Multitier Reactive Abstractions

Pascal Weisenburger
TU Darmstadt, Germany
weisenburger@st.informatik.tu-darmstadt.de

Abstract

Distributed applications are traditionally developed using separate modules for each component in the distributed system, which can even be written in different programming languages. Those modules react on events such as user input, which are produced by other modules, and may in turn produce new events to be handled by different modules. Thus, most distributed applications are reactive in nature. Distributed event-based data flow makes it is hard to reason about the system and therefore makes the development of distributed systems challenging.

In this paper, we present language abstractions for distributed reactive programming easing the development of such applications and supporting various distributed architectures.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords Distributed Programming, Reactive Programming

1. Motivation

Despite a long history of research, developing distributed applications remains challenging. Among the sources of complexity, we find that distributed applications are often event-based [1, 10] and require to transfer both data and control among different hosts [15].

These two aspects make it hard to reason about the behavior of a distributed application because its runtime behavior depends on the interaction between the separate modules via events, whose occurrences can be unpredictable and potentially interleaving [5, 7]. Thus, keeping track of potential events, control flow and data flow between components is challenging. The separation into modules during the development process leads to developers having a local view of each module but may not be aware of the interaction regarding other modules within the entire system. Events neglected by the developer can lead to unexpected behavior of the system. As a result, when developers move from a local setting to a distributed one, they face a serious challenge [12].

Moreover, when developing a distributed application, the developer is required to choose the appropriate software stack for each component and master all different programming languages that come with it. This leads to a lot, potentially complex boilerplate glue code, which needs to be developed and maintained [9].

Aspects of this complexity can be addressed by *multitier* – sometimes referred to as *tierless – programming*, which aims to reduce the complexity of developing distributed applications, and *reactive programming (RP)*, which is a recent technique for developing reactive and event-based applications.

Multitier languages are designed to reduce the complexity and the amount of required boilerplate code needed to develop a distributed application. To achieve this goal, the complete application, including all tiers, is developed in a single multitier language. The gap between the different tiers is typically filled in by the compiler, allowing the developer to focus on writing the actual application logic rather than being occupied with writing boilerplate code.

In the object-oriented paradigm, reactive aspects of applications are traditionally implemented by using the Observer design pattern. This pattern comes with several issues including inversion of the control flow, inducing side-effecting operations to change the state of the application [4].

Functional reactive programming (FRP) [6] is a programming paradigm designed to overcome the problems of the Observer pattern. FRP allows for data flows to be defined directly and in a more intuitive manner. When declaring a reactive value, the reactive system keeps track of all dependencies of the reactive value and updates it whenever one of its dependencies changes. Reactives allow for better maintainability and composability as compared to observers [8, 11].

However, both techniques do not address the scope of problems when dealing with the event-based data flow within distributed applications.

SPLASH Companion'16, October 30 – November 4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4437-1/16/10...$15.00
http://dx.doi.org/10.1145/2984043.2984051

2. Problem

We claim that the complexity described in Section 1 is in a major fraction accidental and due to poor abstractions. Existing abstractions do not address the range of issues for the development of distributed applications and are insufficient to reduce the complexity by

I. allowing data flow spanning over multiple hosts to be specified directly,

II. allowing all components of the distributed system to be developed in an seamless way which allows for reasoning about the complete distributed system, and

III. guaranteeing safety properties across hosts.

The state-of-the-art development of distributed applications disregards one or more of the properties (I, II, III) motivating this work.

Traditionally, the different components of a distributed system are developed as separate modules. Those modules can even be developed in different programming languages, which makes it hard to reason about the component's interaction and thus about the behavior of the distributed system in its entirety. Our proposal to overcome this problem is inspired by the multitier approach (cf. II).

Furthermore, the traditional approach falls short on providing type safety guaranteed by the compiler for data flow across different hosts. Our language abstractions allow for the type-safe specification of data flow over multiple hosts in the system (cf. III).

Existing multitier languages for distributed programming, e.g., Hop [14] or Links [3], usually do not support data flow across hosts to be specified directly. Data flow from one host to another needs to be modeled by remote procedure calls and callbacks. We draw inspiration from reactive languages for specifying data flow and support its specification spanning over several components of the distributed system (cf. I).

Even existing languages like Ur/Web [2] which aim to combine multitier and reactive programming use a reactive system which is confined to a single component of the distributed system. Thus, they do not allow data flow spanning over multiple hosts to be specified directly.

Another common shortcoming regarding existing multitier languages is the lack of support for distributed application architectures other than client–server applications. Multitier languages like Ur/Web, Hop or Links are focused on client–server web development. Notably, this also applies to the existing approaches aiming for a deeper integration of multitier reactive programming, e.g., Scala Multi-Tier FRP [13].

3. Approach

We tackle the problem of developing a distributed application through dedicated language abstractions (w.r.t. I, II, III). Furthermore, the developer can freely define the distributed architecture for the system by specifying the components in the system and the relations between them.

```
1  class Client extends ClientPeer[Server]
2  class Server extends ServerPeer[Client]
3
4  val message: Event[String] on Client = new Event[String]
5
6  val publicMessage = placed[Server] { message.asLocalSeq }
7
8  placed[Client].main {
9    publicMessage.asLocal += println _
10   for (line <— io.Source.stdin.getLines)
11     message(line)
12 }
```

Figure 1. Simple chat application.

We outline the key concepts for our multitier reactive language referring to a concrete example of a small chat application given in Figure 1, where a server makes the messages from all clients available to all clients.

Peers and Peer Ties: Language-Level Distributed Architecture Definition The distributed architecture of an application can be defined within the language by specifying the scheme of the architecture at compile-time, e.g., whether an application belongs to a client–server or peer-to-peer scheme. The architecture is described using *peers* and *ties* between peers.

Peers represent the components of the distributed system while ties characterize the structure of the system as interconnected components approximating the runtime behavior regarding communication channels, i.e., a peer can only access remote abstractions of a tied remote peer. Each peer can be tied to arbitrary peers in the system.

For instance, the client–server architecture is defined by a *server* and a *client* peer that have a single tie to each other. A peer-to-peer architecture can be defined by a *node* peer that has multiple ties to itself because all peers in a peer-to-peer system are considered homogeneous and are represented by the same *node* peer. To differentiate between single and multiple ties, a tie specification states the tie's multiplicity.

The following code shows the previously described type definitions for a client–server and a peer-to-peer architecture:

```
trait Client extends Peer { type Tie = Single[Server] }
trait Server extends Peer { type Tie = Single[Client] }
trait Node   extends Peer { type Tie = Multiple[Node] }
```

Figure 1 uses predefined client and server peers (Line 1 and 2) which define the peer ties for a server that connects to multiple clients.

Abstraction Placement Computations can be *placed* on specific peers. A placed abstraction is an abstraction which is assigned to a certain peer and defines a computation that is executed on the assigned peer only. The placement of an abstraction is statically defined by the developer when declaring the abstraction.

Figure 1 illustrates the way abstractions can be placed on the previously defined peers. Placement can be specified using a *placed* expression (Line 6), where an expression of the form `placed[P] { e }` means that the computation expressed by *e* is placed on *P*. The placement of an abstraction is reflected in its type (Line 4). An abstraction of type `T on P` represents an abstraction of type *T* placed on a peer *P*. [1]

The compiler can check that the way abstractions refer to each other is consistent to the architecture defined through peer ties. For example, for an architecture that specifies that the two peers *A* and *B* and the two peers *B* and *C* are tied, a remote access from peer *A* to an abstraction placed on peer *C* will not pass compiler checks. Thus invalid access to placed abstractions is prevented to compile.

Cross-Peer Access Code placed on a peer can access the abstractions placed on the same peer. Furthermore, it can access abstractions placed another peer if and only if both peers are tied, thus realizing remote access across peers based on their ties. Remote abstractions, e.g., remote reactives, can be accessed in the same way as local abstractions except for a syntactic marker to make potentially costly remote communication visible to the developer.

Figure 1 shows how the placed abstractions *message* and *publicMessage* can be accessed remotely (Line 6 and 9). Remote access almost looks like local access and is statically typed, but requires an *asLocal* form, to explicitly mark the involvement of network communication.

When accessing remote abstractions, the multiplicity of the tie to the respective remote peer determines how values are represented locally, e.g., as a single value for a single tie or as an aggregation of multiple values for multiple ties.

Multitier Reactivity By supporting multitier reactive programming on the language level, data flow between peers can be directly specified to build reactive dependency graphs spanning over multiple hosts.

Figure 1 specifies a reactive dependency from the server-side *messagePublic* event (Line 6) to the client-side *message* event (Line 4). The call to *asLocalSeq* provides events from all connected clients sequentially. The client contains again a dependency to the server-side *messagePublic* by attaching an event handler and printing each new message (Line 9).

The data flow from all clients to the server and back to all clients over *message* and *publicMessage* is explicitly declared and type-checked.

4. Evaluation Methodology

Our objective is to provide multitier reactive language abstractions to ease the development of distributed systems. We claim that providing such abstractions leads to a reduction of boilerplate code and higher design quality of distributed reactive applications. To validate this claim we plan to perform side-by-side comparisons of alternative designs of distributed reactive applications. We plan to compare versions using a traditional approach of implementing all peers as separate modules and using the multitier language and both variants once using reactive abstractions and once not using reactive abstractions.

In order to affirm that the language can meet real-world requirements, we intend to reimplement existing open-source applications and show how our multitier reactive language compares in relation to the base version.

We plan to compare code metrics, e.g. lines of code, needed callbacks etc., as well as giving a side-by-side overview on how concrete problems encountered during the implementation are solved with and without our language to show the influence of our approach on the application design.

[1] `T on P` is syntactic sugar for `on[T, P]`

References

[1] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 2001.

[2] A. Chlipala. Ur/Web: A simple model for programming the web. POPL '15. ACM, 2015.

[3] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. FMCO '06. Springer-Verlag, 2007.

[4] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. ESOP '06, 2006.

[5] J. Edwards. Coherent reaction. OOPSLA '09. ACM, 2009.

[6] C. Elliott and P. Hudak. Functional reactive animation. ICFP '97. ACM, 1997.

[7] J. Fischer, R. Majumdar, and T. Millstein. Tasks: Language support for event-driven programming. PEPM '07. ACM, 2007.

[8] I. Maier, T. Rompf, and M. Odersky. Deprecating the observer pattern. Technical report, 2010.

[9] D. Manolescu, B. Beckman, and B. Livshits. Volta: Developing distributed applications by recompiling. *IEEE Software*, 2008.

[10] R. Meier and V. Cahill. Taxonomy of distributed event-based programming systems. In *Distributed Computing Systems Workshops*, 2002.

[11] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for Ajax applications. OOPSLA '09. ACM, 2009.

[12] A. Ranganathan and R. H. Campbell. What is the complexity of a distributed system? Technical report, University of Illinois at Urbana-Champaign, 2005.

[13] B. Reynders, D. Devriese, and F. Piessens. Multi-tier functional reactive programming for the web. Onward! 2014. ACM, 2014.

[14] M. Serrano, E. Gallesio, and F. Loitsch. Hop: A language for programming the web 2.0. In P. L. Tarr and W. R. Cook, editors, *Companion to OOPSLA '06*. ACM, 2006.

[15] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Separating data and control transfer in distributed operating systems. ASPLOS VI. ACM, 1994.