An Architecture for Programming Distributed Applications on Fog to Cloud Systems

Francesc Lordan[1,2](✉), Daniele Lezzi[1], Jorge Ejarque[1], and Rosa M. Badia[1,3]

[1] Department of Computer Sciences, Barcelona Supercomputing Center (BSC), Barcelona, Spain
{francesc.lordan,daniele.lezzi,jorge.ejarque,rosa.m.badia}@bsc.es
[2] Department of Computer Architecture, Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
[3] Artificial Intelligence Research Institute, Spanish National Research Council (CSIC), Barcelona, Spain

Abstract. This paper presents a framework to develop and execute applications in distributed and highly dynamic computing systems composed of cloud resources and fog devices such as mobile phones, cloudlets, and micro-clouds. The work builds on the COMPSs programming framework, which includes a programming model and a runtime already validated in HPC and cloud environments for the transparent execution of parallel applications. As part of the proposed contribution, COMPSs has been enhanced to support the execution of applications on mobile platforms that offer GPUs and CPUs. The scheduling component of COMPSs is under design to be able to offload the computation to other fog devices in the same level of the hierarchy and to cloud resources when more computational power is required. The framework has been tested executing a sample application on a mobile phone offloading task to a laptop and a private cloud.

Keywords: Distributed computing · Mobile computing
Fog computing · Programming model · Computation offloading
Fault tolerance · Security

1 Introduction

The traditional cloud computing model, based on a centralized control of computing and data resources, does not provide the proper support to the requirements of big data applications that produce and consume volumes of data through IoT devices, fast mobile networks, AI applications, etc. Fog computing has emerged as a complementary solution to overcome the issues related to real time processing, security, latency and transparent management of a decentralized, heterogeneous and dynamic set of resources.

This paper proposes a Fog-to-Cloud (F2C) ready programming framework to develop applications that involve the use of traditional cloud systems, smart end-user devices, and IoT sensors. The framework transparently offloads parts of the

© Springer International Publishing AG, part of Springer Nature 2018
D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 325–337, 2018.
https://doi.org/10.1007/978-3-319-75178-8_27

computation to fog and cloud resources and optimizes the execution considering time, energy consumption and monetary cost. The proposed solution builds on COMPSs [8], a programming model for distributed computing and its associated run-time. On the one hand, COMPSs distributes the computational load of the application transparently to the user and exploits its inherent parallelism and the heterogeneity of the underlying infrastructure. On the other hand, it also handles the distribution of data to provide a seamless offloading and schedules the data processing in larger nodes considering its locality to optimize the execution. COMPSs applications are completely agnostic to the underlying infrastructure and their code runs, with no changes, in all the backends supported by the runtime: HPC systems and private and public cloud. Recently, COMPSs has been integrated with container solutions based on Docker [9] and Mesos [1]. To support the execution of COMPSs applications from mobile devices, the runtime has been refactored to include the support to Android devices and to improve the data management via a Peer-to-Peer (P2P) mechanism. These new features are basic pillars to develop the proposed framework.

A key feature of COMPSs is the ability to distribute the tasks that compose the application on the available nodes of the computing platform. In the case of traditional cloud environments, the decision where to execute a task considers historical data of previous executions and the locality of the data to process. Moreover, the cloud gives the illusion of having access to infinite computing resources; COMPSs can instantiate additional VMs on cloud providers from a settable list. In contexts more dynamic than traditional cloud computing, such as the ones considered in this work, resources might spontaneously disappear from the pool. Handling this volatility is an additional requirement either for data management and proper work balancing between fog nodes. Another relevant issue addressed in the proposed framework is the security since usually edge devices are located in non-controlled environments.

The paper is structured as follows: Sect. 2 includes an overview of the related work in the field of F2C computing framework; Sect. 3 describes the architecture of the proposed solution while Sect. 4 provides the details of how the COMPSs framework has been extended to support F2C environments. Section 5 presents the results of the tests and Sect. 6 concludes the paper and provides ideas for future work.

2 Related Work

Application partitioning, task scheduling, and offloading mechanisms are all problems widely explored in the field of distributed computing. The main differences between previous work on cloud computing and mobile computing are due to issues related to the high mobility of the device, the limited availability of energy of the devices and the impact of the network (latency, monetary cost, bandwidth) on the performance of the entire framework. This analysis of the related work in the field of fog to cloud computing, takes into account capabilities such as how to fragment the applications in order to offload the parts of

the computation to the resources, the scheduling model and the management of parallelism.

CloneCloud [4] offers the developer a thread level granularity mechanism. The strong point of CloneCloud is its partitioning mechanism that combines a static analysis of the code with a dynamic profiling of the application to pick the optimal migration and re-integration points. When a thread reaches a migration point, it suspends, and its state (including virtual state, program counter, registers, and stack) is shipped to a synchronized clone. When the migrated thread reaches a re-integration point, it is similarly suspended and shipped back to the mobile device. The drawback of this system is that it still requires the developer to manage threads and application parallelism. Cuckoo [6] hides the partitioning problem by exploiting the service component of Android operating systems. During the build process, the stubs generated to access service components are replaced by invocations to the Cuckoo framework that decides, at runtime, whether to run the service on the local device or a remote implementation. Since the framework only replaces calls, all the parallelism must be managed by the programmer on the service invocations. ThinkAir [7] provides a mechanism to automatically parallelize the execution of an offloaded method considering intervals of input variables. The main drawback of ThinkAir is that the offloading mechanism works synchronously: the executing thread is suspended until the method invocation is performed and its result collected. Thus, any subsequent method invocation is not executed until previous ones are executed even when they could run concurrently. Mobile Fog [5] is a high level programming model for the future Internet applications that are geospatially distributed, large-scale, and latency-sensitive. The goal is to allow applications to dynamically scale based on their workload using ondemand resources in the fog and in the cloud. In Mobile Fog, an application consists of distributed Mobile Fog processes that are mapped onto distributed computing instances in the fog and cloud, as well as various edge devices. Mobile Fog API is not hiding the distribution of the infrastructure to the application, requiring a large programming effort to the application developer.

3 Architecture Overview

Figure 1 depicts the layered-based architecture of a Fog-to-Cloud platform where the proposed framework can be instantiated; the architecture is designed following the OpenFog Reference Architecture [3]. The lowest layer represents the low processing capability devices, such as sensors or embedded devices that produce data, while the middle layer contains fog devices that have more processing power (as a smartphone or a tablet) and are able to deploy and orchestrate the execution of a distributed application using other fog devices as workers (fog-to-fog). Clouds are at the top layer, hosting services for the control of the entire stack or used for the execution of computing intensive applications started both from the same layer and from a fog device. It is worth noting, indeed, that the framework can be used to instantiate applications on smart devices on the fog

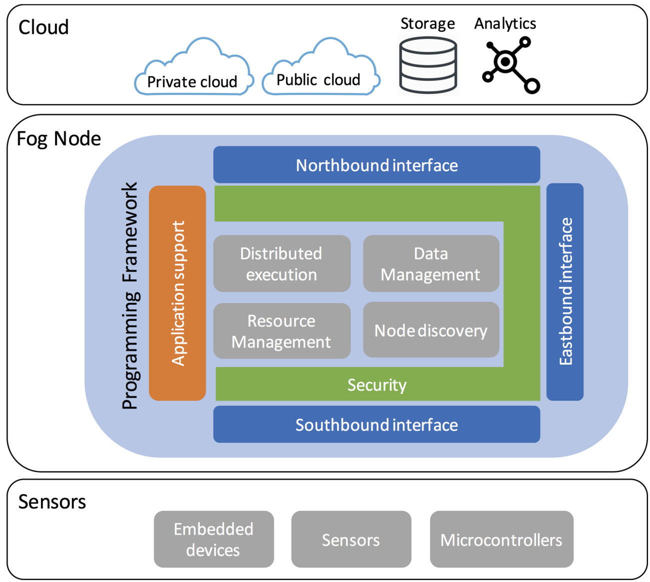


Fig. 1. F2C architecture

layer and to offload part of the computation to the cloud (fog-to-cloud) or use the fog devices as workers for a cloud application.

The main contribution of this work is represented as a programming component in the Fog Node together with the capabilities it offers and the interfaces needed to interact with other elements of the platform. The application support has to be implemented through a high level programming model that enables the development of applications to be executed in distributed, heterogeneous, volatile, data and processing infrastructures. However, these complex infrastructures will remain hidden to the application in such a way that the application can focus on the logic. The aim of this programming model is to keep the code almost untouched avoiding the need for APIs to implement the required functionalities. The application interacts with a runtime that takes care of the coordination of the distributed execution of the applications in a parallel way when possible. The interaction with different computing backends is delegated to a specific component for resource management. Data management is required to let the runtime access to the data produced on the working nodes as well as to synchronize the information on data location in order to proper schedule the tasks on the nodes. The Node Discovery component enables resource discovery and registration. For example, an IoT device coming online "close" to the coordination node can notify its availability to the controller and then this information has to come to the node. Security is a transversal issue common to all the components that have to fulfill a common base set of security and privacy requirements in an environment by nature unsecure and dynamic. Interfaces are needed to ensure communica-

tion between nodes and realizes the data channels. Eastbound interface connects the runtime with other nodes in the same level and allows the sharing of data; Northbound allows to implement the connection with cloud nodes while Southbound interface realizes the connection between a fog node and a sensor or from a cloud application down to the Fog layer.

4 Programming Framework Overview

COMP Superscalar (COMPSs) is a programming model that aims to ease the development of parallel applications to run atop distributed infrastructures. For that purpose, it offers a sequential, infrastructure-agnostic way of programming that abstracts coders from the parallelization and distribution concerns. COMPSs considers applications as composites of invocations to pieces of software encapsulated as methods called Core Elements (CE). To manage the parallelism inherent in the application, the framework instruments the application and replaces CE invocations by calls to a runtime system to execute them atop the infrastructure. Also, accesses to data generated on remote nodes need to synchronize their value before being used. The following subsections introduce the programming model and the architecture of the runtime system, highlighting those aspects relevant to support executions on Fog-to-Cloud environments.

4.1 Programming Model

For developing applications, programmers write their code in a sequential fashion with no references to any COMPSs-specific API or the underlying infrastructure. At execution time, calls to CE methods are transparently replaced by asynchronous tasks whose execution is to be orchestrated by the runtime system. To select which methods become a CE developers define an interface, called Core Element Interface (CEI), where they declare those methods along with some meta-data in the form of annotations. To pick a method as a CE, the programmer annotates the method declaration on the CEI with *@Method* indicating the class containing the method implementation. The code snippet in Fig. 2 contains a simple COMPSs application example. Figure 2(a) shows the sequential code of the application which runs N simulations and selects the best one. As shown in the CEI presented in Fig. 2(b), only two methods are chosen as CE: simulate and getBest. For the runtime system to determine the dependencies between CE invocations, developers specify how each CE operates on the accessed data (its parameters) by adding (*@Parameter*) annotations indicating the parameter type and directionality (in, out, in-out).

4.2 Runtime Library

The main purpose of the runtime toolkit is to orchestrate the execution of CE invocations (tasks) fully exploiting the available computing resources (local devices or remote nodes) guaranteeing the sequential consistency. Applications

```java
public Sim checkSimulation(int N) {
    Sim best = null;
    for (int i=0; i < N; i++) {
        Sim s = new Sim(...);
        s.simulate();
        best = Sim.getBest(best, s);
    }
    return best;
}
```

(a) Application main code

```java
public interface SampleCEI {
    @Method(declaringClass="Sim")
    void simulate();

    @Method(declaringClass = "Sim")
    Sim getBest(
        @Parameter(direction = IN)
        Sim s1,
        @Parameter(direction = IN)
        Sim s2
    );
}
```

(b) Core Element Interface

Fig. 2. Sample application code written in Java

share computing resources and, potentially, data values; therefore, the runtime library is twofold. The front-end of the runtime, instantiated in every application, manages the private aspects of the applications: monitors accesses to private pieces of data, such as objects, and detects the CE invocations. The back-end manages all the aspects that the application can share from computing resources (CPU, GPU, nearby nodes or VM instances on the cloud) to data (currently only files, but we envisage to manage accesses to databases and Content Providers). Since all front-ends contact the same instance of the back-end, it is deployed as an Android service running in an independent process. Figure 3 contains a detailed diagram of the runtime architecture.

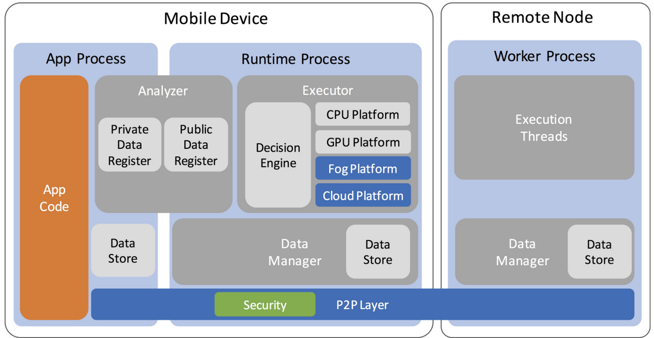


Fig. 3. Runtime system architecture

To monitor the data accessed from each task and the data dependences among task, the runtime processes the parameters of each task upon its detection on the *Analyzer* component. The *Private* and *Public Data Registers*, respectively located on the front-end and back-end of the runtime, record the accessed data values and assign a unique identifier for each version of the value. Once all the accessed values are registered, the *Analyzer* submits the task to the *Executor*, the component of the runtime that manages the resources.

To decide which resources host the execution of a task, the runtime is based on the concept of *Computing Platform*: a logical grouping of computing resources capable of running tasks. The decision is made on the *Decision Engine (DE)*, which is agnostic to the actual computing devices supporting the platform and the details to interact with them. The *DE* requests to each of the available platforms –configured by the user beforehand– a forecast of the expected end time, energy consumption and economic cost of the execution. According to a configurable heuristic, the *DE* picks the best platform to run the task and requests its execution; the selected platform is responsible for monitoring the data dependencies of the task and scheduling the execution of the task on its resources. Currently, there exist three different implementations of *Computing Platform* according to the nature of the computing devices composing it. *CPU Platform* manages the execution of tasks implemented as regular Android methods on the multiple cores of the mobile device CPU. *GPU Platform* executes tasks implemented as OpenCL code on the embedded GPU. Finally, the third implementation, *Remote Platform*, offloads the execution of methods to remote resources. For the runtime to properly exploit Fog-Cloud environments, users can instantiate four platforms: a *CPU Platform*, a *GPU Platform* and two *Remote Platforms*: the *Fog Platform* encapsulating the low-latency remote resources (West-bound) and the *Cloud Platform* representing those VM instances deployed on Cloud Providers (North-bound).

For sharing data across platforms, the runtime hosts a data repository: the *Data Manager (DM)*. Through a publish-subscribe mechanism, the *DM* asynchronously provides information and values of the accessed datums using the unique IDs assigned by the *Analyzer*. Computing Platforms lean on the *DM* for monitoring the data dependencies. When the *Executor* designates a platform to run a task, the platform subscribes for the existence of all the input datums; upon the publication of the creation of any of them, the *DM* forwards the notification to the platform. Once the platform realizes that all of them exist, it plans the execution of the task on its resources and queries the *DM* for the value of each datum. At the end of the task execution, the platform publishes the existence of the output datums and stores their value on the *DM*.

To uncharge the mobile device from the computational load of orchestrating the remote resources, Remote Platforms organize them as a peer-to-peer network. Each node of the network runs a worker process persistently listening to the network for task submissions; these processes are able to autonomously handle the execution of the task on the local computing devices. To ease the management of data dependencies, worker nodes subscribe for and publish information and values of the datums accessed by the tasks on the *DM*, whose content –either information or values– is consistently distributed across the whole infrastructure. The local instance of the *DM* is responsible for fetching the value from any hosting remote node.

The following subsections delve into detail in other features of the runtime specially significant for F2C environments: security on network communications and network-disruption tolerance.

Securing Communications. Data used on Fog applications is likely to be privacy-sensitive (pictures, videos, geolocation, etc.) and networks interconnecting the mobile device with other resources –either on the same layer or the Cloud – tend towards untrustworthiness.

To protect applications from eavesdroppers, the runtime has a security mechanism that provides communications with confidentiality, integrity and authentication. For its implementation the runtime leverages on the Generic Security Services API (GSSAPI) [2], an IETF standard API to access security services, so developers create secure applications while avoiding security-vendor lock-in.

Besides defining a common interface, GSSAPI also settles an operating model where both ends negotiate a secure context – authenticate themselves and agree on the mechanisms for data ciphering and integrity – before transferring any information. Upon the establishment of the context, GSSAPI processes (wraps) the messages and opaques their content returning token thats can be securely shipped to the other end. Although GSSAPI defines the format of the exchanged tokens and its content – actually, the security framework does –, it does not establish nor provide any transmission mechanism. Therefore, applications invoke GSSAPI to wrap a value and obtain a token to ship to the other end. Upon the reception of a token, the receiver invokes GSSAPI to unwrap the token and obtain the original content of the message. In our case, COMPSs uses the Java NIO library to transfer tokens over TCP sockets.

Although GSSAPI provides the infrastructure with an interoperable approach to secure communications, currently there is no generic mechanism to get the required credentials from the Authentication Server automatically. Application users need to manually set up the Authentication Infrastructure and authenticate all the nodes to obtain their credential beforehand. However, we consider this to be the foundational stone to build a platform with Authentication, Authorization and Accounting based on Federated Identity and Single Sign-On. Our ultimate goal is to build a global service where local institutions offer nearby computing resources (Fog nodes) where to offload computation securely from mobile devices belonging to users from other organizations within their federation. Using the same credential, users could always turn to VM instances deployed on the Cloud to obtain additional computing power.

Network Disruption Tolerance. A consequence of the high mobility of Fog devices is instability on the network conditions. Fog devices are likely to face Wi-Fi network handovers, changing the used network interface between Wi-Fi and mobile data, switching to different mobile network protocols (GPRS, EDGE, UMTS, HSPA, LTE, etc.) and eventually the device can disconnect from the network. Controlling all the possibilities is main challenge to tackle not only for Fog Computing but also for IoT and MANET frameworks.

As a first approach to solve the problem, we focused on the device running the application (master) and considered a network disruption that isolates it while the rest of the infrastructure stays up and online. Eventually, the device might

reconnect to the same network recovering access to the same pool of workers, but using a different IP address.

To tolerate short, sporadic network disruptions, the master sends a message to every worker node upon the reconnection indicating its new address. Upon its reception, worker nodes update every reference to the master node with the new IP and re-start any interrupted transaction – transfer of a value or submission of internal COMPSs command.

On long-lasting disruptions, worker nodes should keep progressing on the computation despite the isolation. In the case of reconnection, workers autonomy reduces the impact of the network failure on the performance of the application. Upon the broadcast reconnection notification, *DM* instances synchronize their content, thereby all the components of the infrastructure become aware of the progress done by the other part.

On the other end, the master device should produce the expected result even if the network connection is never re-established. Therefore, the master may need to run all the pending tasks, even those already offloaded. Probably, some input values for a pending task are the output of an offloaded one and they are not likely to be on the master; hence, the value must be computed locally by running the producing task. This mechanism results in a backtracking process that only stops when all the input data required by a task exists in the device. So the runtime can go back in the execution, it keeps track of all the detected tasks and builds a data-dependency graph. Tasks can not be removed from the graph until the master never needs to re-execute them again – i.e., all its output values have a replica on the master or neither the main application nor any task use them.

Upon the detection of a network breakdown, the *Executor* prioritizes the execution of the not offloaded tasks whose input values are already on the mobile. When there are not enough tasks to use all the computing devices within the mobile, the *Executor* picks one of the not offloaded tasks and triggers the backtracking process to generate the missing input values for the task. Finally, once all the not offloaded tasks have started their execution, it runs pending offloaded tasks (if necessary, re-computing the input data values).

To prevent this backtracking process from re-running tasks already executed on the workers, the runtime transfers the output values back to the mobile to establish checkpoints. To avoid transferring every remotely generated value, the runtime picks some strategic values splitting the graph – currently, fixed-size partitions according to the chronological order of task generation – and analyzing each partition for all the output values of the block that succeeding partitions might use. The master fetches these values upon their creation; once the master has all the output values from a block, it removes the whole block from the graph.

5 Experiments

As a proof of concept that validates the feasibility of the described architecture and the proper behavior of the runtime system, we have ported the HeatSweeper

application and executed it on a smartphone that offloads parts of the computation to nearby and remote devices. The following subsections introduce the application, describe the testbed used to conduct the tests, and present the obtained results in terms of execution time and energy consumption.

5.1 Application: HeatSweeper

HeatSweeper is an application to find the optimal placement of 1-to-N heat sources on the surface of a solid body to reduce the time to heat up its whole surface to a certain temperature. Its algorithm consists on an intensive search looking for the best combination of 1-to-N locations for the heat sources, and relies on two different solvers to simulate the heat diffusion based on the Jacobi (used on the tests) and Gauss-Seidel equations.

On the COMPSs version, the application defines two CEs. *Simulate* encapsulates within a task the simulation of the heat transfer over a surface for a specific combination of locations and generates a report summarizing the simulation. In a second phase, the application compares all the simulation reports to select the best combination. To compare two reports the application defines the second CE: *getBest*. On the conducted tests, the application considers 25 different spots of the surface where to locate the heat sources; simulations stop after 10,000 steps if the surface has not reached the desired temperature before. With this configuration, the application generates 325 *simulate* tasks and 323 *getBest*.

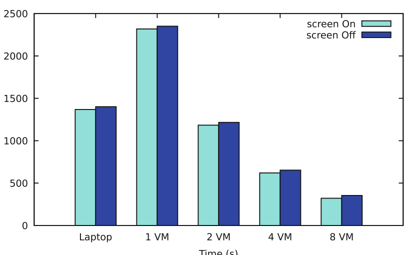
5.2 Testbed

HeatSweeper runs on a OnePlus One (OPO) smartphone, equipped with a Krait 400 quad-core processor at 2.5 GHz and 3 GB of RAM memory. As mentioned above, the defined tests consider two different infrastructures where to offload task. For the fog case, the smartphone offloads the computation to a laptop equipped with an Intel i7-2760QM quad-core processor at 2.40 GhZ and 8 GB of RAM memory. The mobile device connects to the laptop via an 802.11g wireless network. On the Cloud scenario, the phone uses as surrogates up to eight quad-core VM instances deployed on an OpenNebula cloud. The physical nodes supporting the Cloud have six-core Intel Xeon X5650 at 2.67 GHz processors and 24 GB of memory each. Cloud nodes are interconnected through a Gigabit Ethernet network, while the connection between the mobile device and the surrogates goes through the Internet and has an 85.5 ms RTT.

5.3 Results

Measurements of the elapsed time to execute a *simulate* task highlight the performance differences among the devices composing the infrastructure. Running a task on the smartphone takes around 288 s. When the screen of the device is Off, Android reduces the frequency of the processor to a 10% of its regular value. This increases the execution time to 6,794 s; however, it also reduces the

power consumption of the processor from 1.4 W to 0.16 W. Executing the same simulation on the laptop and on a Cloud instance takes 16 and 29 s, respectively. The execution time of running a *getBest* task is negligible. Overall, running the application on the phone – with its screen on – takes 99,641 s (more than 27 h), and it forces the smartphone to stay plugged in and drawning power.



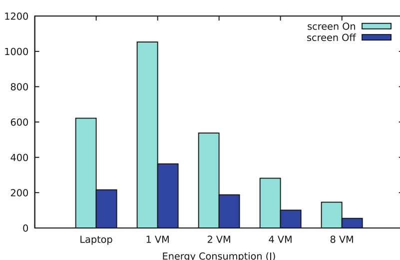


Fig. 4. Elapsed time and energy consumption of executing HeatSweeper according to the surrogate platform

Charts in Fig. 4 illustrate the elapsed time and the energy consumption measured when executing HeatSweeper in the different platforms. Offloading parts of the computation to resources with higher computing capabilities allows a significant reduction of either the execution time and the energy consumed by the smartphone. The laptop is the most powerful resource, and offloading tasks to it reduces the execution time to 1368 s. Although the execution using a single VM instance achieves a worse execution time than the laptop, the cloud provides the runtime with higher amount of resources. The more VMs the application uses, the lower the execution time is; using all eight instances, the application only takes 321 s to finish. Obviously, offloading tasks saves to the master the energy spend on the processor to compute them; however, keeping the mobile on and transferring data through the network maintains part of this consumption. For the 8-VM case, the smartphone consumes up to 146 J. The screen of the devices is responsible for a significant part of this energy; with the screen Off, the application reaches a consumption of less than 55 J. The impact of switching the screen Off on the execution time is not significant. The frequency reduction only affects to the communications and task management performed by the runtime; it does not affect the actual computation of the tasks since remote resources keep their performance.

6 Conclusions and Future Work

This paper presents the preliminary design of an architecture for a programming framework that enables distributed computing on Fog-to-Cloud environments. The baseline of this architecture is COMPSs, a programming tool that has been

successfully applied to port applications and parallelize their execution on clusters, grids and clouds. The COMPSs runtime, as explained in this work, has been extended to be executed on Android devices equipped with CPUs and GPUs and to offload tasks to clouds backends or other fog devices available in the same network. An important improvement and contribution is the design of a new distributed data management mechanism that allows to efficiently share information about data across the different platforms. A checkpointing strategy has been also implemented to make the runtime resilient to network fluctuations and disruptions, allowing to resume the computation after a working node failure. Finally, security mechanisms have been added to secure the communications between the main runtime process and the worker nodes. The results of the tests demonstrate that the refactoring and extensions to the runtime, do not affect the performance of the execution when offloading the tasks to remote nodes.

Future work includes several optimizations as the implementation of a distributed scheduling policy, the improvements on the security mechanisms in order to add authentication at application level, extensions to the resource management to allow elasticity on the Cloud and to use dynamically appearing resources as workers.

Acknowledgment. This work is partly supported by the Spanish Ministry of Science and Technology through project TIN2015-65316-P and grant BES-2013-067167, by the Generalitat de Catalunya under contracts 2014-SGR-1051 and 2014-SGR-1272, and by the European Union through the Horizon 2020 research and innovation programme under grant 730929 (mF2C Project).

References

1. Apache Mesos: http://mesos.apache.org/. Accessed 12 May 2017
2. Linn, J.: Generic security service application program interface version 2, update 1. Internet Requests for Comments, RFC 2743. RFC Editor, January 2000. ISSN 2070-1721
3. OpenFog Reference Architecture. https://www.openfogconsortium.org. Accessed 12 May 2017
4. Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: Proceedings of the Sixth Conference on Computer Systems, EuroSys 2011, pp. 301–314. ACM, New York (2011). https://doi.org/10.1145/1966445.1966473
5. Hong, K., Lillethun, D., Ramachandran, U., Ottenwälder, B., Koldehofe, B.: Mobile fog: a programming model for large-scale applications on the internet of things. In: Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing, MCC 2013, pp. 15–20. ACM, New York (2013). https://doi.org/10.1145/2491266.2491270
6. Kemp, R., Palmer, N., Kielmann, T., Bal, H.: Cuckoo: a computation offloading framework for smartphones. In: Gris, M., Yang, G. (eds.) MobiCASE 2010. LNICST, vol. 76, pp. 59–79. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29336-8_4

7. Kosta, S., Aucinas, A., Hui, P., Mortier, R., Zhang, X.: Thinkair: dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: 2012 Proceedings IEEE INFOCOM, pp. 945–953, March 2012. https://doi.org/10.1109/INFCOM.2012.6195845
8. Lordan, F., Tejedor, E., Ejarque, J., Rafanell, R., Alvarez, J., Marozzo, F., Lezzi, D., Sirvent, R., Talia, D., Badia, R.M.: Servicess: an interoperable programming framework for the cloud. J. Grid Comput. **12**(1), 67–91 (2014). https://doi.org/10.1007/s10723-013-9272-5
9. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)