AmbientJS
A Mobile Cross-Platform Actor Library for Multi-Networked Mobile Applications

Elisa Gonzalez Boix[1], Kevin De Porre[1], Wolfgang De Meuter[1], and Christophe Scholliers[2]

[1] Vrije Universiteit Brussel, Pleinlaan 2, 1050, Brussel, Belgium
[2] Ghent University, St. Pietersnieuwstraat 33, 9000 Gent, Belgium

Abstract. In this paper, we argue that due to technological advances programmers today are faced with a ninth fallacy of distributed computing: "*there is only one fixed application architecture throughout the lifetime of the application*". Mobile devices are nowadays equipped with wireless technology which allows them to interact with one another in both a peer-to-peer way (eg. Wi-Fi-direct, bluethooth,etc.), and via a server in the cloud. Distributed software engineering abstractions, however, do not aid the programmer in developing mobile applications which communicate over multiple networking technologies. This paper introduces AmbientJS, a mobile cross-platform actor library which incorporates a novel type of remote reference, called network transparent references (NTRs), which allows to seamlessly combine multiple application architectures. We give an overview of the NTR model, detail their implementation in a novel actor library called AmbientJS and assess the performance of AmbientJS with benchmarks.

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 a decade ago. Moreover, they are equipped with multiple wireless network capabilities such as cellular network (3G/4G), Wi-Fi, bluetooth, Wi-Fi-direct, and NFC. As to be expected with any new technology, multiple implementation platforms are currently available (being the most relevant ones Android, and iOS). 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). In order to minimize the software development costs, mobile cross-platform tools have emerged which allow the programmer to develop applications which can run on multiple mobile platforms. Many of these mobile cross-platform tools make use of web-based technologies such as Javascript and HTML5[30]. While these tools ease the development of certain application aspects like GUI construction, they still fall short with respect to support for distributed programming.

In this paper, we focus on a new breed of mobile applications which make use of both peer-to-peer communication and centralised wireless network access to coordinate and share data. Such *multi-networked* mobile application enable communication over both infrastructure-less networks of mobile devices, and the cloud. Note that many distributed programming abstractions already abstract away the details of the underlying network technology. For example, a socket abstracts the details of communication over a Wi-Fi connection or a bluetooth connection. However, sockets do not aid the developer in simultaneously using a Wi-Fi and bluetooth connection and seamlessly switch between them based on the connectivity. Developing rich mobile applications thus burdens developers with the following tasks:

– Programmers need to implement a different version of the network layer for each network technology (Bluetooth, Wi-Fi, 3G, etc).
– Programmers need to adapt the application to support multiple architectures (peer-2-peer, client-server) depending on the network layer employed.
– Programmers need to write complex failure handling code to be able to reliably combine multiple network interfaces and application architectures.

To overcome these issues we propose AmbientJS, a mobile cross-platform development library for multi-networked mobile applications based on the actor model. In order to be able to seamlessly communicate over multiple networking technologies, AmbientJS introduces a novel kind of *extensible remote object reference* which abstracts over the kind of network interface being used. We call such object references network transparent references (NTR). As a result, applications can seamlessly communicate over the cloud or use 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 this paper we argue that the use of NTRs implemented in the AmbientJS middleware greatly simplifies the creation of multi-networked rich mobile applications.

2 Motivation

Mobile cloud computing was initially employed for applications such as Google's Gmail which run on a rich server and the mobile device acts as a thin client connecting to it via 3G. However, due to the recent availability of networking capabilities on mobile devices, mobile cloud computing is converging to what we call *rich mobile applications*, in which the mobile devices themselves can also act as service providers and communicate with one another. In this section, we highlight the need for programming abstractions for rich mobile applications. Based on the analysis of an illustrative application, we derive a number of software engineering issues that programmers face when implementing such applications.

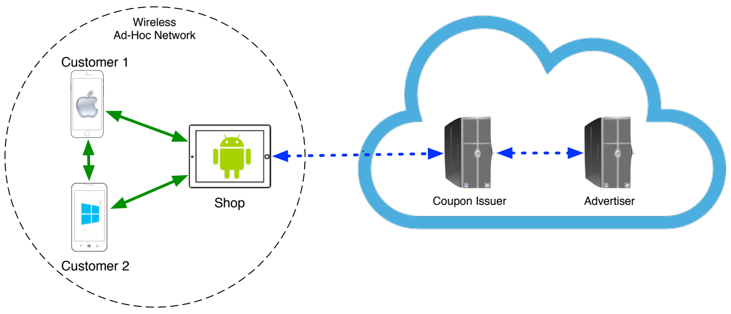

Fig. 1. Coupon Go architectural overview

2.1 Case Study: Coupon Go

We now introduce an industrial case study, called *Coupon Go*, devised together with a Brussels Region company specialized in digital vouchers. This application allows the distribution and redemption of digital, electronic coupons. Figure 1, provides an architectural overview of the system. As shown in the figure, the architecture of Coupon Go consists of four different actors:

- *Coupon Issuer.* The coupon issuer is in charge of the distribution of coupons to the different users upon request of an advertiser. The issuer maintains the digital wallets of the customers, and has access to the shopping history of the customer which can be used for constructing user profiles.
- *Customers.* The users of the electronic coupon system hold a digital wallet on their mobile device, which is populated in two ways: (1) users receive a coupon from the coupon issuer or from a merchant while they are in a shop, and (2) they can get coupons from other digital wallets, e.g members of the same family transferring a coupon from one wallet to another one.
- *Shops.* Shops notify the coupon issuer when customers redeem a valid coupon. This can happen instantly if there is an internet connection available at the store, or asynchronously, e.g. at the end of the day with all the sales. Once a redeemed coupon is accepted by the issuer, the coupon is said to be *granted*, and process of repaying the store can be initiated.
- *Advertisers.* Advertisers request the generation of digital coupons to the coupon issuer for a specific product or group of products. An advertiser can also make use of the user profiles constructed by the coupon issuer to allow for more targeted advertising.

Note that the envisioned electronic coupon system, the digital coupons are first created by the coupon issuer but customers receive a copy of the coupon which is stored in their wallet. This provides offline functionality so that coupons can be validated at stores or transfer between wallets in the absence of an internet connection.

2.2 Analysis

The programmer of a rich mobile application is faced with all the traditional software engineering problems of distributed applications. However, rich mobile applications like the Coupon Go application exhibit several properties that distinguishes them from other types of distributed applications.

- First, the entities of the distributed system employ various network technologies to communicate (i.e. WPAN, WLAN etc.).
- Second, they combine multiple distributed application architectures within the same system, i.e. client-server and peer-to-peer.
- Finally, those applications need to be deployed on multiple software platforms e.g. iOS, Android, Windows mobile devices, back-end servers.

This puts extra burden on software developers. As a matter of fact, these properties act as different dimensions impacting software development.

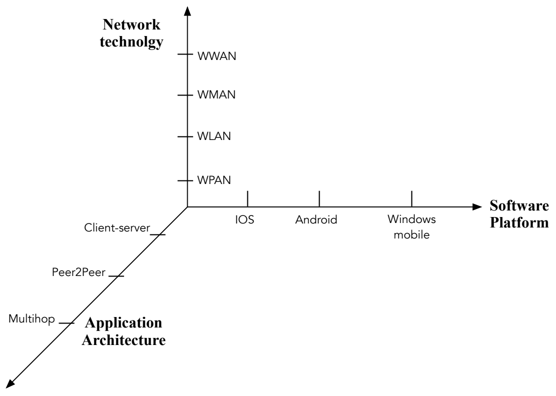

Fig. 2. Design Space of Rich Mobile Applications

Figure 2 shows the design space of rich mobile applications. From the point of view of the software platform, the tools and environments provided by the different platforms are often not compatible with each other. In many cases the implementation language itself is different, e.g, Java for Android devices and Swift for iOS devices. Moving from one platform to another requires programmers to rewrite the application from scratch.

From the distributed application architecture perspective, programming a traditional client-server application is very different from programming a peer-to-peer application over e.g. Wi-Fi-direct. A peer-to-peer application includes a discovery process which is not required when contacting a centralised server in a client-server architecture. When the application is not designed from scratch with multiple network technologies large portions of the application need to be rewritten.

Finally, from the network technology perspective, the networking libraries available for communicating over a bluetooth connection are very different from the libraries for creating HTTP request. Again when the application is not written with these different application requirements from the start there is a big impact on the overall architecture of the application when using a different network technology.

2.3 Problem Statement

There is a long history in distributed computing in order to pinpoint appropriate methodologies and software practices for writing distributed applications. One particular important seminal article called "the fallacies of distributed computing" gives an overview of a number of misconceptions that traditional programmers have when first implementing a distributed application. These fallacies are: the network is reliable, latency is zero, bandwidth is infinite, the network is secure, topology doesn't change, there is one administrator, transport cost is zero and the network is homogeneous [5].

Traditional distributed applications typically only exhibit one single application architecture which does not change during the lifetime of the application. In a rich mobile application, on the other hand, the kind of architecture evolves during the applications lifetime depending on the networking technology available for communication. Moreover, these network architectures can be active a the same time. For example, a device could communicate with a nearby device with ad hoc networking technology while communicating with a centralized server. In our case study, a customer can get coupons directly from a shop or a nearby digital wallet. However, when that user moves out of direct communication range, the application can seamlessly switch communication through a centralized server available on the Internet by 3G or Wi-Fi connection. These kind of scenarios were not within the reach of everyday mobile applications when the eight fallacies of distributed computing were conceived. Today, however, every mobile device has a multitude of network facilities which can only be put to good use when developers are given the right software engineering principles. We argue that due to the technological advances programmer today are faced with a ninth fallacy when writing rich mobile applications. **The ninth fallacy of distributed computing:**

There is only one fixed application architecture throughout the lifetime of the application.

In the rest of this paper we formulate our answer to tackle this ninth fallacy of distributed computing under the form of a new distributed library called AmbientJS. AmbientJS introduces a novel extensible remote object reference which abstracts over the kind of network interface being used.

3 AmbientJS

AmbientJS is a mobile cross-platform library for JavaScript specially designed to ease the development of rich mobile applications. The library embodies the principles of ambient-oriented programming model from the AmbientTalk language[4] and extends them to support multiple distributed application architectures employing different network technologies. The idea of AmbientJS is that programmers can write their rich mobile applications in JavaScript as *one single application* which consists of different *actors* that can be distributed over mobile devices (iOS and Android devices). At server side, AmbientJS can be used in combination with `node.js`. Communication between different distributed actors employs a uniform distributed object model which abstracts over the different networking technologies employed by the application architecture. This includes *direct* peer2peer communication and *indirect* communication through a server. We will first introduce the general architecture of AmbientJS and then focus on its programming support.

3.1 AmbientJS General Architecture

In order to deal with the diversity of software platforms, AmbientJS has been integrated as a JavaScript library to be used on top of mobile cross-platform technology. However, mobile cross-platforms frameworks are not homogenous. There exists two big families of approaches: interpreted and hybrid technologies [30]. Nevertheless, AmbientJS has been designed as a mobile cross-platform agnostic library. This means it can be used on top of the most relevant incarnations of interpreted and hybrid mobile cross-platform technologies, namely Cordova and Titanium.

Figure 3 shows the general architecture of AmbientJS which consists of three main components:

1. AmbientJS's core which provides distributed programming abstractions on top of ambient-oriented programming model,
2. the platform bridge which is in charge of loading the right mobile cross-platform framework, either Titanium or Cordova and dispatches distributed communication from the core to the corresponding networking layer,
3. the AmbientJS networking layer built in JavaScript which provides service discovery and communication services to the core. This layer is implemented on top of the underlying networking libraries for each mobile cross-platform frameworks (namely, Bonjour and NSD for Titanium, and zeroconf and Chrome sockets for Cordova).

AmbientJS's core in turn consists of three cornerstones components:

1. An actor model translating the principles of ambient-oriented programming to rich mobile applications (explained in section 3.2).

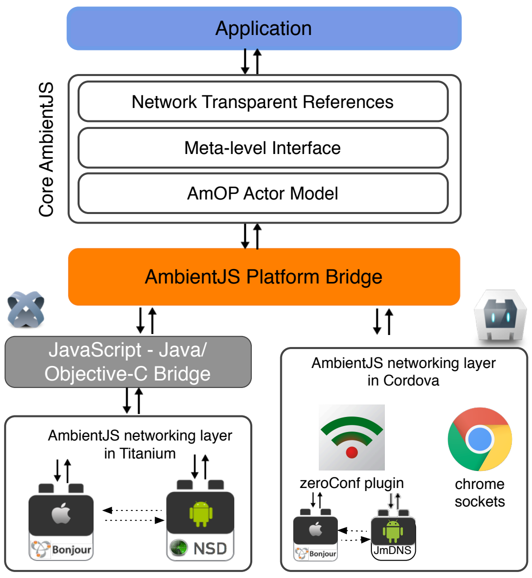
Application
Core AmbientJS
Network Transparent References
Meta-level Interface
AmOP Actor Model
AmbientJS Platform Bridge
JavaScript - Java/ Objective-C Bridge
AmbientJS networking layer in Titanium
Bonjour
NSD
AmbientJS networking layer in Cordova
zeroConf plugin
chrome sockets
Bonjour
JmDNS

Fig. 3. Architectural Overview of AmbientJS.

2. A novel object referencing abstraction called *Network Transparent References* (NTRs) which abstracts away the complexities of the basic networking facilities. NTRs are the main abstraction to help developers deal with the different distributed application architectures as we explain section 3.3.
3. A metalevel interface on which NTRs have been built. The meta-level interface is the key feature in order to allow for extensibility of NTRs as we explain in section 3.5.

Before delving into each of the main components of AmbientJS' core, let us briefly explain the development process of AmbientJS application for each platform. Titanium applications consists of a number of JavaScript modules in which GUI and device specific APIs are offered by the platform as JavaScript modules. AmbientJS applications are just regular Titanium applications which import the AmbientJS library in their project as shown below.

```
var AmbientJS = require('js/AmbientJS/AmbientJS');
AmbientJS.online();
// ... use AmbientJS ...
```

AmbientJS is packaged as a third party Titanium plugin which, once loaded, it can be accessed as any default APIs supported by Titanium (e.g. for GUI construction and accessing phone APIs). In the code snippet, the `online` function is called to connect the library to the network and start exporting and discovering service objects by means of the underlying networking facilities. AmbientJS then relies on the JavaScript - Java/ObjectiveC bridge from Titanium (marked in grey in Figure 3) to transform such JavaScript library code into native applications in the targeted mobile platform. Note that the same kind of code is required to load AmbientJS on top of `node.js`.

In contrast, when using AmbientJS in Cordova, programmers are required to use an asynchronous programming style since the platform only execute code upon all required libraries are ready (i.e. when the `deviceready` is emitted). As such, when employing AmbientJS in Cordova, programmers need to first register to the `AmbientJSready` event as follows:

```
var AmbientJS = require('./AmbientJS/AmbientJS');
AmbientJS.events.addListener('AmbientJSready', function() {
// ... use AmbientJS ...
});
```

The `AmbientJSready` event is emitted once Cordova notifies that AmbientJS is available in the platform. Cordova's development environment is also different than Titanium since it expects a HTML5 file for the UI code and a JavaScript file for the application logic. In AmbientJS, application logic is divided in a number of JavaScript files. It then relies on the Browserify library [3] to create a packaged file used as input to Cordova.

3.2 Ambient-oriented Actor Model

As previously mentioned, AmbientJS incorporates the Ambient-Oriented Programming (AmOP) paradigm at the heart of its programming model. The model advocates a non-blocking communication model to ensure autonomy of devices in face of partial failures so frequent in a mobile environment. Similar to the AmbientTalk language, the unit of concurrency and distribution is an actor represented by an event loop which encapsulates one or more objects. As such, two objects are said to be *remote* when they are owned by different actors. All distributed communication is enqueued in the message queue of the owner of the object and processed by the owner itself. To this end, AmbientJS reuses the already existing event loop concurrency from JavaScript. As such, AmbientJS assumes one event loop per device and does not provide dedicated support to spawn new event loops in the library.

Communication between remote objects happens by means of the so-called *far references*. Far references are a special kind of remote object references which can be only used by sending asynchronous messages. Figure 4 shows the conceptual representation of a far reference with a dashed line. A far reference is reified into two meta-objects encapsulating all aspects of interactions between senders and receivers, called a *transmitter-receptor* pair. Any message sent via a far reference to an object is first enqueued by the far reference itself (by its transmitter) which then delivers it to the receiver actor using the underlying communication channel (depicted with a double line). When the message is received at the recipient actor, the receptor will first receive the message on its mailbox, and further invoke a method on the remote object when appropriate.

[3] http://browserify.org

By manipulating these two metaobjects, developers can handle remote interactions between two objects in a *modular way* as they encapsulate the whole distributed behaviour of a remote object and the references that are handed to client objects. We use them to implement the programming abstractions to support multiple application architectures in an object-oriented programming style. We will further explain their API in section 3.5.

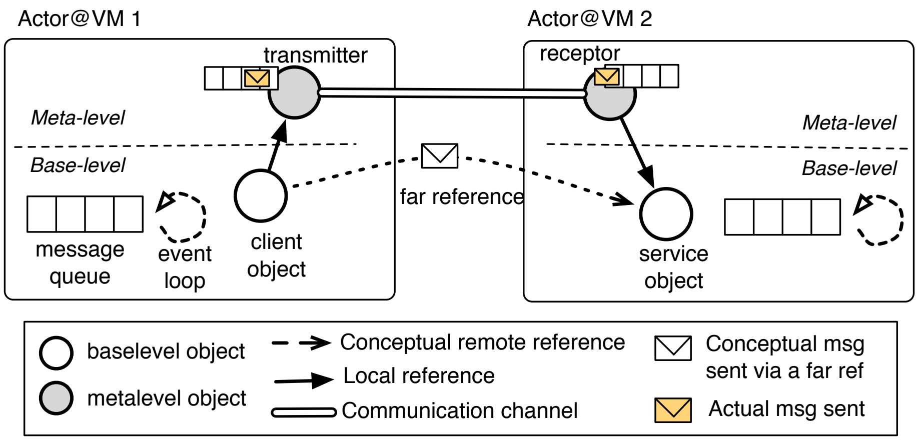

Fig. 4. Event loop concurrency in AmbientJS

In AmbientJS, objects are by default passed by far reference to objects owned by other actors. Objects can also be sent to a different AmbientJS VM by (deep) copy, which allows the recipient actor to operate on the copy by means of regular synchronous communication. Like in AmbientTalk, we refer to those objects as *isolates*.

3.3 Network Transparent References (NTRs)

As previously mentioned, the only type of communication allowed on far references is asynchronous message passing. AmbientJS extends the concept of a far reference with support for implementing different application architectures which in turn can use multiple networking technologies. This allows AmbientJS actors to communicate with one another over wireless links or mobile broadband access. As such, remote references in AmbientJS abstract the underlying networking technology being used for communication. Such *network transparent references* (NTRs) are resilient to network fluctuations by default.

Basic Network Transparent References. In order to illustrate the workings of the NTR model consider figure 5. Figure 5(i) shows the far reference model, in which a conceptual remote reference (dashed arrow) between objects is actually implemented by two meta-level objects. Those meta-level objects acts as proxies at each side of the reference, and enqueue messages sent from client to a service object over a communication channel. Those communication channels are built on top of one networking facilities, eg. Wi-Fi. In such a model, if the application can communicate over different communication channels, programmers needs to do manual bookkeeping for every far references being created over each network technology.

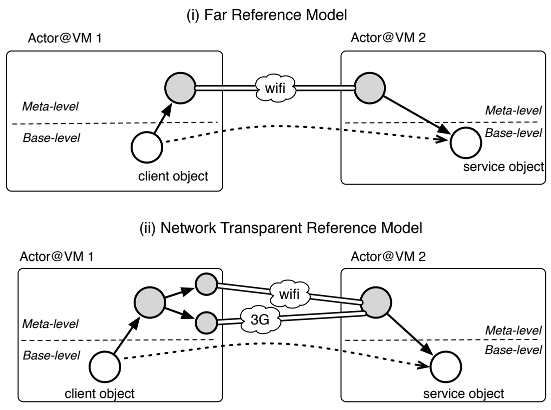

Fig. 5. Far References vs. Network Transparent References

Figure 5(ii) illustrates the network transparent references model. Instead of having a network reference per network technology there is conceptually only one reference. The conceptual far reference abstracts away the network technology for the programmer. Note that the network transparent reference is a distributed referencing abstraction which is active at both the sender and receiver. Part of the difficulty of having multiple references to the same object is to make sure that messages are not duplicated or lost. Therefore there is a part of the network transparent reference which is active at the receiver side in order to filter possible duplicate messages.

A NTR appears as a far reference which enqueues and transmits asynchronous messages sent to the remote object. Unlike normal far references when a network technology (e.g. bluetooth) is not available, the NTR attempts to transmit messages sent to it using another networking technology, e.g. 3G. If all network 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 NTR flushes 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. In order to distinguish partial failures from permanent failures the programmer can make use of leasing. The inner workings of this leasing mechanism are however, outside of the scope of this paper.

Network Transparent References over Multiple Network Architectures. When direct communication between two entities is not possible the traditional way of establishing a communication link with each other is by making use of known centralised server. The *network architecture* of such a distributed application is quite different from the network architecture used for a peer-to-peer application. As shown in the Coupon Go scenario both network architectures are useful during the life time of the application. The situation where communication is possible both over a centralised server and through Wi-Fi direct is sketched in figure 6.

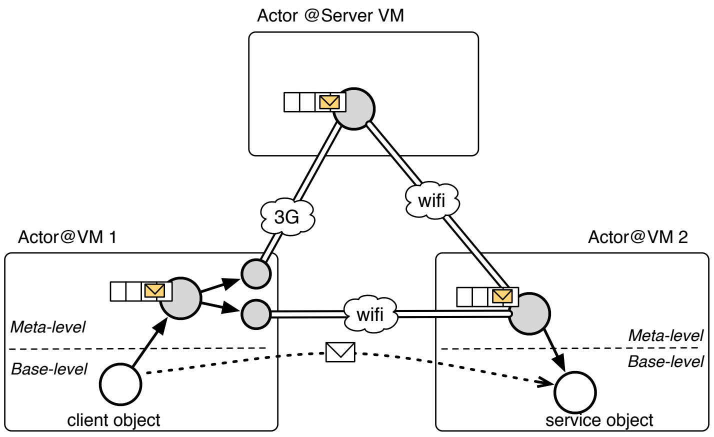

Fig. 6. Indirect Network Transparent References

As shown in this figure when adding a centralised sever there is a step increase in the possible ways packages can travel over the different communication channels. Important to note is that even in this complex architecture conceptually there is still only one reference between the distributed objects.

Unfortunately with existing distribution libraries the programmer needs to completely rewrite his application in order to account for the different ways that communication can be established. Because the server acts as a broker between the different communication partners it becomes even more difficult for the programmer to know exactly from which sender each message comes. For each exported object there might be multiple incoming links from the server to the object. Moreover, for each sender there might be multiple ways from the sender to the sever. The programmer thus manually needs to keep track of which messages are received in order to avoid executing the same message multiple times for each communication path. With NTR the programmer does not need to be concerned about the details of routing the messages through the centralised server and can think in terms of the conceptual reference.

3.4 NTRs in Action

To illustrate NTRs and the different distributed programming constructs in AmbientJS consider the following code snippet from a rich mobile chat application.

Listing 1: Example use of asynchronous message passing over NTRs

In this application the programmer keeps a list of buddies (`buddylist`) of all the available communication partners (line 1). After initialising the library the programmer registers a callback function to be notified about the discovery of nearby communication partners (line 5). 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. This means that while the programmer conceptually receives one reference there might be multiple underlying network technologies to the same remote object.

Once the programmer has obtained the network transparent reference he proceeds by sending a message `getName` over the NTR (line 7-8). The result of sending this asynchronous message is a future on which the programmer registers a callback to receive the reply (line 8-9). When the remote object returns the result of processing the `getName` message the callback function is applied and the return value is bound to the variable `reply`. The programmer then simply stores the network transparent reference into the buddy list and uses the name of the remote buddy as a key (line 10). Note that in a real application care needs to be taken to make these names unique. This concludes the discovery part of the chat application.

We now show how programmers can export a local object so that other mobile phones can discover it. In order to export objects to the network, the `exportAs` function is employed. The code snippet below shows how to create an object which implements a service corresponding to the chat application. Then we export this object with `MESSENGER` string as service type so that other devices in the neighbourhood can discover it (line 5).

Extending the Peer-to-Peer Chat Application to use a Centralised Server. So far our chat application exhibits a peer to peer architecture. 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, AmbientJS 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 can also be exported via an intermediary server in the cloud. The entire code to configure the server is shown below.

The only change needed at the client side of the application is to add one line in the configuration object of AmbientJS. For example the following configuration file is sufficient for the clients to be able to connect to the software languages lab server to discover each other. Because the application was written with network transparent references there are no further changes necessary to the application.

3.5 Meta-level Interface

AmbientJS features a meta-level interface inspired by the transmitter-receptor meta model [10] which reifies the most important aspects of distributed communication amongst remote objects. As illustrated in figure 4, a remote reference is represented at the meta level as a pair of metaobjects, named *transmitter* and *receptor*, representing the source and the target of a far reference, respectively. Transmitter and receptors provide a meta-object protocol (MOP) [15] that allows developers to modify message sending semantics via far references as well as how references are shared in the network. It has been used to implement NTRs and provide them as the default kind of communication mechanism in AmbientJS.

Each service object is bound to at least one receptor. The receptor intercepts each asynchronous message received by its associated service object in a transparent way. It can then perform actions before or after the service object sends or receives a message to handle aspects such as persistence, replication, security, etc. A transmitter, on the other hand, is transparently created on the client device when a receptor is unmarshalled, and is used to transmit asynchronous messages to the service object (via the receptor). A transmitter can perform some actions before or after a message send to handle communication aspects such as providing one-to-many communication, applying delivery guarantees, logging successful message sends, etc. In addition, a transmitter exposes the network connectivity of the physical communication with the device hosting the service object.

We now explain the core API exposed by transmitters and receptors and explain their relevance to implement NTRs.

`onReceive(message)` This hook allows changing the default behaviour of the reference when a message is sent to the reference. The default behaviour at transmitter side is to remove a letter from the far reference's mailbox containing the message and receiver, and transmits it. At receptor side, it makes the service object accept the delivery of an asynchronous message. We employ this hook to ensure message order and eliminate duplicates at the receptor side of NTRs.

`onPassReference(reference)` This hook reifies the act of marshalling objects when they are passed as argument of a message sent to another actor, or passed via the service discovery mechanism. It returns either the receptor or the transmitter to be marshalled instead of this receptor or transmitter. This hook is employed in NTRs to be able to implement indirect access via a server.

Programmers can employ such MOP to extend NTRs to support other application architectures, or to build other families of referencing abstractions. In the context of the Coupon Go, we employed such a meta-level interface to implement different consistency policies for objects distributed between virtual wallets. For example, we built a *single use* object to model transferable coupons which can only be redeemed once, even if they are transferred amongst the wallets of the members of a family. Listing 1.1 shows a simplification of the implementation of the reference abstraction allowing for single use objects.

```
function singleUseConsistencyRef(){
  var receptor = AmbientJS.createReferenceProxy(function(delegate){
    var blocked = false;
    this.onReceive = function(msg){
       if (!this.blocked) {
        delegate.onReceive(msg);
        if (msg == "redeem") {
          this.blocked = true
        }
        return true;
      } else { return false;}
    }
  /// rest code
  }
}
```

Listing 1.1. Creating a custom far reference abstraction for a single use consistency object.

`createReferenceProxy` function creates a receptor for a delegate object (which is passed by parameter). The receptor then mediates all distributed message passing and serialization of the delegate object. In this case, the receptor keeps a boolean to limit the access to the delegate object after receiving the message `redeem`. To this end, it overrides the `onReceive` function to block the object upon reception of the first redeem message. Listing below shows how to use the newly created receptor for creating a twix coupon object. The code employs extended version of the `createObject` function in which we can pass a custom receptor which will control the reception of messages sent to the `twixCoupon` (which acts as the delegate object from the point of view of the receptor).

```
var twixCoupon = AmbientJS.createObject({
  "getDescription" : function() {...},
  "getStockCount" : function() {...},
  "redeem" : function(amount) {...}
}, receptor);
```

4 Developing Applications with AmbientJS

In this section, we describe the implementation of a rich mobile application developed with AmbientJS. This application is a mobile variant of the well known arcade game called pong. In our explanation we stress the distributed aspects of the application and omit the details of the application logic.

4.1 wePong Application

Similar to pong, wePong allows two players to defeat each other by playing table tennis. The player controls a paddle to hit a ball back and forth in a game field. The game differentiates from the original game in two fundamental aspects: (1) players control the paddle by moving the mobile device, and (2) the table field is distributed amongst players. Each player's mobile device represents the game field for one player, which can interact with the game field of another player nearby (by means of ad-hoc networking technology) or a remote player (over the Internet).

Figure 7 shows the game field of a player which initially contains a paddle, a portal and one ball. A portal is a special object located somewhere on the game field (depicted by a green circle in the figure). When a ball traverses the portal, the ball is sent to the opponent. To make the game more competitive, each player has a power-up bar which contains the following types of items:

– extra ball power-up, which adds an extra ball to the game field of the player.
– multi-ball power-up, which adds three to five new balls to the game.
– long pallet power-up, which enlarges pallet for a duration of 20 seconds.
– score power-up, which appears for few seconds on the field game and allows the player to earn extra 100 points when it is hitted.

Power-ups appear randomly in the bar of each player. The figure shows the power-up bar on the left hand side of the field with unused 3 power-ups, and an active score power-up on the game field. In order to use power-ups in the game,

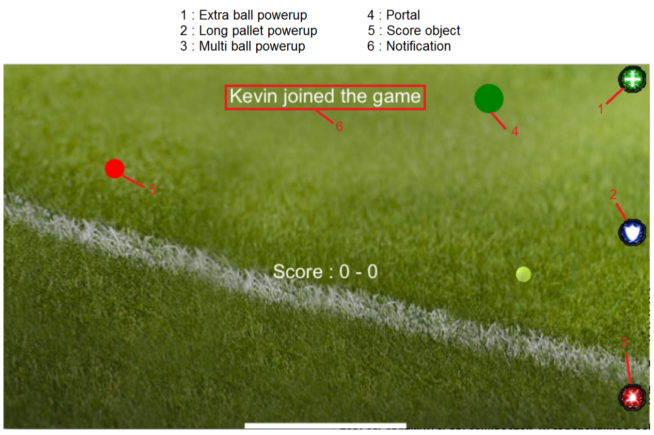

Fig. 7. Screenshot of the wePong game

a player can either tap it or drag it to a portal. When a power-up is tapped, the effect is applied to the player's own game field. A power-up can also be dragged to the portal which will apply it to the opponent.

4.2 Implementation

The first step in the implementation of wePong, is to create and export a game object to representing a playing session in the network. When the user clicks on the `newRoomButton` button, the application creates a game session and waits for an opponent to join it. Listing 1.2 shows the code exporting the game object to other wePong applications instances. The game object is stored in the `remoteInterface` variable at line 3 and contains four methods that can be called by remote wePong applications: (1) `getGameName` returns the name of the game session being exported so that a list of available games is shown to the user, (2)`joiningGame` is called by an opponent who wants to join the game session, (3) `scoreChange` is used to receive the score updates from an opponent, and (4) `receiveGameElement` is used to receive power-ups or balls sent by the opponent. As shown in line 11 power-ups are model by `gameElement` objects which need to implement the `doAction` function.

Listing 1.3 shows the discovery of a wePong game instance. Upon discovery, the application requests the name of the game session being exported by a player by sending the `getGameName` message (line 2). Once the future for that asynchronous message is resolved, the `addRoomToTable` method adds a new name to the list of available game sessions (line 5), and the `joinGame` is called if the user selects that game session by clicking on the name, and will notify the opponent of the wish to join the game session.

The application contains a gameloop which periodically updates the position of the balls and the pallet. The gameloop also detects collisions between a ball

```
function startGame(roomName) {
  currentGame = new game(AmbientJS, roomName, instrumenting);
  var remoteInterface = AmbientJS.createObject(
    "getGameName":
      function () { return roomName; },
    "joiningGame" :
      function (nickname) { currentGame.playerJoined(nickname); }
    "scoreChange" :
      function (score) { currentGame.receiveOpponentScore(score); },
    "receiveGameElement" :
      function(gameElement) { gameElement.doAction(currentGame)},
  });
  AmbientJS.exportAs(remoteInterface, "WePong_Game");
  currentGame.start(true);
}
```

Listing 1.2. Exporting a wePong game session on the network

```
AmbientJS.wheneverDiscovered("WePong_Game", function(reference) {
  var getNameMsg = AmbientJS.createMessage("getGameName", []);{}
  var future     = reference.asyncSend(getNameMsg, "twoway");
  future.whenBecomes(function(name) {
    var row = addRoomToTable(name);
    row.addEventListener('click', function(e) {
      joinGame(name, reference);
    });
  });
});
```

Listing 1.3. Discovering a wePong game session on the network

and some game object. When a collision happens between a ball and the portal, the ball is sent to the opponent by calling the `sendBall` function.

Listing 1.4 shows the `sendBall` function. Each applications keeps an array of balls being displayed on the game field. Before sending the ball to the opponent, it is first remove from the player's game field (in lines 2 - 3). Line 4 creates an AmbientJSobject which is a copy of the ball to be passed to the opponent. In this case the `createObjectTaggedAs` function is used to send the ball by copy. The `teleportedBall` implements the required `doAction` method which will basically add the ball to the opponent's game field upon arrival. Lines 10 and 12 create the `receiveGameElement` message which carries the `teleportedBall` and sends it as one way message (i.e. we do not request a future for the result of the function). The similar implementation strategy is followed to send power-ups to the opponents when they are dragged into a portal.

```
 1  function sendBall(ball) {
 2    var i = balls.indexOf(ball);
 3    balls.splice(i, 1);
 4    var teleportedBall = AmbientJS.createObjectTaggedAs({
 5      var copyBall = [ball.x, ball.y, ball.vx, ball.vy];
 6      "doAction" : function (opponentsGame) {
 7        opponentsGame.receiveBall(copyBall);
 8      }
 9    }, [Isolate]);
10    var ballMsg = AmbientJS.createMessage(
11      "receiveGameElement", [teleportedBall]);
12    opponent.reference.asyncSend(ballMsg, "oneway");
13    delete ball;
14  }
```

Listing 1.4. Sending a ball to an opponent through a portal

This concludes the relevant parts of the distribution aspects of wePong. We have actually built two variants of wePong application, one to be deployed on mobile devices natively with Titanium and one as a web application with Cordova. Both variants employ the same AmbientJS code for implementing the game functionality and distribution aspects. Only native functionality like acceleration data, screen touches and user interface elements are different.

5 Performance Evaluation

Recall that AmbientJS is implemented on top of two existing cross-platform technology: Cordova and Titanium. Cordova is a hybrid technology where the application is written in a mixture between HTML5 and Javascript while Titanium is an so called interpreted approach where all the UI elements are native components and the application logic is written in Javascript. In order to showcase the usability of AmbientJS we performed benchmarks with respect to CPU-load, memory consecution and network throughput for both implementations.

Each benchmark is conducted by monitoring the execution performance of the wePong game running on an iPhone 4S and an iPhone 6S. Note that the implementation of the game does not rely on the GPU. Therefore, all calculations of the trajectory of the balls, the collision detection and the network connections with other players need to be handled by the CPU.

In order to ensure that the game is executing exactly the same code on both platforms, the user interaction is completely automated. The automated script goes through a typical scenario where two users connect with each other and play the wePong game. The script fully automates all the interactions including the device orientation and user touches such as tapping and dragging. Every source of randomness was also removed from the game i.e. we ran it with the same seed.

All performance experiments were measured with the Xcode performance tools[4]. For each run of the benchmark, the **CPU**, **memory usage** and **network throughput** is measured. A first observation is that *both versions of AmbientJS are able to run the wePong game smoothly at 80 frames per second.* There are, however, small differences between the interpreted version and the hybrid approach. We give an overview of these difference in the following sections.

5.1 CPU Usage

The CPU usage is determined by measuring the CPU load at discrete time intervals during the execution of the application. Each execution of the wePong game lasts a bit more than a minute. During that time the Xcode Activity Monitor is able to make 38 measurements of the CPU-load. We take the average CPU

[4] https://tinyurl.com/instrumentsUserGuide

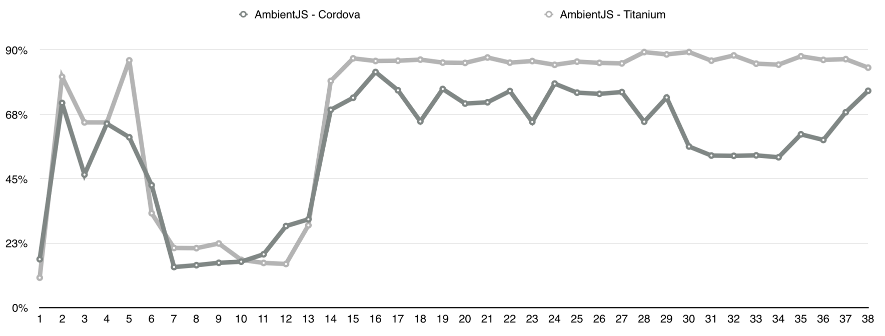

Fig. 8. Average CPU-usage over time

load at each time interval for multiple runs of the application. Figure 8 shows the average CPU-loads for both the Cordova and Titanium implementation.

From this graph it is clear that the CPU-load for both Cordova and Titanium has a similar shape. At the start of the application there is a heavy startup-cost after which the CPU-load becomes less intensive. When the connection between the phones is established and the wePong game is running, the CPU load peeks again. The graph clearly shows that the CPU load for wePong is less for Cordova than for the Titanium implementation. The reason for this difference probably lies in the fact that the Cordova version runs inside an HTML5 canvas, while in Titanium it uses native UI elements. Hence, Titanium needs glue code to communicate between the JavaScript interpreter and native components which requires more CPU-load than its browser-based counterpart.

5.2 Memory Usage

To monitor memory usage throughout a wePong game, we used the Allocations instrument included in Xcode Instruments. This instrument reports every memory allocation together with its size and timestamp. We processed the output of this tool and imported them into SPSS. We then further processed these files in order to measure the overall memory usage. Figure 9 shows the memory consumption of the wePong application in three different runs of the benchmarked scenario.

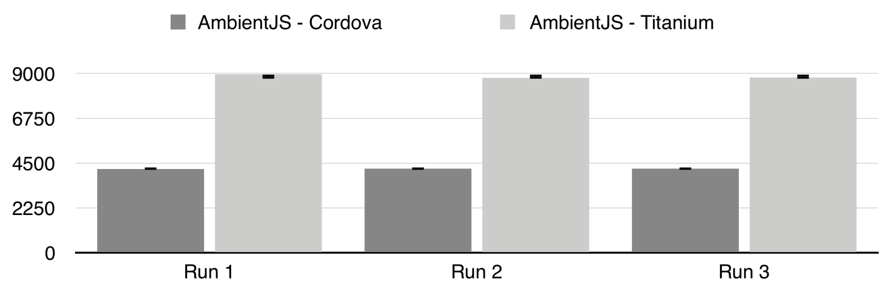

Fig. 9. Memory Consumption of the wePong Application

Based on figure 9, we conclude that the Titanium implementation of wePong is using significantly more memory than the Cordova one. This is because Titanium uses native UI elements to provide a better looking UI and bundles a self contained JavaScript interpreter, while Cordova does not use any extra styling to make the user interface look better and relies on the built in JavaScript interpreter. Hence, as Titanium provides a full SDK the memory usage is significantly higher than with plain HTML5 and CSS.

5.3 Network Throughput

The last performance benchmark we conducted measures the network throughput during the lifetime of the wePong application. Again the network throughput is measured by taking discrete slices of the application and measure the throughput at that time slice. Note that the data that is being sent is exactly the same for each of the implementations. The result of measuring the throughput is shown in figure 10.

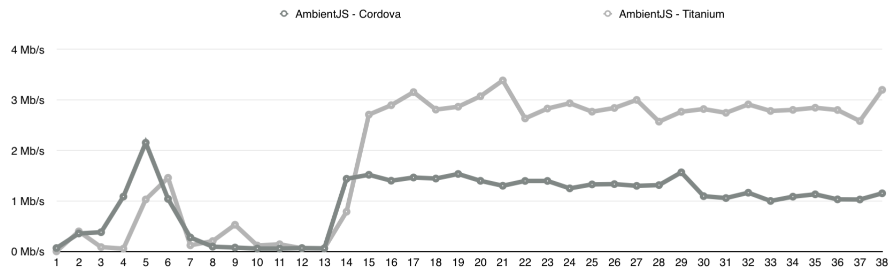

Fig. 10. Network throughput

From this benchmarks we can conclude that our network layer implemented on top of Titanium performs significantly better with respect to throughput. The most probable reason for this difference in throughput is that for the implementation in Titanium we directly access the native libraries of the the phone. In the case of Cordova we make use of the web sockets library which seems to perform a bit worse than the native libraries. Note that in practice both versions of AmbientJS run the application smoothly and there is no noticeable network lag. But this performance impact may be noticed by applications transferring large amount of data like like sharing or video streaming applications.

5.4 Benchmark Conclusions

Our benchmarks unveiled that making use of the native UI components is not necessarily more performant. In the wePong application there are many calls from the Javascript to the UI component to draw on the screen. In the Titanium implementation all these calls need to be translated to calls to the UI component which is less efficient than directly using a HTML5 webview. We also noticed that making use of Titanium implies a significant memory overhead compared to the Cordova approach. Finally, we observed that employing native components as done by Titanium does pay off in terms of network throughput which is significantly higher than in the Cordova implementation.

When one has to choose a suited mobile cross-platform approach for his application, various factors need to be considered. Important factors are performance as well as user experience. Because in Titanium the GUI components are native the user experience is much better.

6 Related Work

In what follows we describe the different advances in the field of mobile and cloud computing with respect to the design space of rich mobile applications. We give an overview of the related work in three dimensions: the network technology, distributed application architectures and the software platforms.

Network technology. Mobile rich applications employ a wide variety of connection protocols, being the most relevant ones Wi-Fi, Bluethooth, and 3G[8]. Wi-Fi was intended as replacement for cabling amongst computers for wireless local area networks (WLANs). Wi-Fi's communication range is within 100 meters and supports up to 11Mbps data rates. Bluetooth, on the other hand, was intended for wireless personal area networks (WPAN) uses and it is characterized by low-power shorter communication range (up to 10 meters) requirements. Bluetooth Low Energy (BLE) has reduced power consumption while keeping a similar range. Finally, 3G provide broadband access to mobile devices of several Mbps, which is slower than Wi-Fi but can be deployed over wider areas.

Since energy consumption varies amongst those technologies (e.g 3G has been shown to be higher than Wi-Fi), many research in mobile cloud computing has focused on offloading workload from mobile devices depending on the connection protocols used [8]. Satyanarayanan et al. proposed the concept of a *cloudlet*[24] in which part of the computation of the cloud is offloaded to several multi-core computers which has access to the cloud (forming a cloudlet). This solution advocates VM technology which automatically offloads application workload from the mobile device (which acts as a thin client) to a nearby cloudlets situated on designated places like coffee shops or airports. Such VM migration technique is employed by many mobile computing frameworks since it requires no or very little application rewriting[8]. Common to our approach their VM migration techniques can be deployed over several network interfaces. Their approach is to hide the complexity in the VM while we offer it at the middleware layer. Moreover, they do not provide any cross-platform facilities.

Distributed Application Architectures. An application architecture basically materializes the abstraction of a communication link used to represent the interactions between entities of a distributed system[3]. In traditional distributed applications, interactions are often established between two entities or process. The most representative case of this type of interactions are client-server applications, supported by point-to-point communication abstractions. One of the most recurrent communication abstractions for building distributed client-server applications has been Remote Procedure Call(RPC). RPC-based solutions include distributed middleware like RMI[25], or RBI[13], but it has been also employed in the context of web programming in service-oriented architectures like SOAP[29] and Apache Thrift developed by Facebook[5]. Nevertheless, RPC has been repeatedly highlighted to be unappropriated for distributed programming [27, 28]. Several extensions were proposed to overcome the so-criticized synchronous request-response messaging promoted by RPC like queueing RPCs [14]. In the context of rich mobile applications, one of the main critique on RPC is the lack of support for disconnected operations[8].

Peer-to-peer architectures, one the other hand, considers that processes can both provide and request services. Those architectures are often supported by group communication abstractions. One of the representative approach of this kind of interactions is the publish/subscribe communication paradigm[6] which allows processes to interact by publishing event notification (often called *events*) and subscribing to the type of events they are interested in. The first publish/subscribe systems assumed that components comprising an application are stationary and interact by means of a fixed, reliable network of event brokers. Adaptations for mobile computing does not rely on intermediate infrastructure, but rely on broadcasting of subscriptions to reachable hosts (e.g. EMMA [20]), or to a certain geographical area closeby the producer, e.g. STEAM [17]. Such group communication abstractions have recently penetrated mainstream SDKs like allJoyn2, Qeo library3, Intel CCF SDK4.

Another group of models and languages in mobile computing are based on the concept of *coordination* in which interactions are established between two or more process by means of a shared tuple space[9] by reading and writing tuples. Mobile middleware such as LIME [19] and TOTA [16] are the most representative examples based on distributed peer-to-peer variants of the original, shared memory tuple space model. Mobile tuple space systems that have adopted a replication model [16, 18, 11] allow for multi-hop architectures in which devices in the network can be used as routers of messages.

Although variations of RPC, publish/subscribe and tuple spaces are plentiful, the integration of those interactions has received very attention. Eugster et al.[7] proposed support for publish/subscribe programming into an object-oriented language. However, that work does not integrate the paradigms, rather allows programmers to use both event notifications and message passing and objects. To the best of our knowledge, reconciling the communication properties of point-to-point and group communication models has only been studied in the context of distributed programming language AmbientTalk [4]. Van Cutsem et al. [26] explored a novel remote object reference abstraction called *ambient reference* which unifies point-to-point with group communication for mobile ad hoc networking applications. The main novelty of *ambient references* lies in the declarative designation of a group of communication partners. Ambient references, however, do not provide any means to combine multiple network technologies and present it as a single reference abstraciton to the programmer. In the context of mobile RFID-enabled applications, *multiway references* [21, 22] were introduced as a reference abstraction that allows to address RFID-enabled

[5] http://thrift.apache.org/

objects through different networking technologies. This paper builds on this work and applies it in the context of rich mobile applications.

Software Platforms. From the platform perspective, developing rich mobile applications entails developing one mobile application for each platforms. However, this is costly and requires deep knowledge of several operating systems and programming languages, and imposes redudancy at the whole software development cycle from design to testing[1]. In oder to reduce that cost, mobile cross-platform frameworks have emerged as a solution to alleviate the issue [30]. They advocate the use of a single code base which can be deployed on multiple mobile plaftorms.

There are two different kinds of mobile cross-platform solutions: hybrid approaches and interpreted approaches (also called "self-contained runtime environments" [12]). The most prominent exponent of the hybrid approach is PhoneGap (currently developed in Apache Cordova) which combines web technology and native functionality. More precisely, hybrid mobile applications combine HTML5 web applications inside a native container (UIWebView in iOS and WebView in Android). On the other hand, the most relevant interpreted technology is Appcelerator Titanium mobile which generates native code for the UI and application logic implemented using JavaScript. Interpreted applications are said to be more efficient and provide better user experience than hybrid approaches because of the native user interfaces. However, they are mainly criticized by the complete dependence on the software development environment as the UI is implemented completely programmatically using the provided APIs.

What both families of mobile cross-platform frameworks have in common is that they provide a number of built-in APIs for GUI construction and accessing the underlying hardware without requiring detailed knowledge of the targeted platform. However, developers still need to deal with many of the difficulties for distributed programming, as they only provide low-level libraries directly on top of networking protocols for communication (e.g. HTTP request and TCP/IP sockets in Titanium, WebRTC or Chrome sockets in the case of Cordova).

While most popular mobile cross-platform solutions are built for JavaScript, alternatives exists employing other languages like C# in Xamarin[6], or C/C++ in Marmalade SDK[7]. Employing JavaScript has the potential to allow developers to write both mobile client and back-ends on the same programming language (e.g by Cordova and `node.js` at client and server side respectively). Note, however, there is no distributed object model that can be used on both mobile clients and back-end JavaScript code running on `node.js`. Recent trends include the use of reactive programming libraries[2] for JavaScript for communication between client and server. However, distributed reactive programming is its infancy[23], and developers need to manually program the interactions, ensuring consistency and offline functionality.

[6] https://www.xamarin.com/

[7] https://marmaladegamestudio.com/tech/

7 Conclusion

In this paper, we argued that due to mobile computing advances, programmers today are faced with a ninth fallacy of distributed computing: "*there is only one fixed application architecture throughout the lifetime of the application*". When programmers do not take into account that their mobile applications can operate both in a peer-to-peer fashion as well as in a client-server architecture adding support for both kinds of application architecture usually requires a rewrite of the networking layer. We have shown that such problems do not arise when making use our mobile cross-platform actor library called AmbientJS. The main innovation of AmbientJSis the embodiment of a special kind of extensible remote reference, called network transparent references (NTRs), which abstracts away from the underlying network technology used. We have given an overview of the NTR model, detailed their implementation in AmbientJS and assessed the performance of AmbientJS with benchmarks.

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.
2. E. Bainomugisha, A. L. Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 45(4):52:1–52:34, Aug. 2013.
3. C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.
4. 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.
5. P. Deutch. The eight fallacies of distributed computing, 1994. https://blogs.oracle.com/jag/resource/Fallacies.html (captured February 2017).
6. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.Kermarrec. The many faces of publish/subscribe. *ACM Computing Survey*, 35(2):114–131, 2003.
7. P. T. Eugster, R. Guerraoui, and C. H. Damm. On objects and events. *SIGPLAN Not.*, 36(11):254–269, Oct. 2001.
8. N. Fernando, S. W. Loke, and W. Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1):84 – 106, 2013. Including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures.
9. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan 1985.
10. E. Gonzalez Boix. *Handling Partial Failures in Mobile Ad hoc Network Applications: From Programming Language Design to Tool Support.* PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Software Languages Lab, October 2012.
11. E. Gonzalez Boix, C. Scholliers, W. De Meuter, and T. D'Hondt. Programming mobile context-aware applications with TOTAM. *Journal of Systems and Software, SCI Impact factor in 2013:1.135 (5-year impact factor 1.322)*, 92:3–19, June 2014.

12. H. Heitkötter, S. Hanschke, and T. A. Majchrzak. *Evaluating Cross-Platform Development Approaches for Mobile Applications*, pages 120–138. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
13. A. Ibrahim, Y. Jiao, E. Tilevich, and W. R. Cook. Remote batch invocation for compositional object services. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 595–617, Berlin, Heidelberg, 2009. Springer-Verlag.
14. A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, 1997.
15. G. Kiczales and A. Paepcke. Open implementations and metaobject protocols. Tutorial slides and notes, Software Design Area, Xerox Corporation, 1996. http://www.parc.xerox.com/csl/groups/sda/publications.
16. M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4):15:1–15:56, July 2009.
17. R. Meier and V. Cahill. Steam: Event-based middleware for wireless ad hoc networks. In *22nd International Conference on Distributed Computing Systems*, pages 639–644, Washington, DC, USA, 2002. IEEE Computer Society.
18. A. Murphy and G. Picco. Using lime to support replication for availability in mobile ad hoc networks. In *8th International Conference on Coordination Models and Languages (COORDINATION)*, LNCS, pages 194–211. Springer-Verlag, 2006.
19. A. Murphy, G. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 524–536. IEEE Computer Society, 2001.
20. M. Musolesi, C. Mascolo, and S. Hailes. Emma: Epidemic messaging middleware for ad hoc networks. *Personal Ubiquitous Comput.*, 10(1):28–36, 2005.
21. K. Pinte, D. Harnie, and T. D'Hondt. Enabling cross-technology mobile applications with network-aware references. In *Proceedings of the 13th International Conference on Coordination Models and Languages*, COORDINATION'11, pages 142–156, Berlin, Heidelberg, 2011. Springer-Verlag.
22. K. Pinte, D. Harnie, E. Gonzalez Boix, and W. De Meuter. Network-aware references for pervasive social applications. In *Second IEEE Workshop on Pervasive Collaboration and Social Networking (PERCOM workshops)*, pages 537–542, March 2011.
23. G. Salvaneschi, J. Drechsler, and M. Mezini. *Towards Distributed Reactive Programming*, pages 226–235. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
24. M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct. 2009.
25. Sun Microsystems. Java RMI specification, 1998. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.
26. T. Van Cutsem, J. Dedecker, and W. De Meuter. Object-oriented coordination in mobile ad hoc networks. In *Proceedings of the 9th International Conference on Coordination Models and Languages*, COORDINATION'07, pages 231–248, Berlin, Heidelberg, 2007. Springer-Verlag.
27. S. Vinoski. Rpc under fire. *IEEE Internet Computing*, 9(5):93–95, Sept 2005.
28. J. Waldo, G. Wyant, A. Wollrath, and S. C. Kendall. A note on distributed computing. In *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 49–64. Springer-Verlag, 1996.
29. World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.2 W3C Recommendation, 2007. https://www.w3.org/TR/soap12//.

30. 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.