Distributed Network Architecture For Mobile Devices

Christina Zacharoula Chaniotaki, Konstantinos Kopanidis, Panos Louridas,
Dimitris Mitropoulos and George Xylomenos
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{zchaniotaki,kkopanidis,louridas,dimitro,xgeorge}@aueb.gr

***Abstract*—The next big step of distributed systems and internet is through the use of computing power of mobile phones. None of the existing devices uses 100% of its power, causing the surplus to be wasted. Creating a system that allows distributed data collection and processing via mobile phones, a whole new range of capabilities is introduced, to share tasks, reduce costs and exploit existing computing capabilities and connectivity. With the use of mobile phone operating systems, in particular with the use of android, and including the mobile phone in a network, it can be used to process information and analyze data. Thus, reducing the machines needed to process this data. Our thesis aims to create a Distributed Network Architecture that uses Android Smartphones to run server-side NodeJS code and plain Javascript Scripts. To achieve these, we created a library for Android applications, an API that handles all the jobs and a Graphical User Interface for the users.**

***Index Terms*—Distributed Network, Decentralized applications, Functions as a service**

I. Introduction

Since 1980s, the nature of computing systems was characterized by the existence of powerful mainframes in which applications were installed that users could access through their terminal systems. In the 90s, when computer programs began to grow and become complicated, there came the next step, distributed systems. The model that prevailed was the client/server model, and thus, applications began to split in two key segments communicating with each other through various types of services. As far as plant operations, production processes, and so on, they were all handled by mainframes. Due to the fact that the operations started to increase in size and the installation became increasingly difficult, unsustainable but also vulnerable it was a solution to create a distributed system to control these large processes through a computer network. These interconnected distributed data collection and processing systems function as a single, decentralized system of centralized information and control.

Over the years, technology evolved, and other solutions emerged to meet needs such as electronic mobile devices and mobile phones for communication between people. But the next big change in mobile phone history came in 2007 with the Apple iPhone, creating smartphone with advanced computing capability for the era [9]. The evolution progressed even more rapidly with the creation of Android smartphones and various other competitors that emerged over the years [4].

Google announced in 2017 that there are over 2 billion android devices active, and increasing daily [7]. An average android device has 4 cores, that stay idle for most of the time, especially when the device charges, which amount for at least an hour of idle time, since it is not being used by the user. Another paper found out that users usually charge their devices at night, and the charging time amounts for an average of 8 hours [5]. Billions of devices all over the world have idle processing power that remains untapped to this day, and this paper proposes a way to take advantage of this, increasing the available computing power worldwide and providing profit for the users.

The next big step of distributed systems and internet is through the use of computing power of mobile phones. None of the existing devices uses 100% of its power, causing the surplus to be wasted. Creating a system that allows distributed data collection and processing via mobile phones, a whole new range of capabilities is introduced, to share tasks, reduce costs and exploit existing computing capabilities and connectivity. With the use of mobile phone operating systems, in particular with the use of android, and including the mobile phone in a network, it can be used to process information and analyze data, thus reducing the machines needed to process this data.

While a program running on a cloud machine costs less than buying a machine in private space, using the computing power of mobile phones, the cost is further reduced because no new machine is bought or rented, but one application is shared across multiple devices, reducing costs significantly, exploiting existing computing power, and at the same time creating a secure, linearly expandable and distributed network of mobile phones [1].

II. Overview

Using the computing power of mobile devices, specifically that of Android smartphones, coupled with the creation of an Application Programming Interface (API), a library and an administration panel, we created a system that allows the distribution of functions, processes and procedures that are written in JavaScript, to mobile devices for calculations, analytical computations, data processing and process execution. For the creation of the tool, we first engineered a Java library that allows for the completion of tasks, a server for

processing and routing tasks, as well as an admin panel for data visualization and easier system management. Users can add tasks through the panel or via interacting with the API directly.

III. Architecture

The software consists of a Java-based library for Android Applications, an API server written in NodeJs, a MongoDB for storing the users, the mobile devices, the processes, the procedures, and the functions. There is also an Administration panel that visualizes the API that is written in Angular 4.

A. Android

Android is the Mobile Operating System (OS) developed and maintained primarily by Google [4]. It is an open source OS, based on the Linux Kernel that uses sandboxed Java Virtual Machine (JVM) instances to run applications. There are two types of apps for the android OS. Those built with the Java SDK, written and run in sandboxed JVM instances, and those build using the Native Development Kit (NDK), that run in a sandboxed JVM instance but with native bindings. The difference between these two is that applications that using the NDK, can write C/C++ code and Java Native Interfaces (JNI) bindings so that high performance tasks can be completed without the added JVM overhead. Java is needed in all cases to provide the User Interface (UI) and power to all user interactions.

Android applications have private storage for their SQLite databases, files and dex files (compiled code for Android). Each application is completely isolated and interacts only with the Android system.

1) Library: For our system to function as designed we needed two services. Firstly, we need to have a way to run JavaScript and NodeJS in Android using the official distributions of V8 (JavaScript Engine)[1] and NodeJS engines. For the JavaScript part the solution is fairly easy, since android provides an API to interact with the V8 engine included in most modern smartphones [2]. The issue here is that our library would have to be developed using the Android NDK[2] which can add overhead and if coded poorly, it would make the application vulnerable to exploits and memory leaks. We wanted the safest solution so we will further research about this. Android as described in the Section I, has the ability to run C and C++ code natively, so theoretically integrating NodeJS should be feasible. And we proved it. Thus, we needed to solve two problems. The solution came from another library, J2V8[3], which we built from the open source repository and created an Android Archive (aar), which includes a native official V8 engine, and the NodeJS engine, both exposed through Java APIs, and internally used with JNIs. This allowed us to simply plug in a library and focus on the workflow and not the integration of such complex engines with the Android OS. Thus, the DNA library was created and we are going to look into several concepts implemented and its overall architecture.

Once the DNA process is started, the network daemon communicates with the master server, announcing its availability for job/task completion while sending its current status along. This process is scheduled to be executed every one second. Every 10 seconds, the performance monitor measures the time the device requires to execute common procedures. Like calculating 10!. This allows us to have a metric which can be used to classify the device and its capabilities. The time to run for each procedure is sent to the DNA server at each update. Once a job is retrieved it is added to the DNA queue, where the queue daemon will monitor its status. The queue daemon, keeps track of jobs that need to be run and tries to execute them, as well as keeping track of finished jobs and sending their results to the master server. This process is run every one second.

The queue daemon interacts with the Warden. This is another daemon that keeps track of running job threads, their status and constantly checks for need of termination. Jobs are run by the Warden only if the limits imposed by the library's configuration are not exceeded, and jobs are terminated either by self-termination or by the Warden issuing an interrupt in the event that a job has exceeded the time to live limit. Every job, once run, is assigned to a DNA Executor thread of the appropriate type (Node Executor, V8 Executor). Each thread contains all the logic needed to complete the job assigned for the given type. In Figure 1 we present the architectural representation of the library.

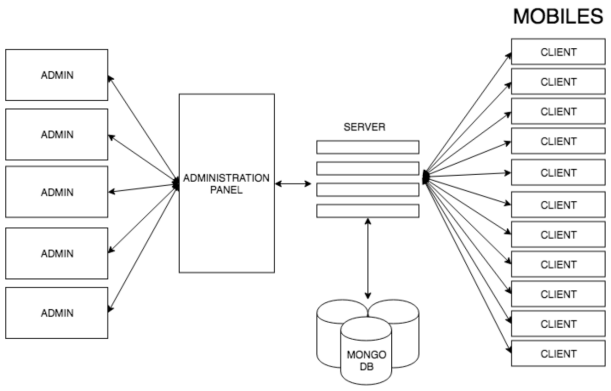

Fig. 1. Application architecture

2) Application: From a developer perspective, looking to use DNA in their application, the library is fairly straightforward. The package needs to be included as a dependency, and then the exposes DNA class needs to be instantiated. This starts the DNA daemon that will make everything work. From that point on, the client app does no other interaction with the library.

In the demonstrator app we created we do this on app open. From then on the DNA library just works. From the beginning of the project it was of absolute importance to hide the library's

[1]https://v8.dev/

[2]https://developer.android.com/ndk

[3]https://eclipsesource.com/blogs/tutorials/getting-started-with-j2v8/

complexity. So, from the developers standpoint, the lib just works, without the need of any complex configuration and/or monitoring.

Furthermore, if the developer wishes to run a script locally, meaning that the script is not retrieved from the DNA server, then there are the appropriate functions exposed to do so. These override the DNA queue, and the scripts are instantly run by the Warden. In addition, we included the ability to bind message listeners to the script requested to be executed so that the developer may log actions or trigger others on script events.

B. API

The API manages the various processes both running and in queue, as well as the various mobile devices that receive and execute tasks. It is a master server that at any moment controls and routes tasks and monitors the availability of mobile devices.

1) Database (Virtual) Schema: We using Mongo Database[4] to store all necessary attributes and the entire schema. The database has five document types. Device, Script, User, Job and JobExecution.

Device: Has the information for the device (Android smart phones currently). For each device we want to know some information like, its online status, job availability for scripts or processes and how many processes and jobs running. Besides these, we have for every device a score. When a device has a good score, that is when it has acquired scripts and returned the results without errors. Additionally, if it is online and completes its jobs, it has a high score and is one of the first to be assigned with jobs. However, if the device is not good, e.g. takes jobs and never returns, or returns with errors, then its score decreases, and it goes to the last places of the job assignment list. Nevertheless, if the device starts again to respond with correct results its score increases again. That happens because we do not want to reject devices instantly but we want to do it only if it is necessary, gradually to keep the consistency of the network. And that's because maybe a device, since of connectivity problems or any other reason, does not respond to the server or does not complete correct the jobs. But as soon as the issues are resolved, it can respond again, and so the score increase again.

User: If someone wants to upload a script, create jobs, and run them, they need first to create a profile. From the Graphic User Interface (GUI) that we offer or by sending the appropriate request to the API. The user schema includes information about the name, company, address, city and so on. When the user logs in, a Json Web Toekn (JWT) [5] token is created and they can send requests with scripts, jobs and so on. In any other case no action is authorized.

Script: The script schema has all the information about the script. Contents, name, type (JavaScript, process or node script), execution time and some Analytics like number of devices that the script run on, mean input and output size, mean execution time, last execution and so on. Other fields include the user that uploaded the script. The script has three types:

- **Process:** If the script may never stop on its own.
- **JavaScript:** If it should terminate forcefully after time to live of the device is reached. Otherwise, it will return on its own and will produce a result.
- **Node Script:** If the script produces no result, but should be terminated if the time to live of the device is reached. If it terminates at an earlier time, no further action is required.

The Analytics for the script created, when a job for the script runs and return results from the device.

Jobs: To run a script, it is needed to create a job. The job schema has information about running the script, like:

- **Input:** If the script needs input.
- **Callback url:** If the script does not want to return on the server but somewhere else.
- **Interval:** Every how long should the job be repeated. This is used for a repeating job (Cron syntax).[6]
- **Executed on:** When the job should start executing. This is used if it is a non-repeating job.
- **Execution status:** An object that describes the current status of the job.
  - Status: The available statuses for the job.
    - Waiting: when the job has waiting to executed.
    - Pending executed: when the job has not been executed to a device yet.
    - Executing: when a job has begun execution.
    - Executed: when a job has finished executing and the results have been sent back.
    - Finished: when the job is done, meaning that it will not be executed again through interval.
  - Assigned Devices: The device to which the job has been assigned.
  - Log: The execution log, which keeps track of the various status changes and the time that they happened.

Except of these, it also has the user that submitted the job (this may not be the same user that uploaded the script) and the script. A script has many jobs. A job belongs to one script (Many to One Relationship).

Job Execution: The job Execution schema concerns the result from the job when it runs on a device. A Job Execution belongs to one Job. A job has many Job Executions. The job execution schema has:

- Execution Time: time from client when run
- Number of Devices: number of devices that run the job
- Input size: If job needs input
- Output size: The output in Base64
- Has error: If the job return with error the Has Error becomes true. Otherwise is false.

[4]https://www.mongodb.com/

[5]https://jwt.io/

[6]https://crontab.guru/

Logic: The logic includes ten files. Device, Job, Script, User, Device Monitor, Job Classifier, Job Execution Manager, Job Monitor, Job Manager, Session Manager.

Regarding the login on the server, firstly, we have the Job Manager. The Job Manager takes all online devices and accordingly with theirs info gives to them jobs. The priority for the jobs defined from Job Classifier. The Job Classifier takes all the jobs with status pending assigned and creates a score for them accordingly on some information like execution time, input, and so on. If the score is high then the job has high priority for a device. When a job send to a device takes the status executing. When the device send the result the job, returns with status executed. From there we have the Job Execution Manager. The Job Execution Manager checks the results for the script and if returns an error and then updates the script analytics and changes the job status. In case that job has an interval different to one, the job takes status waiting in order to run again. Otherwise, takes the status finished which means that the job will not run again on a device.

However, because it is possible to have problems on this, we created some services for checking the process. For example one job is possible to never arrived to the device (has status pending assigned for long time) or never returns from the device (has status executing). The Job Monitor, takes all these jobs and search the scripts (script analytics, if has other jobs and so on) and checks the job executions and depending on the results even deletes the job or updates the status to run later.

Except of the Job Monitor, we need to check the devices too. For that, we have the Device Monitor. The Device Monitor selects all the online devices and checks which of them never returns results or return results with errors and they take again jobs. In each case, the Job Monitor forms a score for the device to change the priority on taking jobs from the Job Manager. The Device Monitor never cancel a device from the first error but it do it gradually depending on their status. And this because, after some time, a device could respond correct again.

C. Portal

Except of the API and the Android Library we also have a portal. The portal is a visualization for the Server. If a developer does not know how to send a request to the API, he/she can use the portal to upload scripts, create jobs and see the results. Additionally, the portal provides the devices who runs the jobs with some information (e.g location). The portal was created with Angular[7]. It enables the user to interact with API throughout a User Interface. The template that was used was created by Creative Tim[8] and a lot of changes were made to fit the application and incorporate all the functionality we needed. With the portal the user can register and then can log in with their details. Once connected, it provides an initial screen with general information about its scripts and also in the menu there are the following options:

1) Profile Details
2) Upload a Script
3) Create a new Job
4) All Script
5) All Jobs (for the user)
6) All Devices and how many jobs have run on each of them.

All of these features can be used directly by calling the API, but through the panel they can see them in a graphical environment. The data changes whenever a change is made.

D. Future - Prospects

The project presents many possibilities on a practical level that combined with the widespread adoption of Cryptocurrencies, the Blockchain and proof of work philosophy, can aid create truly Serverless applications, and potentially a new kind of internet for devices or a platform for Blockchain-based distributed applications.

Below we describe several potential applications of the technology:

1) Google Firebase-like system: A distributed system that works by embedding the DNA library in a developers application. A server described as a knowledge node that will facilitate backup storage and network support when there are few nodes online. The system will work as so Each device will carry some non-modifiable code, that will allow for database storage in the users device, login/register operations, serverless/lambda function calls from the network. For every operation, the node will have to broadcast the record of the transaction to the entire network in a similar manner as the bitcoin network. Upon consensus that the operation is valid, all instances of the database distributed in the clients, will be updated.

All operations are secured via the proof of work philosophy and the consensus requirements. Until a consensus is reached the users device continues operating normally without showing action delays, or by making the user wait for consensus. If the database gets large enough that device storage is no longer an option, there can be two alternatives. N-segment splits to devices, so that a database is distributed between nodes, and twice in the network for redundancy. A large enough database will mean that the application has many users and thus a model such as this is feasible. The 2nd alternative, is migrating all data to the knowledge node, and access keys will be generated by authenticating the device with the entire network, and generating a unique access key for the Database for read operations on specific database collections/tables. For write operations the same procedure will be required as above.

2) A new kind of internet for devices: This required the 1st project, along with the ability to share device CPU, Network and storage with the network. This means that there will be a DNA network, where applications that use the DNA library, will essentially use 80% of the device capabilities while sharing the rest 20% with the network. It will allow for an increase in available nodes for each application, since an app will not need to have 50+ users online to work, since it

[7]https://angular.io/

[8]https://www.creative-tim.com/

can use resources outside its own network. This eliminates the case of the network failing, since there will always be devices that can act as backups. This can be further optimized, by creating backup nodes, that will be computer servers, with the necessary code that will allow for them to further enhance the network and support operations, even when the node count is at its lowest.

3) A platform for distributed applications: The DNA project can be used to create an application for Android, Windows, Mac OSX and Linux that will be able to run JavaScript and NodeJS code in all clients. This will allow the creation of distributed applications that use DNA as their basis, while being agnostic of the platform they are currently being run on. The closest resemblance is Apache Cordova [9] but instead of being used for client apps, it will be used for server-grade software. DNA, will provide the developers with several adapter functions so that the scripts may access device functions like secure, script-specific storage.

There are many more uses for the framework, but these are the ones we believe are the most promising. They can be implemented separately or as one framework.

IV. Implementation

A. Christina Zacharoula Chaniotaki

The parts that were fully implemented from Christina were the portal (90%) and the creation of the server (80%). The Server as well as the panel were analyzed in Section III-B and III-C. Specifically, they were fully implemented:

1) The selection of the GUI
2) The changes in the CSS code
3) The interaction with the API
4) All the components and the processes

Regarding the API she created: the 80% of the routes and their functionality, and the models such us user creation, editing jobs, schedulers an so on.

B. Konstantinos Kopanidis

The part that was fully implemented from Konstantinos was the smart phone library as well as improvements and changes on the portal and the server. Additionally, he creates same periodically running scripts that control various statuses and accordingly run several procedures, for the server. All of the smart phone library has been analyzed in Sections III-A and III-A1.T he library is a java library which means that it is not necessary to run through mobile phones but it can be used in any program in any java-supported platform.

V. Benchmarks

Using the existing test scenarios in the library that we build, we have the following results:

We notice that trying to calculate a larger factor does not cause an increase in runtime either on the computer or in the mobile. The difference remains constant, around 100ms above, which we attribute both to the increased power of the computer and to the "cold start"[10] of Node on the mobile. However, even twice the execution time is acceptable, given the environment.

TABLE I
Benchmark Results

Test	Library	PC
FactorialTest(10)	200ms	100ms%
FactorialTest(100)	200ms	100ms%
FactorialTest(1000)	100ms	100ms%
FactorialTest(10000)	200ms	CALL STACK SIZE EXCEEDED %

In terms of network, we test the speed of the mobile and the computer, on resolving a domain and making a ping. The computer manages to complete the ping at 60ms, while the mobile at 80ms.

The computer that we used was a Macbook Pro with CPU: i7, four cores, eight Threads and a 16GB RAM. The Network was a VDSL with 50mbps. The mobile was with CPU: four, four cores, 3GB RAM and a 3G network at 14mbps.

VI. Relative Work

During the creation of the system we found various existing solutions that can be considered, up to a certain degree, similar to our own. We will present these solutions and the ways that our system is different.

A. Microsoft IoT Edge

Its a platform by Microsoft that provides a cross-platform programming environment for Internet of Things (IoT) devices that run on Linux or Windows. The idea behind it is to enable IoT devices to do computationally intensive tasks and allow them to be used in a secure and predictable way for calculations, data analysis or any other task. The programming languages that supports are Java, .NET Core 2.0, Node.js, C, and Python [3]. In contrast, our implementation supports Mac OS, Windows, Linux and Android. Practically any software that can run Java. Hence, we support a greater device spectrum. Regarding the languages, NodeJS is supported, but with few code changes, Python, Java and C/C++ can also be supported, on the operating systems that can support each language. Our solution is a superset of Microsoft IoT edge.

B. Webinos

Webinos is an interesting concept that focuses in connecting various devices so that they can communicate, share data and allow for the usage of the same applications by the user using the Webinos runtime [6]. Webinos uses NodeJS but is not based on the philosophy of distributed computing and is more focused on sharing information between various applications so that users will have a unified experience. This is similar to the Apple ecosystem, where all devices continuously communicate and synchronize data. Webinos takes this one step further, using multiple technologies to achieve better and more accurate synchronization even without an internet connection. Our solution has nothing to do with this though, since its an

[9]https://cordova.apache.org/

[10]Is the 1st request that a new Lambda worker handles

application development platform, and Webinos could have been built using our solution.

C. Moitree

Moitree is a technology that allows applications to share their workload to the cloud, so that device resources may be freed, both hardware wise and network wise, while requiring supposedly no changes for existing applications [5]. It is created on the philosophy that mobile devices are weak and should be de-congested. On the contrary, our implementation is based upon the fact that mobile devices increase their processing power constantly as well as their energy efficiency and the fact that for a great part of the day they remain idle and for charging. We try to use existing processing power that is not in user, while Moitree wants to bring more power in mobile devices. We believe that our solution is more realistic given the current smartphone market trend.

D. Edge computing

Edge computing is the basic idea of mobile computing and of the processing power enhancement on mobile devices from base stations/servers. It usually refers to nodes that are installed in mobile cell towers and the processing power becomes available to the phones as they enter the cell [8]. Our idea is very different from that, since it is based, as it was stated above, on using idle processing power from mobile devices, not enhancing that power through stationary resources.

VII. Conclusion

Having noted the above, one could argue that our approach seems wrong, concerning the subject of mobile computing, since everyone is trying to bring more processing power to mobile devices, while we are trying to provide them with more processes to handle when idle. The answer lies in both directions. During usage, mobile devices, even today, are in a constant need to improve their energy efficiency as well as increase their processing power. Thus, all the aforementioned ideas help achieve that goal. From the other hand, when these devices are on idle, all their processing power is wasted. That's where our idea comes in, to take advantage of these circumstances to the benefit of everyone.

Furthermore, it is the only idea that could survive in the decentralized world that is currently being created slowly by steadily. In such a reality, none of the other ideas can be applied. In contrast, our idea, which depends on no central point, and when it does is chosen, could provide the backbone of decentralized applications and networks.

References

[1] 12 benefits of cloud computing and its advantages.

[2] Features and apis overview.

[3] Iot edge: Cloud intelligence: Microsoft azure.

[4] *Android*, pages 289–306. Apress, Berkeley, CA, 2009.

[5] Mohammad A. Khan, Hillol Debnath, Nafize Rabbani Paiker, Narain H. Gehani, Xiaoning Ding, Reza Curtmola, and C. Borcea. Moitree: A middleware for cloud-assisted mobile distributed apps. *2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 21–30, 2016.

[6] John Lyle, Shamal Faily, Ivan Fléchais, André Paul, Ayşe Göker, Hans Myrhaug, Heiko Desruelle, and Andrew Martin. On the design and development of webinos: A distributed mobile application middleware. In Karl Michael Göschka and Seif Haridi, editors, *Distributed Applications and Interoperable Systems*, pages 140–147, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[7] Ben Popper. Google announces over 2 billion monthly active devices on android, May 2017.

[8] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[9] Joel West and Michael Mace. Browsing as the killer app: Explaining the rapid success of apple's iphone. *Telecommunications Policy*, 34(5):270–286, 2010.