Towards Distributed Reactive Programming

Guido Salvaneschi, Joscha Drechsler, and Mira Mezini

Technische Universität Darmstadt
lastname@informatik.tu-darmstadt.de

Abstract. Reactive applications is a wide class of software that responds to user input, network messages, and other events. Recent research on reactive languages successfully addresses the drawbacks of the Observer pattern – the traditional way reactive applications are implemented in the object-oriented setting – by introducing time-changing values and other ad-hoc programming abstractions.

However, those approaches are limited to local settings, but most applications are distributed. We highlight the research challenges of distributed reactive programming and present a research roadmap. We argue that distributed reactive programming not only moves reactive languages to the distributed setting, but is a promising concept for middleware and distributed systems design.

Keywords: Functional-reactive Programming, Scala, Event-driven Programming.

1 Introduction

Reactive applications respond to user input, packets from the network, new values form sensors and other external or internal events. Traditionally, reactive applications are implemented by using the Observer design pattern. The disadvantages of this style, however, have been studied for a long time now [7,15,13]. The major points of criticism include verbosity, lack of static reasoning, and inversion of the logic flow of the application.

A first solution proposed by researchers is to provide language-level support for the Observer pattern. For example, in C# classes can expose events to clients beside fields and methods. Other languages that provide advanced event systems include Ptolemy [16] – which supports event quantification – and EScala [9] – which supports event combination and implicit events.

A different approach is adopted by *reactive languages* that directly represent time-changing values and remove inversion of control. Among the others, we mention FrTime [7] (Scheme), FlapJax [15] (Javascript), AmbientTalk/R [12] (Smalltalk) and Scala.React [13] (Scala). This idea originates from functional reactive programming, which explored the semantics of time-changing values in the setting of (strict) functional languages. Another source of inspiration for reactive languages can be found in illustrious ancestors like the Esterel [4] and Signal [10] dataflow languages that focus on realtime constraints. Reactive languages are a promising solution and active research is ongoing in this field.

Reactive languages require careful design to avoid inconsistencies – referred to as *glitches* [15] – that can arise in the update process of time-changing values. All reactive

R. De Nicola and C. Julien (Eds.): COORDINATION 2013, LNCS 7890, pp. 226–235, 2013.

languages proposed so far do not provide these guarantees in the distributed setting. However, applications rarely span over a single host. Rather, most applications are distributed – the client-server architecture being the simplest case.

In summary, distributed reactive programming is an open challenge. Existing reactive languages take into account distribution in the sense that they are capable of transmitting events or propagating function calls among remote hosts that independently run a reactive system. In this way, however, consistency properties, locally enforced by a reactive programming system, are not guaranteed over reactive data flows that cross hosts [2]. We show that this limitation can lead to serious errors in the behavior of applications. Despite reactive programming having huge potential as a means to coordinate complex computing systems in a simple way, it struggles to succeed in this context. We argue that the aforementioned limitation is one reason for this.

The contributions of this paper are the following:

- We analyze the drawbacks of the existing approaches and motivate the need for reactive languages that uphold their guarantees in distributed applications.
- We propose a research roadmap for distributed reactive programming, including a first solution that demonstrates the feasibility of this goal and can be used for further design refinement and optimization.

Our vision is that distributed reactive programming can be a new way of structuring distributed systems. It centers around the concept of *Remote Reactives*, remote entities characterized by time-changing values, which clients can compose to model reactive computations and effects that span across multiple hosts.

2 Motivation

In this section, we introduce the basic concepts of reactive programming, show its advantages over traditional techniques, and describe the challenges of achieving the same result in a distributed setting.

Reactive Programming in a Nutshell. Reactive languages provide dedicated abstractions to model time-changing values. Time-changing values are usually referred to as *behaviors* [7] or *signals* [13]. For example, in Figure 1 we show a code snippet in Scala.React [13] that highlights the mouse cursor when it hits the borders of a window. In Scala.React, signals are time-changing values that are updated automatically; vars are time-changing values that are directly modified. In the example, the `position` signal models the time-changing value of the mouse position (Line 1). In Line 2, the `showCursorChoice` var is declared, which is changed by the user settings (not shown) to enable or disable the highlighting feature. In the rest of the paper, we generically refer to vars and signals as *reactives*. Developers can build expressions upon reactives. In Line 3 we create the `isMouseOnBorders` signal. It depends on the `position` signal and evaluates to true if the current position overlaps the window border. The key point is that reactive values that depend on other reactive values are automatically updated when any of their dependencies change. For example, when the position of the mouse changes,

```
1 val position: Signal[(Int,Int)] = GUI.getMouse().position
2 val showCursorChoice: Var[Boolean] = Var(false)
3 val isMouseOnBorders = Signal{ overlap(position(), GUI.getWindowBorders) }
4 val shouldHighlight = Signal{ isMouseOnBorders() && showCursorChoice() }
5 observe(shouldHighlight) { value : Boolean => setHighlightVisible(value) }
```

Fig. 1. An example of reactive programming

any expression associated with the `position` signal is reevaluated. In our example, the value of `isMouseOnBorders` is recalculated. This in turn causes the expression defined at Line 4, which combines both the `isMouseOnBorders` signal and the `showCursorChoice` var, to be reevaluated. Finally, every time this value changes, it is used to update the highlighting state through the `observe` statement in Line 5.

A detailed comparison of reactive programming with the Observer pattern is out of the scope of this paper. The advantages of this style over the Observer pattern have been extensively discussed in literature. The interested readers can refer for example to [7,15,13]. The main points are summarized hereafter. Callbacks typically have no return type and change the state of the application via side effects. Instead, reactives enforce a more functional style and return a value. As a result, types can guide developers and prevent errors. More importantly, differently from callbacks, reactives are composable, enhancing maintainability and software reuse. Finally, callbacks invert the logic of the application. Instead, reactives express dependencies in a direct way, improving readability and reducing the boilerplate code for callback registration.

Challenges of the Distributed Setting. As we have shown, reactive programming is an appealing solution to implement reactive applications. As such, it is desirable to move the advantages of reactive programming to a distributed setting, in the same way as, historically, functions were generalized to remote procedure calls and local events inspired publish-subscribe distributed middleware.

To discuss the challenges of this scenario, we present SimpleEmail, a minimal email application. In SimpleEmail, the server keeps the list of the emails received for an account. The client can request the emails that were received within the last n days and contain a given word and display them on the screen. The client highlights the desired word in the text of each email. Since an email text can span over multiple screen pages, it can be the case that there is no highlighted word on the current page, even if the server returned that email. To make this case less confusing for the user, the client displays a warning message. A proof-of-concept implementation is presented in Figure 2. We assume that reactive values can be shared with other hosts by publishing them on a registry, similarly to Java RMI remote objects. These objects are referred to as *Remote Reactives* (RRs). Clients can obtain a reference to RRs from the registry. For simplicity, we do not distinguish between remote vars and remote signals and we assume that a RR can be changed only by the client that publishes it.

The client (Figure 2a) publishes a RR that holds the word to look for (Line 2a) and another one with the number of days n (Line 4a). These reactives are automatically updated when the user inserts a new word or a new number in the GUI. The server (Figure 2b) retrieves the word and the days (Lines 5b and 7b) and uses these values to filter the stored emails (Lines 9b-14b). It publishes the RR that holds the result (Line 16b), which is in turn looked up by the client (Line 7a) and displayed in the GUI (Line 9a). The client additionally combines the filtered list of emails with the searched word to determine whether the word is visible on the first page (signal `hInFirstPage`, Line 12a). This value is then used to decide, whether or not to display the alert (Line 16a) informing the user that the search term is only visible on a different page.

Figure 3 models the dependencies among the reactive values in the SimpleEmail application (the c – respectively s – subscript refers to the client – respectively server – copy of a RR). Suppose, the user now inputs a different word. In the program, this would update the $\texttt{days}_c$ reactive. This change would be propagated to the server's instance $\texttt{days}_s$ of the reactive, where it causes the reevaluation of the $\texttt{filteredEmails}_s$ reactive. The new list of emails is then filtered with the current $\texttt{word}_s$ search term and the result is propagated back to the client to updates the display. Now suppose that, instead of $\texttt{days}_c$, $\texttt{word}_c$ is updated. This change is likely to propagate to $\texttt{hInFirstPage}_c$ first, because the propagation of the value to $\texttt{word}_s$ requires network communication. As a result, $\texttt{hInFirstPage}_c$ is evaluated with the *new* word, but the *old* list of emails. This can result in the word not being found on the current page and, in consequence, the warning message being displayed. Only later, when the change propagated through $\texttt{word}_s$, $\texttt{filteredEmails}_s$ and back to $\texttt{filteredEmails}_c$, $\texttt{hInFirstPage}_c$ reevaluates to false and the warning is switched back off.

In summary, spurious executions can be generated, depending on the evaluation order of the values in the graph. In our example, this issue only led to the erroneous display of a warning message. However, it is easy to see that in a real system that involves physical actuators, temporary spurious values can have more serious consequences. For example, in a train control system, a rail crossing could be temporarily opened, letting cars cross the railway while a train is approaching.

Temporary inconsistencies due to the propagation order are a known problem in reactive languages, commonly referred to as *glitches* [7]. Current reactive languages achieve glitch freedom by keeping the dependency graph of the reactive values topologically sorted; reevaluations are triggered in the correct order, avoiding spurious values. More details on this technique can be found in [15,13]. Current reactive languages, however, enforce glitch freedom only locally, but give no guarantees when the communication spans over several hosts as in the presented example. The fundamental point is, that true distributed reactive programming cannot be achieved by naively *connecting the dots* among single (individually glitch-free) reactive applications. On the contrary, dedicated logic must ensure glitch freedom as a global property of the entire system. Re-using topological sorting in the distributed setting would however force a single-threaded, centralized execution of updates across the entire application – an approach that we consider unacceptable. Next to finding and implementing a suitable remoting mechanism for reactives, developing an acceptable solution for ensuring glitch freedom in a distributed reactive application is the main challenge in supporting distributed reactive programming.

```
val word: Signal[String] = GUI.wordInput
publishRR{"word", word}
val days: Signal[Int] = GUI.daysInput
publishRR{"days", days}

val filteredEmails: Signal[List[Email]]] =
  lookupRR("filteredEmails")

observe(filteredEmails) { showEmails(_); }
observe(word) { setHighlightedWord(_); }

val hInFirstPage : Signal[Boolean] = Signal{
  isInFirstPage(word(), filteredEmails())
}

observe(hInFirstPage) {
  setShowHighlightOnNextPageWarning(_)
}
```

(a)

```
val allEmails : Signal[List[Email] =
  Database.emails

val word: Signal[String] =
  lookupRR{"word"}
val days: Signal[Int] =
  lookupRR{"days"}

val filteredEmails: Signal[List[Email]] =
  Signal{ allEmails().filter( email =>
    email.date < (Date.today() − days())
    &&
    email.text.contains(word())
  ) }

publishRR{"filteredEmails", filteredEmails}
```

(b)

Fig. 2. The SimpleEmail application. Client-side (a) and server-side (b).

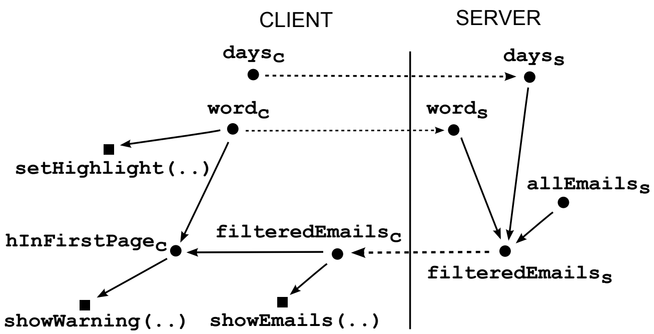

Fig. 3. Dependency graph among reactive values in the SimpleEmail application. Dashed arrows indicate over-network dependencies, square nodes indicate observers with side effects.

3 A Research Roadmap

In this section, we design an initial solution for distributed reactive programming, describe our plans to improve upon it, and present possible application domains.

Distributed Reactive Programming. We formulate our solution as a distributed transaction [18], for which well-known algorithms are available (e.g. distributed 2PL). Since persistency is not necessarily required, an implementation can leverage distributed software transactional memories [17], which have been proved to efficiently scale up to large-scale clusters [5]. A promising alternative for the implementation of our system is the automatic code generation from a specification. The advantage of such an approach is that an efficient and correct-by-construction distributed implementation is automatically derived from a formal model [11,6]. A more detailed comparison with automatic generation of distributed applications can be found in Section 4.

The following operations are a sufficient, minimal set of primitive operations to drive a distributed reactive system and must be executed transactionally. Without loss of generality, we only consider the case in which all the nodes in the distributed graph are remote. Possible optimizations can take advantage of the fact that local reactives are not visible by other participants.

A Participant Updates a RR Value. The participant starts a distributed transaction. The change is propagated across the distributed graph, each participant updating the RRs inside the same transaction. In case of a conflict between two transactions executing at the same time, one transaction rolls back, restoring the previous values of all affected reactives, and retries its execution. For this reason, the system requires separation of purely functional update propagation among reactives and side effects in the same style as Scala.React. The execution of the code preforming side-effects is deferred to after the transaction has successfully committed, thereby ensuring glitch freedom even in case a transaction has to be rolled back. Topological sorting of the dependency graph could still be used inside the update transaction to prevent multiple recalculations of the same functional parts. Doing so would, however, imply forcing all hosts' local update threads that are involved in the transaction to emulate a single-threaded execution. Ignoring the topological order would exploit parallel processing capabilities of multiple hosts and could even be used locally on multicore systems, but at the cost of multiple reevaluations. An ideal solution would be designing a new algorithm, which incorporates both concurrent updates of independent reactives (those that are incomparable in the topological order) and prevention of multiple recalculations of affected reactives. It is our goal to provide this hybrid solution.

The case of a *participant establishing a new dependency*, either independently or as the result of a reevaluation, can be considered a subcase of updating a reactive. Instead of updating the reactive's value, the participant updates the set of dependencies of the reactive. It is even possible, to update both at once. This must be done transactionally as well, to avoid interference with other changes. In addition, all graph modifications must be checked to prevent the introduction of dependency cycles.

A Participant Reads a RR Value. Reading must also be transactional, to avoid inconsistent state between subsequent reads of different reactive values – some of which could have been updated by a change propagation in the meantime. This can easily be achieved by executing all the reads inside a transaction.

The solution sketched above allows a certain degree of parallelism between transactions that do not interfere. In this sense, it is already an improvement over transferring the algorithms available in literature to a distributed setting by adopting a naive, centralized execution, regardless of how much independencies in the topological ordering are exploited. Our research plan is to use this solution as the first step to improve upon. For example, as sketched above, it requires finding a performant way to check the dependency graph for cycles. Based on the assumption that dependency changes are less frequent than value updates, using a centralized master that keeps a representation of the entire dependency graph would be a simple way to enable performant execution of this validation while still providing an improvement over executing all updates from a centralized coordinator. Still, this would imply a single point of failure and prevent concurrent changes to the dependency graph. Our plan includes exploring better design options that allow more decentralized approaches. Also, we expect to identify various trade-offs between provided guarantees and processing speed, some of which are described in the following paragraphs.

As long term goals, we envisage three application scenarios: Web applications, enterprise systems, and computing systems in an open environment. The remainder of this section elaborates our visions for each of these.

Web Applications. The solution presented so far provides a general model that is valid on distributed systems with an arbitrary number of hosts. This model directly applies when several browsers *share* resources among each other through a server[1]. However, we expect that in typical web applications only a client and a server share the RRs. While the aforementioned model proves the feasibility of our research vision, we expect that in a client-server model it can be significantly simplified, reducing the number of messages sent over network and gaining in performance. For example, Javascript applications enforce a single thread model, which solves the problem of synchronizing between concurrent changes for free.

Enterprise Distributed Systems. We envisage a scenario in which distributed reactive programming can be used to implement large-scale enterprise systems. In those environments, the consistency constraints required by distributed reactive programming should be provided by a dedicated middleware in the same way e.g. the JMS middleware provides *guaranteed delivery* and *once-and-only-once* semantics.

This research direction must take into account that enterprise software often does not run in isolation, but operates inside containers, like Tomcat or JBoss. For example, containers can provide persistence for Enterprise Java Beans (EJB). In a similar way, RRs can be modeled by EJBs, whose consistency is supported by the container

Highly Dynamic and Unreliable Environments. Distributed systems pose a number of challenges that include slowdown of network communication, node failures, communication errors and network partitioning. Those issues have been studied for a long time by the distributed systems community. An open challenge is to define proper behavior of a reactive system in such conditions. Proper language abstractions should support dealing with those cases with clear semantics. For example, in case of network partitioning, the system should be able to restructure the distributed graph with only the available hosts, but this clearly has an impact on the semantics of the application.

Another aspect we plan to explore is to relax the constraints as a possible reaction to network performance degradation. Reactive programming has been successfully used in the field of ambient-oriented programming [12], where systems are dynamic and unreliable and links can become slow. We envisage a scenario in which glitch-freedom guarantees can occasionally be switched off, in case maintaining them becomes too onerous. Similarly to before, proper abstractions should be introduced to deal with this case.

Evaluation A primary contribution of our research is to provide a new conceptual tool to support the design and the implementation of distributed software. The achievement

[1] For example, the AtomizeJS distributed software transactional memory supports this functionality, `http://atomizejs.github.com/`

of such a goal is hard to measure numerically, so we mostly expect feedback from the scientific community. More concretely, we plan to empirically evaluate our findings along two dimensions: On the one hand, we want to evaluate the impact of our solution on the design of a distributed system. Since these phenomena have already been observed in the local setting, we expect that static metrics can significantly improve, including reduced lines of code, reduction of the number of callbacks, and increased code reuse. On the other hand, we plan to evaluate the performances of our solution. Our goal here is not to make it competitive with models that require less stringent consistency guarantees, like publish-subscribe systems or complex event processing engines, but to provide a solution with significant design advantages at a minimum performance penalty.

4 Related Work

Due to space limitations, related work cannot be discussed extensively. Several languages implement concepts from reactive programming in various flavors. The interested reader can refer to [2] for an overview. None of them provides glitch-freedom in the distributed setting. Among the existing reactive languages, AmbientTalk/R [12] is the closest to our approach, and, to the best of our knowledge, it is the only one that has been specifically designed to support distribution in a form similar to remote reactives. In the remainder of this section, we point to the main research areas related to reactive programming and summarize their relation with the subjects discussed in this paper.

Dataflow languages, like Esterel [4] and Signal [10] focus on provable time and memory bounds. Differently form reactive languages and functional reactive programming, reactive abstractions are typically second-order, to support compilation to efficient finite state machines.

Self-adjusting computation [1] automatically derives an incremental version of an algorithm. It mostly focuses on efficiency and algorithmic complexity of the incremental solution, while reactive programming focuses on language abstractions for time-changing values.

The Implementation of distributed algorithms has been explored in form of code generation from formal specifications of I/O automatons [11]. A similar approach [6] generates the distributed programs from a description based on BIP (Behavior, Interaction and Priority) models [3]. We believe that the core of a middleware for distributed reactive programming can be conveniently specified in such a model and automatically generated. Consistency properties like glitch freedom would be provided through the synchronized state transitions offered by petri nets. However, we expect that the application code that builds on top of such a middleware (i.e. on remote reactives) is still written in a “traditional” language. As such, we believe it to be beneficial to allow programmers the implementation of all parts of their application seamlessly in their regular language in the style of reactive programming.

Publish-subscribe systems [8] leverage inversion of control to loosely couple interacting systems. Differently from publish-subscribe systems, we want to enforce a more tight coupling in the change propagation, to transfer the advantages of the local reactive semantics into the distributed setting.

Complex event processing (CEP) [14] is about performing queries over streams of data. Similarly to reactive languages, changes in the data result in an update to dependent values – in CEP, query results. Differently from reactive programming, CEP expresses dependencies via SQL-like queries that are usually expressed as strings and not integrated into the language.

Acknowledgments. This work has been supported by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant No. 16BY1206E and by the European Research Council, grant No. 321217.

References

1. Acar, U.A., Ahmed, A., Blume, M.: Imperative self-adjusting computation. In: POPL 2008, pp. 309–322. ACM (2008)
2. Bainomugisha, E., Lombide Carreton, A., Van Cutsem, T., Mostinckx, S., De Meuter, W.: A survey on reactive programming. In: ACM Comput. Surv. (2013) (To appear)
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: Fourth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2006, pp. 3–12. IEEE (2006)
4. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming 19(2), 87–152 (1992)
5. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2008, pp. 247–258. ACM, New York (2008)
6. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. In: Distributed Computing, pp. 1–27 (2012)
7. Cooper, G.H., Krishnamurthi, S.: Embedding dynamic dataflow in a call-by-value language. In: ESOP, pp. 294–308 (2006)
8. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
9. Gasiunas, V., Satabin, L., Mezini, M., Núñez, A., Noyé, J.: EScala: modular event-driven object interactions in Scala. In: AOSD 2011, pp. 227–240. ACM (2011)
10. Gautier, T., Le Guernic, P., Besnard, L.: SIGNAL: A declarative language for synchronous programming of real-time systems. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, Springer, Heidelberg (1987)
11. Georgiou, C., Lynch, N., Mavrommatis, P., Tauber, J.A.: Automated implementation of complex distributed algorithms specified in the ioa language. International Journal on Software Tools for Technology Transfer (STTT) 11(2), 153–171 (2009)
12. Lombide Carreton, A., Mostinckx, S., Van Cutsem, T., De Meuter, W.: Loosely-coupled distributed reactive programming in mobile ad hoc networks. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 41–60. Springer, Heidelberg (2010)
13. Maier, I., Odersky, M.: Deprecating the Observer Pattern with Scala.react. Technical report (2012)
14. Margara, A., Cugola, G.: Processing flows of information: from data stream to complex event processing. In: Proceedings of the 5th ACM International Conference on Distributed Event-Based System, DEBS 2011, pp. 359–360. ACM, New York (2011)

15. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for ajax applications. In: OOPSLA 2009, pp. 1–20. ACM (2009)
16. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 155–179. Springer, Heidelberg (2008)
17. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing 10(2), 99–116 (1997)
18. Weikum, G., Vossen, G.: Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. Morgan Kaufmann Publishers Inc., San Francisco (2001)