data conversions and brings to the world of multitier languages the possibility to specify an architecture based on peer types, thus, supporting generic distributed systems (not just client-server), whose architecture can be defined by the developer. Also, with ScalaLoci is possible to associate locations to data rather than to computations only, like existing approaches.

The ScalaLoci model is developed over the *placement types*, a concept that enables reasoning about distributed data flows and supports different architectures.

There is support for reactive programming, a programming paradigm that allows the programmer to express the flow of data and the propagation of changes. Reactive programming is a natural fit for distributed systems since distributed applications are in many cases reactive (e.g. events from the network, user input, transfer data among hosts and trigger state changes or new events). Having reactive programming in ScalaLoci is extremely useful since developers can compose data flows over multiple hosts.

In the following subsections, we will discuss the main concepts of ScalaLoci in more detail.

3.2. Language constructs

The main language constructs of ScalaLoci are *peers* and *ties*. A peer represents the different kinds of components of the system, like a server, a client, a database, etc. A tie specifies the kind of relation among peers, like Single, Multiple, or Optional. Remote access is possible between tied peers e.g. the multi-client-server architecture has a single tie from the client to the server, and the server has multiple ties to the client. Peers, ties, and placements are known statically and the compiler checks that

access to remote values is consistent with the architecture definition. The following code snippet shows a multi-client-server architecture in ScalaLoci:

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

Another example could be the peer-to-peer architecture where each peer has multiple ties to other peers:

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

3.2.1. Placement types

Placement types are the core concept of ScalaLoci. ScalaLoci uses the peers defined in the system architecture to allow specifying the placement of data and computations. Placement is statically defined and part of a value's type. Placed data of type `T on P` represent a value of type T that is placed on peer P. For instance, is possible to define a value of type `Int` (e.g. the node's ids) that is placed on a peer of type `Node`:

```
@peer type Node <: { type Tie <: Multiple[Node] }
val id: Int on Node = UUID.randomUUID().hashCode()
```

To access the value of a placed data there are different ways depending on the type of tie. For instance, to access the value of a placed data on a peer with a single tie, it is used the keyword `asLocal`. Accessing remote values using the keyword `asLocal` creates a local representation of the remote value by transmitting it over the network or by establishing a remote dependency.

To access the value of a placed data on a peer with multiple ties, the keyword `asLocalFromAll` or `asLocalFromAllSeq` have to be used. The following snippet shows an example of accessing multiple remote values, making the id of the connected nodes locally available:

```
@peer type Node <: { type Tie <: Multiple[Node] }
val connectedNodesIds = id.asLocalFromAll.observe(m => m.foreach { case
(remote, event) => event observe println })
```

By default placed data is accessible remotely from other nodes. The visibility of a value can be restricted by using `Local`:

```
val id: Local[Int] on Node = UUID.randomUUID().hashCode()
```

Now the value `id` can only be accessed locally from the same peer instance. This is useful to prevent other peers from accessing the value.

It is important to notice that ScalaLoci makes remote communication explicit to remind the developer of the latency of the network and potential failures of network communication.

3.2.2. Multitier Reactives

As previously mentioned, ScalaLoci supports reactive programming in the style of systems like REScala[5]. ScalaLoci's reactive abstractions are `Events` and `Signals`. Events are used to represent discrete changes, while Signals are used to represent continuous time-changing values. Signals' values are updated by the language runtime. It is possible to define the sending of a message as an `Event` on the `Client` node as follows:

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

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

Every time the `Client` wants to send a message, an event has to be triggered:

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

Now, the `Server` can observe the event and react to it:

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

3.3. Key Concepts of Distributed Systems in ScalaLoci

3.3.1. Asynchronicity and Fault Tolerance

ScalaLoci handles distributed systems issues like asynchronicity and fault tolerance. For instance, remote access to reactive values (via `asLocal` or `asLocalFromAll`) is asynchronous and creates a local representation of the reactive extending the data flow graph without blocking. In the RP style, propagating values asynchronously is transparent to the programmer but is also possible to access the current value of a reactive from imperative code. This is safe since accessing the current value of a reactive from imperative code returns a `Future`.

Failures are handled by ScalaLoci similarly to how it is done with actors. This is achieved thanks to a dependency relationship between reactive elements, where one reactive "supervises" another and is notified if the supervised reactive fails. In this approach, errors can propagate through the data flow graph, similar to frameworks like Rx and Reactive Streams. Reactives can either ignore computation failures or handle them by recovering from errors. The goal is to ensure that a failed computation doesn't hinder processing of subsequent inputs.

[5] https://www.rescala-lang.com/

3.3.2. Consistency

ScalaLoci adopts a consistency model similar to many actor systems, involving at-most-once delivery for remote reactives and order preservation for sender-receiver pairs. Developers can also achieve stronger consistency on top of ScalaLoci, such as at-least-once delivery, both manually through message resending or using one of the supported backends like for instance Rx.

3.3.3. Concurrency

In ScalaLoci, there's a default approach of utilizing a single thread for both evaluating values and processing remote blocks that are initiated by remote instances. This default behavior can be changed to enhance concurrent access by implementing a thread pool. While sequential remote accesses originating from the same remote instance are processed in a consistent order, there's no assurance regarding the order of remote access performed by different instances.

3.4. Programming large systems in ScalaLoci

As shown in the previous sections, ScalaLoci provides ways to specify different architectures supporting larger systems. This means that a mechanism to separate and modularize the code is necessary. The ScalaLoci module system supports encapsulating each functionality (that could be cross-peer) and defining it over abstract peer types. This enables the realization of distributed systems as a composition of multitier modules, each representing a subsystem.

3.4.1. Multitier modules

LociMod[1] supports strong interfaces to achieve encapsulation and information hiding, such that implementations can be easily exchanged. This approach makes peer types abstract enabling the definition of abstract modules, which capture a specific component of a distributed system. Every module can be further composed with other abstract modules and can eventually be instantiated for a concrete software architecture. Moreover, it is possible to apply mechanisms like encapsulation. This allows developers to design applications based on logical functionalities rather than network boundaries.

The following example shows an example of encapsulation using a module for a Chat that defines a `Client` peer type and a `Server` peer type.s The `ChatApp` also requires an instance of the MessageService multitier module to handle messaging functionalities.

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

The `MessageService` module specifies two peer types: UserInterface for handling user interactions and Database for storing and retrieving messages. The sendMessage and fetchMessages methods can be invoked on the UserInterface peer and make remote calls to storeMessage and retrieveMessages methods on the Database peer. This module encapsulates all the messaging functionalities, including the communication between the `UserInterface` and the `Database`.

```
@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
}
```

The same example can be realized using modules as interfaces. For instance, considering the previously mentioned `MessageService` trait as an interface we can implement this interface in various ways. For example, one implementation might use a file system to store messages and another might use a database.

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

  def sendMessage(message: String): Unit on UserInterface
  def fetchMessages(): Future[List[String]] on UserInterface
}
```

The following snippet shows an implementation of the `MessageServiceAsInterface` module using a file system to store messages.

```
@multitier trait FileMessageService extends MessageServiceAsInterface {
  override def sendMessage(message: String): Unit on UserInterface =
    placed { remote call writeToFile(message) }

  override def fetchMessages(): Future[List[String]] on UserInterface =
    placed { (remote call readFromFile()).asLocal }

  private def writeToFile(message: String): Unit on Database = // logics
  private def readFromFile(): List[String] on Database = // logics
}
```

The following snippet instead, shows an implementation of the `MessageServiceAsInterface` module using a database to store messages.

```
@multitier trait DatabaseMessageService extends MessageServiceAsInterface {

  override def sendMessage(message: String): Unit on UserInterface =
    placed { remote call dbInsert(message) }

  override def fetchMessages(): Future[List[String]] on UserInterface =
    placed { (remote call dbQuery()).asLocal }

  private def dbInsert(message: String): Unit on Database = // logics

  private def dbQuery(): Future[List[String]] on Database = // logics
}
```

To instantiate concrete implementations of the `MessageServiceAsInterface` module, the following code can be used:

```
@multitier object Chat extends ChatAppInterface {
  @multitier object messageService extends DatabaseMessageService
}
```

Every peer instance only contains the values placed on the respective peer.

Other than encapsulation and the definition of module interfaces, LociMod modules enable abstracting 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.

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

As shown in the example, The peers defined in a module can be specialized with the role of other modules' peers. This is useful if you don't need the `UserInterface` or the `Database` to be a physical peer in the system, but you want to define a logical place.

As previously mentioned, multiter modules can be mixed-in and this enables including the implementations of different subsystems in a single module. Mixing modules doesn't mean that the peers are merged, so it is necessary to use mechanisms like subtyping or overriding to specify that a peer also implements the placed values of the overridden or subtyped peers.

```
@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] }
}
```

One last important feature provided by LociMod, is the possibility to specify constrained modules. This means that it is possible to express that a functionality is required by a module to make it work. This is realized by Scala's self-type annotations. An example can be the realization of a module that ensure mutual exclusion for a shared resource. This can be done using a coordinator that handles the access to the shared resource. Such coordinator can be chosen using a leader election algorithm.

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


@multitier trait LeaderElection[T] {
@peer type Node
 def electLeader(): Unit on Node
 def electedAsLeader(): Unit on Node
}
```

To use the `MutualExclusion` module, it is necessary to mix it in with a concrete implementation of the `LeaderElection` module.

A particularly useful usage of the self-type constraint is the possibility of abtracting over the concrete architecture of a system.

Taking as an example an `Executor`, we want to define a module where this class can be executed regardless the architecture of the system.

```
@multitier trait Architecture {
  @peer type Node <: { type Tie <: Multiple[Node] }
}

@multitier class Executor { this: Architecture =>
  private def execute(): Unit on Node = println("execute")
  def main(): Unit on Node = execute()
}
```

The `Executor` module can be instantiated with different architectures, like for instance a peer-to-peer architecture

```
@multitier trait P2P extends Architecture {
  @peer type Peer <: Node { type Tie <: Single[Peer] }
}

@multitier object ExecuteSystem extends Executor with P2P
```

or any other architecture that satisfies the `Architecture` module's constraints

```
@multitier trait ClientServer extends Architecture {
  @peer type Client <: Node { type Tie <: Single[Server] with Single[Node] }
  @peer type Server <: Node { type Tie <: Single[Client] }
}
```

In this example, the `Executor` module is instantiated with the `ClientServer` module and the `Server` runs the `execute` method of the executor.

```
@multitier object ExecuteSystem extends Executor with ClientServer{
  override def main(): Unit on Node = {
    on[Node] {
      super.main()
    } and on[Server] {
      println("Server")
      super.main()
    } and on[Client] {
      println("Client")
    }
  }
}
```

4. Case Studies

In this section, there will be presented some case studies to show the differences between a "traditional" approach and a ScalaLoci approach. The first example is an authentication service, the second is a word counter.

4.1. Authentication Service

The first example is a simple authentication service that provides the following functionalities:

- 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

4.1.1. Traditional approach

The architecture is composed of a server and a client. The application components are shown in the following figure:

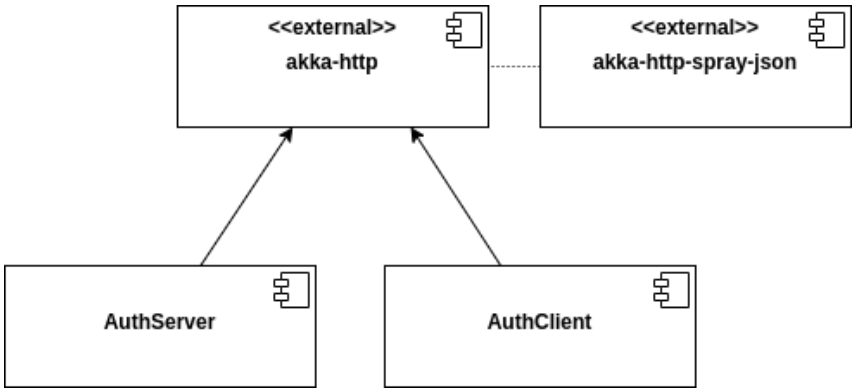

Figure 2: Traditional approach. Application's components

Both the server and the client are implemented using Akka HTTP and use Spray-JSON to serialize/deserialize data.

The server initializes a mutable list of users, which serves as a mock database to store registered users. Then, it sets up the Akka HTTP routes for register, login, and secure functionalities:

- When the server receives a POST request at the `/register` endpoint with `User` data (containing a name and password), it adds this user to its list of registered users if the username is not already taken.
- When the server receives a POST request at the `/login` endpoint with `User` data (containing a name and password), it checks if the user is registered and if the

password is correct. The server generates a token (for demonstration purposes, it's a hardcoded "token" string) and responds with this token.
- When the server receives a POST request at the `/login` endpoint with `User` data (a name and password), it checks its list of registered users to verify the provided credentials. If the credentials are valid, returns a token.
- The secure route represents a secure endpoint that requires authentication. When the server receives a GET request at the `/secure` endpoint with a valid token, it responds with a message (for simplicity). If the token is invalid, it sends an error message.

The client initializes the Akka HTTP system and provides methods for the user to interact with the server, specifically for registration, login, and accessing secure content.
- The client displays a main menu allowing the user to choose an action: register, login, access secure content, or exit.
- If the user chooses to register, the client prompts the user for a name and password.

The client then sends a POST request to the server's /register endpoint with the provided user data. Once the registration response is received, the client displays the server's response.
- If the user chooses to login, the client prompts for a name and password.

The client then sends a POST request to the server's /login endpoint with the provided user data. Once the login response is received, if successful, the client stores the token and displays it. If not successful, it displays an error message.
- If the user chooses to access secure content, the client sends a GET request to the server's /secure endpoint, providing the previously received token as authentication.

The server's response is then displayed to the user, which will either be a secure message or an error message, depending on the token's validity.

The application handles the errors that may occur during the execution of the above functionalities.

4.1.2. ScalaLoci approach

The multitier example in ScalaLoci has a client-server architecture and provides the same functionalities as the traditional approach. Using this approach the code can be written switching between the client and server within the same code. The programmer can simply specify whether a certain functionality is implemented on the server or the client leveraging the *placement types*. The architecture is specified in two lines of code:

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

In this example, a Server has a single tie with a Client and vice versa. The register, login, and secure functionalities are implemented as events triggered by the client and observed by the server leveraging the reactive primitives provided by ScalaLoci.

```
  val _register: Evt[User] on Client = Evt[User]()
  val _login: Evt[User] on Client = Evt[User]()
  val _secure: Evt[String] on Client = Evt[String]()
```

Once the client fires one of the above actions, the server reacts accordingly by remotely accessing the events using `asLocal`:

```
  on[Server] {
    _register.asLocal observe { case newUser: User =>
       // register logics
    }

    _login.asLocal observe { case incomingUser: User =>
      // login logics
    }

    _secure.asLocal observe { case token: String =>
      // secure logics
    }
  }
```

The success and failure messages are handled as events as well, and the client can observe them and display the messages to the user:

```
   val success: Evt[(String, AuthElem)] on Server = Evt[(String, AuthElem)]
()
  val failure: Evt[String] on Server = Evt[String]()

  on[Client] {
    success.asLocal observe { case (message, value) =>
      println(s"Message: $message value: $value")
    }
    failure.asLocal observe { case message => println(s"Error: $message") }
  }
```

The application is deployed as follows:

```
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))
  )
}
```

The client and server are connected using the TCP protocol.

The serialization/deserialization of data is handled by ScalaLoci, and the programmer does not need to worry about it when accessing remote values. In fact, statically typed access to remote values in ScalaLoci ensures that they are serializable. In this example, the values `User` and `Token` are made transmittable by adding the following implicits:

```
case class User(name: String, password: String)
object User {
  implicit val authorTransmittable: IdenticallyTransmittable[User] =
      IdenticallyTransmittable()
  implicit val authorSerializer: ReadWriter[User] = macroRW[User]
}

case class Token(value: String)
object Token {
  implicit val tokenTransmittable: IdenticallyTransmittable[Token] =
      IdenticallyTransmittable()
  implicit val tokenSerializer: ReadWriter[Token] = macroRW[Token]
}
```

4.2. Words Counter

The second example is a word counter and implements a master-worker pattern where the master provides PDF files for workers to process. The workers process these PDFs, count the number of occurrences of each word, and then send the counts back to the master. The master then aggregates these word counts and ranks them.

4.2.1. Traditional approach

The master-worker communication in the traditional approach is facilitated using Akka HTTP. Also, for processing PDFs and extracting text, the code uses PDFBox, a popular Java library for working with PDF documents. JSON serialization/deserialization is handled by Spray-Json.

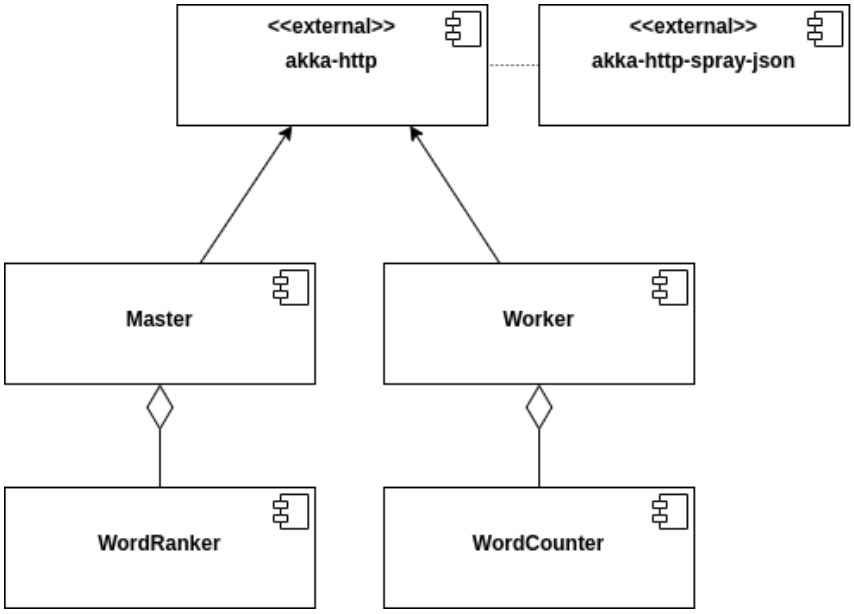

Figure 3: Traditional approach. Application's components

- **WordsRanker**: This class maintains an aggregated count of word occurrences. It provides methods to add new word counts and retrieve the top-ranked words based on their count.
- **Master**: It initializes the actor system and starts an HTTP server. Exposes two endpoints:
  - `/task`: Workers request for a PDF task. The master dequeues a PDF path from its internal queue and provides it to the worker.
  - `/collect`: Workers post their word count results for a given PDF. The master aggregates these counts.
- **WordsCounter**: This class provides a method to count word occurrences in a given PDF.
- **Worker**: It starts by requesting a PDF task from the master. Once received, a worker processes the PDF, counts the words, and sends the result back to the master's /collect endpoint.

The master is started with a directory path containing PDF files as an argument. It loads all PDF filenames into a queue. Multiple worker instances are started (10 in the example). Each worker requests a PDF file from the master to process. A worker reads its assigned PDF, counts the words, and then sends the word count to the master. The master aggregates word counts from all workers. The master continuously logs the top 10 words based on the aggregated counts received so far.

4.2.2. ScalaLoci approach

The master-worker example in ScalaLoci has a master-worker architecture and provides the same functionalities as the traditional approach. The system consists of two peers:

- Master: Responsible for managing tasks, collecting results, and communicating with multiple workers.
- Worker: Processes the assigned task and returns the result to the master.

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

Task distribution and processing are implemented as events triggered by the master and observed by the workers leveraging the reactive primitives provided by ScalaLoci.

- `pdfTaskStream`: is a local event stream on the `Master` for new PDF tasks.
- `deployedTask`: a signal defined on the `Master`, captures the current task assigned to a `Worker`.
- `assocs`: a signal on the `Master` maintaining the association of connected workers with their tasks. The logic to assign free workers to tasks or queue tasks if no workers are available is captured in the `assignTasks` function.
- `aggregatedResults`: accumulates the word counts received from all workers.
- `pdfTaskResult`: on the `Worker` side processes the assigned task and produces the word counts.

The same consideration as the previous example about serialization and deserialization can be done.

4.3. Run the applications

4.3.1. Authentication Service

- To run the **traditional approach**, execute the following commands using `sbt` on different terminals:

  Run the server:

  ```
  > sbt "runMain traditional.authenticator.AuthServer"
  ```

  Run the client:

  ```
  > sbt "runMain traditional.authenticator.AuthClient"
  ```

- To run the **ScalaLoci approach**, execute the following commands using `sbt` on different terminals:

  Run the server:

```
> sbt "runMain multitier.AuthServiceServer"
```

Run the client:

```
> sbt "runMain multitier.AuthServiceClient"
```

4.3.2. Words Counter

To run both the traditional and the ScalaLoci example a directory containing PDF files is required.

- To run the **traditional approach**, execute the following command using `sbt`:

  Launch the `Master`

  ```
  > sbt "runMain traditional.wordcounter.Master /path/to/pdf/directory"
  ```

  Launch the `Worker` (this will launch 10 workers):

  ```
  > sbt "runMain traditional.wordcounter.Workers"
  ```

- To run the **ScalaLoci approach**, execute the following command using `sbt`:

  ```
  > sbt "runMain multitier.Main /path/to/pdf/directory"
  ```

5. Conclusion

Multitier Programming (MT) has emerged as a promising paradigm for distributed systems, offering advantages such as higher levels of abstraction, improved software design, and easier code maintenance. With origins deeply rooted in web development, MT allows for the development of client and server functionalities within a single codebase, simplifying the complexities traditionally associated with distributed systems. However, there are still challenges to tackle, such as the focus on client-server architecture, the lack of flexibility for different levels of consistency, and certain limitations concerning fault tolerance.

The ScalaLoci language has come to address some of these challenges, offering comprehensive solutions for generic distributed systems and code modularization, introducing concepts like placement types, and leveraging multitier reactives.

The module system in ScalaLoci adds another layer of sophistication, enabling the design of large systems through encapsulation and abstract modules.

Yet, the field is young and much is left to explore and require further research. As the demand for distributed systems continues to grow, the continued development and maturation of multitier programming languages are expected.

This project has been a great opportunity to learn about multitier programming and ScalaLoci, and to explore the possibilities of the ScalaLoci module system. Most importantly, implementing the same example with different approaches has allowed me to better understand the advantages and disadvantages of each approach, and to compare the different implementations.

Bibliography

[1] P. Weisenburger, and G. Salvaneschi, "Multitier Modules," in *33rd Eur. Conf. Object-Oriented Program. (ECOOP 2019)* in Leibniz International Proceedings in Informatics (Lipics), vol. 134, Dagstuhl, Germany, 2019, pp. 1–29, doi: 10.4230/LIPIcs.ECOOP.2019.3. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10795 (Keywords: Distributed Programming, Multitier Programming, Abstract Peer Types, Placement Types, Module Systems, Scala)

[2] M. Serrano, and G. Berry, "Multitier programming in hop," *Commun. ACM - Cacm*, vol. 55, 2012, doi: 10.1145/2240236.2240253.

[3] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *Proc. 21st Int. Conf. Softw. Eng.* in Icse '99, Los Angeles, California, USA, 1999, p. 107, doi: 10.1145/302405.302457. [Online]. Available: https://doi.org/10.1145/302405.302457

[4] P. Weisenburger, J. Wirth, and G. Salvaneschi, "A survey of multitier programming," *ACM Comput. Surveys*, vol. 53, 2020, doi: 10.1145/3397495.

[5] P. Weisenburger, and G. Salvaneschi, "Developing distributed systems with multitier programming," in *Proc. 13th ACM Int. Conf. Distrib. Event-Based Syst.* in Debs '19, Darmstadt, Germany, 2019, p. 203, doi: 10.1145/3328905.3332465. [Online]. Available: https://doi.org/10.1145/3328905.3332465

[6] S. Giallorenzo, F. Montesi, et al., "Multiparty Languages: The Choreographic and Multitier Cases," in *35th Eur. Conf. Object-Oriented Program. (ECOOP 2021)* in Leibniz International Proceedings in Informatics (Lipics), vol. 194, Dagstuhl, Germany, 2021, pp. 1–27, doi: 10.4230/LIPIcs.ECOOP.2021.22. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14065 (Keywords: Distributed Programming, Choreographies, Multitier Languages)

[7] P. W. and Guido Salvaneschi, "Implementing a language for distributed systems: choices and experiences with type level and macro programming in scala," *Corr*, 2020. [Online]. Available: https://arxiv.org/abs/2002.06184