Bridging Between Active Objects: Multitier Programming for Distributed, Concurrent Systems

Guido Salvaneschi[0000−0002−9324−8894] and Pascal Weisenburger[0000−0003−1288−1485]

University of St. Gallen, Switzerland
{pascal.weisenburger,guido.salvaneschi}@unisg.ch
https://programming-group.com/

Abstract. Programming distributed and concurrent systems is notoriously hard. Active objects, which encapsulate operations, state and control flow, have been investigated by researchers to alleviate this issue. In a distributed system, message exchange among active objects or actors often coincides with network boundaries, and determines a major modularization direction for the application. Yet, certain application functionalities naturally crosscut such modularization direction. For those, structuring the application architecture around network boundaries is purely accidental and does not help reasoning about programs.
Recently, multitier programming has been proposed as a programming paradigm that enables code that belongs to different peers to be developed together, in the same compilation unit. The compiler then splits the code and generates the required deployment components.
In this work we explore the relation between multitier programming and active objects. Multitier programming can be considered a programming paradigm based on active objects with a focus on application domains where functionalities span multiple active objects, and allows such functionalities to be encapsulated into a single object. The multitier approach keeps the asynchronous model of active objects and actors but provides a holistic view of distributed components and their interactions. Multitier programming addresses the use cases where separating components into different active objects or actors hinders encapsulation and modularization across functional boundaries. In such use cases, multitier programming can increase the level of abstraction, improve software design, simplify code maintenance, aid program comprehension and enable formal reasoning. A number of features of active objects are directly visible to programmers also in the multitier programming, resulting in an interesting combination of language abstractions available to developers.

1 Introduction

Modern-day ubiquitous services – including search engines, online social networks and streaming platforms – run on a network of interconnected computers. Typically, the components of such distributed systems are developed as separate modules. This separation, however, comes with a number of difficulties [23].

Composing the modules correctly into a complete distributed system is a highly difficult challenge that requires intensive integration work and the manual implementation of communication protocols. Thus, programmers are faced with the complex task of implementing complicated communication schemes between hosts, which frequently involves low-level operations prone to errors. As a result, the distributed data flows that arise from the approach are in many cases convoluted and scattered among several modules, making it difficult to fully comprehend the behavior of the system as a whole. Despite the prevalence of distributed software, the design and development of distributed systems remains an extremely challenging task.

Multitier programming [95] presents a promising approach for taming the complexities of developing distributed systems through language abstractions focusing on interacting distributed components, i.e., interacting active objects where distributed objects are bound to different threads of control and communicate asynchronously with other objects.

Active objects. Active objects [16] build on top of object-oriented abstractions which encapsulate operations and state, and, in addition, encapsulate execution flow. Active objects have been successfully adopted to program distributed systems, which are concurrent by nature. This programming abstraction defines clear boundaries among concurrency units, making them coincide with those naturally defined by the object structure. Also, because of asynchronous message passing, active objects simplify developing concurrent systems where different parts are decoupled and progress independently. Yet, in the case of distributed systems, active objects and their derivatives (e.g., actors) encourage programmers to develop software that is modularized according to network boundaries – where the remote communication occurs. While this has been necessary for technical reasons, software functionalities can logically span over several system components and such separation may be not ideal [84, 72, 37].

Multitier programming. In multitier programming, distributed functionalities that cross different components are developed in a single compilation unit [95]. As a result, programmers do not need to arrange the implementation along network boundaries but along logical functions. Since the distributed parts of such a function naturally run concurrently in a distributed system, multitier programming languages typically further abstract over active objects and rely on active objects for an efficient implementation. Yet, multitier languages provide features to deal with concurrent execution – either by exposing standard concurrency abstractions such as futures or by abstracting concurrency away from the developer through a compilation scheme that ensures that code that appears sequentially in the multitier program is also executed in sequence.

Using a single (distributed) program relieves the developer from having to break down a functionality into the parts that should be executed on different machines. Instead of reasoning in terms of distributed components (that may mix different functions together), developers can reason in terms of different modules that functionally belong together (even though they are distributed themselves) –

leaving the splitting into the components to be distributed to different machines to the compiler [93].

Active objects and multitier programming. While multitier programming provides linguistic abstractions to reason about distribution at the language level and lets the compiler handle the actual partitioning of code and the insertion of remote calls and handlers, the underlying execution model is fundamentally distributed active objects that asynchronously invoke methods on remote objects. Multitier programming, however, goes beyond active objects – and actor systems in particular – and solves some of the issues that often arise in actor-based implementations of distributed systems. In particular, control flow between actors can get quite involved and hard to follow for developers due to the fact that actors are highly decoupled and behavior of actors that interact to provide a logically combined function is modeled through complex message-passing schemes. Such functions can be expressed more directly using multitier programming. Hence, multitier programming can be seen as both an evolution and a combination of active objects for distributed systems.

In this work, we explore the connection between active objects and multitier programming. We show that multitier programming addresses some of the design issues that emerge with active objects, and we show that active objects complement multitier programming when reasoning about concurrency in a distributed system.

2 Background

This sections provides a short overview about (1) active objects and actors as a state-of-the-art programming model for concurrent and distributed systems and (2) multitier programming, a programming paradigm designed to ease the development of distributed applications where functionality spans across multiple components.

2.1 Active Objects and Actors

The actor model [47] is based on independent computational entities – so-called *actors* – that encapsulate both behavior and state and communicate with each other by sending and receiving messages. An actor's internal state can only be modified by processing incoming messages. When an actor receives a message, it can perform computations, modify its state, create new actors or send messages to other actors [44]. Actors process one message at a time – messages are processed sequentially – and do not share their state or memory, which eliminates many of the pitfalls of traditional multithreaded programming.

Active objects extend the actor paradigm with structured communication: Instead of message-passing, they use method calls and futures. Futures represent asynchronous return values that will be available at some point in the future.

Listing 1: Akka Typed actor summing up a list of numbers.

```
1  object SumActor {
2    sealed trait Request
3    case class Shutdown() extends Request
4    case class Calculate(
5      numbers: List[Int], replyTo: ActorRef[Result]) extends Request
6
7    case class Result(result: Int)
8
9    def apply(): Behavior[Request] =
10     Behaviors.receive { (context, message) =>
11       message match {
12         case Calculate(numbers, replyTo) =>
13           replyTo ! Result(numbers.sum)
14           Behaviors.same
15         case Shutdown() =>
16           Behaviors.stopped
17       }
18     }
19 }
```

Usually, method calls on active objects look like "normal" method calls on (non-active) objects. They thus remove the fraction in actor systems of fragmenting programs into (1) sending messages to emulate method invocations and (2) sending other messages to emulate their return values. Hence, active object became popular as an improved way to structure concurrent code.

Yet, not being able to easily distinguish synchronous and asynchronous method invocations can also be a disadvantage. Especially in – not only concurrent, but also – distributed systems, remote methods often should not only be executed asynchronously but they also have more fundamentally different invocation semantics. In particular, a remote method may even never return in case of partial failures (i.e., the remote system crashes) or network partitions (i.e., the remote system is not reachable over the network anymore). Due to these differences between local and remote methods, actors – that distinguish between local method calls and remote message-passing – remain widely used for distributed systems.

Listing 1 shows an actor implemented in Akka Typed [59]. Lines 2 to 7 define the messages that the actor can send and receive. Line 10 defines the actor's message handler that has to pattern-match on every type message the actor could receive (Lines 11 to 17). In case the actor receives a `Calculate` message with a list of numbers (Line 12), it calculates the sum of the numbers and sends out the result using the `!` send operator (Line 13). In the example, the `Calculate` message includes an actor reference `replyTo` to which to send the result – a common pattern to model returning values in actor systems.

Another reason for using actors for implementing distributed systems is their support for fault tolerance using supervision hierarchies. Supervision hierarchies

organize actors into a tree, where an actor acts as a supervisor for its children and monitors their behavior [24]. Supervisors can then take appropriate actions to handle errors in supervised actors, such as restarting a failed actor, stopping it or propagating the error up the hierarchy.

Whereas actors proved effective to implement fault-tolerant distributed systems and actor systems are widely deployed, such systems may fall short of achieving encapsulation of distributed functionalities because the scope of a component in a distributed system is tied to an object or actor, i.e., to call a method asynchronously, it has to be part of the public interface of an object. Remote calls across objects or messages sent across actors often lead to code with obscured data and control flow that is hard to read and follow.

2.2 Multitier Programming

Multitier programming is an approach for developing distributed systems, which provides language abstractions to reason about different tiers of a distributed system – for example, a client, server and a database tier – in the same compilation unit. The code for the different tiers is either generated at run time or created by the compiler. Code annotations, static analysis, types, or a combination of these approaches are used to separate the code into components that correspond to the various tiers.

A distributed application is composed of several tiers that can run on various computers connected through a network. A typical three-tier architecture, for example, consists of the presentation, application logic, and data management tiers, each of which runs at a different network location. The benefit of this approach is that each tier's functionality can be updated independently.

However, because of this architectural choice, a functionality that cuts across multiple tiers is now scattered across numerous compilation units. For instance, functionality on the Web is frequently spread across the client and the server. The tiers of a Web application are further typically implemented using different programming languages, such as JavaScript for the browser interface, Java for the server-side application logic and SQL for the database. Multitier languages aim to reduce the separation between client and server by compiling client-side code to JavaScript or by running JavaScript on the server.

In a multitier programming language, the different tiers can be programmed in a single language. Depending on the target tier, different compilation backends (such as Java for the server and JavaScript for the browser) are used. Consequently, functionality that spans multiple tiers can be developed within a single compilation unit. The compiler automatically adds the communication code necessary for components to interact while the program is being executed, generating numerous deployable units from a single multitier program (Figure 1).

The multitier approach's ultimate goal is to improve program comprehension, make maintenance easier and enable formal reasoning about the entire distributed application. A number of research languages that adopt multitier concepts have been proposed and show the advantages of the approach, such as improving software comprehension, design, reasoning and maintenance. As a result, ideas

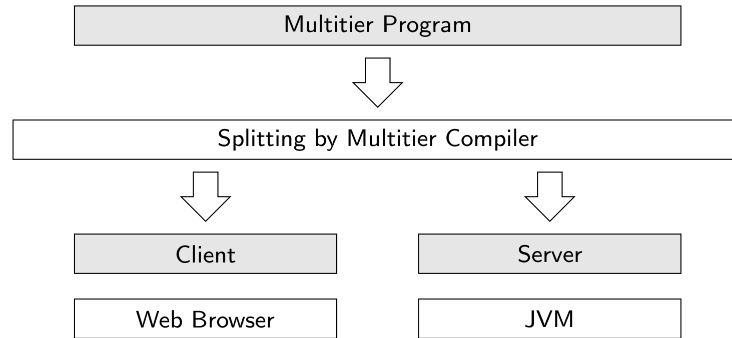

Fig. 1: Multitier programming [adopted from 95].

from multitier programming have been included into a number of industrial solutions, demonstrating the potential of this approach, such as Ocsigen [12], Opa [83], WebSharper [14], Meteor [89] or GWT [52]. Different multitier languages cover different areas of the design space, integrating various techniques (such as compile time vs. run time splitting) and design choices (such as the placement of compilation units vs. individual functions), which frequently depend on the application domain and the software stack.

3 Modular Structuring of Asynchronous Communication

This section illustrates how developers structure asynchronous communication in distributed systems, comparing actor systems and multitier programming. For both approaches, we first describe them conceptually and then we demonstrate how these concepts are applied in a real-world stream processing system.

Active objects and actors proved to be an effective abstraction for developers to organize concurrent code [2]. They allow developers to reason in terms of a sequential execution environment within an active object or actor and communicating with others through asynchronous method calls or message-passing. Although actor framework implementations can reuse threads across actors, processing messages sequentially in every actors hides a potentially multithreaded execution environment.

The downside of this approach is that concurrency boundaries are closely tied to objects boundaries. This means that the concurrent parts of a system need to be separated into different objects. In some cases, splitting concurrent parts into separate objects aligns naturally with the problem domain and allows for independent reasoning about the concurrency aspects. In other application scenarios, however, this separation may increase the complexity of the implementation. Dependencies and interactions between concurrent objects can become intricate, making it harder to comprehend and maintain the system.

A Peak at Distribution and Concurrency in Apache Flink. To support the discussion about the design enabled by actor systems, we introduce a concrete application which makes extensive use of actors. Apache Flink [4] is a widely-used stream processing system. It features a distributed data-flow engine implemented in Scala and Java, which can pipeline and execute data-parallel programs. To increase performance, Flink is able to run different components on different machines in a computer cluster to distribute the load for processing a data stream across computers.

We look specifically into the task distribution system of Apache Flink, which provides Flink's core task scheduling and deployment logic. The task distribution system is based on Akka actors [58] and consists of 23 remote procedures in six *gateways* – an API that encapsulates sending actor messages into asynchronous RPCs – amounting to ~ 500 source lines of Scala code with complex interaction patterns. In the Flink task distribution system, a *JobManager* actor is responsible for assigning data processing tasks to *TaskManager* actors.

3.1 The Actor Approach

Actor languages provide dedicated features to represent different concurrently executing components of a distributed system – so-called actors. Actors naturally capture the concurrent nature of distributed systems and significantly simplify the development of such systems in several ways, making them suitable for building distributed and highly available systems.

Concurrency Abstraction. The primary feature of the actor model – both for distributed and for local concurrent systems – is that it offers a structured way to manage concurrency without worrying about low-level synchronization primitives [1]. Further, the model ensures that actors operate in isolation and their internal state is not directly accessible by other actors. This isolation simplifies concurrent programming as actors do not need to be aware of each other's internal state or execution details.

Fault Tolerance and Scalability. A notable benefit of the actor model, in particular in a distributed setting, is its fault tolerance [64]. Since actors are isolated from each other, failures in one actor do not directly impact others. If an actor crashes or becomes unresponsive, it can be restarted or replaced without affecting the overall system. For the same reason, the actor model also promotes scalability. New actors can be added or removed dynamically without affecting the overall system, enabling flexible scaling of the application to changing demands or requirements.

Modularity. The actor model also fosters modularity since actors are independent entities, which both encapsulate their private state and can be tested individually. However, an important aspect of the behavior of the entire distributed system stems from the communication and interaction between the actors, which can become quite complex, especially in systems with a large number of actors and

Listing 2: Communicating Flink actors [adopted from 93].

(a) Message definition.

```
package flink.runtime

case class SubmitTask(td: TaskDeployment)
```

(b) Calling side.

```
package flink.runtime.job

case class SubmitTask(td: TaskDeployment)

class TaskManagerGateway {
  def submitTask(td: TaskDeployment, mgr: ActorRef) =
    (mgr ? SubmitTask(td)).mapTo[Acknowledge]
}
```

(c) Responding side.

```
package flink.runtime.task

class TaskManager extends Actor {
  def receive = {
    case SubmitTask(td) =>
      val task = new Task(td)
      task.start()
      sender ! Acknowledge()
  }
}
```

intricate dependencies. Understanding the behavior of individual actors in a complex system can be challenging [94]. As actors operate independently and asynchronously, tracing the flow of messages and identifying the root cause of issues can be more difficult compared to more traditional programming models.

The Actor Version of Apache Flink. As it is commonly done in actor-based distributed systems today, the different distributed components of Apache Flink are implemented as different actors. As usual, communication between Flink actors is based on message-passing. Besides Flink, a number of other open-source projects (e.g., the Play Framework for web applications [57] or the Gatling load- and performance-testing framework [40]) and companies (e.g., PayPal [60] or Capital One [61]) use Akka actors.

Concurrency Abstraction. Listing 2 shows an excerpt of the – extensively simplified – interaction of the `TaskManagerGateway` with the `TaskManager`, taken from Apache Flink's task distribution system. The snippets show an example

of sending and receiving of only a single message. The `TaskManagerGateway` is used by the *JobManager* actor to communicate with the *TaskManager* actor to submit data processing tasks to the *TaskManager*. Note that, in contrast to Listing 1, Flink uses the untyped version of Akka with a slightly different syntax. Listing 2a defines the `SubmitTask` message that is exchanged between the actors and which contains the meta data for the task to be executed by the TaskManager. Listing 2b shows the sending of the message (Line 7) from the JobManager using the `?` send operator. As opposed to the fire-and-forget style of the `!` send operator (shown in Listing 1), the `?` operator implements a request-response pattern. With this operator, the next message from the addressed actor is treated as a response, which is then made available as result of `?` in the form of a future containing the response message. Listing 2c shows the receiving of the message on the TaskManager. The TaskManager defines the actor message loop as its `receive` method (Line 4) that pattern-matching on the received messages (Line 5) and carries out a computation that depends on he received message (Lines 6 to 8), e.g., starting the task that was assigned by the JobManager.

The full `receive` methods of course contains a multitude of cases for the different messages which the actor can handle. While messages can be sent to an actor concurrently, programmers can safely assume that only one message is processed in the message loop at a time, relieving them from the complexities of handling intricate concurrency problems such as race conditions when accessing the actor's internal state from inside the message loop.

Fault Tolerance and Scalability. The actor-based design allows Flink to easily scale up to a large number of nodes to keep up with an increasing incoming stream of data to be processed. To achieve this, Flink can spawn actors on additional computer node to handle processing parts of the stream. For example, if the system requires additional computing power to process increased amounts of data, the JobManager can submit processing task to additional TaskManagers to carry out the processing work. Further, thanks to the actor model, if nodes fail or become unresponsive, Flink can re-spawn the respective actor (potentially on another node), making the system highly tolerant to faults.

Modularity. A potential issue of the actor model's message-passing scheme – where messages sent in some part of the code are processed by a completely separated part – in terms of code comprehension and maintenance is that it is not straightforward to map call sites modeled by sending messages to the sites where the messages are handled, which convolutes the control flow between the different actors, making it hard for developers to keep track.

The small code excerpt (Listing 2), illustrates how the task submission functionality is scattered over different modules, making it difficult to correlate sent messages (Listing 2b, Line 7) with the remote computations they initiate by pattern-matching on the received message (Listing 2c, Lines 6 to 8). Further, it is worth noting that the message loop of the TaskManager does not only handle a single type of message sent via the TaskManagerGateway. Due to the modularization enforced by the actor's remote communication boundaries, the

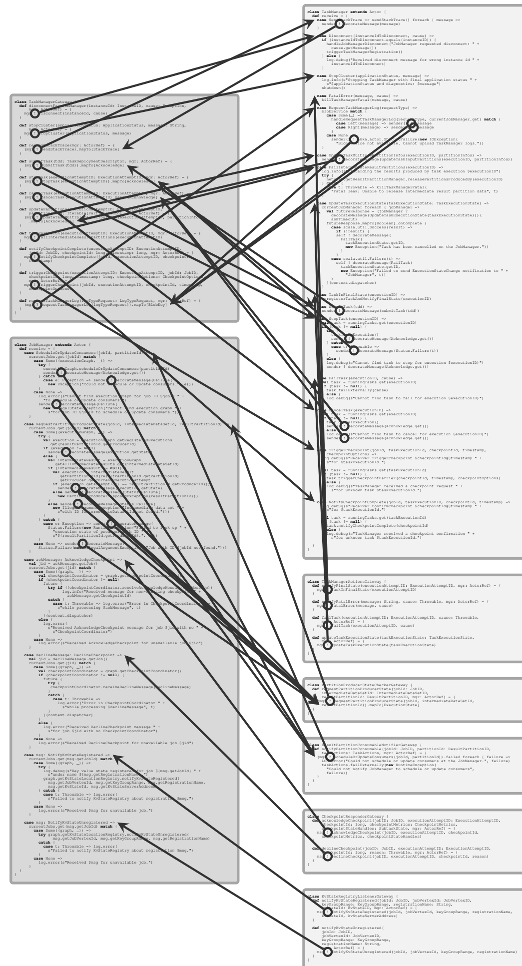

Fig. 2: Communication of two actors in Apache Flink [adopted from 94].

message loop also needs to handle messages belonging to unrelated functions that should be executed on the TaskManager.

To provide a broader overview of the scattered control flow in Apache Flink, Figure 2 depicts a larger portion of the communication between the two actors. The figure shows a part of the JobManager implementation (dark gray boxes, left), the TaskManager implementation (light gray boxes, right) and their communication (arrows). Every box is an actor which is confined by network boundaries. Thus, cross-host data flow belonging to the same (distributed) functionality is scattered over multiple objects.

Notably, Flink implements its own abstraction over message-passing that encapsulates the sending of messages into asynchronous remote procedure calls.

As such, Flink is essentially using active objects, built on top of actors. Most of these calls are processed in a different compilation unit within another package, making it difficult to correlate the messages sent with the remote computations they trigger.

A closer look at the code reveals that the reason these remote procedures are implemented as public object methods is not because they represent a reusable function – in fact they often only have a single call site. Instead, the reason is that this structure to organize distribution is imposed by the actor model, which tempts to combine unrelated functionalities into a single actor because they incidentally run on the same component rather than properly separating them.

3.2 The Multitier Approach

Multitier programming follows the active objects or actor systems approach of providing developers with language abstractions for explicitly defining concurrent entities and the code they execute. The focus of multitier programming is to provide and enhance the language features for handling the communication between these entities [85]. Such entities, like active object or actors, are represented by different *tiers* in multitier programming [30]. Tiers decouple the concurrency (and distribution) abstractions from the objects/actors. Hence, the concurrency boundaries do not need to be at the level of objects. Different methods of the same object can run concurrently at different locations.

Multitier programming thus addresses the modularization issues of active objects and actors [93] that especially arise in application scenarios where remote functions are not loosely coupled but work closely together to achieve a common goal and provide a joint functionality.

Basic multitier language features. Multitier languages typically give the developer full control over where values are placed and computations are executed using a variety of different techniques such as annotations, types or multi-stage programming. For illustration, we will use a language where placement is expressed in the types. For example, a value of type `Int on Server` represents an integer value that lives on the server, and a method of type `String on Client` represents a method that will execute on the client and return a string. A main method, which runs on the client when it starts, has the signature `main(): Unit on Client` – i.e., the method receives no arguments, has a void return value (i.e., it returns the singleton unit value) and lives on the client. Calling methods that live on other tiers looks similar to traditional (local) method calls. In particular, remote calls are fully type-checked across distributed components and remote methods are looked up according to the usual scoping rules (e.g., defined in the same lexical scope, imported, inherited, etc.). In the following presentation, remote calls are explicit through the `remote call` marker. Note that there also exist multitier languages in which remote calls are transparent.

Listing 3 places the main method on the client (Line 1), where it keeps reading line-by-line from standard input (Line 2). For every line, it calls the `fire` method

Listing 3: Method with specified placement.

```
1 def main(): Unit on Client =
2   for (line <- io.Source.stdin.getLines)
3     remote call fire(line)
4
5 def fire(message: String): Unit on Server =
6   remote call show(line)
7
8 def show(message: String): Unit on Client =
9   println(line)
```

on the server remotely (Line 3), which in turn calls the `show` method on the client remotely (Line 6) to print the line to standard output (Line 9). The remote call to `fire` (Line 3) requires the `remote call` marker since the `fire` method is placed on the server (Line 5) but is invoked within the `main` method placed on the client (Line 1). Hence, it is statically known where remote calls appear and which method is invoked to handle them.

As the example illustrates, multitier programming brings programming distributed applications closer to the development of "traditional" non-distributed applications. Both method definitions and calls look similar to the usual way of defining and calling methods – with the only new language features being the ones required for distribution, namely the specification of the placement and marking remote accesses.

Multitier languages typically take the possibility to compose code on different tiers one step further and do not only allow methods to be placed on tiers but also expressions inside methods. For example, Listing 4 implements the same logic as the snippet above but nests the expressions to be run on the client (Lines 1 and 4) and on the server (Line 3) inside each other.

In our example language, an expression of the form `on[T].run` is used to divert the control flow to another tier. In this language design, we also require developers to explicitly list the values that should be transferred to another tier using the `capture` clause. This design choice aims to avoid accidental captures which are distinctively more costly in a distributed setting – compared to captures in local closures, for example – as they require additional data to be transmitted over the network. The compiler issues an error if variables are used in a nested placed block without being explicitly captured, as this situation may indicate a potentially expensive programming mistake.

Distributed architectures. Multitier languages use different underlying system architectures or application topologies. Historically, multitier programming focused on client–server Web applications. Hence, most approaches have a server and a client – and sometimes a database – as the only supported tiers baked into the language model. They differ in whether they treat the server side as the single instance of the server code that serves a connected client or as one server instance

Listing 4: Nested code blocks with specified placement.

```
def main(): Unit on Client =
  for (line <- io.Source.stdin.getLines)
    on[Server].run.capture(line) {
      on[Client].run.capture(line) {
        println(line)
      }
    }
```

Listing 5: Distributed architecture specification.

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

  def main(): Unit on Client = /* ... */
  def fire(message: String): Unit on Server = /* ... */
  def show(message: String): Unit on Client = /* ... */
}
```

serving all connected clients. The former case leads to a one-to-one connection between server and client sides. In such case, the example presented in Listing 3 will lead to the server echoing the message from the client back to the same client that sent the message. The latter case leads to a one-to-many connection between server and clients. In the example (Listing 3), the server would then forward the message from one client to all connected clients, essentially implementing a simple command line chat.

While the underling topology may be implicit and built into the language, we will use a multitier language that makes the involved peers and their architectural connection explicit – extending the scope beyond the Web and the client–server model. In Listing 5, we assume that the `main`, `fire` and `show` methods are implemented as before (Listing 3) and part of the `Chat` object. Lines 2 and 3 define the peers and their relation: A server that can handle multiple clients and a client that is connected to a single server.

Modularization, encapsulation, composition. Separating the distribution aspect from the object structure in multitier programming allows developers to return to using OOP abstractions for structuring, modularizing and composing their code based on distinguishing functionalities rather than locations. A single module – e.g., an object, class, trait, mixin, depending on the abstractions offered by the language – can contain functionalities that are themselves distributed. Hence, a module can abstract also over distributed functionalities: Distribution will not leak if it should not be exposed as part of the public interface. To integrate

Listing 6: Abstract multitier module.

```
1  @multitier trait Chat {
2    @peer type Server <: { type Tie <: Multiple[Client] }
3    @peer type Client <: { type Tie <: Single[Server] }
4
5    def main(): Unit on Client
6
7    protected def fire(message: String): Unit on Server =
8      remote call show(line)
9
10   protected def show(message: String): Unit on Client
11 }
```

Listing 7: Concrete implementation of abstract multitier module.

```
1  @multitier object CommandLineChat extends Chat {
2    def main(): Unit on Client =
3      for (line <- io.Source.stdin.getLines)
4        remote call fire(line)
5
6    protected def show(message: String): Unit on Client =
7      println(line)
8  }
```

functionality defined in different modules, developers can use the usual techniques such as inheritance, delegation, composition or mixins.

For example, we can define different variants for the chat examples, e.g., one using a command line interface (like before, Listings 3 and 5) and another one using a graphical user interface. First, as shown in Listing 6, we can factor out the architecture (Lines 2 and 3) and the common methods (Lines 5, 7 and 10), leaving the implementation of the methods abstract (Lines 5 and 10) that are to be implemented by a specific variant.

The command line chat variant (Listing 7) then only needs to implement the abstract methods (Lines 2 and 6), inheriting the architecture and the distributed functionalities form the `Chat` trait defined in Listing 6. In the example, the methods that are only relevant to the module or its sub-modules are access-protected using the usual `protected` visibility modifier.

Multitier programming goes beyond active objects and the actor model by enabling the separation of distribution concerns and OOP mechanisms used for modularization and composition. The ability to declare placement as an orthogonal dimension in the language relieves the developer from having to manually model placement and being forced to align the structure of the program with the boundaries of active objects or actors. Multitier objects can be composed

Listing 8: Communicating Flink peers [adopted from 93].

```
1  @multitier object TaskManagerGateway {
2    @peer type JobManager <: { type Tie <: Multiple[TaskManager] }
3    @peer type TaskManager <: { type Tie <: Single[JobManager] }
4
5    def submitTask(td: TaskDeployment, tm: Remote[TaskManager]) =
6      on[JobManager] {
7        on(tm).run.capture(td) {
8          val task = new Task(td)
9          task.start()
10         Acknowledge()
11       }
12     }
13 }
```

like standard objects but different parts of their code can be run across different distributed components.

A Multitier Version of Apache Flink. In the case of Apache Flink, many remote procedures could be expressed more directly using nested remote expressions. Listing 8 shows a multitier variant of the interaction between JobManager and the TaskManager of Listing 2. The multitier version uses an intra-module cross-peer remote call (Line 7) to execute the data processing task on the TaskManager (Lines 8 to 10). Thus, related functionalities are kept inside the same `TaskManagerGateway` module and the multitier module contains the functionality that is executed on both the `JobManager` and the `TaskManager` peer.

Figure 3 shows a reimplementation of Figure 2 using the multitier approach. The cross-peer data flow in the system is much more regular – thanks to the reorganization of the same code in a single unit – and thus much easier to track.

In the Flink example, the different distributed sub-functionalities of the task distribution system can be encapsulated into their own module. Besides the module already shown in Figures 2 and 3, the task distribution system consists of five further individual functionalities. Figure 4 shows the task distribution system module (background), composed by mixing together the modules for the different sub-functionalities (foreground). Cross-peer data flow (arrows) is encapsulated within modules and is not split over different modules. As before, the data flow in each module module spans across the JobManager (dark gray) and TaskManager (light gray) peers.

4 Discussion

This section discusses the similarities and differences among active objects, actors and multitier programming. Finally, we highlight the areas where multitier programming strives for improvement compared to alternative approaches.

4.1 Active Objects vs. Actors vs. Tiers

Fig. 3: Flink: Multitier approach [from 94].

Multitier programming adopts the same approach of actors and active objects to decouple method invocation and method execution. Invoking a method on an active object – or sending a message to an actor – returns immediately. The method itself is executed – or the message is dispatched – asynchronously. Hence, the multitier approach retains the basic asynchronous execution model of active objects and actors. In fact, every tier can be thought of as an active object on which the methods that are placed on the tier can be executed. The execution of a remote method is necessarily asynchronous since the threads of execution of the different tiers are – even physically – separated. Some languages hide the asynchronicity from the developer by compiling the multitier code to continuation-passing style and invoking the continuation only when the remote result becomes locally available [30]. Other languages make the asynchronicity of the result explicit by having remote methods return futures [94] – similar to how asynchronicity is often handled in active objects – or by employing coroutine-based approaches for cooperative multitasking [29].

Multitier languages, however, provide a holistic view on the distributed components and their interaction. They tackle situations where the interaction between active objects or actors hinders encapsulation and proper modularization along meaningful functional boundaries, when different places are treated as different objects or actors. To achieve this goal, the multitier paradigm is characterized by linguistic features for expressing different tiers and their interaction. Therefore, multitier programming especially focuses on application scenarios where systems that are designed "as a whole" and a holistic view simplifies the reasoning about the system for the developer. In these scenarios, multitier programming can serve to bridge the communication across distributed active entities.

4.2 Development Benefits

In summary, the development benefits of the multitier programming paradigm revolve mainly around the following aspects.

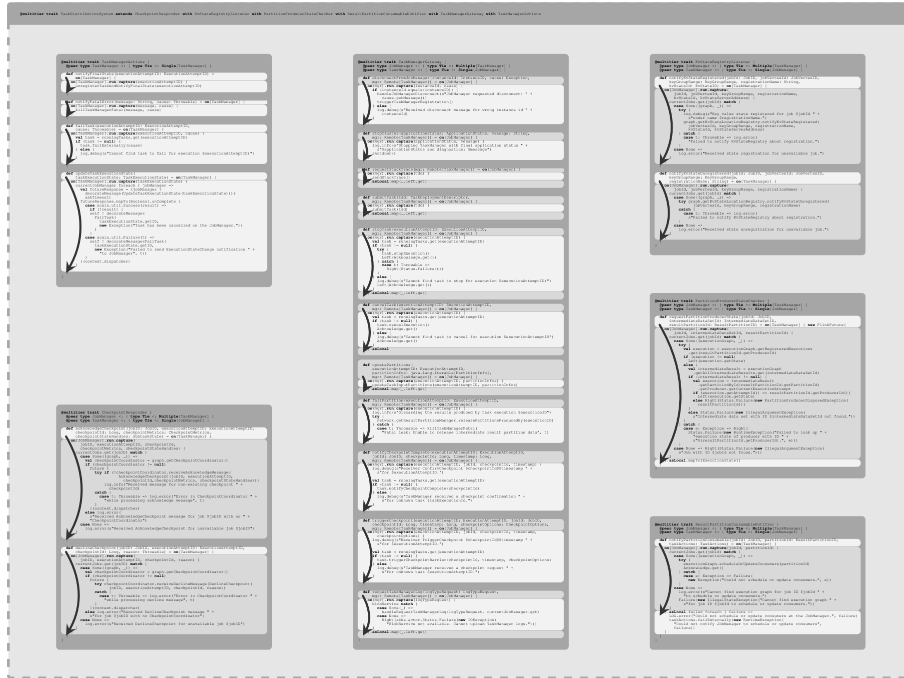

Fig. 4: Communication of two actors in Apache Flink: Multitier modularization [adopted from 93].

Higher abstraction level. Multitier programming simplifies software development for distributed systems by abstracting away low-level details such as network communication, serialization, and data format conversions [82], allowing developers to work at a higher level of abstraction [94]. With multitier programming, there is no need for manual design of inter-tier APIs, as the underlying technologies used for inter-tier communication are transparent to the developer [85].

Improved software design. In distributed applications, the boundaries between hosts and functionalities may not always align, with functionalities spanning multiple locations and a single location hosting multiple functionalities. Programming each location separately introduces two issues: compromised modularity and code repetition. Multitier programming addresses these problems by enabling the development of a functionality once and placing it where needed [36].

Formal reasoning. Multitier design improves formal reasoning by explicitly modeling distributed applications and capturing important aspects such as placement, system components, and tier boundaries. This approach enables thorough analysis of software properties across the entire system, instead of treating components in isolation. It supports reasoning about concurrency [76], security [13], performance optimization [26], as well as domain-specific properties like reachability in software-defined networks [75].

Code maintenance. Multitier programming simplifies the process of modifying software systems in two notable ways. First, it allows for migrating functionality between different tiers without the need for a rewrite in a different language [43] (i.e., validating user input on both the client and server sides without code duplication). Second, it provides easier migration of applications across different platforms [46] (e.g., simply changing the compilation target to JavaScript for a Web client).

Program comprehension. Multitier programming simplifies program comprehension – i.e., the complexity that programmers face to develop a correct mental model of a program [88] – by enabling seamless data flow across multiple hosts, eliminating communication code details and interruptions from forcing modularization along peer boundaries. Thereby, multitier programming simplifies development and debugging [66]. Yet, so far, we lack empirical studies or controlled experiments measuring the specific advantage of multitier programming in terms of program comprehension.

5 A Research Roadmap

This section outlines open challenges and opportunities for future research on multitier programming.

Dynamic placement. Existing multitier languages assign the application functionalities to the nodes in the system based on various mechanisms, such as user annotations, types, and static analysis. Serrano et al. [85] and Cooper et al. [30] introduced multitier languages that incorporate two places as annotations on functions: client and server. Murphy et al. [74] developed a type system based on modal logic to represent different places as *possible worlds*. Type-based approaches have also been used to describe the interaction of places. Notably, multiparty session types [49] provide static specifications for communication protocols. Choreographic programming [41] ensures safe communication protocols across different locations encoded by different type parameters. Information flow type systems have been employed to define the placement of data and computation, preventing the leakage of private data to untrusted parties [99].

All these mechanisms are based on *compile time* assignment of functionalities to nodes. Yet, in distributed systems, often, functionalities need to be assigned to nodes during the program execution, usually to improve performance.

Dynamic placement decisions play a crucial role in various computing domains. First, both computation and the resulting data placement can be dynamically decided. For instance, in a master–worker system, the master leverages information about the execution environment and the job's parameters to determine the most suitable worker [97]. In this scenario, the computed result is dynamically placed at the location where it was generated. Second, computation can depend on data that is being transferred between different places. For instance, data that is frequently accessed in a remote database is often stored in a cache for access

speed [7, 101]. An application that operates on data should be able to handle both the database-stored data and the data residing in the cache. Conversely, data that does not exist in either the database or the cache may require distinct handling, e.g., when dealing with data received from a client, it is necessary to sanitize the data prior to storage.

Developers need to tackle the lack of programming abstractions for dynamic placement themselves by manually encoding placement information into the program. The first option is to extract a common interface, treating the system as homogeneous, consisting of a single type of place. This approach, however, leads to a loss of precision necessary to distinguish between various types of places. Consequently, this approach can potentially cause run time errors as the distinctions among nodes are abstracted away. The second option involves not explicitly extracting any interface but relying solely on the programmer's awareness of equivalent functionalities across different places. This approach, however, does not support dynamic placement efficiently as the same functionality needs to be implemented multiple times for the different places, resulting in code repetitions.

Active objects provide a high degree of flexibility to decide their placement dynamically. Actor systems in particular typically provide location transparency, i.e., they abstract away the actors' placement. However, these models lack the static reasoning about placement and the automatic compiler checks found in multitier languages. So far, we lack programming models that combine the strength of both approaches. Finding a practical language abstractions to trade-off static safety guarantees and the ability to decide placement dynamically is still an open problem.

Error handling. In software execution, error occurs in various circumstances, for example when dealing with the external environment (e.g., I/O). Another class of errors occurs in the case of software bugs (e.g., in the case of null values). In distributed systems, software applications are executed on several nodes. Hence the overall probability that (at least) one of the nodes fails increases with the number of nodes. In addition, network connections can also fail, leading to packet loss and system partitioning. These potential errors need to be modeled in the application to give developers the opportunity to execute a reaction. This mechanism can be implemented in various ways. Actor systems provide supervision hierarchies where certain actors that "supervise", i.e., monitor, other actors – the supervisor can react to failing supervised actors. Different actor systems can offer different means to setup supervisors.

In Akka, supervision trees are defined by configuring parent actors with supervisor strategies, which dictate the parent's response to child failures. Akka provides several built-in strategies, such as the `OneForOneStrategy`, which restarts only the failing child, or the `AllForOneStrategy`, which restarts all children upon a single child's failure. Erlang OTP [8] utilizes a similar approach with its supervisor behavior, where a supervisor process is responsible for starting, stopping, and monitoring its child processes. In this case, supervision trees are built by composing supervisors and worker processes, with supervisors specifying

restart strategies and intensity, allowing for fine-grained control over failure handling.

Supervision trees offer several advantages for fault-tolerant systems. First, the hierarchical organization of actors provides a clear structure for error handling, with each level of the tree responsible for a specific subset of actors. Second, the isolation between actors prevents errors from cascading uncontrollably through the system. Also, supervision trees enable self-healing capabilities in a system by restarting failed actors, often allowing the system to recover from failures without manual intervention. This promotes system resilience, as the impact of isolated errors can be minimized and the system can continue functioning despite component failures.

As most multitier languages make the underlying actors visible, e.g., as types or annotations, it is conceivable that a multitier language can provide similar features to define supervision relations. So far, however, multitier languages have not developed such specific fault tolerance mechanisms yet. This is partly due to their origins as languages for web development, where neither the server has the ability to stop or restart clients nor the other way around. Yet, with multitier languages expanding their scope beyond web applications, dedicated features for fault tolerance will become more important. For example, a multitier programming framework could allow developers to specify whether only the failed or all supervised peers should be automatically restarted in case of failures, while also deciding whether to wipe or retain their state – akin to supervision strategies in actor systems.

Consistency. Message ordering guarantees vary in different actor systems. One basic requirement is that messages sent by the same source actor are processed in the same receiving actor in the exact order they were sent. This ordering property transitively holds true even when messages are relayed through multiple actors. For example, if actor A sends messages to actor B, and B forwards them to actor C, actor C receives the messages in the same order they were generated by actor A. However, actor systems like Erlang, do not provide ordering guarantees when actor A sends two messages to actors B and C, and both B and C forward these messages to the same actor D. In such cases, there is no guarantee on the order in which messages are received by D.

Generally, this kind of non-deterministic message ordering can introduce inconsistencies if the processing order of messages affects the outcome. Yet, in distributed systems based on message-passing, guaranteeing a specific sequence of messages may incur overhead, both in terms of maintenance complexities for the developers and in terms of potential run time performance implications.

Further, in distributed actor systems, issues like network problems or node failures can cause delays in message transmission or even result in message loss. If a particular actor relies on a specific message to carry out its operations, the loss of that message can introduce inconsistencies and disrupt the desired behavior. Actor systems are typically stateful with each actor maintaining its own private state. An actor's state can only be modified by the actor itself, usually in response to received messages. Hence, when different actors receive messages in a

different order or miss certain messages, their internal states can diverge. These data inconsistencies across actors can lead to unexpected behavior and undesired outcomes. Thus, handling message loss is crucial to maintaining consistency.

Traditionally, synchronization techniques such as locks are used to prevent such inconsistencies. To ensure that state changes are carried out consistently across multiple operations, operations are grouped into transactions, which are executed atomically, guaranteeing that either all operations take effect or none of them do. Such concurrency control mechanisms, however, are challenging in distributed actor systems. The distributed nature of actor systems, where actors work independently and communicate through message passing, adds complexity to the coordination process. Coordinating multiple actors to achieve consistent outcomes throughout the system requires careful design and synchronization.

These issues often render strong consistency impractical in distributed systems. Instead, these systems often resort to a weaker consistency model, such as eventual consistency. In an eventual consistency model, a certain level of consistency is guaranteed over time but is not enforced immediately.

Reasoning about consistency in distributed system is still an active area of research. To aid developers in understanding the behavior and the consistency of distributed systems, multitier programming offers a valuable approach by providing a holistic view of the system and the interactions among the different actors, multitier programming facilitates reasoning about consistency in distributed systems. One promising direction yet to be explored is the automatic generation of consistency schemes based on a hypothetical sequential execution of multitier code. This approach could offer insights into effectively achieving consistency in distributed systems by automatically deriving suitable consistency mechanisms from the code structure. Additionally, consistency types [48, 53] are a promising technique for statically reasoning about consistency properties. They allow developers to analyze and reason about the expected consistency of distributed systems at compile-time, catching potential inconsistencies early on.

6 Related Work

Programming languages and calculi for distributed systems. Multitier programming emerged from a rich lineage of programming language design for distributed systems, influenced by notable distributed languages such as Argus [62], Emerald [15], Distributed Oz [45, 90], Dist-Orc [5] and Jolie [73]. Additionally, various frameworks for big data processing have emerged in recent years, including Flink [20], Spark [98], Dryad [51], PigLatin [77] and FlumeJava [25]. These frameworks refine and generalize the original MapReduce [35] model, transparently handling fault tolerance, replication, and task distribution. Further, significant contributions have been made towards designing programming languages that cater to specific aspects of distributed systems. For example, conflict-free replicated data types (CRDT) [86, 87] or cloud types [18] ensure eventual consistency, Ericsson's Calvin [79] provides a programming frameworks for mixed IoT/Cloud

development and Spores [68] provide language support for distribution of computations and fault tolerance [69].

Formal calculi have been developed to model distributed systems. They provide varying levels of abstraction for placement and communication across peers. Process calculi such as the π-calculus [70, 71] are especially common to model the behavior of distributed systems. In the π-calculus and its variants, different processes represent the execution threads of the different peers. In particular, the join-calculus [38] defines processes communicating through asynchronous message-passing over channels. The Ambient calculus [21] describes concurrent systems involving mobile devices and computation. It allows the definition of named places where computations take place, with the ability to move ambients between nested places, representing administrative domains and access control. The Cloud Programming Language (CPL) [17] serves as a core calculus for composing services in cloud computing environments. CPL employs an event-based approach and provides combinators that enable secure composition of cloud applications.

Choreographies. Choreographic programming defines a concurrent system as a unified compilation unit, which provides a global description of the interactions and computations among connected components within a distributed system, known as a *choreography* [56, 73, 92]. Similar to multitier programming, the compiler automatically generates implementations for each component [19]. However, choreographic programming differs in that it makes the communication protocol between peers explicit. The compiler ensures that the generated code strictly adheres to this defined flow. The foundations of choreographic programming lie in process calculi [11], which has been used to explore novel techniques in information flow control [63], deadlock-free distributed algorithms [32], and protocols for dynamic run time code updates for components [81]. Giallorenzo et al. [42] make a first attempt to systematically compare choreographic programming and multitier programming.

Aggregate programming. The concept behind aggregate programming is to allow the specification of global behaviors for distributed systems by defining local computations. The Field calculus [9] is designed to specify and execute distributed computations over devices embedded in a spatial environment. This environment might include diverse entities like sensor networks, mobile robots, or other distributed systems where there is a notion of spatial distribution. In Field calculus, computations are expressed in terms of fields, which are functions from space-time to data values. Devices can read and modify the local values of these fields and use the information from neighboring devices to compute new field values. The paradigm promotes the idea that by providing the right local interactions and computations, more complex global behaviors can emerge without the need for central coordination. *Field-based computing* [91, 54, 65] is a programming model where the overall distributed system behavior is understood as producing a *computational field*, i.e., a map from network nodes to values. Among the most important advantages, by abstracting over the role of individual devices, it is

possible to define a programming paradigm where concurrency, asynchronicity, network communication, message loss and failures do not need to be handled explicitly [10]. Both aggregate and multitier programming aim to improve the development of complex distributed systems. In multitier programming, the system architecture is explicitly defined, composed of heterogeneous tiers, each representing a specific function or component. Aggregate computing, on the other hand, builds on a network of homogeneous devices that execute localized computations, which collectively contribute to the overall behavior of the system.

PGAS languages. Partitioned global address space languages (PGAS) [34] offer a programming model designed for high-performance parallel execution. The X10 language [27], for example, parallelizes task execution using a work-stealing scheduler, enabling developers to write highly scalable code. The X10 programming model features explicit fork/join operations to make the communication cost visible. The language's advanced dependent type system [26] captures the specific place to which a reference points. While both PGAS and multitier languages aim to reduce host boundaries between places for simplified development, their scopes differ significantly. PGAS languages primarily target high-performance computing in dedicated clusters, whereas multitier programming focuses on networked distributed systems on the Internet. Hence, places in multitier programming represent the different peers of a distributed system, whereas places in PGAS refer to partitions in a shared global heap address space.

Software architectures. Software architectures [39, 78] organize software systems into components with defined connections and interaction constraints. Architecture description languages (ADLs) [67] specify components, connectors, architectural invariants, and a mapping of architectural models to implementation infrastructure. ArchJava [3] aims to combine architecture specification in the style of ADLs with the actual system implementation in a single language. Hilda [96] is a language at the intersection of multitier and modeling languages and enables automatic partitioning of data-driven multitier software using a declarative language similar to UML. Component models [31], influenced by ADLs and object-oriented design, separate concerns in the entire software system, defining component interfaces and composition mechanisms, and enforcing strong interfaces with other modules. In the distributed setting, component-based development typically models the distributed system components as separate units, forcing developers to modularize along network boundaries.

Big data processing systems. A significant factor contributing to the success of modern big data systems is the availability of a programming interface that – similar to multitier programming – enables developers to program components running on different hosts within the same compilation unit, with the big data processing framework handling communication and scheduling. This kind of systems includes batch processing frameworks like Hadoop [35] and Spark [98] and stream processing systems such as Flink [4] and Storm [6]. Yet, the domain of big data processing systems is limited enough that they can completely abstract

distribution concerns away. Further, the language semantics of these systems visibly differs, e.g., mutable shared variables are transformed into non-shared separated copies.

Operator placement. In contrast to explicit placement methods – such as using annotations as typically found in multitier programming – the operator placement problem focuses on determining the best host for deploying each operator in a distributed system. In this domain, the best placement is the one that maximizes a specific metric like throughput [33, 55] or load [28]. Various approaches have been proposed to address the operator placement problem, including the use of overlay networks where operators are assigned to hosts through random selection [50], network modeling [80] and linear optimization techniques for finding the optimal solution to a constraint problem [22]. While these systems typically consider operators as the deployment unit, Zhou et al. [100] suggest a coarser granularity approach where query fragments, i.e., groups of operators, are deployed to reduce the load on the placement algorithm.

7 Conclusion

Active objects have been studied for long as a language abstraction that encapsulates not only state and operations, like objects, but also a process. This work delved into the multitier programming language paradigm, which is rooted in active objects and improves on some aspects of active objects when a distributed system is conceived as a cohesive unit. In multitier programming, code that belongs to different peers within a distributed system can coexist within the same compilation unit. It is the responsibility of the compiler to split the code into deployment components and add the necessary networking code. We showed that multitier programming achieves positive results in modularizing distributed and concurrent applications, abstracting over network communication and host boundaries, and outlined areas that present open challenges. As active objects are the ideal compilation target for multitier languages and their execution model of such languages is based on interacting active entities, active objects remain visible for developers when they implement data exchange across peers using multitier programming.

Acknowledgements This work has been supported by the Swiss National Science Foundation (SNSF), grant 200429.

Bibliography

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[2] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 19–41, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. ISBN 978-3-540-39722-9. https://doi.org/10.1007/3-540-16042-6_2.

[3] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. https://doi.org/10.1145/581339.581365.

[4] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, December 2014. ISSN 1066-8888. https://doi.org/10.1007/s00778-014-0357-y.

[5] Musab AlTurki and José Meseguer. Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis. In *Proceedings First International Workshop on Rewriting Techniques for Real-Time Systems*, RTRTS '10, pages 26–45, 2010. https://doi.org/10.4204/EPTCS.36.2.

[6] Apache Software Foundation. Storm, 2011. URL http://storm.apache.org/.

[7] Zachary Arani, Drake Chapman, Chenxiao Wang, Le Gruenwald, Laurent d'Orazio, and Taras Basiuk. A scored semantic cache replacement strategy for mobile cloud database systems. In Ladjel Bellatreche, Mária Bieliková, Omar Boussaïd, Barbara Catania, Jérôme Darmont, Elena Demidova, Fabien Duchateau, Mark Hall, Tanja Merčun, Boris Novikov, Christos Papatheodorou, Thomas Risse, Oscar Romero, Lucile Sautot, Guilaine Talens, Robert Wrembel, and Maja Žumer, editors, *ADBIS, TPDL and EDA 2020 Common Workshops and Doctoral Consortium*, pages 237–248, Cham, 2020. Springer International Publishing. ISBN 978-3-030-55814-7. https://doi.org/10.1007/978-3-030-55814-7_20.

[8] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, September 2010. ISSN 0001-0782. https://doi.org/10.1145/1810891.1810910.

[9] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *ACM Transactions on Computational Logic*, 20(1), January 2019. ISSN 1529-3785. https://doi.org/10.1145/3285956.

[10] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi, and Mirko Viroli. Functional programming for distributed systems with XC. In Karim Ali and Jan Vitek, editors, *Proceedings of the 36th European Conference on Object-Oriented Programming (ECOOP '22)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:28, Dagstuhl, Germany, June 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-225-9. `https://doi.org/10.4230/LIPIcs.ECOOP.2022.20`.

[11] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2–113):131–146, May 2005. ISSN 0304-3975. `https://doi.org/10.1016/j.tcs.2004.07.036`.

[12] Vincent Balat. Ocsigen: Typing web interaction with Objective Caml. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 84–94, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9. `https://doi.org/10.1145/1159876.1159889`.

[13] Ioannis G. Baltopoulos and Andrew D. Gordon. Secure compilation of a multi-tier web language. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 27–38, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-420-1. `https://doi.org/10.1145/1481861.1481866`.

[14] Joel Bjornson, Anton Tayanovskyy, and Adam Granicz. Composing reactive GUIs in F# using WebSharper. In Jurriaan Hage and Marco T. Morazán, editors, *Proceedings of the 22nd International Conference on Implementation and Application of Functional Languages*, IFL '10, pages 203–216, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 978-3-642-24275-5. `https://doi.org/10.1007/978-3-642-24276-2_13`.

[15] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. The development of the Emerald programming language. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 11:1–11:51, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. `https://doi.org/10.1145/1238844.1238855`.

[16] Frank de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Computing Surveys*, 50(5), October 2017. ISSN 0360-0300. `https://doi.org/10.1145/3122848`.

[17] Oliver Bračevac, Sebastian Erdweg, Guido Salvaneschi, and Mira Mezini. CPL: A core language for cloud computing. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY '16, pages 94–105, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3995-7. `https://doi.org/10.1145/2889443.2889452`.

[18] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In James Noble, editor, *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP '12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31056-0. `https://doi.org/10.1007/978-3-642-31057-7_14`.

[19] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 263–274, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. https://doi.org/10.1145/2429069.2429101.

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38(4):28–38, 2015. URL http://sites.computer.org/debull/A15dec/p28.pdf.

[21] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, FoSSaCS '98, pages 140–155, Berlin, Heidelberg, 1998. Springer-Verlag. ISBN 978-3-540-64300-5. https://doi.org/10.1007/BFb0053547.

[22] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, DEBS '16, pages 69–80, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4021-2. https://doi.org/10.1145/2933267.2933312.

[23] Mark Cavage. There's just no getting around it: You're building a distributed system: Building a distributed system requires a methodical approach to requirements. *Queue*, 11(4):30–41, April 2013. ISSN 1542-7730. https://doi.org/10.1145/2466486.2482856.

[24] Francesca Cesarini and Simon Thompson. *Erlang Programming – A Concurrent Approach to Software Development*. O'Reilly, Sebastopol, CA, USA, 2009. ISBN 978-0-596-51818-9.

[25] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. https://doi.org/10.1145/1806596.1806638.

[26] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 11–22, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. https://doi.org/10.1145/1345206.1345211.

[27] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. https://doi.org/10.1145/1094811.1094852.

[28] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, CIDR '03, January 2003. URL `http://nms.csail.mit.edu/papers/CIDR_CRC.pdf`.

[29] Kwanghoon Choi and Byeong-Mo Chang. A theory of RPC calculi for client–server model. *Journal of Functional Programming*, 29, 2019. `https://doi.org/10.1017/S0956796819000029`.

[30] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO '06, pages 266–296, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 978-3-540-74791-8. `https://doi.org/10.1007/978-3-540-74792-5_12`.

[31] Ivica Crnkovic, Severine Sentilles, Vulgarakis Aneta, and Michel R. V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, September 2011. ISSN 0098-5589. `https://doi.org/10.1109/TSE.2010.83`.

[32] Luís Cruz-Filipe and Fabrizio Montesi. Choreographies in practice. In Elvira Albert and Ivan Lanese, editors, *Proceedings of the 36th IFIP International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, FORTE '16, pages 114–123, Berlin, Heidelberg, 2016. Springer-Verlag. ISBN 978-3-319-39569-2. `https://doi.org/10.1007/978-3-319-39570-8_8`.

[33] Gianpaolo Cugola and Alessandro Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, February 2013. ISSN 0010-485X. `https://doi.org/10.1007/s00607-012-0217-9`.

[34] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Computing Surveys*, 47(4):62:1–62:27, May 2015. ISSN 0360-0300. `https://doi.org/10.1145/2716320`.

[35] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. `https://doi.org/10.1145/1327452.1327492`.

[36] Gwenaël Delaval, Alain Girault, and Marc Pouzet. A type system for the automatic distribution of higher-order synchronous dataflow programs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '08, pages 101–110, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-104-0. `https://doi.org/10.1145/1375657.1375672`.

[37] Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. Thread-safe reactive programming. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), October 2018. `https://doi.org/10.1145/3276477`.

[38] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, POPL '96, pages 372–385, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. https://doi.org/10.1145/237721.237805.

[39] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994. URL http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf.

[40] Gatling Corp. Gatling, 2011. URL https://gatling.io/.

[41] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects, 2020.

[42] Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. Multiparty languages: The choreographic and multitier cases. In Anders Møller and Manu Sridharan, editors, *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP '21)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:27, Dagstuhl, Germany, July 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-190-0. https://doi.org/10.4230/LIPIcs.ECOOP.2021.22.

[43] Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. WebDSL: A domain-specific language for dynamic web applications. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA Companion '08, pages 779–780, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-220-7. https://doi.org/10.1145/1449814.1449858.

[44] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3): 202–220, February 2009. ISSN 0304-3975. https://doi.org/10.1016/j.tcs.2008.09.019.

[45] Seif Haridi, Peter Van Roy, and Gert Smolka. An overview of the design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation*, PASCO '97, pages 176–187, New York, NY, USA, 1997. ACM. ISBN 0-89791-951-3. https://doi.org/10.1145/266670.266726.

[46] Haxe Foundation. Haxe cross-platform toolkit, 2005. URL http://haxe.org.

[47] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular Actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI '73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL http://ijcai.org/Proceedings/73/Papers/027B.pdf.

[48] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. Disciplined inconsistency with consistency types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 279–293, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. https://doi.org/10.1145/2987550.2987559. URL https://doi.org/10.1145/2987550.2987559.

[49] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. https://doi.org/10.1145/1328438.1328472.

[50] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB '03, pages 321–332. VLDB Endowment, 2003. ISBN 0-12-722442-4. https://doi.org/10.1016/B978-012722442-8/50036-7.

[51] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 987–994, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. https://doi.org/10.1145/1559845.1559962.

[52] Federico Kereki. *Essential GWT: Building for the Web with Google Web Toolkit 2*. Addison-Wesley Professional, 1st edition, 2010. ISBN 978-0-321-70514-3.

[53] Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. Rethinking safe consistency in distributed object-oriented programming. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. https://doi.org/10.1145/3428256. URL https://doi.org/10.1145/3428256.

[54] Alberto Lluch Lafuente, Michele Loreti, and Ugo Montanari. Asynchronous distributed execution of fixpoint-based computational fields. *Logical Methods in Computer Science*, 13, 2017. https://doi.org/10.23638/LMCS-13(1:13)2017.

[55] Geetika T. Lakshmanan, Ying Li, and Rob Strom. Placement strategies for Internet-scale data stream systems. *IEEE Internet Computing*, 12(6): 50–60, November 2008. ISSN 1089-7801. https://doi.org/10.1109/MIC.2008.129.

[56] Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '08, pages 323–332, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3437-4. https://doi.org/10.1109/SEFM.2008.11.

[57] Lightbend. Play Framework, 2007. URL http://playframework.com/.

[58] Lightbend. Akka Classic Actors, 2009. URL https://doc.akka.io/docs/akka/current/actors.html.

[59] Lightbend. Akka Typed Actors, 2015. URL https://doc.akka.io/docs/akka/current/typed.html.

[60] Lightbend. Case study: Capital one scales real-time auto loan decisioning with lightbend's akka platform, 2017. URL https://www.lightbend.com/case-studies/paypal-blows-past-

1-billion-transactions-per-day-using-just-8-vms-and-akka-scala-kafka-and-akka-streams.

[61] Lightbend. Case study: Capital one scales real-time auto loan decisioning with lightbend's akka platform, 2020. URL https://www.lightbend.com/case-studies/real-time-decision-making-for-auto-loans.

[62] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988. ISSN 0001-0782. https://doi.org/10.1145/42392.42399.

[63] Alberto Lluch Lafuente, Flemming Nielson, and Hanne Riis Nielson. *Discretionary Information Flow Control for Interaction-Oriented Specifications*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer-Verlag, Berlin, Heidelberg, 2015. ISBN 978-3-319-23164-8. https://doi.org/10.1007/978-3-319-23165-5_20.

[64] Martin Logan, Eric Merritt, and Richard Carlsson. *Erlang and OTP in Action*. Manning Publications, Shelter Island, NY, USA, 2010. ISBN 1-933988-78-9.

[65] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications*, pages 263–273, Piscataway, NJ, USA, 2004. IEEE Press. ISBN 0-7695-2090-1. https://doi.org/10.1109/PERCOM.2004.1276864.

[66] Dragos Manolescu, Brian Beckman, and Benjamin Livshits. Volta: Developing distributed applications by recompiling. *IEEE Software*, 25(5):53–59, September 2008. ISSN 0740-7459. https://doi.org/10.1109/MS.2008.131.

[67] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000. ISSN 0098-5589. https://doi.org/10.1109/32.825767.

[68] Heather Miller, Philipp Haller, and Martin Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In Richard E. Jones, editor, *Proceedings of the 28th European Conference on Object-Oriented Programming*, ECOOP '14, pages 308–333, Berlin, Heidelberg, 2014. Springer-Verlag. ISBN 978-3-662-44201-2. https://doi.org/10.1007/978-3-662-44202-9_13.

[69] Heather Miller, Philipp Haller, Normen Müller, and Jocelyn Boullier. Function passing: A model for typed, distributed functional programming. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, pages 82–97, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4076-2. https://doi.org/10.1145/2986012.2986014.

[70] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992. ISSN 0890-5401. https://doi.org/10.1016/0890-5401(92)90008-4.

[71] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Information and Computation*, 100(1):41–77, 1992. ISSN 0890-5401. https://doi.org/10.1016/0890-5401(92)90009-5.

[72] Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), October 2019. https://doi.org/10.1145/3360570.

[73] Fabrizio Montesi. Kickstarting choreographic programming. In Thomas Hildebrandt, António Ravara, Jan Martijn van der Werf, and Matthias Weidlich, editors, *Proceedings of the 13th International Workshop on Web Services and Formal Methods*, WS-FM '14, pages 3–10, Berlin, Heidelberg, 2014. Springer-Verlag. ISBN 978-3-319-33611-4. https://doi.org/10.1007/978-3-319-33612-1_1.

[74] Tom Murphy, VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In Gilles Barthe and Cédric Fournet, editors, *Proceedings of the 3rd Conference on Trustworthy Global Computing*, TGC '07, pages 108–123, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-78662-7. https://doi.org/10.1007/978-3-540-78663-4_9.

[75] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, pages 519–531, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-09-6. URL http://usenix.org/system/files/conference/nsdi14/nsdi14-paper-nelson.pdf.

[76] Matthias Neubauer and Peter Thiemann. From sequential programs to multi-tier applications by program transformation. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 221–232, New York, NY, USA, 2005. ACM. ISBN 978-1-58113-830-6. https://doi.org/10.1145/1040305.1040324.

[77] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. https://doi.org/10.1145/1376616.1376726.

[78] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17 (4):40–52, October 1992. ISSN 0163-5948. https://doi.org/10.1145/141874.141884.

[79] Per Persson and Ola Angelsmark. Calvin – merging Cloud and IoT. *Procedia Computer Science*, 52(The 6th International Conference on Ambient Systems, Networks and Technologies, the 5th International Conference on Sustainable Energy Information Technology):210–217, 2015. ISSN 1877-0509. https://doi.org/10.1016/j.procs.2015.05.059.

[80] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 49–60, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2570-9. `https://doi.org/10.1109/ICDE.2006.105`.

[81] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), April 2017. `https://doi.org/10.23638/LMCS-13(2:1)2017`.

[82] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. Eliom: A core ML language for tierless web programming. In Atsushi Igarashi, editor, *Proceedings of the 14th Asian Symposium on Programming Languages and Systems*, APLAS '16, pages 377–397, Berlin, Heidelberg, November 2016. Springer-Verlag. ISBN 978-3-319-47957-6. `https://doi.org/10.1007/978-3-319-47958-3_20`.

[83] David Rajchenbach-Teller and Franois-Régis Sinot. Opa: Language support for a sane, safe and secure web. In *Proceedings of the OWASP AppSec Research*, 2010. URL `http://owasp.org/www-pdf-archive/OWASP_AppSec_Research_2010_OPA_by_Rajchenbach-Teller.pdf`.

[84] Guido Salvaneschi, Joscha Drechsler, and Mira Mezini. Towards distributed reactive programming. In Rocco De Nicola and Christine Julien, editors, *Coordination Models and Languages*, pages 226–235, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38493-6. `https://doi.org/10.1007/978-3-642-38493-6_16`.

[85] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the web 2.0. In *Companion to the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA Companion '06, New York, NY, USA, 2006. ACM. URL `https://www.lri.fr/~conchon/TER/2012/3/dls06.pdf`.

[86] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. page 47, January 2011. URL `http://hal.inria.fr/inria-00555588`.

[87] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS '11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24549-7. `https://doi.org/10.1007/978-3-642-24550-3_29`.

[88] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, September 1984. ISSN 0098-5589. `https://doi.org/10.1109/TSE.1984.5010283`.

[89] Isaac Strack. *Getting Started with Meteor.js JavaScript Framework*. Packt Publishing, 1st edition, 2012. ISBN 978-0-321-70514-3.

[90] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997. ISSN 0164-0925. https://doi.org/10.1145/265943.265972.

[91] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *Journal of Logical and Algebraic Methods in Programming*, 109, 2019. ISSN 2352-2208. https://doi.org/10.1016/j.jlamp.2019.100486.

[92] W3C WS-CDL Working Group. Web services choreography description language version 1.0, 2005. URL http://www.w3.org/TR/ws-cdl-10/.

[93] Pascal Weisenburger and Guido Salvaneschi. Multitier modules. In Alastair F. Donaldson, editor, *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP '19)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:29, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-111-5. https://doi.org/10.4230/LIPIcs.ECOOP.2019.3.

[94] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with ScalaLoci. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):129:1–129:30, October 2018. ISSN 2475-1421. https://doi.org/10.1145/3276499.

[95] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. A survey of multitier programming. *ACM Computing Surveys*, 53(4), September 2020. ISSN 0360-0300. https://doi.org/10.1145/3397495.

[96] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 341–350, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. https://doi.org/10.1145/1242572.1242619.

[97] Fan Yang, Jinfeng Li, and James Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *Proc. VLDB Endow.*, 9 (5):420–431, January 2016. ISSN 2150-8097. https://doi.org/10.14778/2876473.2876477. URL https://doi.org/10.14778/2876473.2876477.

[98] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, Berkeley, CA, USA, 2012. USENIX Association. URL http://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.

[99] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20 (3):283–328, August 2002. ISSN 0734-2071. https://doi.org/10.1145/566340.566343.

[100] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In Robert Meersman and Zahir Tari, editors, *Proceedings of the 2006 Confederated International Conference "On the Move to Meaningful Internet Systems": CoopIS, DOA, GADA, and ODBASE*, OTM '06, pages 54–71, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 978-3-540-48287-1. https://doi.org/10.1007/11914853_5.

[101] Mulki Indana Zulfa, Rudy Hartanto, and Adhistya Erna Permanasari. Caching strategy for Web application – a systematic literature review. *International Journal of Web Information Systems*, 2020. https://doi.org/10.1108/IJWIS-06-2020-0032.