2018 Third ACM/IEEE Symposium on Edge Computing

Amino - A Distributed Runtime for Applications Running Dynamically Across Device, Edge and Cloud

Ying Xiong, ying.xiong1@huawei.com; Donghui Zhuo, terry.zhuo@huawei.com; Sungwook Moon, sungwook.moon@huawei.com; Michael Xie, Haibin.Michael.Xie@huawei.com; Isaac Ackerman, isaac.ackerman@huawei.com Quinton Hoole, quinton.hoole@huawei.com;

Seattle Cloud Lab, Huawei R&D USA, Bellevue WA

ABSTRACT: This paper presents a framework and runtime system, Amino, for developing and executing distributed applications in highly dynamic computing environment consisting of cloud resources, edge nodes and/or devices such as phones and smart cameras. This work is based on Sapphire [1] - a general-purpose distributed programming platform. In Sapphire, application objects (called Sapphire Objects) run inside kernel servers (KS), and Kernel server instance runs on every device or cloud node. Between Kernel Server and an application object is a layer called Deployment Manager (DM). Inbound and outbound communications to/from Sapphire objects will be intercepted and processed by deployment managers. Each DM provides one specific distributed system capabilities, e.g. caching, resource leasing, replication, data partitioning etc. Developers selectively choose DMs to manage Sapphire objects. As part of this work (Amino), we re-implemented and extended Sapphire platform to support Sapphire objects written in multiple languages and to support attaching multiple DMs to a Sapphire object for increased distribution capabilities. Finally, in the work, we introduced a code offloading design for dynamically moving application objects between devices and cloud servers at runtime to optimize a user specified objective, e.g. to reduce latency or to save energy consumption.

KEYWORDS; Edge Computing, Distributed Systems, Cloud Computing and Edge Programming Framework

I. INTRODUCTION

Edge computing is a new paradigm to provide computing capabilities at the edge of internet in close proximity to mobile users or devices. Unlike cloud computing which consolidates computing capacity into datacenters, edge computing drives computing toward decentralized architecture, which presents runtime and management challenges for distributed applications running across device, edge and cloud.

Today, most distributed applications are designed with server components running in cloud data center and client components running on devices with. The separation between server components and client components is introduced at design phase and the design is "fixed", in terms of which component runs where (device, edge or cloud). As more applications, such as AI, stream data analytics and IoT apps, want to leverage computing resources both on the edge and in the cloud, there is a need to dynamically schedule application components to run at the right place at right time to optimize the overall application performance and availability. Building large scale and highly available distributed applications is difficult. Developers are forced to deal with complex distributed problems like concurrency, partitioning, consistency, fault tolerance, etc. Building distributed edge applications across device, edge and cloud is even more challenging, due to the great variance of underlying heterogeneous hardware, the higher device failure rate, network instability and restricted resource constrains.

In this paper, we will introduce Amino, a distributed runtime system built on top of Sapphire programming framework [1] developed by the system group at University of Washington. Amino provides a unified runtime layer across device, edge and cloud. Application objects written in different programming languages can run and communicate each other on this layer. Amino uses GraalVM [2] to manage multi-language objects. Precisely, Kernel Server in Amino uses GraalVM Polyglot APIs to create, manage and invoke multi-language objects. Supported languages include JavaScript, Python, Ruby, JVM based languages such as Java, Scala and Kotlin, and LLVM based languages such as C and C++. Amino also provides a collection of built-in plugins (DMs) for common distributed system tasks needed by an application, such as fault-tolerance, code offloading, and caching, etc. Developers choose plugins by specifying policies to meet their distribution requirements. For example, developers can enable code offloading on a face-detection object by providing an offloading policy that contains offloading conditions - when to run the face-detection object on device and when to run the same object in the cloud. Amino will monitor the application and perform offloading when conditions are satisfied. Amino allows developers to focus on business logics without dealing with complex distributed systems problems which increases software quality and development productivity.

978-1-5386-9445-9/18/$31.00 ©2018 IEEE
DOI 10.1109/SEC.2018.00046

II. RELATED WORKS

Application availability, state partitioning, replication, and code offloading are problems that have been widely studied in the field of distributed computing. Comparing with conventional distributed application, the development of edge applications faces extra challenges due to the issues of strict resource constraints, network instability and great mobility of edge servers and/or devices. This section tries to analyze and summarize the related works with respect to distributed programming model and unified runtime across edge devices, edge servers and cloud resources.

Most of previous works focused on application task partitioning and code offloading between devices and cloud resources. Cuckoo [3] framework integrates Android operating system service component with Eclipse development tool to provide computation offloading for smartphone applications. Cuckoo generates the stubs for accessing service components, which are replaced by invocations to the Cuckoo framework that decides, at run-time, whether to run the service on the local device or a remote implementation. However, Cuckoo does not provide unified programming model for applications running cross edge devices and cloud servers. It supports only Java on Android smartphones and provides only offloading capability. It does not provide a framework to solve complex distributed system problems as a whole. Mobile Fog [4] is a high level programming model for developing geographically distributed, large-scale, and latency-sensitive applications. Its goal is to allow applications ~~to~~ dynamically scale based on their workload using on-demand resources in the fog (edge nodes) 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. While Mobile Fog programing model simplifies the development on heterogeneous servers and devices distributed over a wide area, it requires application developers to partition their application components (Mobile Fog processes) onto these devices and/or cloud instances, and the partitioning is not dynamic at runtime. It also does not solve other distributed system problems such as leader selection, replication and caching. CloneCloud [5] offers elastic execution framework at thread level between mobile devices and cloud VMs. 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. The partitioning is offline and pre-computed partitions are stored in a database. At runtime, the distributed execution mechanism picks a partition from the database and implements it via a small and fast set of modifications of the executable before invocation. The drawback of CloneCloud is that it still requires developers to manage threads. Similar to others, CloneCloud is not designed to solve distributed system problems other than offloading. Finally, COMPSs [6] [7] [8], a programming model for distributed computing and associated runtime, provides a framework to develop and run your application components across edge devices and cloud servers. COMPSs considers applications as composites of invocations to pieces of software encapsulated as methods called Core Elements (CE). At execution time, calls to CE methods are transparently replaced by asynchronous tasks whose execution is to be orchestrated by the runtime system, fully exploiting the available computing resources (local devices or remote nodes) and guaranteeing the sequential consistency. On the other hand, COMPSs 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. The main differences between COMPSs and Amino are that Amino chooses objects as the software pieces that can be instantiated on local devices, edge nodes or cloud servers at runtime, and each object (called Sapphire object) can be attached to one or more DM (Deployment Manager) for its distributed capabilities. For example, if a Sapphire object is attached to the high availability DM, Amino will create multiple instances (replica) of the object at runtime on different computing resources. The object state will be synchronized among these instances, and if the primary object instance crashes, one of the replica will be used for subsequent method invocations on this object, thus, providing HA capability to this object.

III. ARCHITETCURE AND PROGRAMMING MODEL

The base management unit in Amino distributed system is Sapphire object shown as circles in **Figure 1**. Sapphire objects are remotely accessible distributed objects. Developers first build the application as a single object-oriented program with their familiar OO programming model. They then break the application into distributed components by declaring a set of local objects to be Sapphire Objects.

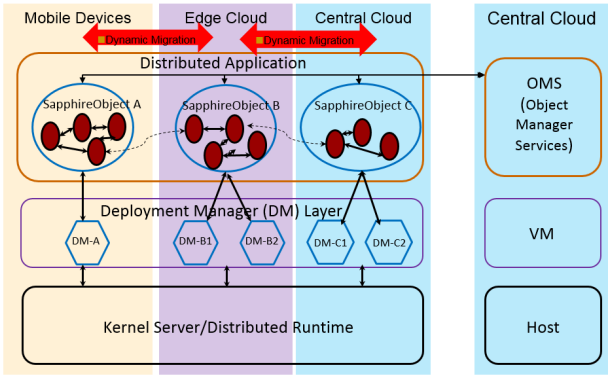

Figure 1: Layered Architecture of Amino

The dots inside each Sapphire object represents local objects with no remote invocation and distributed capabilities. The solid arrow lines between dots are method invocations between local objects. The dashed arrow lines between circles are remote method invocations between Sapphire objects. Methods on local objects can only be invoked locally by objects residing on the same host. Sapphire objects however may have remote methods which can be invoked by objects residing on different hosts. To convert a local object to a Sapphire object, a developer simply makes the class

implement the "SapphireObject" interface, as shown in Listing 1 below:

```
public class TodoList implements SapphireObject {
    String name;
    ArrayList<Object> toDos;

    public TodoList(String name) {
        toDos = new ArrayList<>();
        this.name = name;
    }

    public String addToDo(String todo) {
        toDos.add(todo);
        return "OK!";
    }
}
```

Listing 1: Define a Sapphire Object

At runtime, Sapphire objects are managed by Kernel Server and OMS for distribution and invocations. Kernel Server and OMS are two important components in Amino architecture, and are described in the next section.

A. Kernel Server, OMS and DM

Kernel Server provides runtime environment for Sapphire objects. Each host runs a Kernel Server instance, which exposes a set of remote API. Amino assumes that any Kernel Server instance can invoke the remote API on any other Kernel Server regardless where the instance lives. Object Management Service (OMS) keeps track of the locations of all Sapphire objects. Sapphire object must be created with a special Sapphire helper method Sapphire.new_(). Upon Sapphire object creation, the helper method will generate a globally unique ID for the Sapphire object, and register the object in OMS. OMS provides API to search Sapphire objects. Given a Sapphire object ID, OMS can tell the IP of the host on which the Sapphire object runs. Whenever a Sapphire object is moved or deleted, OMS will be updated accordingly.

OMS server contains a Kernel Object Manager which keeps track of the mapping between kernel object ID to the IP address of the kernel server in which the object runs. Given a kernel object ID, a client can call OMS server to get the IP of the host where the object runs. Kernel server contains an object manager which keeps track of the mapping between kernel object ID and the references of the object.

Figure 2 below depicts the design and interaction between Kernel Server and OMS.

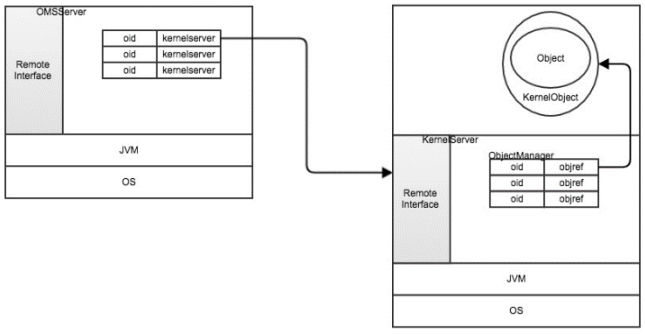

Figure 2: Kernel Server and OMS in Amino

Between Kernel Server and application Sapphire objects in Amino architecture shown in **Figure** 1 is a layer called Deployment Manager (DM). DMs provide distributed system capabilities for Sapphire objects. To associate a DM to a Sapphire object, simply specify the DM when defining the object, as shown in the following Listing (Listing 2). In this example, "TodoList" is a Sapphire object with one associated DM(LoadBalanceMasterSlaveSyncPolicy). This makes "TodoList" a highly available object at runtime, with one primary instance (Master), and multiple replica (Slaves) for HA. The DM and runtime takes care of monitoring master and slave objects, syncing object states and promoting slave to master once the master crashes.

```
public class TodoList implements SapphireObject<LoadBalancedMasterSlaveSyncPolicy> {
    String name;
    ArrayList<Object> toDos;

    public TodoList(String name) {
        toDos = new ArrayList<>();
        this.name = name;
    }

    public String addToDo(String todo) {
        toDos.add(todo);
        return "OK!";
    }
}
```

Listing 2: Associate DM to A Sapphire Object

Every DM has three components: a proxy, an instance manager, and a coordinator. When a user creates a Sapphire object, he/she can associate a DM to the Sapphire object. During the creation of the Sapphire object, Amino will inject codes into the stub of the Sapphire object, in which case any method invocation on the Sapphire object will first be processed by the proxy, instance manager and the coordinator of the DM before reaching the actual Sapphire object. Multiple DMs can be associated to a Sapphire object and each DM provides a specific distributed system functionality. Most popular DMs are Scalability DM, Transaction/Replication DM, Caching DM, and Offloading DM, which will be discussed later in the paper.

Sapphire objects on different hosts communicate each other through generated DM stub classes. **Figure 3** below describes the communication through Kernel Server.

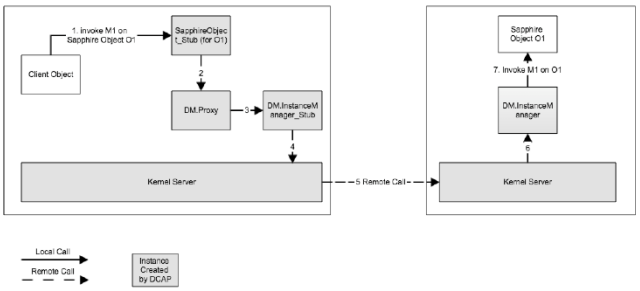

Figure 3: Sapphire Object Communication through Kernel Server

B. Multi-Language Support

Cloud native applications are normally designed with micro-service architecture, and are implemented in multiple languages. Supporting Sapphire objects written in different languages is a requirement for Amino project, for which, we use open source GraalVM[2] technology to support multiple languages for Sapphire objects. GraalVM provides the virtualization layer representing programming languages, allowing the execution of guest programming languages (for Sapphire objects), namely JavaScript, Ruby, R, Python and LLVM bit-code, in the same runtime as the host JVM-based application. The host language and guest languages can directly interoperate with each other and pass data back and forth in the same memory space. In this section, we introduce how we design and integrate GraalVM into Amino runtime system. For details of GraalVM itself, please refer to the links in reference section of this paper. Figure 4 describes the design of GraalVM solution in Amino runtime, with an example of an application written in Ruby invoking methods on a Sapphire object written in JavaScript.

Both Kernel Server and OMS in the figure run inside GraalVM. Kernel Server uses GraalVM Polyglot API to create Sapphire objects written in multiple languages as shown in step #4 in the diagram.

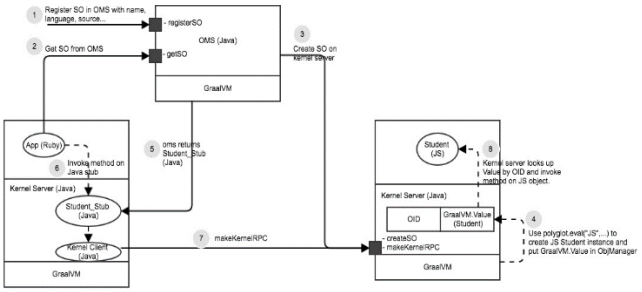

Figure 4: Multi-Language Support with GraalVM in Amino

When OMS creates and registers a Sapphire object written in one of the supported languages, it creates a java stub for the object, and passes the stub to the client as shown in step #5 in the diagram. The end-to-end process in multi-language support at execution time is like this:

1) Sapphire object provider calls OMS to register new sapphire objects. Upon registration, sapphire object provider needs to provide the name of the object, the programming language in which the object is written, and the artifact that contains the object implementation.
2) OMS sends request to kernel server to trigger sapphire object creation. Kernel server uses Graal API to create object instance.
3) To use a sapphire object, an application needs to get the stub of the sapphire object by querying OMS with sapphire object name. Amino provides helper classes in different programming languages to assist developers in discovering sapphire objects and retrieving sapphire object stubs.
4) Client (written in one language) invokes methods on a stub (returned from OMS) to interact with the sapphire object. Even though stubs are java classes, they appear as Graal Value instances in non-java applications and can be consumed directly by non-java applications.
5) The Kernel Server on the client host intercepts the method invocations on stubs and consults OMS to determine the host where the actual Sapphire object runs
6) The kernel Server serializes parameters into bytes and makes a RMI call to the target Kernel Server where the real object runs
7) The Kernel Server on the target host de-serializes bytes back into parameters, looks up the Sapphire object (written in another language), and invokes the methods on the object via Graal API.

Please note that a) Kernel Server, OMS and all DMs are still in Java; b) Sapphire objects can be written in different languages, but corresponding stubs are in Java; c) Kernel Server uses Graal API polyglot.eval() to create Sapphire object instances, and saves sapphire object as Graal Value instance (polyglot.Value) in Object Manager; and d) It is possible for Object Manager to store server policies that refer to the Graal value; e) Kernel Server uses Graal API (polyglot.Value.getMembers(…).execute()) to invoke methods on the Sapphire object.

IV. MULTIPLE DM CHAINING

As mentioned previously, Deployment Manager (DM) in Amino provides distributed capabilities to Sapphire objects through plugin mechanism. In Sapphire [1], only one DM can be associated with a Sapphire object. In real world, it is desirable to associate multiple DMs with one Sapphire object. For example, Distributed Hashing Table (DHT) is a deployment manager which provides request partition to sapphire objects. Based on configurations, DHT is able to maintain multiple instances of a sapphire object, and route requests to one specific instance based on the hash value of the request. Suppose we have a key value store class. By applying DHT, we get a partitioned key value store which is able to scale out smoothly with more instances when workload increases. Consensus DM is a deployment manager which provides fault tolerance to sapphire objects. When Consensus DM is applied to a sapphire object, the DM will automatically create three instances of the sapphire object, use Raft protocol to keep them in consistent, and provides fail over when one instance dies. By chaining DHT DM and Consensus DM together, we can add both scalability and availability on a stateful Sapphire object where DHT provides horizontal scaling through partitioning and Consensus DM provides fault tolerance. Similarly, we can combine Retry DM and Load Balance DM together to provide scalable at-least-once RPC support on a stateless Sapphire object. The following diagram (**Figure** 5) illustrates

how multiple DMs work together. In this diagram, a client DM can either talk to the client of another DM, or talk to the stub of a server DM, and the last client DM will always talk to the stub of its server DM. For this example, DM1.C talks to DM2.C which is the client of DM2, but DM3.C talks to DM3.S_Stub which is the stub of a server DM.

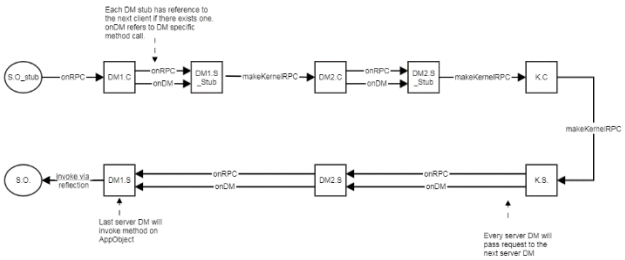

Figure 5: Multi DM Chaining Model

In the model illustrated in the diagram above, DMs on the client side are chained together and DMs on the server side are also chained together. The model allows Amino to support many combinations of DMs where they make sense, e.g. Retry plus Locking Transaction, LoadBalance plus Retry, etc.

V. DESIGN OF CODE OFFLOADING DM

Offloading is a promising way to improve performance as well as reducing energy consumption by executing some parts of the app on remote server. Published papers have shown that code offloading to remote resources leads to energy optimization and application performance. For Amino project, we are implementing a DM specific to code offloading and can be associated with Sapphire objects to be offloaded to other devices or cloud servers. The work is still in process and we started with the basic scenario that all Sapphire objects are assume to belong to one application. The basic design is:

1) For each Sapphire object associated with offloading DM, OMS generates client and server stubs that calculate the execution time of each method invocation on the object instance running on a given host (edge device or cloud server), t(*Object, host*)

2) For each object invocation on a server from a client, the generated DM stubs measure the latency and bandwidth (transmit time) of the link between calling client to the object host, l(object, client, host)

3) For each Sapphire object (configured with Offloading DM) invocation, the generated server DM stub measures the cpu, memory and IO resource consumption as well as energy consumption on a given host, r(object, host) = f(cpu, mem, disk io, network io) and e(object, host) = f (r), where r() and e() are the resource utilization and energy consumption of a given object on a given host respectively.

4) Collected data from DM Stubs are sent to OMS for offloading scheduling algorithm (Offloading Scheduler),

5) For a given client and a set of Sapphire objects to be invoked, offloading algorithm in OMS calculates the offloading value for very host in Amino cluster:

$$V(Host_i) = \sum_{k=1}^{n} \big(t(Obj_k,\ Host_i) + l(Obj_k,\ Client, Host_i) + e(Obj_k,\ Host_i)\big)$$

6) For a given sliding time windows, GroupPolicy DM in OMS will make a decision where (which host) to create and run the Sapphire object, based on the values calculated for all hosts. The actual algorithm will discussed in the future paper. The decision making is based on past experience. If no data is available, objects will run simultaneously on all hosts (edge device or cloud server) to collect data needed.

. .

VI. EXPERIMENTS

As part of Amino project and for validating the Amino architecture and design, we have ported a few applications and executed them on Amoni runtime platform. In this section, we describe two applications (Go Game and License Plate Recognition) and show preliminary test results.

A. Go Game

The first test application is Go Game, a Pachi open source Go engine written in C++. The engine uses both algorithmic and neural networks approaches to calculate moves. The skill level of the Go engine is approximately amateur 1 duan. We ported neural networks on the Android (previously running on desktop only), and we "sapphirized" the game with a Sapphire object for calculating the next move in the game. The Sapphire object runs on the device by default. After a few moves, Amino moved the object to run on cloud servers, and we notice that the performance was **5 times** faster than the calculation on device. More importantly, the application developer enjoyed code offloading capability provided by Amino without writing any code.

B. License Plate Recognition

The second application is the OpenAlpr open source engine written in C++ for license plate recognition, which is a Sapphire object in our test. The Sapphire object is tested on a host with the configuration shown in the **Figure** 6.

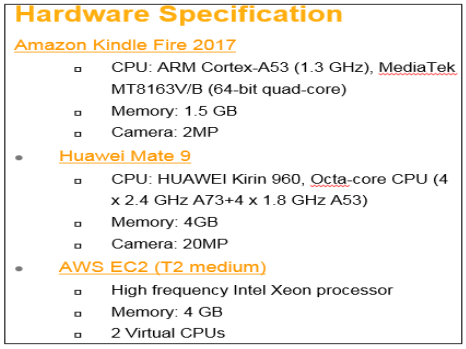

Figure 6: Hardware Configuration in Experiment

The application was installed and run on an Android phone, and when the phone took a picture of a license plate, the Sapphire object performed the license plate recognition function. As a result, it displays a list of the recognized plates with confidence levels. When the picture contains few license plates, the recognition object run on the phone without significant delay. When the picture contains more plates, doing recognition on device becomes very slow, therefore Amino moved the Sapphire object to a remote server in the cloud to improve performance.

As shown in **Figure** 7 below, the same Sapphire object can run on Kindle device, Android phone (Mate 9) and on the AWS Cloud with different cost. The algorithm run time is the highest on Kindler device due to resource constraints, while the upload time is the highest on AWS Cloud, due to network transmission. However, moving Sapphire object to the cloud still results in best performance in license plate recognition.

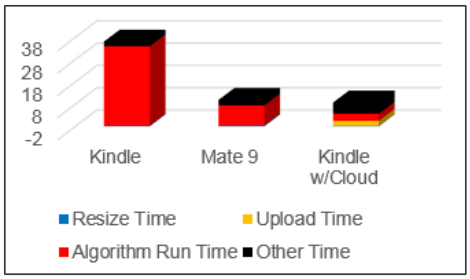

Figure 7: Cost Breakdown on Different Hosts

VII. Conclusion, Futurework and Acknowledgment

We have presented a programming framework and runtime system for distributed computing across device, edge and cloud, based on original Sapphire platform. We re-implemented, enhanced and extended the platform to support multi-language and multi-DM for distributed Sapphire objects. Our goal is to build the platform as commercial ready product or service in our public cloud for real customer use cases. Future work includes developing additional DMs and robust offloading algorithm to further optimize the object placement. We very much appreciate the help from System group at UW, and special thanks to Dr. Irene Zhang for her valuable support in this work.

References.

[1] Irene Zhang, Adriana Szekeres, Dana Van Aken, and Isaac Ackerman, "Customizable and Extensible Deployment for Mobile/Cloud Applications" In the Proc. of the 11th USENIX Symposium 2014

[2] GraalVM, a high performance polyglot VM, https://www.graalvm.org/; https://www.graalvm.org/docs/; https://github.com/graalvm/

[3] Kemp, R., Palmer, N., Kielmann, T., Bal, H.: Cuckoo: A Computation Offloading Framework for Smartphones, pp. 59{79. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-29336-8 4.

[4] Hong, K., Lillethun, D., Ramachandran, U., Ottenwalder, 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. pp. 15{20. MCC '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2491266.2491270.

[5] 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. pp. 301{314. EuroSys '11, ACM, New York, NY, USA (2011).

[6] 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. Journal of grid computing 12(1), 67{91 (2014).

[7] Lordan, Francesc; Lezzi, Daniele; Ejarque, Jorge; Badia, Rosa : "An architecture for programming distributed applications on Fog to Cloud systems". Euro-Par 2017, Parallel Processing Workshops

[8] Rosa M. Badia, Javier Conejero, Carlos Diaz, Jorge Ejarque: "COMP Superscalar, an interoperable programming framework", SoftwareX Volumes 3–4, December 201