Integrated distributed computing between the cloud and mobile devices

Ολοκληρωμένος κατανεμημένος υπολογισμός ανάμεσα στο νέφος και τις κινητές συσκευές

Department of Electrical and Computer Engineering

UNIVERSITY OF THESSALY

FOR GRADUATION OF BACHELOR OF SCIENCE

Diploma Thesis by

Eleftheriadis Alexandros

Supervised by:

Spyros Lalis, Associate Professor

Manolis Vavalis, Professor

Greece, Volos 2015

Acknowledgements

I would like to thank my supervisor Dr. Spyros Lalis about his confidence in my abilities as well as his patient guidance, and Dr. Manolis Vavalis for his constructive comments on my thesis.

I would also like to thank all my teachers and colleagues for their valuable knowledge and experiences they have given me during my studies.

Finally, I thank all the members of my family for their unconditional support over the years, my friends, and especially Apostolis, my roommate for five years, for the beautiful moments we spent together as students.

Περίληψη

Σκοπός αυτής της εργασίας είναι η υποστήριξη κατανεμημένων εφαρμογών μεταξύ του υπολογιστικού νέφους και πολλών κινητών συσκευών. Αυτό επιτυγχάνεται μέσα από ένα μεσοστρωματικό λογισμικό που εκτελείται τόσο στο νέφος όσο και πάνω στην κινητή συσκευή, και παρέχει ένα ενιαίο περιβάλλον εκτέλεσης καθώς και υπηρεσίες επικοινωνίας υψηλού επιπέδου για την αποστολή και παραλαβή/χειρισμό ασύγχρονων γεγονότων της εφαρμογής, αντιμετωπίζοντας με διαφανή τρόπο την διακοπτόμενη επικοινωνία και αποσυνδεδεμένη λειτουργία σε πιο χαμηλό επίπεδο, λόγω μετακίνησης. Η υλοποίηση βασίζεται στις τεχνολογίες Node.js και WebSockets, ενώ η εφαρμογή γράφεται σε Javascipt χρησιμοποιώντας μια απλή διασύνδεση προγραμματισμού.

Λέξεις κλειδιά

Κατανεμημένος υπολογισμός; Υπολογιστικό Νέφος; Node.js; WebSockets; Socket.io; Apache Cordova; Javascript; Android

Abstract

The objective of this work is to support distributed applications between the Cloud and several mobile devices. This is achieved through middleware that runs on both the Cloud and the mobile device, and which provides a uniform runtime environment as well as high-level communication services for sending and receiving/handling asynchronous application-level events, dealing in a transparent way with the intermittent connectivity and disconnected operation at the lower level, due to mobility. The implementation is based on Node.js and WebSockets technology, while the application is written in Javascript using a simple API.

Keywords

Distributed computing; Cloud; Node.js; WebSockets; Socket.io; Apache Cordova; Javascript; Android

Contents

List of figures

List of table

Chapter 1 Introduction

Due to their ubiquitous presence, their powerful computing, sensing and wireless communication capabilities, smartphones enable new distributed computing scenarios, where they can play the role of mobile sense-and-compute nodes. For instance, smartphones can be used to perform trivially parallel computations, or to monitor different environmental conditions such as air pollution.

At the same time, the cloud is gradually becoming the next desktop computer, offering unprecedented flexibility and capacity as a function of the required processing and connectivity requirements of the application. Moreover, the cloud allows one to collect, maintain and update huge amounts of data, without being limited by the storage constraints of privately owned computing infrastructure.

These two trends, pave the way to a new class/type of distributed computing applications, where some parts of the application run on the cloud and some other on smartphones. In fact, the number of smartphones can be very large. As an example, a crowdsensing application could be composed of one part that is deployed on hundreds or thousands of smartphones to collect data based on their onboard sensors as well as input that is provided by user, and another part which runs in the cloud and collects this data for further processing and visualization.

However, even though it is already possible to develop such applications, this development is still far from trivial, as one has to deal with a variety of issues. Firstly, smartphones feature completely different operating systems, such as Apple iOS or Google Android, come with quite different programming languages and environments, e.g., Objective-C vs. Java. Secondly, the system-level APIs of these platforms are again different than the ones offered by the more conventional operating systems used for running cloud-based applications. Thirdly, the connectivity between the smartphones and the cloud is typically sporadic and intermittent due to the mobility of the smartphone, but also provider-specific policies on connection management. Finally, smartphones have energy constraints since they are running on batteries, which requires special care in terms of the amount of processing and communication that is to be performed. As a result, very good know-how on completely different programming environments together with a considerable amount of effort is required in order to write a distributed application that can span across all these platforms.

The objective of this thesis is to make a first step in simplifying the development of such applications. This is achieved through middleware that runs on both the Cloud and the mobile device, and which provides a uniform and cross-platform runtime environment as well as high-level communication services for sending and receiving/handling asynchronous application-level events, dealing in a transparent way with the intermittent connectivity and disconnected operation at the lower level, due to mobility. The implementation is based on Node.js and WebSockets technology, while the application is written in Javascript using a simple API.

The rest of this paper is structured as follows. Chapter 2 discusses related work. Chapter 3 gives an overview of the system architecture of the distributed applications we wish to support. Chapter 4 describes the communication support provided by our middleware, and the respective APIs offered to the application developer. Chapter 5 describes the server-side and mobile-side runtime environments. Chapter 6 shows an application example that runs on top of our middleware. Finally, Chapter 7 concludes the thesis and identifies directions for future work.

Chapter 2 Related work

MagnetOS [1] is a distributed operating system which simplifies the programming of ad hoc networks and extends system lifetime. MagnetOS presents an alternative programming model where a thin distributed operating system layer makes the entire network appear to applications as an extended Java virtual machine. MagnetOS applications are comprised of a set of mobile event (static) handlers, specified by programmers. The MagnetOS runtime is in charge of application partitioning and dynamic migration, to distribute the event handlers to nodes in the ad hoc network.

Coign [2] is a distributed partitioning system that significantly eases the development of distributed applications. Coign is restricted to applications built with components conforming to Microsoft's Component Object Model (COM), which the system software automatically partitions and distributes the components of application to nodes within a local-area network.

Jessica [3] is a middleware that support parallel execution of multithreaded Java applications in a cluster of computers. Jessica combines the parallel execution capability of clusters and the simple thread model of Java for concurrent programming. It is based on preemptive thread migration, which allows a thread to freely move between machines during its execution. Each computer can only access data that are stored in its local memory by following a shared-memory model, non-local data are obtained as messages being sent from remote nodes.

cJVM [4] is a cluster-enabled implementation of a Java Virtual Machine. It distributes an application's threads and objects across the nodes of the cluster, creating the illusion that the cluster is a single computing resource. Building a Java Virtual Machine upon a cluster-enabled infrastructure, cJVM enables exploiting optimizations based upon Java's semantics which includes caching individual fields and also, migrating a Java thread between nodes to improve locality. As a consequence, cJVM completely hides the cluster from the application trying to manage the application's threads and objects to achieve a performance benefit for a large class of real applications.

MagnetOS and Coign reduce the application execution delay due to network communication. They both treat the network as a system of autonomous computers and transparently partitions applications into nodes within the network to reduce energy consumption and to increase system longevity. Jessica and cJVM make the cluster appear as a single computer to

applications, where threads are automatically redistributed across the cluster for achieving the maximal possible parallelism. Also, both target rather tightly-coupled parallel computing applications.

Our work targets distributed applications that span the cloud and mobile computing/sensing devices (e.g., smartphones), aiming to relieve the developer from having to deal with different system platforms and low-level communication issues. The middleware environment we have developed for this purpose is built on top of Node.js, and provides a high-level API over the WebSockets protocol, hiding intermittent communication and disconnected operation issues from the developer.

While our middleware offers a uniform runtime environment, unlike MagnetOS and Coign, it does not perform any automated decomposition, partitioning and placement of the applications on different devices/nodes. In our work, the programmer partitions the application into the server-side part that will run on the cloud and the mobile-side part that will run on the smartphones, and explicitly declares the events and event handlers that drive the respective interaction. Also, contrary to Jessica and cJVM, we do not target traditional parallel computing applications, but focus on distributed applications that work according to 1<->N scheme, and which can involve a very large number of mobile devices that communicate with the cloud over unstable wireless links.

Chapter 3 System Overview

Our work focuses on distributed applications that span the Cloud and powerful wireless mobile devices (like smartphones), and which follow a 1<->N distribution scheme, as illustrated in Figure 1, where one part of the application resides in the cloud and the other part resides on a potentially large number of devices.

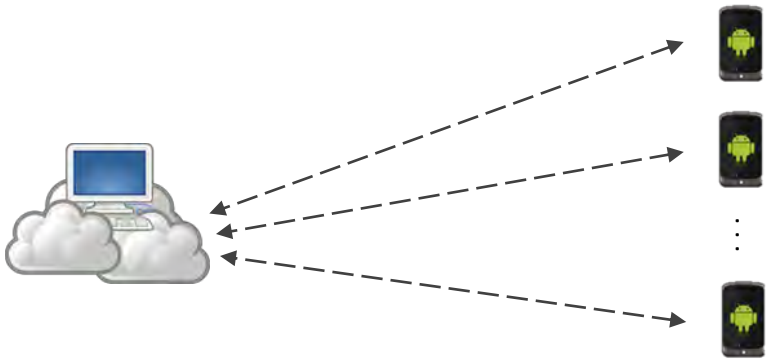


Figure 1: Application structure.

The interaction between these two application parts follows a client-server pattern, with the part that resides in the cloud acting as a server, and the part that resides on the mobile devices acting as a client. Henceforth we will refer to these parts as the "server" and respectively the "mobile" parts of the application. Typically, the mobile part performs some form of sensing and light data pre-processing on the mobile device, while the server part collects the data produced from all mobile parts and performs more heavyweight processing and visualization tasks.

It is important to stress that the server part does not necessarily know the number of mobile parts that exist or that will attempt to interact with it. Not only can this number be very large, but it can also vary dynamically during the application's lifetime. For this reason, the interaction between the server and the mobile part must be supported in a flexible way, also letting the server part address all mobile parts as a whole, without having to deal with each and every one of them independently (unless this is actually required/desired by the application).

To support such applications, we provide a uniform runtime environment for both the server and the mobile part, and allow these two parts of the application to interact in an asynchronous and event-oriented manner via a corresponding API, as shown in Figure 2.

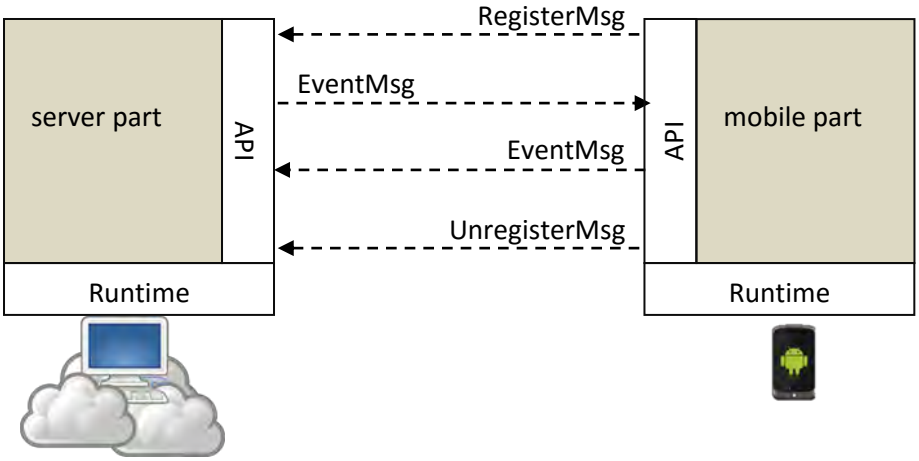


Figure 2: High-level interaction between the server and mobile parts of the application.

At the level of the middleware, the interaction between the server and mobile parts is based on three types of messages. Initially, the mobile part registers with the server part, via a "RegisterMsg" message. Then, the two parts can issue application-level events to each other, which are sent to the other side as part of an "EventMsg" message. Finally, the mobile part can explicitly unregister itself from the server at any point in time, via an "UnregisterMsg" message.

The middleware transparently maintains the list of mobile parts that are registered with the server part. It also automatically unregisters a mobile part that does not show any sign of activity for a longer period of time. Furthermore, the middleware performs the event exchange between the server and mobile part with tolerance to intermittent connectivity and disconnections, buffering events locally until they are successfully sent to their destination, or their lifetime expires. The timeouts for automatic unregistration and event garbage collection on the server side, and the reconnection retry intervals and maximum number of attempts on the mobile side, are specified by the application according to its requirements.

Given this support, the application programmer merely has to define the application-level events and write the server and mobile code that should be invoked in order to handle them. Everything else is done by the middleware behind the scenes.

Charter 4 Communication & runtime support

4.1 Overview of the protocol stack

We have developed our communication support for the Node.js environment, which can be used to run Javascript programs in a platform-neutral way. Our implementation is based on the Socket.io library, an open source project distributed under the MIT license. Socket.io is written in Javascript and provides a real-time transport service on top of the WebSocket protocol. The resulting protocol stack is illustrated in Figure 3.

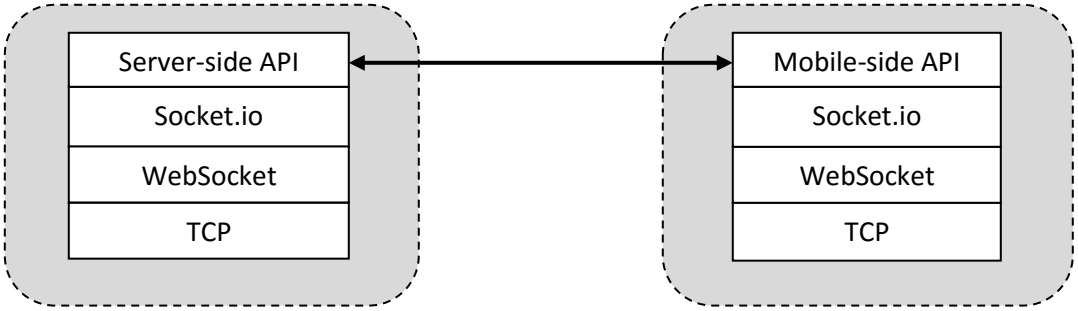


Figure 3: Layers of the protocol stack.

The WebSockets protocol provides a full-duplex communication channel over a single TCP connection, which is kept open and is used to transport messages in both directions over a longer period of time. To establish a WebSocket connection, the client sends a WebSocket handshake request as a special type of HTTP request, which is interpreted by the server as an Upgrade request in order to switch to the corresponding protocol mode. The ability of the server to support the WebSockets protocol is verified via a special HTTP reply. Although WebSockets was designed to be implemented in web servers and web browsers, it can be used by any client or server application.

Socket.io is a Javascript implementation of the WebSockets protocol. It also provides more features, such as broadcasting to multiple sockets, storing data associated with each client, and asynchronous I/O. Another feature is the ability to establish private channels that can be used for client-to-client communication.

Finally, on top of Sockets.io, our implementation provides the following services to the application programmer:

- Abstraction of the underlying communication protocol.
- Support for open-ended application-specific event-driven programming.
- Event broadcasting to all registered mobile parts of the application.
- Event buffering and retransmission upon reconnection.
- Tolerance to disconnections and the intermittent connectivity.

In a nutshell, the developer defines the events for the interaction between the server and the mobile part of the application, and provides the handlers for them in the form of callback functions. The runtime flow of the program is then determined at runtime, depending on the order of event arrivals, and the actions that are taken by the corresponding event handlers.

The middleware keeps an internal list of all application-level events types and handlers, which have to be properly declared by the developer. When the application issues an event, the middleware sends an "EventMsg" message to the other side together with the application-level event type and payload. At the other end, the middleware intercepts "EventMsg" messages, looks-up the list of application-level events for the event in question, and invokes the corresponding handler. Data serialization and deserialization is performed by the middleware.

The next subsections discuss the concrete server-side and mobile-side APIs in more detail.

4.2 Server-side API

The server-side API provides the primitives for listening on a specific port for incoming registration requests, sending events to one or more registered mobile parts of the application, and receiving/handling events that were issued by them. The primitives are summarized in Table 1.

Method *createServer ()* creates and sets up a new server WebSocket, either on an HTTP server or on a plain TCP port. It accepts one argument which is a HTTP server or a common port number.

Method *closeServer()* closes the server socket. However, all application-level state remains intact, in case the applications wish to resume operation at a later point in time. This method does not have any parameters.

Primitive	Description
createServer(*int port / HTTP (server) port* *)*	Create a new server which listens on an arbitrary port or an HTTP server, for connection requests of the mobile part of the application
closeServer()	Close the server.
newEventListener(*String event,* *Function handler* *)*	Define a new application event, and its handler for the server part of the application.
sendEvent(*String event,* *(Any) data,* *Array IDs,* *Boolean buffer,* *int lifetime* *)*	Send an event to one or more mobile parts of the application.
sendEventToAll(*String event,* *(Any) data,* *Boolean buffer,* *int lifetime* *)*	Send an event to all mobile parts of the application.

Table 1: Primitives of the server-side API

Method *newEventListener()* updates the internal list of application-level events types and handlers. The first parameter is the name of event and the second parameter is the callback function for it.

Method *sendEvent()* sends an application-level event to one or more specific mobile parts of the application. It has five parameters. The first parameter is the name of event, which should correspond to an event which can be handled at the other side. The second parameter is event payload/data, which can be of any type – of course, the other side must be prepared to properly receive/interpret it. The third parameter is an array with the identifiers of the mobile parts to which the event should be sent. The fourth parameter indicates whether it is desirable to issue the event towards mobile parts that are (still registered but) currently disconnected. If so, the fifth parameter can be used to set the lifetime of the event in seconds; upon expiration of its lifetime, an event is automatically collected by the middleware.

Method *sendEventToAll()* is similar to the one above, but sends the event to all registered mobile parts of the application.

4.3 Mobile-side API

The mobile-side API provides the primitives for connecting to the server-part of the application, sending events to it, and receiving/handling events that were issued by it. The primitives are summarized in Table 2.

Primitive	Description
newSession(*String url,* *Boolean persistent,* *int attempts,* *int interval* *)*	Establish new session with the server part of the application.
closeSession ()	Close the session to the server part of the application.
newEventListener(*String event,* *Function handler* *)*	Define a new application event, and its handler for the mobile part of the application.
sendEvent(*String event,* *(Any) data,* *Boolean buffer,* *int lifetime* *)*	Send an event to the server part of the application.
setIdentifier(*String id* *)*	Set the identifier for this mobile part of the application.

Table 2: Primitives of the mobile-side API

Method *newSession()* establishes a new session with the server. The first parameter is the URL of server. The second parameter indicates whether an automatic re-connection is desirable in case of a disconnection. If so, the third parameter specifies the number of re-connection attempts, and the fourth parameter is the time in seconds that should elapse between two attempts.

Method *closeSession()* terminates the session. However, all registered events and event handlers remain valid, in case a new session is established at a later point in time.

Method *newEventListener()* works in the same way as in the server-side API. It updates the internal list of application-level events types and handlers. The first parameter is the name of event and the second parameter is the callback function for it.

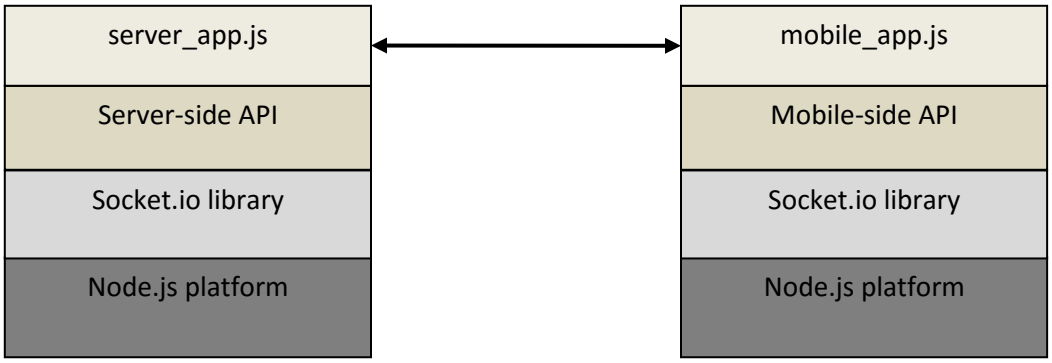
Method *sendEvent()* works in a similar way as in the server-side API, but in this case there is no need to specify the destination – the mobile part of the application always sends events to the server part. The first parameter is the name of event, which should correspond to an event which can be handled at the other side. The second parameter is event payload/data, which

can be of any type – of course, the other side must be prepared to properly receive/interpret it. The third parameter indicates whether it is desirable to buffer the event in case the mobile part is currently disconnected. If so, the fourth parameter can be used to set the lifetime of the event in seconds; upon expiration of its lifetime, an event is automatically collected by the middleware.

Finally, as the name suggests, *setIdentifier()* is used to set the identifier of the mobile part. The application should do this before attempting the first session with the server part. On the server side, this identifier is used to refer to individual mobile parts, if desired, e.g., so that the server part can send events to a subset of the registered mobile parts via *sendEvent()*.

4.4 Server-side and mobile-side runtime

The above communication support is provided on top of Node.js, a Javascript platform that is designed to support scalable networking applications. Besides the inherent portability of the Javascript language, Node.js actively targets server-side functionality by adopting an event-driven architecture and supporting non-blocking communication, thereby enabling the creation



of data-intensive real-time applications that involve a large number of devices. We employ Node.js for both the server and mobile part of the application, as illustrated in Figure 4.

Figure 4: Server-side and mobile-side runtime environment.

As a consequence, with the exception of the different APIs for the server and mobile part of the application, the programmer sees a uniform runtime environment, and does not have to become familiar with different languages, protocols and runtime systems for the cloud and the mobile devices that will be used to run the application.

4.5 Proof-of-concept prototype

We have prepared a server-side runtime image for the Cloud using the Cloud9 Integrated Development Environment (IDE). Cloud9 IDE provides a variety of pre-setup workspaces and supports different programming languages, including Node.js and Javascript, and allows developers to write and run code in the cloud in a straightforward way. To start the server part of the application, one simply logs into our Cloud9 account, uploads the server-side Javascript program, and runs it.

For the mobile side, we have prepared a runtime image for Android smartphones, which can be used via the Apache Cordova framework. Cordova allows us to build native mobile applications using standard web technologies such as HTML5, CSS3, and JavaScript, without being tied to each mobile platform's native development language (such as Java or Objective-C) and runtime environment. It also offers APIs for accessing native device functions, such as the camera or the GPS.

Chapter 5 Application example

To test our middleware, we have developed an application that collects user positions, recorded via the GPS of the smartphone. The server and mobile parts of the applications are written in Javascript, based on the API that was described previously. The server part, which runs on Cloud9, also creates an HTTP server used (a) to push the mobile part of the application on the user's smartphone, and (b) to serve a page where the data that is collected from the mobile parts is displayed on a map (as shown in Figure 5).

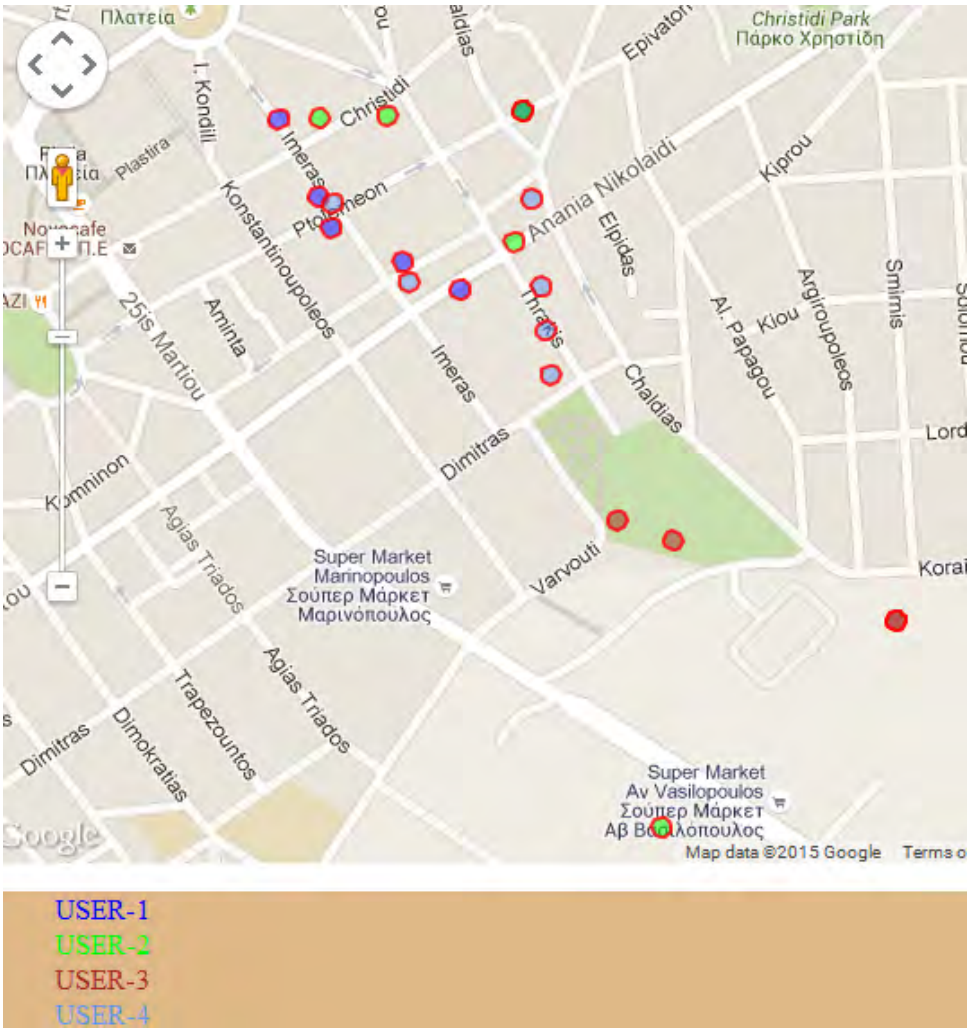


Figure 5: Display of the data/positions collected from different smartphones on a map.

On the smartphone, we have implemented an application that downloads the mobile part from the Cloud, and then starts running it locally (as shown in Figure 6). This makes it possible to change the mobile side code of the application without having to update/re-install the main application that runs on the smartphone, which was particularly useful for testing purposes.

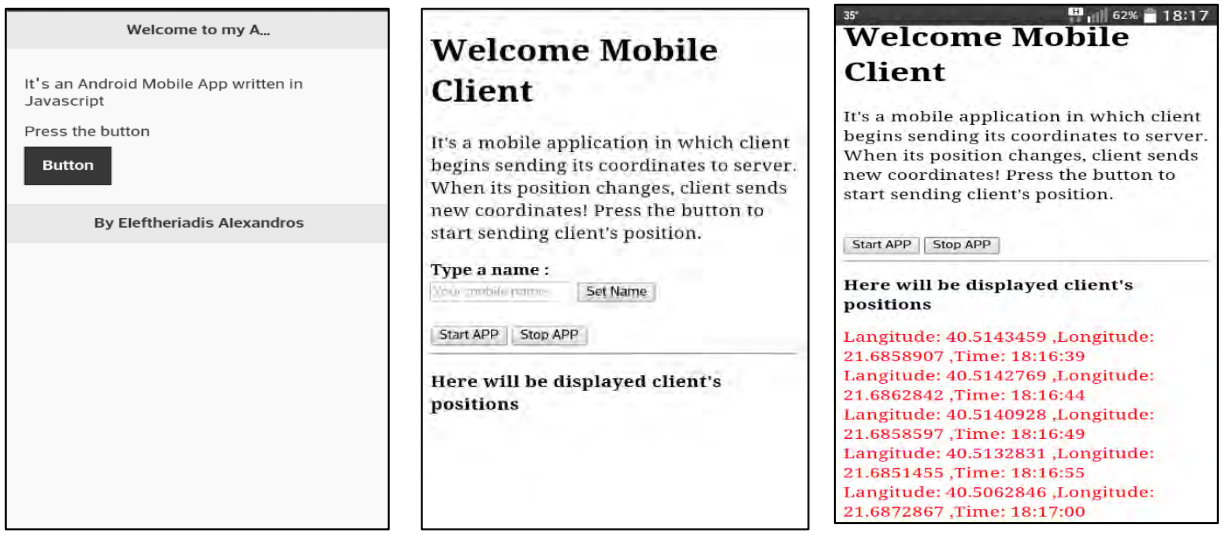


Figure 6: GUI of the main application, prompting the user to download the mobile part, and GUI of the mobile part before establishing the session with the server and during normal operation.

Indicative code snippets from the server part of the application, illustrating the usage of our middleware, are given below:

```
var http = require('http');                          // “HTTP” node.js module
var serverAPI = require("./serverAPI");              // The server-side API of the middleware
var sever = new serverAPI();                         // Create a new server side instance

var httpSrv = http.createServer(function(request, response) {      // HTTP server initialization
    …
}).listen(….);

var updatePos = function(position) {                 // Function for handling new position data
   …  // draw position on the map
 } };

server.newEventListener('newPosition', updatePos);   //  'newPosition' event and its handler

server.createServer (httpSrv);                        // server side part listens on the HTTP server
```

As can be seen, one of the events introduced by the application is the *newPosition* event, which is handled via the *updatePos*() function in order to display the position received on the map.

The corresponding code snippets from the mobile part of the application are shown below:

```
$("#start").click(function() {                  // On click event, (Button) starts Geolocation functions
    var client = new mobileAPI();               // Create a new mobile side instance
    client.setIdentifier(ID);                   // Define client's ID
    client.newSession('https://mynodejs-alexanderuser.c9.io', true,15,60);   // start new session

    function showPosition(position) {      // Receive new positions (Geolocation API)
        data : {clientID: ID, langitude : position.coords.latitude ,
                              longitude : position.coords.longitude,
                              time : position.timestamp }};
        client.sendEvent( 'newPosition', data, true, 60);    // Send new coordinates to server side
      …. // print data on the phone's screen
    }
    ….
} });
```

When the user presses the start button, a new mobile app instance is created, which is then used to send position updates of the mobile device to the server via the *newPosition* event.

Chapter 6 Conclusions

Our middleware greatly simplifies the development of distributed applications with a 1<->N communication pattern, where one part of the application resides on the Cloud and the other part of the application resides on a potentially very large number of mobile devices. The programmer is provided with a uniform event-oriented programming model, for both parts of the application, and can structure the application in a flexible, event-oriented way, without having to deal with intermittent connectivity and disconnections.

Possible next steps could be the extension of our middleware to allow for mobile-to-mobile interactions. It would also be interesting to provide language-level support, so that the entire application is written as a single program which is then automatically split into the server and mobile parts, with the latter being automatically instantiated on the smartphones that connect to the server.

References

[1] H. Liu, T. Roeder, K.Walsh, R. Barr, E. G. Sirer. Design and Implementation of a single System Image Operating System for Ad Hoc Networks, 2005.

[2] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proc. of OSDI*, New Orleans, LA, February 1999.

[3] M. J. M. Ma, C.-L.Wang, F. C. M. Lau, and Z. Xu. JESSICA: Java-Enabled Single System Image Computing Architecture. In *Proc. of PDPTA*, Las Vegas, NV, June 1999.

[4] Y. Aridor, M. Factor, and A. Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *Proc. of ICPP*,Wakamatsu, Japan, September 1999.

[5] Node.js platform https://en.wikipedia.org/wiki/Node.js.

[6] Socket.io library https://en.wikipedia.org/wiki/Socket.IO.

[7] Apache Cordova platform https://cordova.apache.org/.

[8] WebSockets protocol https://en.wikipedia.org/wiki/WebSocket.

[9] Cloud9 IDE https://c9.io/.