Connect.js

A cross mobile platform actor library for multi-networked mobile applications

Elisa Gonzalez Boix Christophe Scholliers Nicolas Larrea Wolfgang De Meuter
Vrije Universiteit Brussel
{egonzale,cfscholl,nlarrea,wdmeuter}@vub.ac.be

Abstract

Developing mobile applications which communicate over multiple networking technology is a difficult task. First, developers usually have to maintain a different version of the application for each mobile platform they target. Recent trends in mobile cross-platform solutions may alleviate this issue. However, developers still need to program a variation of the application for each different network interface. In addition, the APIs for communicating over ad-hoc networking technologies (eg. wifi direct), are very different from the cloud APIs. Finally, developers need to write highly asynchronous code for communication. This is often written with callbacks which invert the control flow of the application leading to code which is hard to debug and maintain. This paper introduces Connect.js, a JavaScript library for writing multi-networked cross-platform mobile applications. Applications consists of distributed objects which communicate with one another by means of asynchronous messages via a special kind of reference which is transparent for the underlying network technology used. Connect.js also provides dedicated language constructs for structuring asynchronous code by means of future combinators.

1. Introduction

Today we are witnessing a convergence in mobile technology and cloud computing trends. One the one hand, mobile devices have become ubiquitous. Many of them have more computing power than high end (fixed) computers developed 15 years ago. Moreover, they are equipped with multiple wireless network capabilities such as cellular network (3G/4G), wifi, bluetooth, wifi-direct, and NFC. As to be expected with any new technology, multiple mobile platforms are currently available (being the most relevant ones Android, iOS, and Windows Mobile). Important for the programmer is that each of these platforms have a radically different programming environment (e.g., Java in Android, Objective C in iOS, etc).

On the other hand, with the advanced on mobile broadband internet access, the web has evolved from data presentation layer to a data sharing and computation platform, to wit the cloud. Implementing mobile applications employing web-based technologies (like HTML and JavaScript) has the potential benefit of running in multiple mobile environments. However, mobile web-based applications usually perform worse, and do not provide a consistent look and feel than their native counterparts.

In order to minimize the software development costs, mobile cross-platform tools allow to develop one application for multiple mobile platforms [4]. Specially relevant are interpreted tools, like Appcelerator Titanium[1] or Xamarin[2], where developers write applications in a specific language (e.g., JavaScript or C++) and the tool builds a native application for the different targeted mobile platforms. They provide a number of built-in APIs for constructing native GUI and accessing the underlying hardware without requiring detailed knowledge of the targeted platform. The resulting rich mobile applications (RMAs) contribute to the intersection of mobile and cloud computing [1].

In this paper, we focus on a new breed of rich mobile applications which make use of both P2P communication and centralised wireless network access to coordinate and share data. Such *multi-networked* RMAs enable communication over both infrastructure-less networks of mobile devices, and the cloud. Developing such multi-networked applications burdens developers with the following tasks:

- Programmers need to implement a variation of the application for each network interface, and write complex failure handling code to support for reliable communication over multiple networking interfaces. Moreover, the API's for communicating over mobile ad hoc networking

[1] http://www.appcelerator.com
[2] http://xamarin.com

technologies like wifi-direct or bluetooth, are very different from the cloud API's. While the cloud typically requires a client-server communication model, the lack of infrastructure in ad hoc networking technologies requires a peer-to-peer communication model, in which services can be directly discovered in proximate devices.

- Programmers need to write highly asynchronous code for the network communication. This is often written with callbacks. However, these callbacks invert the control flow of the application leading to code which is hard to debug and maintain.

To overcome these issues we propose Connect.js, a mobile cross-platform development library for multi-networked mobile applications. In order to be able to communicate over multiple networking technologies, Connect.js introduces a novel kind of remote object reference which abstract over the kind of network interface being used, called network transparent references (NTR). As a result, applications can seamlessly communicate over the cloud or using an infrastructureless mobile network depending on the underlying available networking technology. NTRs offer reliable communication and as such, programmers do not need to manually verify the delivery of each message sent over multiple network interfaces. In order to mitigate the negative effects of callbacks, Connect.js provides dedicated language constructs for structuring asynchronous code by means of future combinators. Future combinators treat futurized messages as monads and provide a number of operators to combine futurized message passing while avoiding deep nesting associated with callbacks. We believe that the combination of NTRs and future combinators eases programming multi-networked rich mobile applications.

2. Connect.js

Connect.js is a mobile cross-platform library integrated in Appcelerator Titanium which allows programmers to write their distributed mobile applications in JavaScript and deploy it on several mobile platforms, namely iOS and Android. Figure 1 shows the general architecture of Connect.js. Programmers write mobile applications in JavaScript importing the Connect.js library in their Titanium project for distributed programming. The abstractions provided by the Connect.js API are based on the ambient-oriented programming model from AmbientTalk[2] which treats network partitions as a normal mode of operation. As in AmbientTalk, every device hosts at least one actor which encapsulates one or more objects. Objects can communicate with objects in another actor system by means of sending asynchronous messages via a special remote reference called a far-reference. To this end, Connect.js incorporates a built-in service discovery mechanism which allows to discover services in devices accessible under the same ad hoc network or via the cloud, independently of the mobile platform of the device (explained in the next section). From an implementation point of view, Connect.js contains two different native modules (also called plugins) for service discovery on an ad hoc network based on zero-configuration networking technologies specific to the mobile platform. The iOS plugin uses Bonjour [3], and the Android plugin employs NDS [4] We then rely on the JavaScript - Java/ObjectiveC bridge from Titanium to transform the JavaScript code into native applications in the targeted mobile platform.

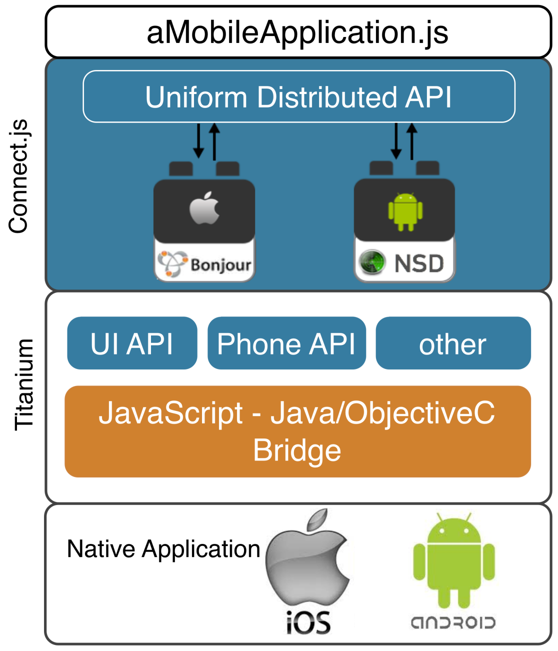

Figure 1. Architectural Overview of Connect.js.

2.1 Network Transparent References (NTRs)

Connect.js considers actors as the unit of distribution, and as such two objects are said to be remote when they are owned by different actors. The only type of communication allowed on remote object references is asynchronous message passing. Any messages sent via a remote reference to an object are enqueued in the message queue of the owner of the object and processed by the owner itself.

In Connect.js actors communicate with one another over wireless links or mobile broadband access. As such, remote references in Connect.js abstract the underlying networking technology being used for communication. Such *network transparent references* (NTRs) are resilient to network fluctuations by default. When a network technology (e.g. wifi) is not available, the NTR attempts to transmit messages sent to it using another networking technology, i.e. 3G. If all networking interfaces are down, the remote reference starts buffering all messages sent to it. When the network partition is restored at a later point in time, the far reference flushes

[3] https://developer.apple.com/bonjour/

[4] http://developer.android.com/reference/android/net/nsd/NsdManager.html

all accumulated messages to the remote object in the same order as they were originally sent. As such, temporary network failures or fluctuations on the availability of the different network interfaces does not have an immediate impact on the applications' control flow.

To illustrate NTRs and the different distributed programming constructs in Connect.jsconsider the following code snippet from a chat application:

```
var buddyList = {};
var Ambient = require("js/connectjs/ConnectJS");
function initializeMessenger(name) {
 //....
  Ambient.wheneverDiscovered("MESSENGER",
   function(ntr){
     var msg = Ambient.createMessage(getName,[]);
     var future = ntr.asyncSend(msg,"twoway");
     future.whenBecomes(function(reply) {
        buddyList[reply] = ntr;
        // send a salute message
     });
   });
}
```

Listing 1: Example use of asynchronous message passing over NTRs

The `wheneverDiscovered` function takes as arguments a string representing the service type and a function serving as callback. Whenever an actor is encountered in the ad hoc network that exports a matching object, the callback function is executed. The `ntr` parameter of the function is bound to a network transparent reference to the exported messenger object of another device.

In order to define objects exporting service to the network, the `exportAs` function is employed. Code snippet below shows how to create an object which implements a service corresponding to the chat application using the string MESSENGER as service type.

```
var remoteObj = Ambient.createObject({
  "getName": function ()  { return myName; },
  "talkTo": function (msg) { displayMessage(msg);}
});
Ambient.exportAs(remoteObj, "MESSENGER");
```

With the code provided two phones can already communicate with each other when they are within direct communication range. However, when the phones are not in direct communication range, Connect.js allows the phones to communicate with each other through a centralised node server. To this end, the programmer needs to configure a node server so that service objects are allowed to be exported as well via in an intermediary server in the cloud. The code to configure the node server is shown below.

```
var Ambient=require("js/connectjs/ConnectJS"),
  express = require("express"),
  nodeServer = require("http").Server(express());
nodeServer.listen(3000);
Ambient.initServer(nodeServer);
```

2.2 Future as a Monad

.

The use of asynchronous communication has proven to be beneficial to build distributed mobile applications because it mitigates the negative effects of frequent network failures. However, asynchronous primitives suffers from similar problems as traditional callbacks. In this section we give an overview of how defining futures in terms of a monad allows programmers to better structure their asynchronous code. We also present a number of combinators which turn out to be useful but do not follow the monadic structure.

2.2.1 The Future Monad

The basic constructs of futures can be formulated as a monad [3]. Listing 2 shows the unit and bind type signature of the monad typeclass in Haskell. A monad has two operations, `bind` (>>=) and `unit`. Bind takes a monad of `x` and a function which takes a value of type `x` and returns a monad `m` y. Unit lifts a regular value of type `x` into a monad of `x`.

```
class Monad m where
 (>>=)  :: m x -> (x -> m y) -> m y
 unit :: x -> m x
```

Listing 2: Monad typeclass in Haskell

While we can not reap the advantages of the Haskell type system to enforce monadic behaviour we can still implement the basic bind an unit operators over futures in Javascript. Pseudo code [5] for the bind operator as defined in the future prototype in Connect.js is defined in listing 3. When `bind` is applied on a future `F`, a new future `R` is created (line 2). This new future is also the result of applying `bind` (line 9). Bind registers an anonymous function on the future `F`. This anonymous function resolves the future `R` with the result of applying the function `f` to the value where the future `F` resolves to (line 4-8).

More easily the return operator lifts a normal value into a future value. Return simply creates a new future and immediately resolves the future with the given value.

2.3 Future Combinators

The basic monadic operators together with the lift construct provide a very useful set of abstractions for composing asynchronous computations. However, in our context programmers often need some more high-level abstractions. Our li-

[5] For clarity we omitted the code for future pipelining here.

```
// bind :: (x-> future[y]) -> future[y]
function bind(f) {
  var R = new Future();
    self.register( function(v) {
      f(v).register( function(res){
          R.resolve(res);
        });
    });
    return R;
}
```

Listing 3: Monad bind in the Future Prototype of Connect.js

brary therefore defines a set of operators implemented on top of these basic monadic combinators. An overview of these operators is shown in tabel 1. Note that the in the type signature `f [a]` should be read as: A future which will resolve to an array of a's.

Conditionals	
if	`f a -> (a -> Bool) -> f a`
or	`f a -> f b -> (f a | f b)`
Simple synchronisation	
first	`f a -> f b -> (f a | f b)`
last	`f a -> f b -> (f a | f b)`
Group synchronisation	
many	`f[a]->(a->Bool)->f[a]`
some	`f[a]->(a->Bool)->f[a]`
all	`f[a]->(a->Bool)->f[a]`
Array operators	
filter	`f[a]->(a->Bool)->f[a]`
map	`f[a]->(a->b)->f[b]`

Table 1. Overview of the basic future combinators.

2.3.1 Future Combinator: Chat example

In this section we give a small example to showcase how the asynchronous nature of the communication can be hidden by the use of our future combinators. Consider the implementation of sending a friendly message to your two best buddies. We first define a (synchronous) function which checks that the user is either Christophe or Elisa (lines 1-4). Next we define a function which sends out a friendly message to each user in a given array of users (lines 5-7). Finally we first get the list of all our friends from the server (line 8), we filter this list (line 10), we transform the list of users to a list of user objects (line 11), and send a message to each of them with the function sendFriendlyMsgs (line 12).

3. Validation

In order to compare our approach to existing approaches we have implemented two example applications: a chat application and a distributed unit test suite. For both applications we have compared the percentage of lines which are attributed

```
1  var filterBuddy = function(user) {
2     return "Christophe" == user.firstname
3             || "Elisa" == user.firstname;
4  };
5  var sendFriendlyMsgs = function(users) {
6          users.map(sendFriendlyMsg)
7  };
8  var msg = Ambient.createMessage("getFriendList");
9  server.asyncSend(msg, "twoway")
10       .filter(filterBuddy)
11       .reduce(makeBudObj,{})
12       .whenBecomes(sendFriendlyMsgs);
```

Listing 4: Example use of future combinators

to the application logic compared to the lines of code for communication. The chat application has been implemented twice in Connect.js, once with network aware references and once without. For the unit testing framework we also implemented the tests twice, once with normal futures and once with future combinators. The results for both the chat application and the unit testing framework are shown in figure 2. As shown in the figure in both cases there is a clear shift from the percentage of lines spent for the application logic compared to the lines of code attributed to the communication logic. While not shown in the figure the absolute lines of code for each application also reduced.

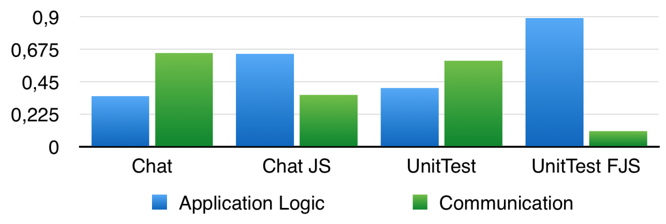

Figure 2. Comparison in usage distribution

4. Conclusion

In this paper we reported on initial work on Connect.js : a cross mobile-platform actor library for multi-networked RIA applications. With Connect.js the programmer does no longer need to write complex network code in order to exploit the use of both P2P communication and communication over a centralised wireless network. To this end, Connect.js provides the programmers with a new kind of distributed object reference called network transparant references. When one network interface fails these reference automatically and transparently tries to send the message over another network interface. Finally, in order to minimise the negative effects of writing asynchronous code Connect.js provides a set of future combinators. Initial validation of our artefact on two small applications shows promising results.

References

[1] S. Abolfazli, Z. Sanaei, A. Gani, F. Xia, and L. T. Yang. Review: Rich mobile applications: Genesis, taxonomy, and open issues. *J. Netw. Comput. Appl.*, 40:345–362, April 2014. ISSN 1084-8045. .

[2] T. V. Cutsem, E. G. Boix, C. Scholliers, A. L. Carreton, D. Harnie, K. Pinte, and W. D. Meuter. Ambienttalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures*, 40(34):112 – 136, 2014. ISSN 1477-8424. .

[3] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press. ISBN 0-8186-1954-6.

[4] S. Xanthopoulos and S. Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, pages 213–220, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1851-8.