# Exploring the Role of Sequential Computation in Distributed Systems

## Motivating a Programming Paradigm Shift

Ivan Kuraj
CSAIL, MIT, USA
ivan.kuraj@csail.mit.edu

Daniel Jackson
CSAIL, MIT, USA
dnj@mit.edu

## Abstract

Despite many advances in programming models and frameworks, writing distributed applications remains hard. Even when the underlying logic is inherently sequential and simple, addressing distributed aspects results in complex cross-cutting code that undermines such simplicity.

This paper analyzes different programming models to motivate a new paradigm that leverages the sequential computation model, while gaining the expressiveness for distribution. The paper argues for an adoption of the paradigm shift by exhibiting a programming model that allows easier reasoning about the conceptual aspects of distributed systems' behavior. The newly proposed programming model provides a clean separation of concerns and retains the simplicity of sequential computation, using it as a basis onto which distributed aspects are added without corrupting the essential sequential structure, while offloading much of the complexity of implementing distributed concerns to the compiler. We demonstrate the feasibility of this model on a case study, identifying key improvements over existing approaches.

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming; D.3.2 [*Language Classifications*]: Very high-level languages; D.2.3 [*Coding Tools and Techniques*]: Structured programming; I.2.2 [*Automatic Programming*]: Program transformation

***Keywords*** reactive, distributed, programming model, declarative programming, software synthesis, separation of concerns



*Onward!'16*, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4076-2/16/11...
http://dx.doi.org/10.1145/2986012.2986015

## 1. Introduction

Distributed, reactive, and interactive applications represent the dominant category of modern software, which includes many types of web applications, data dissemination, data processing, and monitoring systems, ranging over domains from user communication to business analytics [5, 24, 46, 57]. Some reactive applications must exhibit distributed computation and update their state in response to multiple external stimuli, combining incomplete inputs from multiple sources [5]; others must promptly respond to asynchronous user inputs or process incoming protocol messages [24, 50]. Today's era of always available social networks, scalable online services and real-time user collaborations brings new demands for application programming in the field of distributed computing and imposes additional requirements on implementations of such distributed systems.

Both interactiveness and performance are essential features of many successful modern distributed systems that have to ensure timely responses even when they represent values computed at multiple different nodes. This in turn often requires scaling a distributed application by reconfiguring the location of its state and computation, together with the structure of communication across a distributed system, which might contain a large sets of machines [2]. Moreover, many systems must accommodate interactions amongst multiple users, raising demands for clean control over allowed interactions and data synchronization [42, 46]. In sum, modern distributed reactive applications often need to achieve a combination of seemingly conflicting properties even if their underlying functionality and business logic remain relatively simple.

On the other hand, programming such inherently complex software systems in the presence of the variety of non-functional requirements does not seem to become much easier. There are numerous reasons for this complexity, where some of the main ones are: 1. the distributed architecture across multiple nodes in the system that need to participate in complex interactions; 2. the abstraction gap between the problem domain level (business logic) and the implementation level (low-level communication primitives, message handling, scheduling, asynchronous callbacks); 3. concurrency issues within and across different nodes in the system, such as data

races, atomicity violations, deadlocks, etc.; 4. the difficulty in achieving efficiency, responsiveness, scalability, availability and shared data consistency, inherent to distributed computing.

When compared to sequential (specifically, locally-executed and single-threaded code), programming models for distributed systems become inherently more complex, due to multiple additional aspects of dealing with distributed nature of the running implementation [16, 39]. Advances in distributed programming models and web frameworks have addressed some of these challenges, and have made it easier to program certain classes of distributed applications, but they tend to sacrifice flexibility and introduce new complications. For example, some of the flexible and general models, such as the actor model, force the programmer to break computations into distinct message sends and handlers [16, 32, 48]. More specialized models, such as reactive programming, involve propagation algorithms that can become performance bottlenecks [24, 43]. Inherent in all these approaches to distributed programming is the recognition, codified in the so-called CAP theorem, that it is not possible to achieve consistency, availability and resiliency of a distributed system all at once [27]. Each approach therefore adopts some point in the design space, and optimizes some of these at the expense of others [13, 42, 51, 57]. Because these approaches tend to address some of the concerns with hardwired functionality, a developer who wants full control to select an arbitrary point in the design space must resort to a more primitive framework. We propose an approach in which, in contrast—by strictly *separating out the behavior and avoiding the commitment* to where the design will sit in the space dictated by distributed aspects of the system—developers can capture the logic of their systems early, while still being free to make design decisions concerned with distribution later, according to the desired requirements.

The sequential model of computation is well understood and often the most natural way to express system's behavior. From the perspective of distributed computing, however, the sequential model is not sufficiently expressive, because it fails to capture many of the inherent distributed aspects [16]. Many modern programming models and frameworks for developing distributed systems allow the behavior of individual nodes of the system to be expressed in the sequential model, while using specialized abstractions, and language constructs, to capture additional aspects such as communication and remote execution [4, 16, 19, 28, 29]. In this paper, we envision a more radical approach, in which the sequential model is used to express end-to-end behavior of the system, while the concerns of node interaction, data location and consistency constraints are handled by adding special pragmas to the code. The pragmas are then used by the compiler to generate implementations with respect to the specified distributed aspects of the system.

We propose a programming paradigm that *uses sequential computation model* for describing end-to-end behavior of a distributed system and *does not require disruptive changes* in the process of achieving its final implementation. The key insight lies in the notion of identifying *fragments* that capture behaviors for which the *semantics expressed with sequential computation model can be guaranteed* and the idea that such fragments can be *enriched with information about distributed aspects* of the system that is sufficient to produce a complete implementation. We define the semantics of our programming model by restricting the relationship between the behavior captured by the sequential model and possible executions of the resulting system. Effectively, this makes our system *a synthesizer that needs to find a correct distributed implementation* according to the specification expressed by the fragments augmented with distributed concerns. We show that such enriched fragments are *natural and expressive* to define potentially a large class of distributed systems, while allowing orthogonal aspects to be specified independently. More specifically, specifying not just orthogonal aspects of the system, but also additional behaviors, can be done *without changing the existing code.* In the end effect, the programming model achieves a *clear separation of concerns* during development and effectively splits it into two phases: the first one, in which developers focus on the conceptual logic of the end-to-end behavior of the system and write sequential code which is easy to reason about and test; and second one, in which such fragments are enriched with additional information, such as location of data and computation, that allows the system to produce an efficient complete implementation. As a side-effect of such a separation, the ability to specify behaviors of the system independently provides high-degree of *modularity* and *compositionality* as well as *flexibility* in customizing different aspects of the system.

We structure our presentation in the form of a proposal paper. We motivate the paradigm shift by considering a simple but representative example, rendering it in some of the existing representative programming models for programming distributed systems and analyzing the tradeoffs amongst the ease of programming, reasoning and expressiveness. We demonstrate the feasibility and expressiveness of our new programming model through the detailed case study, while realizing the desired requirements and comparing to other analyzed programming models and frameworks. Afterwards, we capture the essential components that are necessary for our programming model and describe its prototype implementation. Finally, we present our vision of what needs to be achieved to make such a programming model practically usable for wide range of distributed applications.

## 2. Modern Approaches

To motivate the programming paradigm shift in development of distributed systems and the need for a new programming model, we chose to analyze several commonly used approaches as representatives of different programming models, which are provided as separate programming languages or programming frameworks. This section introduces the approaches that we consider in our motivating case study.

### 2.1 Criteria and Goals of the Case Study

The main criteria for choosing the particular programming models in our analysis are: 1) the extent of the model to represent

fundamental concepts and abstractions that are both inherent to the model, as well as sufficiently distinct from other models (even though many programming models incorporate and mix multiple such concepts); 2) the ability to implement certain classes of distributed applications that are conceptually simple, but tend to exhibit complex implementations (and thus are usable and flexible to different extents in today's practice of developing distributed systems). The models considered in our analysis share the goal of making development of distributed applications easier, by hiding some of the inherent complexity from the developer. On the other hand, they focus to a different extent on the following aspects of distributed system implementations:

- expressiveness to capture conceptually related behaviors with the same programming abstraction (lack of such expressiveness may result in a single behavior implementation being dispersed over code that implements unrelated behaviors)
- the flexibility and convenience in adding new behaviors that are conceptually independent of other behaviors in the system (lack of such expressiveness may require disruptive changes for the implementation of new behaviors)
- specification, and its flexibility (for subsequent changes), of mapping the system's state and computation to particular nodes
- the support for specifying and ensuring different guarantees for data consistency

Our analysis aims at determining to which extent the analyzed models are suitable for development of the supported class of distributed applications, as well as the flexibility of implementing applications outside that class. Importantly, the analysis points out the extent to which these models become too strict and, by defaulting to inappropriate implementations, hurt efficiency, performance, and flexibility. To this end, our case study implements a fairly complex distributed application with varying requirements, with respect to different distributed aspects (mentioned in the previous section), and tries to analyze the potential issues and quantify their impact when handling those requirements.

## 2.2 Analyzed Programming Models

In our case study, we consider a subset of related programming models in more detail, as representatives of broader category of models that are relatively widely used in practice. We focus on models that have emerged in popularity in distributed software development practice during recent years, especially those well-suited for developing conceptually simple, but sufficiently user-specific, distributed systems and applications. Note that other models, not considered in the analysis, but otherwise relevant showcases of both novel programming concepts and various programming model design decisions, are presented in § 6.

We introduce programming models considered in our analysis, briefly commenting on the motivation for including them. (These models are described in more details in § 6.)

### 2.2.1 Reactive Programming

Reactive programming is motivated by difficult reasoning about complex control flows, concurrency and values changing over time, and is becoming attractive for implementing distributed (interactive) applications [13, 24, 52, 58]. Reactive model of computation is being adopted in modern programming frameworks since it automatically manages the propagation of value changes to their dependencies across distributed nodes [3, 13, 24, 44]. We consider *Functional Reactive Programming (FRP)*, which employs functional approach to abstract the time and compose side-effect free behaviors that represent changing values [7, 13, 22, 50, 58]. Since FRP approaches provide abstractions for behaviors and events that are relatively similar across different realizations, we consider the FRP model described in [50], without loss of generality.

### 2.2.2 Programming with Streams

Stream processing systems include a collection of computational units that compute in parallel and communicate via channels [53]. Due to raising demands for streaming functionality in distributed applications, we've been witnessing increasing support for programming with streams of values [9, 30, 48]. Although fundamentally similar to FRP (and other dataflow models, see § 6), stream programming focuses on managing production and flow of values. We consider Akka streams, a representative of the "Reactive Streams Initiative", that provides efficient and robust streams for programming distributed applications [5, 9].

### 2.2.3 Event-Driven Programming

Event-driven programming models represent applications with a set of events that occur in the system [23, 25, 33, 46]. By focusing on the application logic encapsulated with events, the model offloads the management of communication and concurrent execution to the underlying runtime. We analyze Sunny [46], a high-level event-driven model for programming web applications, which supports first-class events and a global data model that are managed by the runtime.

### 2.2.4 Key-Value Stores

Distributed key-value stores are a popular choice for implementing distributed applications since they provide a uniform interface to a remote storage while allowing good horizontal scaling [18, 59]. In spite of lacking support for defining computation and communication explicitly, many of the data-driven aspects such as replication and consistency can be offloaded to the underlying data store. Modern distributed applications often base their logic solely on the communication with the store through a protocol like REST. We analyze Redis [59], and it's Scala DSL interface scredis [8].

### 2.2.5 Actor Model

Actors are units of concurrent execution that can communicate only by exchanging messages [10, 32]. When distributed across nodes, they become very flexible and fit well for implementing a wide range of distributed systems [1, 12, 17, 34]. We consider

the Akka framework, which implements the actor model on the JVM and offers an advanced runtime with capabilities such as message delivery guarantees, location-transparency, and high-level communication patterns [1].

## 3. Motivating Case Study

In this section, we present our case study that illustrates the ideas behind our proposed programming model and compares it to other approaches for programming distributed systems. Our goal is to motivate the proposal for a new programming model through a detailed analysis of multiple aspects and requirements of implementing distributed applications. We demonstrate disadvantageous consequences of design trade-offs in the existing models and show desirable properties of handling the given requirements in the newly proposed model.

We consider a prototypical example of developing a distributed messaging application. Such an application is commonly used to point out complex aspects of the implementation in spite of representing a conceptually simple system [29, 46, 50]. Throughout the analysis, we progressively consider more (functional) requirements and implement more behaviors in the application. In the interest of presenting key insights and advantages of the proposed model, for each of the requirements, we describe implementation considerations for all analyzed models. In doing so, we begin by considering the sequential computation model and afterwards the aspects that make a transition to distributed implementations challenging (in other models). Finally, we arrive at a messaging application that not only offers the basic expected functionality (of exchanging messages between nodes in the system), but also addresses common efficiency and data consistency concerns.[1]

For each requirement, we present its corresponding implementation in the sequential model as well as in our newly proposed programming model. We present sequential implementations in Scala [47], which the prototype compiler for our programming model reuses in defining the resulting distributed application. As the resulting implementation, our system generates Scala code that uses Akka framework to implement the desired distributed application as a set of distributed communicating actors [1]. This implementation is equivalent to the implementation of the case study, given in the case of the actor model.

### 3.1 Two-Way Interaction for Sending Messages

We start by considering the basic functionality of sending a message (with receiving a response whether it was successful), in a commonly-used way, which allows clients to send messages that are stored on the server. For the sake of focusing on the illustration of main ideas, we assume the server is responsible for only one given chat room (while not going into the details on how can this be generalized, which tends to involve additional requirements). Moreover, to illustrate realistic domain-specific concerns, we do not consider actions like user login; while they can be implemented in our model, they are usually provided through existing libraries [4, 46].

#### 3.1.1 From Sequential to Distributed Implementations

This section presents the implementation of the requirements as simple programs in the sequential computation model and shows that some aspects of the distributed system behavior can be captured with a conceptually matching sequential code.

***Sending Messages as a Sequential Program*** We start by declaring variables that represent the state of the messaging server (assuming users are identified by strings):

```
1 var joinedUsers: Set[String] = Set()          // initially empty
2 var messages: List[String] = List()
3 var last: String = "Initial"                                Sequential
```

We maintain users that joined the room, the log of messages and the last message or event in the system. (For clarity, code snippets with same captions, here Sequential, belong to the same implementation given in a particular programming model.)

Afterwards, the function for sending a message can be straightforwardly implemented as:

```
4 def postMsg(user: String, msg: String) = {
5   if (joinedUsers contains user) {
6     messages +:= msg                  // append new message
7     last = "Last: " + msg
8     true
9   } else false }                                            Sequential
```

With the previously declared variables in scope, the function checks if the user belongs to the room and if so, adds the message to the log and returns true, otherwise just returns false.

Having this code, developers can easily reason about its behavior, run it and write tests for it. As an example, to check correctness of the function, developers can exercise both execution paths that return different outcomes:

```
10 joinedUsers = Set("User 1")                    // User 1 joined
11 assert(postMsg("User 1", "Hello") == true)
12 assert(last == "Last: Hello")
13 joinedUsers = Set("User 2")      // User 1 now cannot post
14 assert(postMsg("User 1", "Onward") == false)
15 assert(messages.size == 1)                          SequentialTesting
```

In spite of being a simple example, it demonstrates that sequential computation model allows easy reasoning about the system's behavior as well as testing its behaviors. The central idea of our programming model is to preserve this simplicity, even in cases where the resulting implementation runs distributed across multiple nodes.

***Implementing Distributed Messaging*** To make the behavior distributed across nodes, even for the functionality simple as sending messages, the implementation needs to take into account multiple aspects of the desired distributed system, in addition to describing the behavior itself. Although conceptually simple, the implementation for such a distributed application requires capturing the computation and communication with respect to the system's node configuration: in our case

[1] A common concern with messaging systems is consistency of observable messages, which might be violated if network instability is not anticipated in the system's implementation [41]

specifically, capturing where the variables used in the function are located and fetched from, and how is the behavior invoked.

In the running example, if we assume a simple client-server architecture, we might make the following decisions:

- assigning messages, last and joinedUsers to the server node
- deciding that postMsg executes fully on the server, to which clients send function arguments user and message
- initiating the communication on the client, handling it on the server, and sending the return value of postMsg back
- ensuring concurrent postMsg invocations keep mutable variables last and messages consistent
- ensuring concurrently (locally) issued postMsg requests result in an appropriate order of matching responses

As we will demonstrate, in most programming models that we consider, these aspects need to be introduced, either with special language higher-level abstractions and constructs (if supported) or manually, as *disruptive changes* to the existing code that implements the business logic, i.e. the *conceptual behavior* itself. Moreover, in addition to those changes, due to requiring special abstractions, these programming models often inherently require a *"paradigm shift"* that results in a significantly different and more complex programming model than the sequential computation model (which is sufficient for capturing the conceptual behavior of the application).

***Sending Messages in Existing Approaches*** The following sections show implementations of the messaging application in the other considered programming models (introduced in § 2), and analyzes their complexity and discrepancy with respect to the starting sequential code.

### 3.1.2 Functional Reactive Programming

In the case of reactive programming we need to declare our data as reactive and define dependencies between all used variables so that they achieve the computation required for the message sending. Effectively, we need to achieve two tasks: 1) specifying dependencies between variables used in postMsg and it's resulting value, and 2) defining reactive variables on the client and providing mechanisms for binding the dependencies from the server to the client (and vice versa for the response).

We declare *reactive behaviors* on the client that capture the values for the message and user, and an *event* that samples an input element (in this case a button) to initiate the behavior.[2] The resulting event captures all the necessary information that needs to be forwarded to the server.

```
1 val usr: Beh[String] = text("user").vals // input as behavior
2 val msg: Beh[String] = text("message").vals
3 val send: Event[MEvent] =          // button clicks as event
4   button("send").toStream(Click)
5 val submit: Event[(String, String)] = {
6   val entry = usr.combine(msg)
7   entry.sampledBy(send) }
```
FRP

[2] Note that for brevity, we omit some code and syntax details.

We construct behaviors from textual input elements. They represent values for the user and message (of type String). We construct a new event that samples from the send event, which is in turn bound to a button, so that each time the button is pressed, the code samples and combines current values of user and msg to create a pairs of current values.

Behaviors constructed on a remote node can be forwarded to other nodes by appropriately binding events. We bind the event submit from the client to the server (which is in this case implicit [50]), by using a specialized method toServer:

```
8 val post: EventS[(String, String)] = submit.toServer
```
FRP

On the server, we encapsulate joinedUsers as a reactive behavior, similarly as we did on the client. Afterwards, the functionality analogous to the sequential postMsg (presented before) can be defined by mapping the bound stream post with an appropriate function. This function, given the values of the user and the message as arguments, computes the necessary information:

```
9  val joinedB: Beh[Set[String]] = ... // joined users as behavior
10 def proc(p: (String, String), joined: Set[String]): Result = {
11   val (usr, msg) = p              // extract values from a pair
12   if (joined contains usr) Success(msg, usr)
13   else Failure(usr) }
14 val results: Event[Result] = post.combine(joinedB) {
15   proc }
```
FRP

where we assume Success and Failure are previously defined case classes as subclasses of Result, that carry given values for the sake of convenience. The resulting values of this behavior can then be bound on the client to be processed as the response.

Afterwards, also on the server, we need to handle values from the created event to update the behaviors that represent the accumulated messages and the last posted message:

```
16 val successfulE = results.filter(_.isSucc)
17 var messages: Beh[List[String]] = successfulE.fold(List()) {
18   case (acc, Success(msg, _)) ⇒
19     msg :: acc }                        // append the message
20 var last: Beh[String] = successfulE.map(_.msg).
21  .fold("Initial"){ "Last: " + _._2 }    // retain last
```
FRP

Behaviors messages and last are determined by folding the list of successes (due to filtering of results with isSucc that returns true only for Success). This effectively means that the current values of behaviors is recomputed (by the fold function) whenever a new request appears: messages are appended, while the last message just gets overwritten.

The resulting event results needs to be further bound from the server to the client as well, so that the resulting values that designate whether the post was successful are transferred back to the client. FRP approaches need to expose a construct similar to toClient, which takes a function that is executed when the event fires, to determine to which clients the event should be transferred [50]:

```
22 def correctClient(r: Result, c: Client): Option[Boolean] =
23   if (r.usr == c.usr)       // check if user belongs to client c
24     Some(r) else None
25 val clientRes = results.toClient(correctClient).
26   map(_.isSucc).hold // behavior as the success flag
```
FRP

The resulting event is bound to a behavior on the right client, as determined by evaluating correctClient for all known clients (which returns Some(r) if value r needs to be forwarded to that client c). Here, by calling hold we define a behavior that emits the last value of the event results, i.e. a Boolean value that designates whether the message was successfully sent.

This implementation demonstrates that flow of values is realized with behaviors and events, dictated by the way of their binding. Effectively, the constructs that achieve such bindings across different nodes (here, toServer and toClient) dictate the flexibility of handling additional distributed aspects within the system, such as different communication patterns or consistency. Purity of behaviors in FRP often implies the need for (un)compressing multiple values (like in the case of constructing Success) and forwarding them along the appropriate flow, whenever they might be needed at another point within the system's behavior. This example shows that handling additional distributed aspects, like additional communication (as in our example, sending the response back to the client), would necessarily need to disrupt the existing code had the structure of the existing code failed to expose appropriate values. In addition, FRP approaches do not support mutability and rely on special combinators for combining and folding behaviors; declaring an event that consumes a behavior but also needs to update it (recursively, as needed in our example, to model a mutable variable) might create cycles in the flow that might become significantly more complex to reason about.

### 3.1.3 Flow Graphs with Akka Streams

With Akka streams, we declare and combine streams of values to achieve a similar end-result flow structure as shown in the previous section with FRP, albeit with different abstractions and capabilities. (Effectively, in the running example, Akka streams generalize the behaviors and events used in FRP.) The central notion that models any activity in Akka streams is a *flow graph*, which is comprised of *sources* that emit streams of values, *sinks* that consume streams of values, and *flows* as processing components between them.

We start by declaring *sources* on the client for streaming current values for the the user and the message. (We omit some details, including the definition of getMsg, which returns the message to send; the function blocks until an UI interaction, effectively capturing same behavior as event sampling in FRP.)

```
1 var user: String = ...; def getMsg(): String = ...
2 val usr = Source(() ⇒ user)      // current values as stream
3 val msg = Source(() ⇒ getMsg())
4 val pair = usr.zip(msg)          // combine streams
```
Streams

We combine streams of values, similarly as we did previously; zip produces a source of pairs of String values.

Preparing requests from the client can be performed by binding the source on the client to the appropriate *flow* on the server. Flows effectively represent components that process values that are pushed to them and output resulting values. We map the previously constructed source pair to produce Request values (we omit definition of toRequest) and bind it to a flow that represents an HTTP connection for the transfer of values to the server. The flow http takes the stream of requests on its input and pushes them out on the network as its output:

```
5 implicit val materializer = ActorMaterializer()
6 val http: Flow[Request, Response, Future[Success]] =
7   Http().connect(params)          // a remotely bound flow
8 pair.map(toRequest).to(http).run
```
Streams

For brevity, we omit some (implementation specific) details, such as materialization (required for scheduling) and details of the construction of the given HTTP flow [9]. By calling run we effectively activate the resulting flow graph.

A *sink* is an element that consumes a stream of values that is pushed to it (through streams from the attached flow or source, as we will show later). For the server code, we define a source for values of joined users (which simply returns the variable joinedUsers), as well as sinks that consume messages:

```
9  val joined = Source(() ⇒ joinedUsers)
10 val msgSink = Sink.fold(List())(_ :: _)  // stores messages
11 val lastSink = Sink.last[String]
```
Streams

The msgSink sink appends newly pushed messages to the message log, while lastSink retains the last message.

Processing on the server can be concisely expressed by constructing a composite *flow graph* that is composed out of simpler flow graph components:

```
12 def procMsg(p: (String, String), join: Set[String]): Result =
13   ...       // process message, similarly as defined previously
14 val serverFlow = FlowGraph() { implicit builder ⇒
15   val zip = builder.add(Zip[Entry, Set[String]])
16   clientSrc ~> zip.in0          // combine connected inputs
17   joinedSrc ~> zip.in1
18   val bcast = builder.add(Broadcast[Result](3))
19   zip ~> process ~> bcast                  // fan out results
20   bcast.filter(_.isSucc).map(_.msg) ~> msgSink
21   bcast.filter(_.isSucc).map("Last: " + _.msg) ~> lastSink
22   bcast.map(_.isSucc) ~> clientRes }
```
Streams

This flow graph executes on the server and handles newly received messages. (We omit code that declares source clientSource and sink clientRes that represent streams of values pushed from and back to the client, respectively, over an HTTP channel.) The flow graph updates the state on the server by extracting appropriate values (with map) and binding the resulting streams to the appropriate sinks. The stream of Boolean values, bcast.map(_.isSucc) represents whether the message was successfully sent and is forwarded to clientRes as the response. For brevity, we omit details of processing responses on the client—the client uses similar mechanisms to unwrap the source from the http flow and connect it to the appropriate sink that handles the received Boolean values.

Although similar in spirit to FRP, Akka streams is not restricted to purely functional style, allows constructing flexible and robust flow graphs, and focuses on controlling flows of values distributed over the network (including the control of back-pressure [5]). However, it suffers from similar issues as FRP, since the flow graph dictates both the communicated values and communication routes. This effectively means that even small changes in the communication within the distributed

system might require disruptive changes to existing flow graphs and their bindings. In addition, the approach requires dealing with additional concerns of running and distributing flows (like flow materialization) and might require constructs outside the model to handle lower-level details (e.g. futures to encapsulate potential failures in the network) [9].

### 3.1.4 Event-Driven Programming with Sunny

Sunny allows developers to declaratively express the structure of the distributed application and its data and define event handling with cleanly abstracted snippets of code. We define events, but omit the code that invokes them in the application, since the Sunny programming model couples event triggering with input elements found on web pages [46].

First, we write a declarative specification of the (global) *data model*. The record User describes data on the client node:

```
1 record User do refs name: String end
```
Sunny

where **refs** denotes simple referencing aggregation (without any constraints) [46].

We then provide a specification of the *network model*, in which we specify the client machine that contains a reference to the user record, and the server that contains messages and joined users:

```
2 machine Client do refs usr: User end
4 machine Server do
5   refs joined: (set User)
6   owns last: String, messages: (seq String)
7 end
```
Sunny

The server node maintains the users that joined the room, the message log and the last message. (Keyword **owns** captures referencing an external field where referential integrity is automatically maintained within the data model.)

To achieve the functionality of sending messages we define an appropriate *event model*:

```
8  event SendMsg do
9    from client: Client; to serv: Server
10   params msg: String
11   fun newMsg { serv.last = "Last: " + msg
12     serv.messages << msg }          // append the message
13   requires { room.members.include?(client.user) }
14   ensures { newMsg() }
15 end
```
Sunny

The event has an appropriate precondition (given in the **requires** clause) that prevents executing the event, i.e. its postcondition, in case of a client state that should not allow it. Here, we require that the user has joined the room. A specification of the effects of an event (**ensures** clause) is concerned only with updating relevant data records, namely adding the message and setting last.

Note that this implementation ignores responding to clients. Interestingly, to specify returning the response, as we formulated it previously, this specific model (adopted in Sunny) would need to be changed. Firstly, since the runtime does not process events if their preconditions are not satisfied, we would need to remove precondition in SendMsg and modify the postcondition code to check the condition explicitly (to prevent discarding events of failed message posting). Secondly, since the only means of communication in the event-driven model is through events, the model forces us to invoke another event to send the response to the client. In that case, the postcondition would look similar to:

```
14 ensures { if (room.members.include?(client.user))
15   then { newMsg(); trigger Response(client, true) }
16   else trigger Response(client, false) }
```
Sunny-Extended

Note that this is not possible to achieve in the existing Sunny model, since Sunny does not support such a general way of triggering events. Events in Sunny have to be explicitly bound to UI components on client machines that trigger events as a result of a user action.[3]

This demonstrates that, by relying on explicitly defined events, in the event-driven programming model we might need to declare separate events to achieve additional behavior even if it conceptually represents the same or related functionality within the application. In our case, we had to declare a new event that represents the response sent back to the client. Thus, such inherent splitting of behaviors in event-driven programming might hinder the ability to define and reason about complex interactions and communication patterns within the distributed application.

### 3.1.5 Programming against the Redis Store

The support for a uniform interface for storing data inside a key-value store, readily accessible from different nodes in the system, makes key-values stores a popular choice when implementing conceptually simple (and data-centered) distributed applications, as is the case with messaging applications. While key-value stores such as Redis provide flexible access to the stored data, even though communication between nodes can be handled through the store itself, it is usually handled independently. We assume the message posting functionality is located on the server, which is invoked by clients by supplying the values for the user and the message. (To focus on the model itself, we omit the actual communication between clients and the server.)

We implement message sending by using the exported API calls to access the state of the application that is stored in the key-value store. As a first step, we initialize the key-value store, by specifying parameters that determine how can the key-value store be accessed (we omit the details of defining the configuration object conf [8]):

```
1 val redis = Redis(conf)      // initialize the KV store
```
Redis

Importantly, to handle potential failures in API calls issued to the Redis store (e.g. due network unavailability), scredis encapsulates *API calls as futures* in Scala—a programming abstraction that is often used to handle asynchronous calls and operations that might fail [29, 47]. We implement the message sending functionality with Scala for-comprehensions that ensure that only when all accesses to the key-value store (which are asynchronously invoked) succeed, the computation in the body (within **yield**) can take place [8, 47]:

[3] Note that Sunny also includes reactive support (not considered here), which tracks dependencies on the data model and allows implementing such behavior as an reactive update of a view on the client [46].

```
2 def postMsg(usr: String, msg: String): Future[Boolean] = {
3   for (isIn ← redis.isMember("joined", usr);
4     if (isIn);       // true if user in the set with key "joined"
5     _ ← redis.lPush("msgs", msg);     // append mesasge
6     _ ← redis.set("last", "Last: " + msg))
7     yield true }}
```
Redis

where placeholders _ designate unused return values from futures (which e.g. might signal how many rows were updated). The result of the for-comprehension is another future, composed of futures that represent individual calls to the store. Scredis supports higher-level API calls, like isMember and lPush, that communicate with the key-value store with POST/GET requests in the background, but transform the request and data according to various supported Scala types, like Set and List.

The resulting future can be invoked to asynchronously issue calls to the key-value store in the order specified in the for-comprehension and return Scala Success only if all calls succeeds. To get and process the final result of the composition of our calls to the key-value store, we handle the resulting future (on the server) with:

```
8 def respond(succeeded: Boolean) = ... // send the response
9 postMsg(user,msg) onComplete {
10   case Success(res) ⇒ respond(res.isSucc)
11   case Failure(ex) ⇒ respond(false) }     // failure
```
Redis

Where onComplete specifies code to execute for both cases: if the future succeeds or fails. In case of successful invocation of all API calls we send the response back to the client, otherwise we handle the failure. We omit sending the response back to the client and comment on some of the possible implementations subsequently.

It is important to note that while key-value stores are becoming more flexible and robust to allow developing distributed applications around them, they clearly cannot handle all the potential aspects of a general purpose application (that requires deriving more complicated views of the stored data model). In most cases, key-value stores handle only storage (with distribution and scalability across nodes), while the computation and the communication between nodes is defined independently, for each node. However, modern key-values stores (including Redis) support realizing communication through the key-value store itself, through mechanisms such as publish-subscribe (see § 6.2) that signal changes in the store, or simply with efficient polling of the store [6, 8]. Interestingly, to some extent, they enable reusing the sequential implementations by changing local accesses to API calls. (In our example, in spite of using futures, the code structure is similar to the sequential code from § 3.1.1.) This model does not inherently provide any particular higher-level abstractions to address some of the concerns of distributed applications; as our running example shows, it is up to a particular framework to decide on the abstractions used, for example for dealing with failures. Moreover, in terms of consistency, our implementation is problematic since it issues requests to the store in the particular order, while some subset of individual requests might fail and leave the store in an inconsistent state. (To this end, KV stores, including Redis, support specialized atomic operations and transactions, which, in turn, are more complicated to reason about and tend to affect performance [6, 59].)

### 3.1.6 The Akka Actor Model

The actor model, being based on *actors as a first-class concept*—effectively, individual live components that encapsulate their state and capture behavior only as a consequence of *exchanging messages* with other actors—represents an excellent fit for implementing distributed applications such as messaging-like services [29].

A straightforward implementation of sending messages with actors splits the behavior into two actors (two classes of actors), one for both the server and the client. We start by defining messages that contain the needed information for the communication between the server and clients within separate case classes,

```
1 case class Msg(usr: String, msg: String)
2 case class Response(successful: Boolean)
```
Actor

in their respective fields.

The client is represented with the following actor class (as a subclass of the standard Akka Actor class) [1]:

```
6 class ClientActor(server: ActorRef) extends Actor {
7   var user: String = ...
8   def notify(msg: String) = ...          // handle notification
9   def send(msg: String) = { // asynchronous message send
10    server ! Msg(user, msg) }
23 }          // we will add some code above later
```
Actor

Given a reference to the server actor, invoking send will send the given message (asynchronously, using !) with current values of user and the given message. (We omit attaining the reference to the server actor—Akka offers multiple different ways of sharing references to actors within the system [1].) Note that this snippet of code belongs to a larger implementation, thus multiple lines are omitted and will be shown later.[4]

The server declares the message handler by overriding the receive of the Actor super class:

```
24 class ServerActor extends Actor {
25   var messages = ...; var last = ...
26   var joinedUsers = ...          // declare variables as before
27   def postMsg(usr: String, msg: String) = ...
28                          // declare the same function as before
31   override def receive = {       // define message handling
32    case Msg(usr, msg) ⇒
36      val flag = postMsg(usr, msg)     // posts the message
37      sender ! Response(flag) }}        // respond
```
Actor

which, upon receiving an instance of Msg as a message, executes postMsg (which is the same as in § 3.1.1), after which it sends its result as a response to the client. Note that sender is implicit in receive and designates the actor that sent the message that is being handled; thus, the right target for sending the Response.

In addition, since we need to handle these responses on the client, we add a method:

```
13 override def receive = {              // handle the response
14   case Response(flag) ⇒ notify(flag) }
```
Actor

[4] We hint the code structure with line numbers to pay a tribute to an old procedural language BASIC, which was considered high-level long time ago.

to define the receive handler in the Client actor that calls the appropriate function to process the results (by invoking notify with the result).

With actors in Akka we can straightforwardly make this implementation distributed by providing a configuration that defines mapping of actor names to particular nodes in the system and instantiating clients with the appropriate server reference. Although defining computations with separate actors and explicit communication makes the model flexible, an inherent drawback is that structure of the code in terms of actors needs to match the structure of the distributed system. A decision to change the data or computation might require disruptive changes, not just to actor classes and handlers, but also messages.

#### 3.1.7 Can We Retain Sequential Computation Model?

The main question we pose in this work is whether we can define a programming model which heavily *relies on the sequential computation model* for specifying the behavior, but *does not break it*, nor significantly complicate it, in the process of capturing aspects of the implementation inherent to the desired distributed application. If so, the goal of such a programming model would be to allow "lifting" conceptually simple programs defined with sequential model of computation (which can be independently executed and tested), to implementations of distributed applications. Since, in that case, additional information about the distributed aspects of the system is inherently required to fully characterize it, there needs to be a way to provide such information. Such a programming model should allow flexible and non-intrusive customization of parameters that dictate distributed aspects, thus effectively choosing a point in the trade-off space inherent to distributed systems.

#### 3.1.8 Concepts of Programming with Scenarios

The central idea of our newly proposed programming model revolves around a notion of a *scenario*. Scenarios are used to capture parts of the overall behavior of a distributed system from *two different perspectives*: a perspective of individual fragments of the overall behavior that can be expressed in the sequential computation model and a perspective of the complete behavior within the desired distributed system.

Scenarios are the central abstraction that defines programs in the scenario-based programming model. From one perspective, scenarios model the behavior of a distributed system when *viewed from the perspective of certain parties* involved in the system, under the condition that the semantics of such an interaction described by the scenario is rightfully respected in the resulting system. Importantly, the specified behavior has the property that its semantics has to be *captured with a sequential model of computation*. Therefore, by defining scenarios, developers are effectively capturing *fragments of interactions* of the resulting distributed system "projected out" as a sequential computation.

We divide scenarios into two types: *basic scenarios* and *distributed scenarios*. Both of them model *(conceptually) the same* behavior, but from different perspectives (as mentioned before), where distributed scenarios effectively just extend basic scenarios to define the role of their behavior within the distributed system. Basic scenarios are represented as simple programs (in our prototype, non-pure functions) that define the behavior of a distributed system on an abstract level, where sequential computation faithfully captures the intended behavior of the system; effectively, the result of projecting out the executions of the distributed system as a sequential computation, see § 4. All the variables used in the function, i.e. its scope, also belong to the basic scenario. On the other hand, distributed scenarios are basic scenarios extended with three components: *function mapping*, *data mapping* and a *trigger*. Trigger represents an occurrence of an event that causes the behavior captured by the given scenario. It defines how the scenario behavior starts manifesting itself in the distributed system. The trigger might represent a stimulus from the environment (e.g. a user making an input action on one of the nodes) or the condition being made true (e.g. triggered at the specified time). The function mapping specifies how are the parameters of the scenario function (that represents the scenario) populated, at the point of triggering. Once the developers define nodes of the distributed system, the given data mapping specifies the location of all variables used in the scenario function.

The key insight behind our programming model is that behaviors captured by basic scenarios admit the semantics of the sequential computation model. (The relation between the behavior of a basic scenario and the resulting distributed implementation is defined in § 4.) Importantly, the information additionally captured by distributed scenarios is sufficient for characterizing full implementations of distributed applications; specifically, we show how scenarios can implement the messaging application from the running example.

#### 3.1.9 Implementing Message Sending with Scenarios

To use our programming model, we start by writing simple functions that represent basic scenarios. In fact, the definition of the sequential computation of postMsg (presented in § 3.1.1) already represents the needed basic scenario. (Our prototype compiler takes such functions and infers the intended behavior, i.e. computation and state, that can be later distributed across nodes.)

Before beginning developing distributed scenarios, we specify the configuration of the system. We declare configuration of nodes that participate in the desired distributed system:

```
1 class Server extends SingletonNode
2 class Client extends SpawningNode {
3   def notify(msg: String) = ...          // same as before
4   var user: String }
```
Scenario

This declaration specifies that there are two types of nodes in the system, one for the server and one for client, where SingletonNode designates a unique Server node (known to other nodes), while the system might have multiple, and spawn new, Client nodes. (This defines the client-server architecture, where the system assumes all nodes know about the singleton node; which is currently supported in our prototype implementation.) Each Client node has variable user assigned to it. Note that instances of these classes represent "logical" nodes, in the sense they might be arbitrarily assigned to physical nodes. (We cur-

rently do not do anything specific in terms of such deployment, and rely on the Akka framework for those tasks; see § 4.)

We define the needed distributed behavior by enriching a basic scenario that specifies the sequential behavior of sending, with an appropriate specification for distributed aspects. The distributed scenario can be defined as:

```
5  scenario Send {
6    @location(Server) {     // same variables as in Sequential
7      var messages: List[String] = ... }   // assigned to server
8    @trigger(Client.action)
9    @input(Client.user,Client.msg)          // bind parameters
10   @output(res⇒Client.notify(res))   // handle return value
11   def behavior(usr: String, msg: String) =
12     postMsg(usr, msg) }    // as defined before
```
Scenario

The scenario defines its behavior with sequential computation simply by calling the function postMsg (as we have defined before, in the case of Sequential, see § 3.1.1). In addition, it also captures other pieces of information that specify which nodes are involved in the behavior, the location of used data variables, and how the scenario is triggered. The *@location* annotation specifies where is the annotated variable located, i.e. on which nodes does it live; last, joined and messages are thus located on the server node. (Note that, in this case, the same effect can be achieved either by declaring variables inside the scenario object and associating them with a location, or by explicitly declaring them in nodes and then binding them as function arguments.) In addition, the behavior function is annotated with *@input* and *@output* that specify how are the values for the function arguments populated, and where and how is the return value processed, respectively. Here, parameter usr and msg originate from the values Client.user and Client.msg, respectively, while the resulting value is used to invoke notify, all located on the Client node. The trigger designates that the scenario is explicitly invoked with Client.action on the client (the system generates this function and its body that starts the scenario).

Our system takes the **scenario** Send as input, performs a simple code analysis (to infer all the necessary information, such as where to map certain portions of the computation) and generates the code that matches the one given for the actor implementation in § 3.1.6, i.e. in the case of Actor. Specifically, the system generates given messages, two actor classes and their fields and methods. Currently, the system executes each defined basic scenario only on one node, while passing all the dependencies (in this case only parameters usr and msg) in a message from other nodes. (More details on the restrictions of the model and the notion of a correct generated implementation are given in § 4.) By specifying the basic scenario as a sequential fragment of system's behavior (essentially the function postMsg) and afterwards, associating the information about the distributed aspects of the system (like data location), we achieved the conceptually simple, but fully distributed, implementation of the messaging application. Note that we first defined the behavior for posting messages with a fully executable (and testable) sequential code and then effectively lifted that code into a distributed application.

### 3.2 Implementing Multi-Node Message Notifications

As the next step in making the case study more realistic, we consider a functionality that requires multi-node interaction. We consider sending notifications to all users in the system whose name is mentioned in the message, in case their status does not say "do not disturb" (as modern messaging platforms, like IRC, usually do). Since notifications are sent after a new message is posted, the functionality potentially requires changing the existing implementation. The need for disrupting existing code when introducing such functionality is one of the main motivation points behind our programming model.

From this section onward, we do not go into details of the implementations in all the considered programming models; rather, we focus on the most relevant comparisons and elaborate the solutions provided by the scenario-based programming model.

#### 3.2.1 Notifications in the Sequential Model

As done previously, we first write a simple sequential program that captures the needed functionality:

```
16 class Client {                    // encapsulate client behavior
17   def notify(msg: String) = ...   // display given notification
18   var usr: String = ...; var status = "Available" }
```
Sequential

However, since the behavior inherently involves interaction between multiple nodes, we arrive at an obstacle in the expressiveness of this model. One alternative is to capture the behavior from the perspective of only one client that gets the notification:

```
19 if (status == "dnd") notify("* New msg")
```
Sequential

(For brevity, we omit checking if the usr variable occurs in the message.) Although quite disconnected from the actual intended functionality (and seemingly useless overall) this implementation indeed captures the behavior when viewed from the given perspective. However, it is not clear what can we do with such a sequential implementation: how to reuse it or transfer it to fit one of the presented models for implementing a distributed application.

On the other hand, if we try to capture the behavior in the sequential computation model by defining multiple clients (as some programming models do [15, 16, 19]):

```
20 val clients: List[Client] = ... // get clients as remote objects
21 for (c ← clients)                // loop through in some order
22   if (c.status == "dnd") c.notify("* New msg")
```
Sequential

it becomes unclear whether such program, together with its semantics, can translate to a distributed application without prohibitive restrictions. Specifically, if we respect the semantics of the sequential model, the flow of control from the server to the clients (and back) *must happen in the order* dictated by the loop traversal. In many distributed applications, including the one in our case study, this is not acceptable since such a restriction would incur high performance penalties; the optimal implementation would send notifications without the need to respect any particular order.

Due to the strictness of the sequential computation, i.e. the mismatch in the semantics, implementing distributed systems by modelling clients explicitly and respecting the

sequential semantics may be practically unacceptable. One potential solution is introducing constructs for relaxing the semantics of sequential computation (e.g. to allow out-of-order execution), which seems sufficient only in a limited set of cases [16, 35]. Unlike other considered programming models, which introduce additional abstractions, our model aims at fully reusing sequential implementations without complicating the computation model by only considering behaviors from *the perspective amenable to the sequential computation model*.

### 3.2.2 Notifications as Events in Sunny

When implementing notifications in Sunny, we end up having a similar issue as with returning a response to the client (described in § 3.1.4), since we need to define a new event that needs to be triggered after a message is received. In an extended event-driven Sunny model, this could be achieved by adding an additional status variable to the client node,

```
2 machine Client do ...              // other variables unchanged
3   refs status: String end                                  Sunny-Extended
```

declaring a new event to be triggered for notifications,

```
16 event Notify do
17   from serv: Server; to c: Client; params msg: String
18   requires { c.status != "dnd" && in(c.usr, msg) }
19   ensures { ... }          // perform notification on the client
20 end                                                      Sunny-Extended
```

and finally, triggering the event Notify whenever a new message is received. (We omit the code that triggers Notify, as it requires similar changes as for implementing responses to message sending; see § 3.1.4.) The event Notify simply checks the status and whether the received message mentions the user in the precondition, and performs notification if needed.

This programming model suffers from the need to modify existing code when new behaviors are added. Note that not only that we needed to change the definitions of machines, but the existing SendMsg event as well (to trigger Notify). Although flexible for defining additional behaviors and interactions between multiple nodes (event-driven programming excels at defining many-to-many communication patterns [25]), the issue remains that conceptually connected behaviors are spread across different abstractions that might need to be modified when introducing new behaviors.

### 3.2.3 Reactive Notifications

To implement the notification functionality with functional reactive programming, we need to bind the existing reactive values—namely, the one resulting from new messages—to behaviors located on all clients.

We simply bind the event that streams new messages, filter the ones that succeeded, and send them to all clients:

```
27 val notification = successfulE.map(_.msg).toAllClients
```

The code uses previously declared event successfulE (in FRP code, line 18; see § 3.1.2) and binds it to be sent to all clients. Afterwards, on the client, we use the event to invoke the notification for all messages that mention the given user:

```
28 val cn = ...                  // bind the notification on the client
29 val status: Beh[String] = text("available")
30 cn.zip(status).zip(user).filter({ case ((msg, st), user) ⇒
31   st != "dnd" && msg contains user }).map(_ ⇒
32     notify("*" + msg))                                        FRP
```

On the client, we combine the behavior of the current status and value usr to invoke notify if needed. (We omit the code that binds the behavior cn, which represents received notifications.)

Note that while we reused previously declared event successfulE, adding notifications would require modifying the existing implementation if the binding of behaviors was not structured such that the appropriate event (which passes the right information needed for notifications), was already exposed. Moreover, note that the natural solution of binding events of all successful message postings to all clients, and then filtering to invoke functionality if needed, might incur significant penalties. On the other hand, a better solution could be to check to which clients should the event be bound on the server (as discussed in § 3.2.6); a solution that tends to be less straightforward in FRP and forces developers to explicitly think about the boundaries between nodes and the overall structure of the system.

### 3.2.4 Notifications with Actors

The flexibility of the actor model allows us to implement the additional notification functionality with a few changes to the existing code. Assuming the server maintains a list of all clients in the Server class (as actor references, that might be located on remote nodes [1]),

```
29 val clients: List[ActorRef] = ...                          Actor
```

we create a new message class for signalling notifications,

```
3 case class NotifyMsg(msg: String)                           Actor
```

and add code for sending notification messages after a new message has been received,

```
32 case Msg(usr, msg) ⇒ ...          // code the same as before
34   for (client ← clients) client ! NotifyMsg(msg)           Actor
```

which uses a simple for loop to send messages to all clients. Note that although the sending is executed in the given particular order, it is executed asynchronously, thus no blocking is incurred (and the code does not match the behavior defined by the sequential semantics in § 3.2.1). Afterwards, we add additional code to the Client class to handle notifications at the client:

```
11 var status = "available"
13 def receive = { ...          // handle additional message class
15   case NotifyMsg(msg) ⇒ if (status != "dnd" &&
16     msg.contains(user)) notify("*" + msg) }                  Actor
```

As described before, the code simply examines the status and the current message and invokes notify if needed.

Although all the required changes are conceptually simple overall, the implementation of this additional functionality requires multiple, relatively smaller, separate changes that are dispersed across the existing actor code. The additional functionality requires changing at least the code of all the participating actors. Note that, had we decided to check whether users are mentioned in the message on the server (for performance rea-

sons), we would not only need to move the part of the condition from the client to the server actor, but also write code to manage and appropriately update the variable used in that condition.

### 3.2.5 Notifications in Other Programming Models

We omit detailed analysis of implementations of notifications in the case of stream programming and key-value stores. In the case of Akka streams, the implementation would be similar in structure to the one given for FRP, since the structure of bindings of reactive behaviors in FRP is similar to the one for stream binding we showed previously. Implementing notifications with a key-value store would effectively use the same approach of accessing appropriate parts of stored state as in the case of message sending; however the concerns of disseminating notifications would need to be handled separately as well, as previously discussed.

### 3.2.6 Implementing Notifications with a Scenario

One of the key points behind introducing the new scenario-based programming model is to allow implementing new functionality, like notifications in this case, while focusing solely on the conceptual logic of the functionality without the need to consult, or introduce changes to, the existing code.

We define a new scenario that implements the notifications functionality without changing the existing code:

```
13 scenario Notify {
14   @location(Client) def notify(msg: String) = ... // as before
15   @location(Server) var last: String          // put into scope
16   @input(Client.status,Client.user)             // bind inputs
17   @trigger def cond(status: String, name: String) =
18     status != "dnd" && last.contains(name)
19   @output(Client.notify) def action = "*" + last }
```

Scenario

The scenario is declared to be triggered with a *trigger condition* that starts a scenario when the given predicate (function cond, of type Boolean) becomes true. The trigger might happen as a consequence of other behaviors in the system, i.e. execution of other scenarios. Our prototype compiler analyzes all the code in the current implementation (more specifically, the existing scenario defined before, in § 3.1.9) to determine all possible places where the condition can trigger. Currently, the compiler considers all statements after which the condition might be true in a pessimistic fashion (possibly emitting condition checks at places where they might not be necessary). Next, it determines that message should be sent from the server to all clients and splices the code that represents the behavior, defined by action, into the implementation of the server. In this case, action simply returns last (located on the server); this value will be sent and used as an argument to notify on the client. Our compiler, when given the two input scenarios Send and Notify, produces the same code as the Actor implementation given in the previous section (§ 3.2.4), including the new class for notification messages (Actor, line 3), the modified handler that uses a for loop to send notifications (lines 32-37), and the message handler on the client that handles notifications (lines 15-16).

This example demonstrates the modularity of adding new functionality in the scenario-based programming model. The Notify scenario demonstrates that some behaviors might be triggered as consequences of other behaviors indirectly, by defining triggers that depend on the application state. In certain development scenarios, as in the case of adding notifications, such behaviors are more easily captured with such a predicate on the application's state. Importantly, note that we did not merge message sending and notifications into a single scenario—although this can be achieved in our model, such an implementation would not precisely capture the intended behavior, with respect to the strict interpretation of the sequential model semantics within a scenario (as discussed in § 3.2.1). We define message sending and notifications as two separate scenarios since in that case we can precisely capture the behavioral fragments of the distributed application in the sequential model (with guarantees of the sequential semantics covering the necessary portions of the behavior) and allow an efficient resulting implementation (that sends notifications our of order, as given in § 3.2.4).

Note that the body of action simply returns last (the last message). Our model could be extended to support declaring a trigger as a consequence of the SendMsg scenario and extend the scope to allow using the current message from that scenario. (Note that, this would be semantically different, as it would make notifications strictly dependent on the scenario Send.)

***Optimizing Notification Sending*** One (mentioned) optimization of the functionality would be to track and check usernames on the server and send notifications only to those users who are mentioned in the message. (Such optimizations are required for a reasonably practical distributed chat application.)

To emit an implementation that performs a local check before executing the whole scenario, we add a special annotation *@replicated*; see § 4.4.1. If Client.user is annotated with *@replicated(Server)*, the compiler infers that the value is accessible on the server as well and generates the optimized implementation (which, in addition, handles data replication).

This optimization demonstrates the advantage of the separation of concerns, where small changes in the specification of distributed aspects lead to a substantially different implementation. In contrast to other approaches that might require structural code changes, here, adding one annotation is sufficient.

## 3.3 Chat Topics and Handling Consistency

As the last piece of the functional requirements we consider, we explore functionality that operates on the same data as previously described behaviors, but might represent a challenge for incorporating it into the existing system due to data consistency concerns. It allows setting a discussion topic for the room, which in turn sends an appropriate message to all clients to notify them that the topic has changed (like in e.g. IRC). We model this functionality with a special command message that is sent to the server node, after which the server has the responsibility to change the topic.

An interesting observation and a challenging aspect of this functionality is handling the data consistency with respect to posting messages and receiving them on clients on the one hand, and changing the topic and topic notifications at clients on the

other. If the system exhibits an execution with a particular order of concurrent requests for sending a message and changing the topic, then observing those requests on the clients in different order might lead to inconsistent views (of the last message and the topic notification). Specifically, this might mean that a user's message is seen as posted in a different topic. Here, we consider causal consistency, which dictates that behaviors which are potentially causally related are seen by every node of the system in the same order [40]. (The details about consistency concerns, and guarantees in our model, are given in § 4.3.)

For brevity, we analyze approaches that are directly related to the solution in the scenario-based model and comment on the required implementation changes for the other programming models.

### 3.3.1 Changing Topics in Sequential Code

We model the behavior with a simple function that modifies the last message and notifies clients:

```
16 class Client { ...
23   def changeTopic(topic: String) = {
24     last = "Now discussing: " + topic
25     notify("Topic: " + topic) }}
```
Sequential

Note that, as in the case of implementing notifications, the behavior is conceptually simple and can be expressed with straight-line sequential code. However, we encounter issues similar to the ones we had when expressing notifications in the sequential model: it is unclear how should the defined function be executed in a distributed fashion across different nodes, and how should we model the message dissemination and behaviors on the clients. Moreover, the sequential computation model is oblivious to the notion of data consistency.

### 3.3.2 Changing Topics with Actors

Due to the possibility of concurrent execution of existing and new behaviors that might access and modify the same parts of the application's state, we have to enforce the right level of consistency in potentially conflicting behaviors. In the actor model, this usually reduces to preventing reordering of sending and handling of certain messages on different nodes. Here, the inconsistent view might occur if we allow clients to observe message sending and topic changing in different orders. To prevent receiving those messages out of order on the client, since messages for conflicting behaviors originate on the server, it is sufficient to impose an ordering on those messages and make sure clients handle them in the correct order.[5]

Since we need to establish an ordering between messages, we also need to change the existing class for notification messages and add a new class of messages for topic changes, while including an index, used for ordering, in both of them:

```
4 case class NotifyMsg(ind: Int, m: String) // change existing
5 case class TopicChangeS(topic: String) // handled at server
6 case class TopicChange(ind: Int, topic: String)
```
Actor

[5] Akka framework can guarantee that messages get delivered in the same order as they were sent by a particular actor, but only between two actors; nevertheless, we consider a more general solution for ensuring consistency.

Note that this requires changing all the places where NotifyMsg was sent or received in the existing code.

The server actor now needs to track the current index of the message that was delivered last and use it when sending either of the two messages. We declare a variable to track the index and define a case handler for changing the topic:

```
24 class ServerActor extends Actor { ...
30   var currentIndex = 0
31   override def receive = { ...                    // add a new case
38    case TopicChangeS(topic: String) {
39     last = "Now discussing: " + topic
40     currentIndex += 1                          // update the index
41     for (client ← clients) client !
42       TopicChange(currentIndex, topic) } }
```
Actor

In addition, we also need to change the code that sends NotifyMsg (inside the receive in the server actor), to include the correct index of the message:

```
33 currentIndex += 1                              // update the index
34 for (client ← clients) client !
35   NotifyMsg(currentIndex)
```
Actor

The client uses *stashing*, a feature in Akka that allows storing messages to be handled later [1]. We need to add and modify the handling of messages on the Client to use currentIndex, stashing, and later handling, messages received out of order. We do this for both notification and topic change messages:

```
17 case m@TopicChange(name) ⇒
18   if (m.ind == currentIndex + 1) {
19     notify("Topic: " + name)                  // notify the user
20     currentIndex += 1; hStash() // handle stashed messages
21   } else stash()                             // stash the messages
22 case NotifyMsg(product) ⇒ ...                 // similar
```
Actor

When the client receives a TopicChange message, it first checks whether the received message is the next message the client can handle, by comparing the last index the client saw (currentIndex) and the index of the message (m.ind). If the message is next in the sequence, the client handles it by notifying the user about the topic change, updates the current index, and handles the stashed messages. We omit the definition of hStash; it simply goes through all the stashed messages and delivers them in order. Otherwise, the message is not delivered at that point as it is received out of order. Such messages are stashed to be handled later by calling stash [1]. (We omit the modifications to the code for handling NotifyMsg, as it handles comparing and updating currentIndex similarly as in the handler of TopicChange.)

Although quite flexible in that it offers controlling the way messages are communicated and delivered, like custom orders of message delivery in this case, the actor model might require heavy disruptive changes to the existing implementations. Note that the implementation of the new functionality required not only disruptive changes to the existing code, but also adding substantial amount of boilerplate code that was needed for delivering messages in the specific order. Moreover, it required inspecting all places in the code where involved messages are sent or received. Although Akka provides built-in support for features like message stashing and customizing actor mailboxes,

handling distributed aspects such as consistency usually requires fairly general, but ultimately application-specific mechanisms, thus ultimately manual effort (such as identifying and ordering appropriate messages) [1, 29].

#### 3.3.3 Changing Topics with a Simple Scenario

In the scenario-based programming model, we again only need to declare one additional, relatively simple scenario to achieve the needed functionality, without making any changes to the existing code, thanks to guarantees of the compiler for ensuring consistency in the resulting implementation.

We declare a new scenario TopicChange that achieves the needed functionality and allows issuing requests for changing the topic on the client:

```
20 scenario TopicChange {
21   @location(Server) var last: String        // put into scope
22   @trigger(Client.topicAction)          // new client method
23   @input(Client.msg) @output(Client.notify)
24   def action(name: String) = {
25     last = "Now discussing: " + name
26     "Topic: " + name }}
```

Scenario

This scenario gets triggered when a user-defined (to be generated) method topicAction is invoked on the client. Since the scenario function action updates a variable bound to the server and takes an input parameter located on the client (we reuse msg), this scenario gets triggered at a client and processed at the server. Output of action is bound as an argument to notify that is located on the client (as defined in § 3.2.6). Therefore, the scenario sets the last seen message on the server (variable last) and notifies all clients about the topic change.

Note that we did not explicitly specify the consistency of data and behaviors that are involved in this scenario and existing scenarios from before. Our compiler uses a simple program analysis to check whether scenario behaviors might trigger at different nodes but modify the same variables. If so, it ensures that ordering of scenario executions, when observed at any node in the system, is preserved. (Our prototype guarantees causal consistency of scenarios, as described in § 4.3). The compiler generates the same code as presented in the Actor case, which tracks indexes and performs message stashing.

This example demonstrates that our programming model allows defining and intuitive reasoning about behaviors, while the consistency is implicitly guaranteed. This is done automatically, without any additional input from the programmer, due to the compiler ensuring the given semantics. Additionally, the sequential code that models the behavior of changing topics is again safely reused.

#### 3.3.4 Changing Topics in Other Models

In other considered models, preventing inconsistencies requires manually preventing reordering of certain requests or relying on provided centralized means for data control:

- In the event-driven model, reordering needs to be prevented manually (e.g. by explicitly marking events and enforcing their ordering). Therefore, developers need to reason and identify potential conflicts between events. Note that Sunny does not suffer from this problem directly, since it orders all possible events on the single server and manages view updates for all clients (this might come at high performance costs, as the server becomes a bottleneck) [46].
- In reactive and stream programming, consistency is implicitly preserved if potentially dependent requests are grouped as a single event that is delivered to all clients: in that case, all values are guaranteed to be delivered in the same order as they are created on the server. However, if behaviors are bound separately, in general, there are no guarantees on the ordering and consistency needs to be ensured by additional means or specialized propagation algorithms [24, 42].
- In our example with Redis, we could delegate preserving consistency to the store by grouping dependent data and declaring it with a stronger consistency level or perform the requests in a transactional way (sacrificing the amount of allowed concurrency and performance) [6].

### 3.4 The Role of the Sequential Model

The goal of our case study was to analyze the development process of a conceptually simple distributed application that exhibits a complex resulting implementation, throughout the process of adding new requirements. We believe the chosen example captures the requirements that are *common to a wider range of modern distributed applications*. Since some requirements cannot be directly handled in existing programming models, developers often face subtle difficulties and potential pitfalls during development, depending on the model they chose. Through examining the advantages and drawbacks of existing approaches, we showed that the scenario-based programming model offers a separation of concerns that *exposes conceptual behaviors in the sequential model*, delays dealing with the complexity inherent to distributed applications, and allows *hiding much of the implementation details behind declarative specifications*.

#### 3.4.1 Expressiveness of Analyzed Approaches

Most of the programming models we analyzed rely on the sequential computation model to some extent, as the basis for expressing computation, and use additional means to specify distributed aspects (such as communication and concurrency) of the intended application, while alleviating some of the burden of handling those aspects (usually by managing parts of the low-level implementations). However, the extent of this support tends to be limited to a certain class of supported applications; this comes at the expense of losing flexibility for implementing even conceptually simple applications *outside that class*. Therefore, these models tend to become *too strict and default to a sub-optimal predefined behaviors* (often unacceptable) that might end up hurting performance and the ability for further customization; both of which are often necessary properties for distributed application development. In general, implementing such applications amounts to additional complexity that usually needs to be handled manually, using lower-level language constructs. One

of the main points of our analysis was to examine *how disruptive is the transition of development and reasoning*, when going from the sequential model to the model that achieves full distributed implementations, depending on various requirements.

#### 3.4.2 Shared Aspects Between Different Models

Although we have identified several categories of programming models, many of their concrete representatives overlap on the adopted ideas and programming abstractions. Some of the overlaps include the support for: reactive client views in event-driven programming model Sunny [46], publish-subscribe support in Redis [6], and most of the actor functionality and features from the Akka framework naturally supported in Akka streams [1]. Nevertheless, our aim was to present main ideas of each approach that are faithful in spirit to the category they represent. In spite of a non-negligible overlap, we believe our analysis captures a representative sample of modern approaches for development of distributed applications, together with their shortcomings, to motivate an alternative, potentially more suitable, paradigm.

## 4. Scenario-Based Programming Model

In this section we describe the scenario-based programming model in more detail and define its key components. We present language elements and their semantics (that any chosen realization of the programming model should be compliant with) that capture all the needed properties of the model. We restrict possible executions of valid generated implementations, with respect to given specifications. We assume code fragments that represent basic scenarios (in a sequential model) are defined as simple functions within an existing host language and they follow its operational semantics (as shown in § 3). Our prototype compiler takes as input, as well as generates, code written in Scala [47].

### 4.1 Programming in Two Separate Phases

Since the goal of the programming model is to allow capturing, reasoning and testing behaviour of the system with sequential computation model, and later adding additional specification to produce a full implementation, the compiler should allow specifying scenarios in two incremental phases:

***Sequential computation phase*** In the first phase, developers identify and capture fragments of system's behavior with basic scenarios, which are simply represented with function (closure) definitions. This phase effectively corresponds to writing simple programs in the host language, while having the opportunity to use its compiler and execute them sequentially.

***Distributed computation specification phase*** In the second phase, developers specify how the captured fragments participate in the resulting distributed system by enriching basic scenarios with additional information that is sufficient to fully characterize the implementation. Developers need to add annotations to basic scenarios (from the first phase) to define distributed scenarios and specify the node configuration of the resulting system.

### 4.2 Scenario as a Language Element

The notion of a scenario is based on the idea of capturing an independent, conceptually self-contained, fragment of computation within the desired distributed application. We define a basic scenario to consist of a simple function together with all variables that are accessed within the function, i.e. a subset of variables in its scope (as described in § 3.1.8). On the other hand, the definition of a distributed scenario needs to capture additional information that determine the distributed aspects of the application and allow generating a full implementation.

A program consists of definitions of distributed scenarios and nodes of the system. We define distributed scenarios as an extension of basic scenarios with specifications of distributed aspects. A distributed scenario $s = (s_b, s_d)$ is defined with two components:

***basic scenario*** $s_b = (f, V)$, where $f$ is a function with a set of free variables $V$ (that are used inside $f$)

***distributed specification*** $s_d = (t, M, M_f)$, which is associated with a basic scenario $s_b$, where $t$ is the scenario trigger, $M$ is mapping of free variables of $s_b$ to nodes, and $M_f$ defines binding of function arguments and the return value.

In our prototype, nodes are defined by extending predefined `Node` classes, where all declared fields are associated with the containing node (as shown in § 3.1.9). Distributed scenarios are declared using the keyword **scenario** (which just represents a special object in Scala). Scenario objects define one function that represents the basic scenario, as well as the used variables that are allocated to nodes by using the *@location* annotation. Other distributed aspects are given as annotations to that function: *@trigger* (given a function) defines the trigger, and *@input* and *@output* bind the function arguments and the return value.

### 4.3 Scenario Semantics

Intuitively, the second phase should result in an implementation of a distributed system that behaves according to both the behavior defined with sequential execution of basic scenarios and the distributed specifications. We define the semantics of scenarios by constraining possible observable behaviors across all executions of the resulting distributed system.

#### 4.3.1 Semantics of Executing a Single Scenario

To capture the fact that a basic scenario is fully and faithfully represented in an execution of the generated distributed system, we consider a projection of the execution on each node in the system. We say that a basic scenario is *correctly matched* in the execution of the system if: some node executing an operation `op1` causes some other node to execute `op2`, only if executing the scenario (i.e. its function) in the sequential model would execute `op1` before `op2` (i.e. projected operations belonging to a scenario are executed in the same order), where the state used by `op1` and `op2` satisfies the distributed specification for that scenario.

Figure 1 depicts a correct matching. Boxes designate executions of potentially multiple operations, from a message receive (or a scenario trigger, in the case of the initial dotted

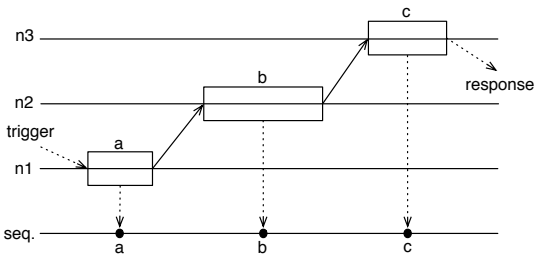


**Figure 1.** Scenario matched in a distributed execution. Timelines denote execution traces on different nodes, boxes the start and end of executions, and arrows denote communication.

arrow) that initiates the behavior on the given node, to the time the node sends a message to another node (or ends the scenario). The three given sequences of operations *a*, *b*, and *c* belong to a single scenario (i.e. its function). The sequential execution of the scenario is depicted on the *seq.* line. Within the whole system, the scenario is triggered at the node *n1* and in turn causes flow of messages and fragments to be executed on *n2* and *n3*. Therefore, assuming the basic scenario specifies a sequential execution order as given on the *seq.* line (and the state used in *a*, *b*, and *c* satisfies to the specification), since the execution order of operations matches the sequential execution of the scenario, the given scenario is correctly matched in this execution.

#### 4.3.2 Semantics of Scenario Composition

Intuitively, we define a composition of multiple scenarios as valid if for all possible executions of the implemented system, each executed operation and communication belongs to an execution trace that correctly matches at least one defined scenario. In addition, for all defined scenarios there needs to exist some execution of the system that contains a flow that correctly matches it. This ensures that all the specified functionality is indeed respected by the system, with respect to the programming model: *nothing is missed or artificially introduced* by the compiler.

Figure 2 depicts two scenarios, $s1$ and $s2$, with dotted line boxes. Even though their executions overlap across different nodes and within node *n3*, we map operations to corresponding scenarios with respect to the used state and given distributed specification. Thus, if we assume that program defines only the two given scenarios, since the given execution of the distributed system correctly matches both scenarios, this execution trace of the resulting implementation is considered valid.

#### 4.3.3 Scenario-Based Consistency

Having the semantics of scenario composition defined, there remains the issue of consistency across multiple scenario executions. Such an execution is similar to an execution of concurrent transactions on a multicore machine (with a subtle difference of semantically grouping operations that belong to the same scenario, which might be executed on different nodes). Since different scenarios might run concurrently, span across multiple different nodes, and involve reading and writing overlapping subsets of the application's state, allowing *arbitrary executions in the system might violate desired assumptions about data consistency*. This is not isolated only to concurrent executions of different scenarios, but also of the same scenario; such executions often span across different physical nodes, since scenarios capture behaviors with respect to the node type, which might be instantiated at different nodes during the system's execution (as it was the case in § 3).

Since scenarios capture fragments of behaviors that are conceptually self-contained, the goal of our programming model is to enable *natural and convenient reasoning* about (possibly concurrent) executions of scenarios within the resulting distributed system. To that end, we define consistency requirements that allow developers to view scenarios as happening atomically. (Note that this is conceptually different than providing atomicity of request handling, often supported in other approaches [19, 46, 57].) We introduce *linearization points* for scenarios, which allow reasoning about end-effects of possible executions as if *whole scenarios executed atomically*. Such a notion is reused, but slightly modified from the traditional definition of a linearization point, as it is applied to whole scenarios [31]. In Figure 2, the dots, in their respective boxes, designate scenario linearization points; even though scenarios might overlap in execution on different nodes, linearization points allow us to reason about their execution (i.e. their observable effects) as if they were happening atomically at the designated points (in time). The linearization points for scenarios $s1$ and $s2$ are defined with $f1$ and $f2$ on the *seq.* timeline. This particular execution trace entails the same effects as executing scenarios $s1$ and $s2$ in that order sequentially.

***Semantic Dependencies Between Scenarios*** Guaranteeing linearization points of scenarios alone is not sufficient for avoiding inconsistent state or improper ordering of scenarios (as demonstrated in our case study § 3, in the case of changing topics). Even if linearizable, scenarios might start executing (concurrently on multiple different nodes) as a consequence of other scenarios; in such cases *scenario linearization points need to be ordered according to such (semantic) dependencies*. Therefore, arbitrary orderings of linearization points are not acceptable in general. To achieve the desired consistency guarantees, we can impose additional rules on scenario executions, according to the dependencies defined by triggers. Effectively, our model allows *choosing a consistency model for scenario executions*, with respect to such dependencies, including some well-studied models in distributed computing, such as strong and causal consistency (which our prototype uses) [49, 57].

We describe the intuition behind defining different consistency models with respect to linearized executions of scenarios. To that end, we consider the system and its execution given in Figure 2, with only two scenarios $s1$ and $s2$. Here, the linearization points of scenarios, $f1$ and $f2$ on the line *seq.*, dictate the possible observable effects of the whole execution as if the system executed the two scenarios in the given order. A valid execution is then defined in the spirit of classical reasoning about consistency: the effects of the execution must be equal as if the scenarios were executed in some order that is compliant with the chosen con-

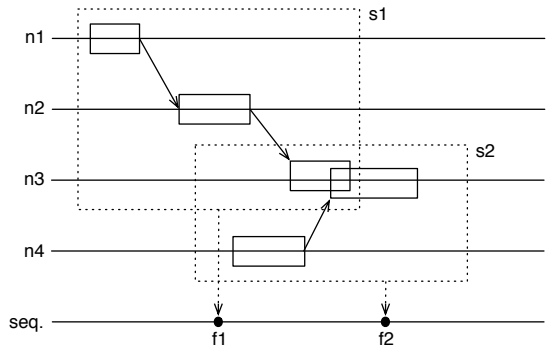


**Figure 2.** Multiple scenarios matched in a distributed execution

sistency model. Therefore, in the example from Figure 2, valid effects of executing scenarios in the distributed system must correspond to executing $f2$ after $f1$ sequentially, assuming that the order is consistent with the semantics of defined scenarios—e.g. for causal consistency defined below, $f2$ does not cause $f1$.

***Causally Consistent Scenarios*** For brevity, we only define causal consistency for scenarios [49] (that is currently supported in our prototype). We define *causality of scenarios through triggers*: if a scenario can cause another scenario to start, the two scenarios are causally related. We require that if an execution of scenario $s1$ caused the trigger for scenario $s2$ (on the same or different node), then their linearization points must respect the causal ordering. Namely, we reuse the standard definition of the *"happened-before" relation* and apply it to scenario linearization points [31, 40]. We define that $s1$ happened before $s2$ if either:

- some operation of $s1$ occurred before some operation of $s2$ on at least one node, or $s1$ caused the trigger of $s2$
- exists $s3$ that happened before $s2$ and $s1$ happened before $s3$

Valid implementations may allow only executions that respect this definition of causality. (We showed an implementation that satisfies this requirement in § 3.3.)

### 4.4 Implementation Concerns

Our prototype implementation is itself written in Scala and uses the Leon synthesis framework to parse and analyze scenario definitions and specifications, as well as to generate the resulting actor implementations [38]. The compiler processes scenarios one by one, producing an intermediate executable implementation after each step. The generated implementation relies on a small library that implements predefined constructs from the programming model using the Akka framework. All the synthesized implementations follow the constraints of the presented semantics and assume causal consistency of scenarios.

#### 4.4.1 Extensions of the Basic Model

Our programming model allows multiple extensions for supporting a wider class of applications and optimized implementations. Our prototype supports the annotation `@replicated(n)` which designates that the value is replicated on the node `n`. For such values our compiler allocates the data on all the specified nodes and emits additional code that propagates the value as soon as it's updated (akin to "push-based" reactive updates [13]). The messages for the state replication are ordered to achieve consistent view of data at times of scenario execution.

## 5. Future Work and Vision

We envision a more general, highly expressive language with a compiler that supports efficient program analyses and emits optimized implementations for a wide range of distributed applications (that go beyond the requirements considered in § 3). The language, together with the compiler, i.e. the synthesizer, should:

**support syntax** that allows convenient definition and reusability of scenarios depending on already defined fragments, through language features such as inheritance

**define full operational semantics** of the language and allow extensions such as different node classes and communication patterns to widen the class of supported applications

**provide choice of consistency levels** in a non-intrusive way

**leverage program reasoning** within the compiler to avoid using pessimistic mechanisms (e.g. for consistency) if it can be statically proved that they are not necessary

**require less input** by supporting partial programs (e.g. that omit full data mappings) and strengthening the synthesizer

**generate higher-level calls** to external frameworks to reuse common functionality (e.g. for communication or storage)

## 6. Related Work

This presentation focuses on the detailed analysis of only a subset of related programming models (presented in § 2), with a goal to summarize different related, but fundamentally distinct, perspectives on programming models for distributed systems. Note that not only other programming models might be sufficient in implementing certain pieces of functionality of our case study, they might be a better fit than some of the analyzed ones.

### 6.1 Analyzed Programming Models

In this section, we expand the description and general discussion about programming models we analyzed in our case study.

***Reactive Programming*** One of the difficulties in implementing interactive distributed applications is the "callback hell" problem—the excessive use of callbacks (imperative components that are invoked in response to asynchronous events) which results in complex control flow within and across different parts of the application [13]. Such code can be very difficult to reason about, especially since callbacks might modify application's mutable state. Reactive programming (RP) model tries to avoid this problem by capturing control flow and values changing over time behind clean abstractions [13, 52, 58]. Although initially proposed for modelling applications with dependent values in customized domains (such as programming animations), the model has been applied to distributed programming in several practical

frameworks and languages [13, 22]. One of the central notions in RP is a behavior, which abstracts a value that can change over time. The dependencies between behaviors are tracked: changes to any behavior automatically cause recalculation of all dependent behaviors. Functional Reactive Programming (FRP) does this by composing pure, side-effect free behaviors (also known as signals) and events [50, 58], while other frameworks use mechanisms such as reactive collections [4, 13].

Although offering flexibility in declaring dependencies between reactive values, the mechanisms for defining them tend to be rigid and need to be declared explicitly; hence, adding new functionality often requires heavy modifications of the existing code [50]. Due to relying on the structure of reactive dependencies, enforcing different consistency models, especially between unrelated behaviors, can be hard to achieve [24]. Moreover, RP approaches tend to be specialized towards restricted classes of applications (e.g. with a single server, reactivity only on clients [4]) or adopt communication schemas that do not offer handling of concerns such as scalability and consistency [13, 24, 42].

***Programming with Streams*** Programming with streams focuses on structuring programs as collections of modules that compute in parallel and communicate data via channels [53, 55]. Stream programming approaches offer primitives for constructing and managing streams and can be found implemented in various paradigms: logic, functional, imperative programming. Similarly in the basic idea to RP, they use abstractions that capture and transform sequences of values, albeit with different mechanisms and constructs [36, 50]. Even though they offer more explicit control of the flow of values and robust handling of changes within the system, they suffer from issues similar as in RP. Akka streams represents a modern and evolving instantiation of the model that tries to remedy some of those issues by allowing more control over the streams [5, 9].

***Event-driven Programming*** Event-driven programs are organized around event processing [23, 26, 33, 46]. Event-driven programming models thus mainly focus on representing applications as sets of events that occur in the system, usually modifying its state. The goal is to allow developers to focus on the business logic encapsulated with events, while alleviating the burden of managing communication and distribution to the underlying runtime. We analyzed Sunny, a high-level (and multi-tier) event-driven model that targets programming web applications with independent events and clear separation of the data model, views, and behavior [46].

Event-driven models are often inflexible for defining complex behaviors and communication patterns, since such patterns require defining multiple dependent events and handling their dependencies, in spite of those events representing the same behavior [46]. The concerns of the performance of event processing, event ordering and data consistency usually require manual effort from programmers; event-driven approaches offer only limited predefined data allocation models (such as a central storage) and mechanisms for event control.

***Actor Model*** The actor model gained popularity for developing distributed systems in the recent years and spawned multiple implementations in both academia and industry [1, 12, 17, 34]. In addition to representing concurrently executing units that communicate only by exchanging messages, actors impose additional restrictions such as data encapsulation and non-blocking communication, which are crucial properties for reasoning, robustness and scalability [10, 32, 34]. The model itself focuses on concurrent execution, while modern implementations allow allocation and communication between actors deployed across a network of nodes [1].

Although very general and flexible for implementing distributed applications of various architectures and requirements, actor model is often viewed as low-level: complex communication patterns force implementations to be spread into multiple message sends and handlers across different actors [29, 48]. With actors, developers need to structure their actor code according to appropriate data placement and communication patterns. Often, to achieve favorable performance, developers cannot rely on the predefined features such as location-transparency (due to performance considerations) and default schedulers (due to dependencies between messages) [48, 56].

***Key-Value Stores*** Distributed key-value stores provide strong support for data-centric distributed applications and allow easy control of aspects such as consistency levels, availability, as well as horizontal scaling [18, 57, 59]. They allow managing, evolving, and specializing data to support data-driven distributed applications, while offloading the burden of managing fault tolerance and replication needed for scalability. In contrast to other programming models, key-value stores clearly separate the concerns of storing and handling data, and implementing business logic, which has to be achieved separately. Even though the functionality of key-value stores is often limited, coupled with a good interface, the model provides additional features such as storing structured data and robust access through protocols like REST [8, 59].

Besides focusing only on managing data, key-value stores usually offer only a limited set of consistency modes that can be set in a coarse-grain fashion, usually per different segments of the store [6, 57]. This makes them harder to use if nodes have different consistency requirements, effectively pushing the concerns back to the application logic.

### 6.2 Other Related Approaches and Techniques

Next, we present other approaches and techniques related to high-level programming of distributed systems.

***Programming with RPCs*** Many approaches leverage some form of remote procedure calls to simplify achieving communication and abstracting away the necessary boilerplate code behind seemingly standard procedure calls [3, 4, 16, 19, 22]. RPC tries to remove unnecessary difficulties of building distributed systems like timing and communication in the context of distributed execution environments [15]. Interestingly, some of the emphasized issues of the analyzed models we discussed in § 3,

are related to the ones present in the RPC model and pointed out in prior work [54]. One of the main concerns with RPCs is inability to interchangeably mix normal calls and RPCs, due to both semantic (e.g. interference of global variables) and performance issues. (Functional approaches solve the former class of issues [3, 19].) Behaviors with complex communication patterns, even if conceptually coherent, might require splitting them into multiple RPC calls [54]. Scenario-based model separates the two responsibilities of programming to capture necessary information at the right level of abstraction and avoid such issues. Interestingly, it shares some of the potential issues with RPCs, like agreeing on completion and failure handling [54]. While such issues are inherent to the distributed nature of systems, our approach emits code that might employ external mechanisms to solve them.

***Publish-Subscribe Model*** The ability to define flexible many-to-many communication through channels and subscriptions offers an advantage for implementing a large class of distributed systems [16]. Many approaches incorporate publish-subscribe functionality, some of them mixing it with other models [4, 19, 22, 25, 42, 59]. On top of providing flexible means of connecting nodes (e.g. by transparently subscribing to topics), some approaches allow more complex patterns by composing subscriptions [16, 19]. However, publish-subscribe model often entails either inflexible runtime, or manual handling of lower-level details (which hinders the ability to schedule computation or allocate state on different nodes) and distributed aspects like consistency.

***Distributed Dataflow*** Dataflow languages model distributed applications as reactive networks which propagate signals, synchronously or asynchronously between their components [11, 30, 45, 53]. They resemble programming with streams, in terms of both expressiveness and issues, and are often considered as a specialized category of stream processing [30]. Many of such languages focus on programs in specific settings, such as real-time and critical systems like microcontrollers, which then map to specific low-level components. In distributed settings, they often exhibit performance issues and need to rely on specific schedulers and lightweight threads to manage computations [53].

***Language Design for Distribution*** A large body of research in programming languages brought many languages, such as Occam, Ada, and Linda, with primitives based on message passing, rendezvous, and remote procedure calls [14]. Moreover, constructs in certain paradigms were utilized for handling aspects like communication and error handling [30, 39, 43]. Recently, researchers started recognizing the role of data semantics for such concerns; conflict-free replicated data types, albeit currently limited, offer guarantees for consistency even with eventual communication [45]. Prior work that aims at specifying concerns (such as concurrency) orthogonally to sequential implementations extends languages with aspects [21] and pragmas [35]. While such approaches usually rely on deterministic code transformations, in our approach, distributed specifications dictate searching for a correct implementation in a potentially large space.

***Code Generation and Program Analysis*** The idea of increasing the level of abstractions in programming, to the point of declarative specifications, was attracting interest for a long time [28, 38]. By leveraging program analysis techniques, the goal is to allow programmers focus on editing high-level code and specifications, rather than final (optimized) implementations [38, 46]. Many systems rely on program analysis in order to generate and optimize (certain parts of) distributed applications [11, 19, 33]. Our approach is well aligned with this idea, with a strong emphasis on program analyses for searching for, or synthesizing, efficient implementations that satisfy distributed aspects given as specifications.

***Large Scale Distributed Processing Systems*** An interesting line of research involves programming models in the context of large scale distributed architectures. Many large scale processing systems provide frameworks based on specialized abstractions such as resilient distributed data types from Apache Spark, Google's MapReduce model of distributing computation, and publish-subscribe mechanisms like Apache Kafka. While these systems are usually built with low-level primitives (like RPCs), their frameworks do not expose them; their model is usually specialized towards specific kinds of distributed computation and is not flexible to address user-specific requirements.

***Multi-Tier Programming Models*** Developing distributed applications in a single language (and framework) is an idea shared by many approaches [3, 19, 20, 37, 50]. The focus of multi-tier approaches includes strong static checking, handling low-level boilerplate code, and leveraging programming abstractions across tiers (usually, for web technologies). However, these approaches tend to rely on some existing model for handling distributed aspects, such as RPCs ([3, 19]) and reactive components ([50]) for communication, and as such, offer similar perspectives (and issues) for development of distributed applications.

## 7. Concluding Remarks

We posed, and made an attempt to answer, a fundamental question about the sequential computation model and its relevance and applicability in the development of modern distributed systems. There are several conclusions we arrived at based on our exploration of the practice of programming distributed applications. Although sequential computation is used as basis in many approaches for programming distributed systems, the inherent complexity of dealing with distributed aspects is the essence of a large set of issues during development. Our perspective on the sequential computation model is that it remains a crucial component in the specification of a distributed system's behavior. To emphasize it, we hinted at a programming paradigm shift that could free the sequential model from the inherent complexity of distributed systems and allow retaining its simplicity.

We proposed scenario-based programming as a case in point and a step towards achieving the paradigm shift. By separating the concerns of modelling the sequential behavior of the application, the programming model allows specifying distributed

aspects, including data and computation location, reaction to stimuli, and data consistency, by writing orthogonal constraints without disrupting the existing code. We look forward to seeing the model as a fully-developed language and hearing reports on it from researchers and practitioners that explore this frontier.

## Acknowledgments

We thank Eunsuk Kang and Sean McDirmid for insightful discussions and the anonymous reviewers for feedback on a paper draft. This work was supported by the NSF grant *CCF-1438969*.

## References

[1] Akka – actor toolkit and runtime. `http://akka.io/`.

[2] Amazon elastic compute cloud. `https://aws.amazon.com/ec2/`.

[3] Jmacrorpc - reactive client/server web programming. `http://hackage.haskell.org/package/jmacro-rpc`.

[4] Meteor - pure javascript web framework. `http://meteor.com`.

[5] Reactive-streams - standard for asynchronous stream processing. `http://www.reactive-streams.org/`.

[6] Redis - in-memory data structure store. `http://redis.io/`.

[7] Rx - reactive extensions. `https://rx.codeplex.com/`.

[8] scredis – scala redis client. `github.com/Livestream/scredis`.

[9] Akka streams. `doc.akka.io/docs/akka/2.4.2/scala/stream/`.

[10] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[11] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, 2011.

[12] J. Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2007.

[13] E. Bainomugisha, A. L. Carreton, T. V. Cutsem, S. Mostinckx, and W. D. Meuter. A Survey on Reactive Programming. *CSUR*, 2013.

[14] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 1989.

[15] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 1984.

[16] J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and Distribution in Object-oriented Programming. *ACM CSUR*, 1998.

[17] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud Computing for Everyone. In *SoCC*, 2011.

[18] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 2011.

[19] A. Chlipala. Ur/Web. In *POPL*, 2015.

[20] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. *FMCO*, 2006.

[21] K. J. L. Cristina Videira Lopes. Abstracting process-to-function relations in concurrent object-oriented applications. *ECOOP*, 1994.

[22] E. Czaplicki and S. Chong. Asynchronous Functional Reactive Programming for GUIs. In *PLDI*, 2013.

[23] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *SIGOPS*, 2002.

[24] J. Drechsler et al. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *OOPSLA*, 2014.

[25] P. Eugster and K. R. Jayaram. EventJava: An Extension of Java for Event Correlation. In *ECOOP*, 2009.

[26] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 2003.

[27] S. Gilbert and N. Lynch. Perspectives on the CAP Theorem. *Computer*, 2012.

[28] P. T. Graunke et al. Programming the Web with High-Level Programming Languages. In *ESOP*, 2001.

[29] P. Haller and F. Sommers. *Actors in Scala*. Artima Incorporation, 2012.

[30] M. Henz, G. Smolka, and J. Würtz. Oz-a programming language for multi-agent systems. In *IJCAI*, 1993.

[31] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *Trans. Program. Lang. Syst.*, 1990.

[32] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, 1973.

[33] K. R. Jayaram and P. Eugster. Program analysis for event-based distributed systems. In *DEBS*, 2011.

[34] R. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: A comparative analysis. *PPPJ*, 2009.

[35] H. Kasim, V. March, R. Zhang, and S. See. Survey on parallel programming model. In *IFIP*, 2008.

[36] S. Khare, K. An, A. Gokhale, S. Tambe, and A. Meena. Reactive stream processing for data-centric publish/subscribe. In *DEBS*, 2015.

[37] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI*, 2007.

[38] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.

[39] S. Krishnamurthi et al. Modeling web interactions and errors. In *Interactive Computation: The New Paradigm*. Springer, 2006.

[40] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978.

[41] B. B. Lowekamp. Centrally distributed: P2P and the cloud for communications. In *IPTcomm*, 2011.

[42] A. Margara and G. Salvaneschi. We have a DREAM. In *DEBS*, 2014.

[43] M. F. Matthews, J., R. B. Findler, P. T. Graunke, S. Krishnamurthi. Automatically Restructuring Programs for the Web. In *ASE*, 2003.

[44] S. McDirmid. Taking Back Control (Flow) of Reactive Programming. In *REBLS*, 2014.

[45] C. Meiklejohn and P. Van Roy. Lasp: A language for distributed, coordination-free programming. In *PPDP*, 2015.

[46] A. Milicevic et al. Model-Based, Event-Driven Programming Paradigm for Interactive Web Applications. In *Onward!*, 2013.

[47] M. Odersky et al. An Overview of the Scala Programming Language (Second edition). Technical report, EPFL, 2006.

[48] A. Prokopec and M. Odersky. Isolates, Channels, and Event Streams for Composable Distributed Programming. In *Onward!*, 2015.

[49] M. Raynal and K. Vidyasankar. A distributed implementation of sequential consistency with multi-object operations. *ICDCS*, 2004.

[50] B. Reynders, D. Devriese, and F. Piessens. Multi-tier Functional Reactive Programming for the Web. In *Onward!*, 2014.

[51] S. S. Shim. The CAP Theorem's Growing Impact. *Computer*, 2012.

[52] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided... *Artif. Intell.*, 1977.

[53] R. Stephens. A survey of stream processing. *Acta Inform.*, 1997.

[54] A. S. Tanenbaum and R. van Renesse. A critique of the remote procedure call paradigm. Technical report, Vrije Universiteit, 1987.

[55] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel computing*, 2002.

[56] N. Venkatasubramanian and C. Talcott. A metaarchitecture for distributed resource management. In *ISCOPE*, 1999.

[57] N. Viennot et al. Synapse : A Microservices Architecture for Heterogeneous-Database Web Applications. *EuroSys*, 2015.

[58] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *PLDI*, 2000.

[59] J. Zawodny. Redis: Lightweight key/value store that goes the extra mile. *Linux Mag.*, 79, 2009.