Journal of Grid Computing manuscript No.
(will be inserted by the editor)

Towards Mobile Cloud Computing with Single Sign-On Access

F. Lordan · J. Jensen · R. M. Badia

Received: date / Accepted: date

Abstract The low computing power of mobile devices impedes the development of mobile applications with a heavy computing load. Mobile Cloud Computing (MCC) has emerged as the solution to this by connecting mobile devices with the "infinite" computing power of the Cloud. As mobile devices typically communicate over untrusted networks, it becomes necessary to secure the communications to avoid privacy-sensitive data breaches. This paper presents work on implementing MCC applications with secure communications. For that purpose, we built on COMPSs-Mobile, a redesigned implementation of the COMP Superscalar (COMPSs) framework aiming to MCC platforms. COMPSs-Mobile automatically exploits the parallelism inherent in an application and orchestrates its execution on loosely-coupled distributed environment.

To avoid a vendor lock-in, this extension leverages on the Generic Security Services Application Program Interface (GSSAPI) (RFC2743) as a generic way to access security services to provide communications with authentication, secrecy and integrity. Besides, GSSAPI allows applications to take profit of more advanced features, such as Federated Identity or Single Sign-On (SSO), which the underlying security framework could provide. To validate the practicality of the proposal, we use Kerberos as the security services provider to implement SSO; however, applications do not authenticate themselves and require users to obtain and place the credentials beforehand. To evaluate the performance, we conducted some tests running an application on a smartphone offloading tasks to a private cloud. Our results show that the overhead of securing the communications is acceptable.

F. Lordan and R. M. Badia
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
E-mail: {francesc.lordan, rosa.m.badia}@bsc.es

J.Jensen
Science and Technology Facilities Council (STFC-RAL)
E-mail: jens.jensen@stfc.ac.uk

R. M. Badia
Spanish National Research Council (CSIC)

Keywords distributed computing, high performance computing, mobile computing, authentication, data security

1 Introduction

Mobile or handheld devices are computers small enough to be held in hand and easily carried by users wherever they go. Smartphones and tablets are the most typical examples of this kind of devices. In recent years, their popularity has increased [28], and applications for them are abundant. Although these devices have high capabilities for user interaction and network connectivity, their computing power is low and limited by the battery lifetime. Mobile Cloud Computing (MCC) [17] tackles this issue by bringing together the mobility of mobile devices with the vast computing power of the Cloud [8]. Applications run on the mobile device, and when they reach a compute-intensive point, the execution is offloaded to better-performing resources in the Cloud.

A paradigmatic example of MCC is an organization offering its IT resources to its members so that they accelerate applications running on their mobile devices. To avoid unauthorized users and to reliably account for resource usage, the organization needs users to authenticate through the Identity Provider (IdP) of the organization [34]. In addition to its own affiliates, this organization could also offer the IT resources to members of other organizations – with their own IdP – with which it is federated. Organizations need to define a set of common policies and protocols to manage and trust the identity of the users (Federated Identity Management (FIM)). In this case, using Single Sign-On (SSO) techniques would benefit organizations and users. Resource providers would be released of user account management (managing password strength, keeping account details up to date, resetting passwords) since they delegate it to the home organization of the user. It is also more comfortable for users since they no longer have to remember a large number of passwords or reuse a single password for multiple services. Instead, they have a single password – or some other means of authenticating, like a smart card – with which they authenticate to their IdP. Because they typically use this method frequently, they are also less likely to forget. Finally, they do not expose their passwords to remote systems, only to their (trusted) IdP.

On the user end, the data contained on the phone or collected through running applications (pictures, videos, lists of contacts, geolocation, movement, etc.) can be privacy-sensitive and should not be accessed without permission of the user. Given the sensitivity of the data, data breaches are a main concern on MCC environments [25]. Clouds often run on resources owned by private companies that offer them as a utility [10]. Other cloud users (multi-tenancy) are a potential threat; however, virtualization should isolate the resources assigned to each user and protect them from the attacks from its neighbors. Malicious insiders are another potential hazard since providers could snoop on the hardware resources and obtain information stored or processed on it. The strong laws enforcing data protection and the strict personnel background checks of commercial providers make malicious insiders not likely to happen, and we shall blissfully ignore it. However, to protect themselves from these attacks, users can apply data-at-rest encryption techniques such as ciphering/deciphering data when interacting with the file sys-

tem or operate directly on encrypted data using fully homomorphic encryption (FHE) [36].

The biggest concern regarding data breaches is attacks from unaffiliated people to in-transit data. Usually, the network interconnecting the nodes of the infrastructure workers is untrustworthy; for instance, the mobile device would normally connect to the remote workers through Wi-Fi and Internet. Attackers could eavesdrop on the interconnecting channel (e.g. the Wi-Fi network); communication has to provide message secrecy to not expose user or application information. Also, a disguised attacker could impersonate either a remote node, to intercept data transmitted from the mobile, or the mobile device, to fetch data stored in a remote worker. Hence, communications require mutual authentication and message integrity to ensure that both ends are part of the trusted infrastructure and that the content of the messages is the original and not a malicious command introduced by the attacker.

In this paper, we propose a solution to develop MCC applications where components interact through communications secured with authenticated encryption to protect the integrity, confidentiality and authenticity of the messages in the system. As for any other distributed computing infrastructure, writing applications for MCC environments is not straightforward. To ease their development without compromising performance, programmers turn to programming models that abstract them from the implementation details of partitioning the application, managing data dependencies, handling data transfers and submitting code executions between the nodes in the underlying infrastructure.

For that purpose, we build on COMPSs-Mobile [30], an implementation of the COMP Superscalar (COMPSs) [31] programming model specifically designed with MCC environments in mind, and enhance its communication mechanism to provide the security described above[1]. Most organizations already have a deployed authentication infrastructure; adopting a generic approach that avoids a security vendor lock-in was an important design consideration. For this reason, our solution leverages Generic Security Services Application Programming Interface (GSSAPI) [29], an interface implemented by most of the security services vendors. Thus, applications following the COMPSs model can replace the security framework without modifying their code. To validate the viability of our solution and evaluate its performance, we conducted some experiments using Kerberos [3] as the security provider.

Although GSSAPI provides the runtime system with the ability to authenticate and encrypt communication using federated credentials – if supported by the used implementation – it does not provide a generic mechanism for obtaining the credential. The described extension of the runtime requires the device to have the credential already. It can obtain the credential through another application that stores it on the file system, or the same application can provide the required mechanism. While the former option keeps the application agnostic to the authentication mechanism, the latter gathers all the functionality within a single application despite binding it to a concrete authentication mechanism. Either way,

[1] Further information on the COMPSs programming model and its runtime system can be found at http://compss.bsc.es. The source code for COMPSs-Mobile is available at http://compssdev.bsc.es/gitlab/flordan/COMPSs-Mobile.

we consider the work presented in this paper a first, but important, step towards achieving a secure MCC platform with SSO.

In a few words, the contribution of this paper consists in the following:

- An extension of the runtime toolkit architecture to integrate GSSAPI and the underlying security framework
- Description of the implemented mechanisms to support GSSAPI on the current runtime
- Discussion around the protection provided by the proposed solution against potential threats
- Validation of the prototype
- Evaluation of the overheads incurred by the extension

The paper continues by giving an overview of the current state of the art in distributed programming and other MCC frameworks in Section 2. Section 3 describes the architecture of our solution, and details related to the implementation are provided in Section 4. Finally, Section 5 evaluates the overheads of our implementation and discusses some technical decisions. To wrap up, Sections 6 and 7 present, respectively, the conclusions and the future directions.

2 Related Work

Having a low-capabilities device dedicated to user interaction while a resource-rich, remote node hosts the compute intensive operations is a technique that appeared with time sharing systems on mainframes when users accessed a central node from simple terminals. Since then, there has been much research on supporting remote execution and computation offloading; however, due to the short existence of smartphones and tablets, research on mobile clouds (MCC) has been done only in the recent years.

Writing applications for distributed platforms is not as straightforward as writing sequential applications that run in a single device. Programmers must deal with technical concerns such as partitioning the application into parallel blocks, deciding which tasks are worth being offloaded (cost/benefit analysis), assigning resources to host each task execution considering the data dependencies that may exist between them (task scheduling), and submitting the execution of each task to a remote node. Thus, the programmer must take into account the infrastructure while programming, and must potentially consider details of each node e.g. its features and the specific middleware to interact with it. Grid Computing – and later on, Cloud Computing – presented similar issues; researchers studying distributed computing already explored these problems and there is a wide bibliography [40,38,47,16,18,7] discussing the topic. In addition to these problems, mobility brings additional concerns that characterize the development of application targeting MCC environments. On the one hand, mobile devices are tied to a battery, a limited source of energy; thus, their lifetime depends on the battery capacity and the usage of the available energy. On the other hand, high mobility incurs network variability: different network protocols and interfaces, signal strength fluctuation, instability, monetary costs, etc. All these aspects should be considered in the cost/benefit analysis and applications should quickly adapt to changes.

To ease the development of applications, developers turn to frameworks with programming models that allow them to describe parallel applications, abstracting them from all these details. Later, at runtime, a toolkit is launched along with the application to orchestrate the proper execution of the application on top of a distributed infrastructure. This section introduces different trends followed by distributed programming models and positions COMPSs in this landscape and compares the COMPSs-Mobile implementation with other MCC frameworks. We also introduce related work in distributed computing security.

To the best of our knowledge, COMPSs-Mobile is the first MCC framework to consider not only authenticated encryption to protect in-transit data on communications over untrusted networks but also using Federated Identity to authenticate both ends. The only framework that envisages implementing mechanisms to secure data transfers is RAPID [1], an extension of ThinkAir, which is currently working on securing the communications by using SSL sockets.

2.1 Distributed programming models

Programming models provide developers with a mechanism to describe their applications, so their runtime toolkit partitions the application and executes it on the distributed infrastructure. Some models, such as Aneka [42], require programmers to create new tasks explicitly and to add them to a bag from which any of them can be selected to be executed. Their main drawback is that developers still need to deal with data dependencies among tasks. Other models opt for predefining a particular parallelism pattern and restrict the workflow, as MapReduce [15] does. In this kind of models, programmers only need to implement a set of methods that compose the predefined workflow.

More ambitious programming model designers allow programmers to describe any workflow. An easy approach to this is to define a specific syntax to describe the application parallelism. JOLIE [33] and Microsoft Dryad [23] have explored this way, and models like Taverna [32] and WS-BPEL [4] have gone one step further by hiding complexity behind a visual development environment. Another approach for the application developer is to write its application as a sequential code along with some meta-data and leverage the programming model to automatically detect code blocks and data dependencies among them through to orchestrate their parallel execution. COMPSs and Swift [44] are examples of programming models with automatic parallelism detection.

In addition to these programming models, there exists another important family of programming models called concurrency-oriented programming models. Applications following these models are implemented as complex software systems composed of a set of isolated components which communicate exclusively by exchanging messages. Each component has an exclusive mailbox where all the messages addressed to it are left; later on, the component processes a single message at a time. The processing of the message varies depending on the current state of the component and its behavior, a function that defines the state changes or message submissions in response to an input message according to the current component state. One of the most successful models within this family is the actors model, implemented by Akka [19] or Erlang [43].

2.2 Mobile Cloud Computing Frameworks

No framework targeting classic distributed systems that implements any the programming models of the previous section considers the energy restrictions due to the battery nor is capable of quickly adapt its behavior to the network. During this section, we introduce some frameworks specially designed for MCC environments and classify them into a taxonomy according to three distinguishing features. The first factor, the *migration granularity*, is determined by the size of the application pieces that are offloaded to the remote resources. The coarser grained it is, the more data needs to be transferred to the resource. Transferring the whole state of a Virtual Machine (VM) (or keeping the state of two VMs synchronized) requires more data than transferring only the state of one single thread and the data values it accesses; in turn, offloading a single method execution avoids shipping all the state of the thread.

The second classifying factor is how the model decides whether a part of the application runs on the local device or is offloaded (*offloading decision*). It could be statically defined in the application code or decided dynamically at runtime depending on environmental conditions.

Finally, every computation has blocks that can be executed concurrently on different resources to reduce the execution time. Depending on the model, the management of the parallelism is left to the programmer, or the runtime exploits it automatically (*automatic parallelization*).

Satyanarayanan et al. define a coarse-grained model where a whole VM is shipped to a nearby resource-rich computer, the cloudlet, taking advantage of hardware VM technology [37]. They propose two approaches: migrate the VM or synthesize a small VM overlay to be applied on a base VM already present in the cloudlet (dynamic VM synthesis). Evidently, offloading a whole VM implies that any parallelism must be explicitly stated in the application. Regarding the offloading decision, they do not specify whether the programmer must specify when to offload or if the runtime toolkit decides it at execution time.

CloneCloud [13, 12] offers the developer a finer level of granularity: threads. 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. Finally, the returned packaged thread is merged into the state of the original process. Although thread level is finer than VM, it still requires the developer to create new threads and manage the application parallelism.

The partition granularity can still be reduced. Many models operate at a method-level granularity. Cuckoo [24] takes advantage of the architecture of Android applications and hides the partitioning problem by exploiting the service component of Android. 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.

Other models force the programmer to identify the methods to offload (or, at least, to consider their offloading). MAUI [14] offloads the execution of .NET methods to a remote clone of the application deployed in the cloud. Developers annotate the remotable methods, and the framework decides whether to offload the method invocation, taking into account the application and network characteristics. For submitting the method, the system computes an incremental delta of the application state (method inputs and some static data) and ships them with the task description. The weakness of this model is the application parallelism. The programmer completely manages it, and the model only exploits the computing resources of a single clone.

ThinkAir [26,27] follows the same partitioning method as MAUI, but it works around MAUI's parallelism limitation by allowing the use of multiple resources. ThinkAir already provides a mechanism to automatically parallelize the execution of an offloaded method by 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.

COMPSs-Mobile also does a method-level partitioning, dynamically decides where to run the method, and allows the use of multiple resources; however, it follows an asynchronous execution model. When a remotable method is invoked, the calling thread creates an asynchronous task and continues executing the application. Thus, the same thread keeps executing the application code and intercepting other invocations to remotable methods that may run concurrently, which results in a higher level of parallelism.

Table 1 summarizes the features of the MCC frameworks discussed in this section and highlights the characteristic features that distinguish them.

	Migration Grain	# Worker Nodes	Execution Model	Automatic Parallelization	Offloading decision
Cloudlets	VM	1	Synchronous	No	Not detailed
CloneCloud	Thread	Not detailed	Synchronous	No	all methods dynamic profile
MAUI	Method	1	Synchronous	No	candidate methods dynamic profile
Cuckoo	Method	N	Synchronous	No	service calls dynamic profile
ThinkAir	Method	N	Synchronous	Intervals	candidate methods dynamic profile
COMPSs-Mobile	Method	N	Asynchronous	Yes	candidate methods dynamic profile

Table 1 Comparison of MCC frameworks

2.3 Securing distributed computing

As discussed in the introduction, distributed computing across a wide-area network needs security. Servers need to identify themselves to the others, and users (or their applications) have to authenticate to the infrastructure and communicate securely.

Grid systems used the Globus toolkit [6] basing their security on X.509 certificates [22], which in practical deployments are issued by trusted certification authorities [9]. While requiring client (certificate) authentication, Globus expanded on the original authentication model by introducing delegation [41]. X.509 then "plugs into" the protocol, whether it is secured HTTP [35], GSS, SOAP, etc.

In general, distributed security can be done in a peer-to-peer manner, but this usually requires humans to connect to each other to establish trust: an example would be to share ssh host and client keys. In larger scale distributed computing, it is more common to use one or more trusted authorities. These do not have to be Certification Authorities (CAs), distributing X.509 certificates, but could also be Kerberos Key Distribution Centers (KDCs), for example. We are not covering this in further detail here; we refer the reader to the study of non-web access in AARC [20].

Using security services incurs an additional overhead to the execution that can become significant for large datasets. Therefore, this overhead is to be considered during the allocation of resources where each task runs. Xiao and Qin analyze different security services implementations in computations distributed over either heterogeneous and homogeneous resources [46]. The authors derive a scheduling model (TAPADS) for workflows (i.e. with task dependencies) running on homogeneous resources where each task has different strength requirements for authentication, integrity, and confidentiality. The goal of its policy is to optimize the compute time while guaranteeing the job precedence and the desired security levels. Chen et al. go one step further and propose a scheduling approach (SOLID [11]) that selectively duplicates predecessor tasks to avoid the overhead of transferring the intermediate data – which includes its encryption when sending it out and its decryption upon reception. Regarding the resource allocation for tasks, COMPSs-Mobile follows a simpler approach. Unlike these methods, tasks cannot have different security constraints. The model considers security as a single service with two levels of strength: enabled or not enabled. Whether a connection needs to be secured or not does not depend on the task or the information to transfer but on the security conditions guaranteed by the interconnecting network. The current scheduling algorithm assumes a high application parallelism with tasks long enough so the transfers of the input values for later-executing tasks can overlap with the execution of other tasks. Therefore, time to transfer data values – including the overhead incurred by security – are not considered by the COMPSs-Mobile scheduling algorithm.

3 Background, Architecture and Design

3.1 Distributed Application Development

COMPSs [31] is a framework that provides developers with a sequential, infrastructure-agnostic programming model, to ease the development of distributed, high-performing applications. Applications developed in the COMPSs model are automatically instrumented to invoke a runtime toolkit that splits the application into computing units (tasks) and orchestrates their parallel execution on top of a distributed platform, guaranteeing the sequential consistency of the application.

Mobile devices have lower computing and storage capacity, and their connectivity is limited over slower network connections that are likely to fail. These peculiarities led to the development of a new runtime toolkit specially designed for supporting the COMPSs programming model on mobile devices: COMPSs-Mobile [30]. With it, developers code mobile applications with the native programming language of the target mobile platform as if application ran completely on the device. Without changing the source code of the applications, the application building process transparently alters the expected behavior of the execution, so the main code of the application keeps running on the mobile device, i.e. the master node, and tasks are offloaded to remote nodes. Consequently, the introduction of any security mechanism (section 3.2) in the framework should also be transparent to the application developer.

Before delving into the details of COMPSs-Mobile, we introduce a few more concepts about COMPSs itself.

3.1.1 COMPSs Programming Model

COMPSs applications are compositions of computations whose remote execution is to be orchestrated to exploit their parallelism. These computations are encapsulated in methods, called Core Elements (CEs); each composite is known as Orchestration Element.

Developers select the CEs by declaring the corresponding methods in the Core Element Interface (CEI) along with a *@Method* annotation that indicates the implementing class. 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 to the CE declaration indicating the data type and directionality (in, out, in-out). The code snippet in Figure 1 contains a simple COMPSs application example. Subfigure 1(a) shows the sequential code of the application which runs N simulations and selects the best one. As shown in the CEI presented in Subfigure 1(b), only two methods are chosen as CE: *simulate* and *getBest*.

```
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

```
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. 1 Sample application code written in Java

3.1.2 Application Instrumentation in Android

Android is an open-source platform targeted to devices with limited computing resources such as smartphones or tablets. The platform offers support at all levels of the software stack: from OS management to end-user applications, including a set of native libraries and other components that applications are likely to use.

Android applications, written in the Java language, are converted to Dalvik bytecode and bundled into Android package (.apk) files for distribution in a four-step process. This process starts with the creation of a Java class (Resource Manager) that eases the access to resources – non-source code entities, such as images and sounds. The Android Development Toolkit completes the user code with the automatically generated proxy-stub classes required for the inter-process communications (Pre-compiler). Once the application code is complete, all the Java classes are compiled to generate Java bytecode (Java Builder) that is translated into Dalvik bytecode and bundled together with all the resources into the Android package file (Package Builder).

To instrument the code and alter the behavior of the application, COMPSs-Mobile extends the default Android building process by adding a step after the Java Builder: Parallelization. Through Javassist [2], a library for editing Java bytecode, it replaces CE invocations by asynchronous runtime calls to create new tasks and to synchronize data accesses with the fetching of their actual value located in the worker nodes. These instrumented versions of the Java classes replace the original ones in the building process, so they are translated into Dalvik bytecode and bundled into the application package along with the COMPSs-Mobile library.

3.1.3 COMPSs-Mobile Runtime System Architecture

COMPSs-Mobile applications follow a master-worker execution model as depicted in Figure 2. The main code of the application runs on the master node, a mobile device. When the end-user launches the application, the instrumented code invokes a runtime toolkit whose main goal is to parallelize the execution of the CE invocations on remote computing resources (workers) while guaranteeing the sequential consistency of the application.

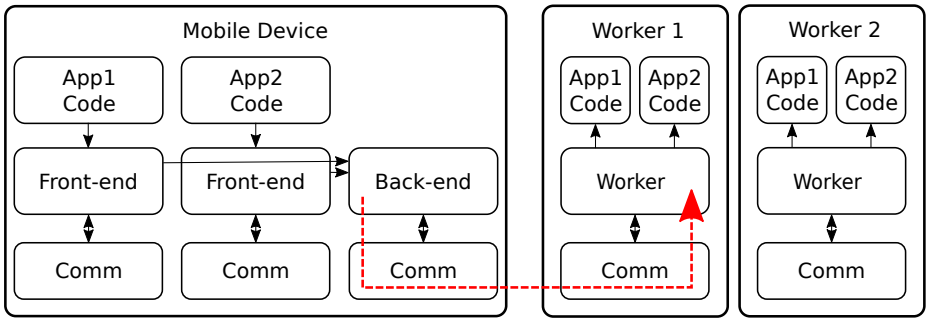


Fig. 2 Deployment architecture diagram. As depicted by the red dashed arrow, the communication component on each process manages all the messages exchanged through the network between runtime components.

The runtime library on the master node analyses the data values accessed by every task to find data dependencies among them and orchestrates their executions on the worker nodes. For this purpose, each COMPSs-Mobile application hosts an instance of the front-end of the runtime that registers accesses to application-private data such as objects. Shared data, such as files, are monitored by the back-end of the runtime which runs on an Android service within an independent process common for all applications.

In addition to the shared data management, the back-end of the runtime also decides whether to offload a task execution or to run it on the internal computing devices. For making a decision, the library follows a heuristic that considers time, monetary cost and energy consumption.

Worker nodes are organized as an autonomous peer-to-peer network. Using a publish-subscribe system, the network monitors the data dependencies of the tasks of all applications and discovers those that become dependency-free and can be executed in one of the peers. When a worker node picks a dependency-free task to run, it fetches all the input values, executes its code, and, at the end of the execution, publishes the existence of its output values so any other node can fetch them from there.

All the communications between components of the runtime toolkit (i.e., data values and message exchanges) are managed by a communication layer that transfers the information through (clear, without end-point authentication nor message encryption) TCP sockets using the non-blocking I/O API offered by Java.

Usually, applications run while the mobile phone is connected to the network; for long executions, the phone might eventually lose connectivity, or reconnect via another network (e.g. switch from wireless to 3G)). In this case, the master notifies the new network conditions (e.g. IP address) to all the workers who update any master reference with the new address and retry any failed communication. If the break in connectivity was short enough, the workers might not notice.

Besides the monitoring of the data dependencies, the peer-to-peer network allows the worker nodes to share data values without connecting to the master; thus contributing to the robustness of the platform and the reduction of the network traffic between a worker node and the master. During network breakdowns, the mobile device becomes isolated for an undetermined amount of time and data values can not be transferred from the mobile to any worker node, or vice versa.

On the master side, the mobile executes all the tasks already assigned to it; upon their completion, it starts to compute offloaded tasks. Since tasks might depend on data values computed by worker nodes that have not been transferred to the master yet, it has a backtracking mechanism to find those tasks generating the unavailable value and execute them.

On the other side, worker nodes can keep with the execution too. Through the peer-to-peer network, workers notice the creation of data values on other workers and retrieve them. Therefore, all tasks can be computed except for those with some input value located only in the master node.

Eventually, the network connection will re-establish, the master will send the new network conditions notification, and the workers will notify to the master all the data values created and task completed during the breakdown and fetch the values for the stalled tasks. Thus, the application execution follows up as if the network breakdown never happened.

Previous work of the authors[30] contains further details about COMPSs-Mobile, the architecture of its runtime, the description of the internal mechanisms and the performance evaluation.

3.2 Security Architecture

A secure system consists of a set of interacting participants which authenticate themselves using the credentials issued by an *authority*. These participants are *end users* (persons), which use a username, e-mail address or real name to be uniquely identified; and *compute nodes* identified either by hostname, IP address or sometimes as individual services or endpoints. Authorities usually are centralized: a single entity manages the credentials for all the participants within the domain (e.g. the members of an organization accessing to its services). However, an identity and its associated attributes can be shared across multiple domains; for instance, the members of *Organization A* use services offered by *Organization B*. Both organizations define a common set of policies and protocols to manage the identity in a federated way.

Several different security technologies implement the described architecture [39]. The Generic Security Services Application Programming Interface (GSSAPI) is an abstraction of the security negotiation that happens when a client – initiator – authenticates to a server and exchange messages securely. The applications at either end call the API and are instructed by the implementation whether authentication is successful, is unsuccessful, or needs more calls – some protocols require several back-and-forth communications. A wide range of mechanisms can implement the underlying authentication: username/password, Kerberos, Moonshot, X.509 certificates; clients can be anonymous or named, and can pass authorization attributes ([45]). Originally, the preferences of the initiator determined the authentication protocol; however, GSSAPI was extended to support a common protocol for negotiation [48]. This negotiation protocol builds on mutually accepted trust anchors, and that might not be sufficient; a further proposed extension ("extended negotiation") supports more sophisticated negotiation protocols (for example, the negotiation in a shared protocol might fail after the protocol has been agreed, e.g. due to a lack of mutual trust anchors.)

For message-level security, GSSAPI supports not only origin authentication (i.e. sender signs the message) but also message encryption, integrity, replay detection, or detection of receipt out of sequence: each block of data will have the selected security features applied to it before it is sent, and checked upon reception.

Compared to just implementing one security protocol, using GSSAPI correctly is more complicated for the application programmer; it is also harder to debug because GSSAPI is implemented using ASN.1 as a layer around the actual protocol. However, the generic service, if coded correctly, can then support a range of mechanisms – including future ones – and delegates on GSSAPI many security tasks that may not be obvious to the programmer, such as preventing replay attacks, checking the server identity correctly (preventing man-in-the-middle attacks), and negotiating shared protocols for message security, etc.

A special feature in GSSAPI, supported by some protocols/implementations, is the support for *delegation*. Delegation can be defined as a temporary reassignment of rights; however, in many practical applications, it means that the client forwards

a credential to the server. Through those credentials, servers "impersonate" the user or, at least, perform certain actions on their behalf: this is typically useful where a worker accesses a resource requiring authentication, e.g. storage, on behalf of the user. In Kerberos, the corresponding technology is a proxiable ticket, and X.509 has the GSI proxy (RFC 3820).

3.3 Architecture of Secure COMPSs-Mobile

The extension to secure COMPSs-Mobile brings two challenges to the current platform architecture. On the one hand, the roles within a secure architecture need to be mapped to the components of a COMPSs-Mobile platform. On the other hand, the runtime has to secure all the communications among the nodes of the platform; therefore, the architecture of the system needs to be adapted to include security frameworks within its stack.

A secure architecture is composed of several interacting participants authenticated by a trusted third party that acts as an authority. In a COMPSs-Mobile environment, participants correspond to the nodes of the computing platform, either the master or the worker nodes. The actual infrastructure acting as the authority and the protocols to interact with it are specific to the used security framework. As deduced from the content of section 3.2, to provide an interoperable solution that works with several security frameworks and avoids a security vendor lock-in, the runtime leverages on GSSAPI.

GSSAPI abstracts away the authentication infrastructure and the used protocols from COMPSs-Mobile and establishes a client-server pattern where the client (GSS initiator) contacts a service (GSS responder) to start a secure connection and exchange messages. This pattern maps easily with the master-worker approach of COMPSs-Mobile where the application running on the mobile device (master node) offloads task executions to remote nodes running a service. Therefore, the master, assuming the GSS initiator role, would authenticate on behalf of the application user to worker services playing the GSS responder role.

However, this model clashes against the peer-to-peer organization used for sharing data in COMPSs-Mobile. Whenever a node of the infrastructure needs some value located on a remote node, it opens a new TCP connection to a server deployed on the remote node, regardless of whether it is a worker or the master. In TCP terms, any node can act as a TCP client, so every node must listen for incoming connections, including the master. To avoid that anyone fetches a value from a node of the infrastructure, both ends authenticate to each other following the protocol established by the specific security framework.

For master-worker communications, the master, as GSS initiator, always triggers the secure context negotiation upon the TCP connection establishment, regardless of the TCP roles of each node. Since establishing a secure context when both ends act as GSS responder is not possible, in worker-worker communications the TCP client assumes the role of GSS initiator.

Traditionally, worker processes are likely to be deployed together on resources interconnected by trustworthy networks, such as HPC clusters. In this case, using secure connections gives no added value but adds unnecessary overhead; for this purpose, the communication library allows a whitelist to be set up for worker-worker communications that do not need security.

The second problem to tackle is the integration of GSSAPI within the software stack. GSSAPI only indicates the format of the messages exchanged among both ends of a connection, but does not define their content, which depends on the message and the selected security framework, nor does it define the transport-layer protocols used to transfer messages through the network. All application messages need to be processed (and likely modified) by the security framework before being sent using the same network protocols that the application would regularly use. Upon arrival of new data from the network, messages need to be processed by the framework to extract the actual application message before forwarding it to the application. The described COMPSs-Mobile architecture already encapsulates all the network interactions within a communication layer component; therefore this is the only component that needs to be modified to secure COMPSs-Mobile communications. Whenever a node of the infrastructure, either master or worker, wants to send a message to another node, it will request to the communication layer to open a new connection to the target node.

Figure 3 depicts an overview of the architecture of a deployment of COMPSs-Mobile with one master node (leftmost part of the figure) and a worker node (right). The security framework used for the validation of the described architecture is Kerberos. The user of the mobile phone had previously obtained the Ticket Granting Ticket from the Kerberos key distribution center. Worker nodes are identified through a Kerberos keytab, although they typically use X.509 certificates, and are authorized to accept connections either from the master or other worker services.

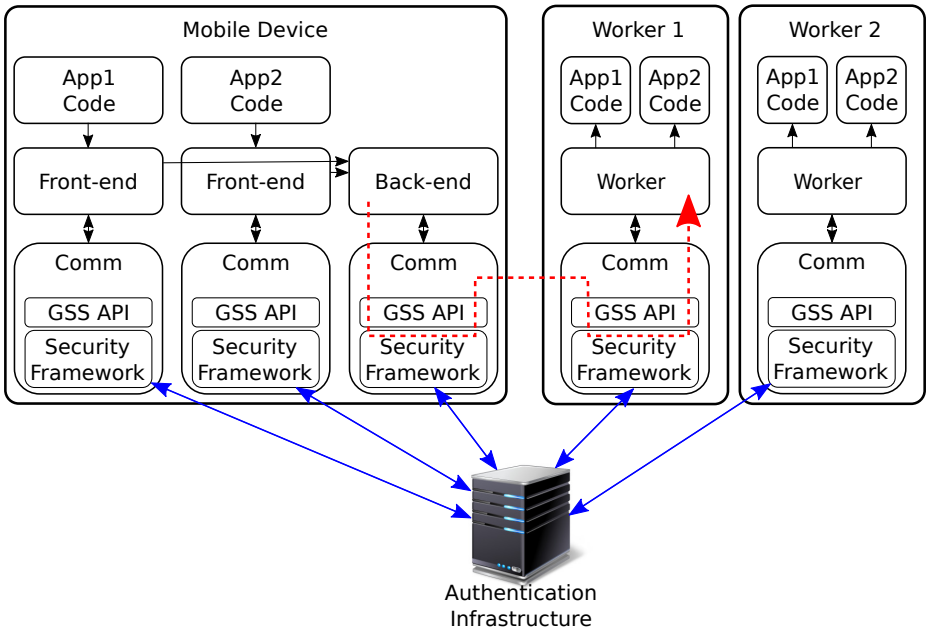


Fig. 3 Secure COMPSs-Mobile architecture diagram. As depicted by the red dashed arrow, the security framework processes the messages exchanged between the runtime components prior the communication component transmits it and upon its reception on the remote node.

4 Implementation

Despite Java being the native language to develop Android applications, not all of the classes and libraries typical from Java are available on Android. GSSAPI is one example of these libraries; although it is included as part of the Java SE, there is no GSSAPI implementation within the Android software stack. For the porting of GSSAPI to Android, we cross-compiled the official release of the MIT Kerberos for Android and created a native library (libkerberosapp.so) that is bundled along with the application and the runtime code in the apk file. The library is dynamically loaded upon the completion of the first TCP handshake. RFC5653 defines the Java binding for the GSSAPI. The wrapper is defined as Java code, the natural language to program for Android, and links with the native library using Java's JNI library.

Java's Non-Blocking IO library encapsulates point-to-point, ordered network connections in stream-oriented channels. This approach guarantees the reception of all the sent bytes in the same order, but it does not necessarily maintain the groupings: the sender could submit a 128-bytes packet and the receiver could get two packets of 96 and 32 bytes, respectively. For abstracting this away from the application, the COMPSs-Mobile communication library adds 4 bytes to the message header indicating the message size. Upon the reception of the whole message, the communication library delivers the messages to the application/runtime level.

To achieve complete secrecy, also the message headers need to be encrypted. Therefore, the length of messages cannot be known by the recipient until the header is decrypted; and ciphertexts cannot be decrypted until they are completely received[2]. For overcoming this problem, we split plaintexts, and pad them if necessary, to fit into a fixed "token" size (of configurable size); encrypted tokens can then be sent across the network[3]. The recipient can start to decrypt a message once it has received enough to decrypt the first token, which includes the length of the plaintext message, as mentioned above. In particular, sender and receiver *must* use the same token size. The chosen token size has a strong impact on the total amount of bytes transferred through the network, and therefore on the time and cost of data transfers. Larger token sizes may add more padding and take more time to transfer; smaller token sizes have more overhead in being processed individually and may be more likely to split important structures. In Section 5, we check the impact on the runtime performance when using different token sizes.

5 Discussion

5.1 Security

As a consequence of implementing security through GSSAPI, security is managed outside of COMPSs-Mobile. Externalizing security is both an advantage because we can connect different implementations of security, and a disadvantage because

[2] For block cipher algorithms, at least one complete block must be received, but it simplifies the implementation to wait for the whole message.

[3] With stream ciphers and, for block ciphers, with the token size being a multiple of the block size, encrypted tokens have the same length as their plaintext. However, GSS adds some bytes to the token, see section 5.3.

three different programming skill-sets are required: parallel programming, mobile cloud computing, and security. COMPSs-Mobile simplifies the development of applications by transparently managing the three of them. However, unlike the handling of the application parallelism and seamless offloading of computation from the mobile to the cloud, security is not intrinsically integrated into the framework. Thus, the end user of the application must currently provide GSSAPI with the required credentials to correctly initialize security context

For instance, when using Kerberos as security services provider, users need to obtain the ticket granting ticket (TGT)). They must be able to communicate with the Authentication Server (AS) and Ticket Granting Service (TGS) (together known as the Key Distribution Center, KDC). Even if the mobile device obtained the initial ticket (from the AS) out of band, it would still need to be able to connect to the TGS at runtime, so practical deployments of Secure COMPSs-Mobile would probably not be using Kerberos.

Delegation is required when the workers need to access other services on behalf of the user; for instance, an external storage service requiring authentication. GSSAPI has support for delegation, but it needs to be backed by the implementation; the delegated credential will belong to the same mechanism as the original user identity. In our case, the master must permit delegation of its client credential, and the worker can then perform steps to obtain the delegated credential.

HPC clusters are often protected behind a firewall that isolates the computing nodes from external networks; usually, one single node is exposed and accepts incoming connections acting as the front-end of the cluster. Future work could look into using this frontier node as the only worker open to incoming connections from the mobile (see Figure 4), as opposed to opening incoming connections from any IP address to a whole HPC cluster. This additional node would forward messages between the worker nodes and the master, and potentially could act as a mirror of the mobile device, caching input data values. Thus, this additional node simplifies the setting up and monitoring of the security of the cluster and reduces communications between master and workers.

However, only the worker(s) which connect directly with the master obtain a delegated credential; the Master node connects to the proxy worker, so only the proxy will obtain a delegated credential through GSSAPI. Thus, if other workers need a credential, they will need it forwarded from those that have it, or they will need to proxy their requests through workers with a delegated credential. In Kerberos terms, the delegated credential is a proxy ticket, generally designed to be issued to a particular network address and, hence, intentionally not useful to other nodes in the cluster.

5.2 Threat Model

One of the basic principles of data security is to know what one aims to protect, and from whom. Based on the list of threats by the Cloud Security Alliance [5], we discuss here briefly whether our proposal addresses these threats or not, and how it does so.

1. **Data breaches** are precisely our main concern; especially those breaches incurred by data transfers. The toolkit chooses to encrypt the data or not depending on whether the end user trusts the interconnecting network not (i.e.

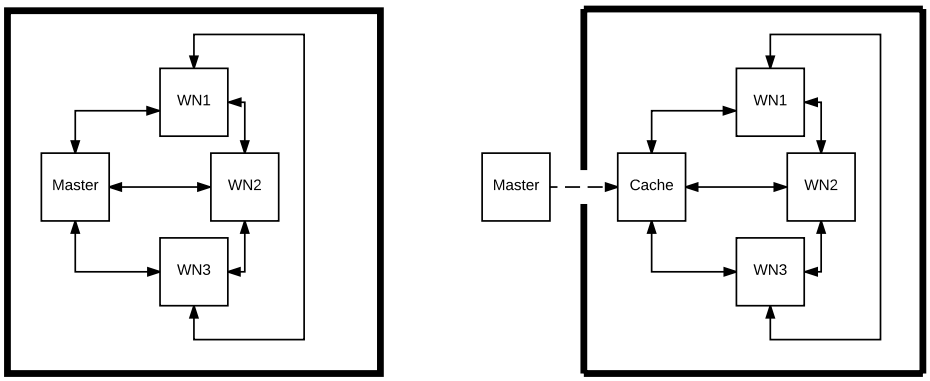


Fig. 4 Application security: LHS the single-cluster scenario, RHS with the master on the mobile; the fat line denotes the site firewall.

there are no vulnerabilities on the network that attackers could exploit them to obtain user's data). In practical deployments, we consider connections wholly within the same cluster to be secure for the reasons exposed in point 6. Therefore, we recommend to turn off encryption for performance purposes. However, if one were concerned, it would, of course, be possible to encrypt every connection.

2. **Insufficient identity/credential/access management** is a vulnerability that our system inherits since the initialization of credentials is out of band to COMPSs-Mobile. A poor management of the identities by the IdP or an improper granting of the permissions could become a vulnerability of the system. Besides, once initialized, the credential typically remains active for a period of time. An insecure handling of the credential could potentially become another vulnerability. Since we use Kerberos in our deployment, the security parameters are well known.
3. **Insecure interfaces/APIs** are addressed through securing the sockets. External users cannot obtain data either being transferred – GSSAPI provides secrecy – nor at-rest since the authentication mechanism would reject connections to invoke the COMPSs-Mobile components. COMPSs-Mobile does not consider data ownership; therefore, other users authorized to interact with the worker could request the transfer of some intermediate value.
4. **System vulnerabilities** remain a concern, particularly for mobile devices that are busy and well-connected; users would be required to ensure they are patched and kept up to date.
5. **Account hijacking** is considered a less likely threat in our scenario; it would be a concern if, and only if, the mobile device hosting the master node (and the user account on it) were shared.
6. **Malicious insiders** are a threat which we do not address. Given the described infrastructure, we look at three different possibilities regarding the clouds hosting the computation: the organization to which user belongs owns the cloud, academic clouds and commercial clouds. For the first two cases, the risk of a malicious insider is low enough to accept the risk without further mitigation. On the first one, the data processed by the application is likely to already belong to the organization and its the same organization who will capitalize the

results of the execution. Academic clouds should be used for research purposes, and the results, published and shared. For commercial clouds, the strong laws protect data and the high competitiveness in the market forces service providers ensure a sense of security; for that, they have very stringent personnel controls. If one were concerned about the risk at an academic research center, one could switch to a commercial cloud.

7. **Advanced persistent threats**. Mobile devices, phones in particular, are busy devices which are designed to connect to many kinds of networks, from wireless and 3G/4G to NFC and Bluetooth. At the same time they are running apps for many different activities, from games to mobile banking. Ruling out threats from all of these would be very difficult.
8. **Data loss** concerns data-at-rest and is a low-risk threat. It would trigger only if the mobile phone were lost and happened to have contained unique data (not yet replicated to the data center), or if a credential is needed to access data services in the data center and it expires before the data can be written and the data is lost.
9. **Insufficient due diligence** is mainly a threat to the operations of the infrastructure; the secure COMPSs-Mobile must not be "tacked on" to a production infrastructure but must be maintained as a part of it. Likewise, some user training/education/awareness in for example credential management is likely to be helpful.
10. **Nefarious use** of compute services deals with the work done by the application on the infrastructure. To mitigate this threat, it suffices that the server keeps sufficient logs of the connections – remote IP address, user principal – so that incidents may be dealt with appropriately.
11. **Denial of service** is mitigated by requiring secure connections. An attacker could still flood the network and listening socket on the data center server, so, in the present, work we cannot fully eliminate this threat.
12. **Shared technology vulnerabilities** are threats due to shared/multi-tenancy use of the compute infrastructure; these are managed by the underlying batch system of the HPC cluster or, in the case of clouds, by the underlying virtaulization technology; hence, they are not relevant here.

5.3 Performance

To compare the performance of secure and clear communications, we open a new TCP connection to transfer data of multiple sizes using different token sizes. The source of the transfer, which takes the role of GSSAPI client, is a OnePlus One smartphone, equipped with a Krait 400 quad-core processor at 2.5GHz and 3GB of RAM and connected to the Internet via an 802.11g wireless network. The receiver, a GSSAPI service, is a quad-core VM on an OpenNebula private cloud hosted by nodes with six-core Intel Xeon X5650 at 2.67 GHz processors and 24 GB of memory interconnected by a Gigabit Ethernet network. In this case, the connection between the mobile device and the worker nodes has a 133 ms RTT (Round Trip Time), and both ends have already obtained their Kerberos tickets (so the time required to contact the KDC is not included.)

As shown in the timeline depicted in Figure 5, after the 3-way handshake to establish the TCP connection, both ends of the connection exchange messages

(plain text) to establish the secure context (Negotiation). First, the GSSAPI client instantiates a new GSSAPI context (average 16 ms) and constructs a message of 612 bytes to authenticate itself and describe the available mechanisms to establish the secure context (average 18 ms). If the library is set up to use very short tokens (256 or 512 bytes) the library may need to split the message into several tokens increasing the total amount of sent bytes (620 bytes in 3 tokens, for 256-bytes; and 616 bytes in 2 tokens for 512-bytes). The GSSAPI service receives the message, verifies the identity of the client and picks the mechanisms and algorithms to establish a secure context (55 ms) and creates a response message of 166 bytes, 142 dedicated to identification and terms of the secure context. Upon the reception of this message, the client verifies the identity of the service (2 ms) and end ups the negotiation. In overall, this process takes around 355 ms (depends on the network conditions). The client emits 632-640 bytes, and the server, 166 bytes.

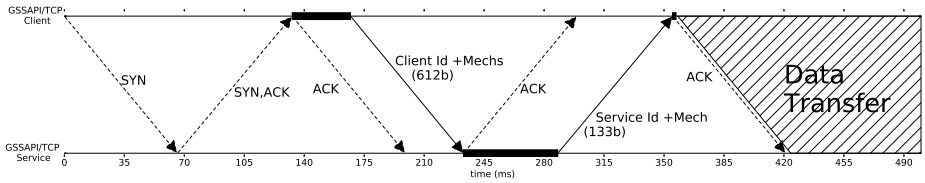


Fig. 5 Timeline of the TCP Connection Establishment and GSSAPI Negotiation

Once the negotiation ends, the actual data transfer begins, and secure tokens ("wrapped" in the GSSAPI terminology) are transferred through the network. It is in this second stage where the token size may have a stronger impact depending on the performance. Tables 2 and 3 compare, respectively, the actual transfer size and the elapsed time when transferring internal commands of COMPSs-Mobile (typically, 250 bytes), small objects (2500 bytes) and files of different sizes (10,000, 100,000, 1,000,000 and 10,000,000 bytes) when different token sizes are used for non-secured and secured transmissions.

Comm. Type	**Token Size**	**# Bytes**					
		250	**2,500**	**10,000**	**100,000**	**1,000,000**	**10,000,000**
non-secure	**256**	274	2,556	10,176	101,604	1,015,892	10,158,748
	512	270	2,536	10,096	100,804	1,007,892	10,078,760
	1,024	270	2,528	10,056	100,412	1,003,940	10,039,232
	2,048	270	2,524	10,036	100,212	1,001,976	10,019,588
	4,096	270	2,520	10,028	100,116	1,000,996	10,009,792
	8,192	270	2,520	10,024	100,068	1,000,508	10,004,904
	16,384	270	2,520	10,020	100,044	1,000,264	10,002,460
secure	**256**	512	3,584	13,568	133,376	1,333,504	13,333,504
	512	512	3,072	11,776	114,688	1,143,296	11,428,864
	1,024	1,024	3,072	11,264	107,520	1,067,008	10,667,008
	2,048	2,048	4,096	12,288	104,448	1,034,240	10,323,968
	4,096	4,096	4,096	12,288	102,400	1,019,904	10,162,176
	8,192	8,192	8,192	16,384	106,496	1,025,808	10,084,352
	16,384	16,384	16,384	16,384	114,688	1,015,808	10,043,392

Table 2 Actual size of transferring 250, 2,500, 10,000, 100,000, 1,000,000 and 10,000,000 bytes according to the token size in bytes (256, 512, 1,024, 2,048, 4,096, 8,192, 16,384)

Regarding the actual transfer size when transmitting non-secure messages, the communication library adds a header of 16 bytes to the message content. The bigger the data is less significant is the overhead of the header (6.4% for a 250-bytes message, 0.0000016% for the largest case). In addition to this header, the library adds four extra bytes for each token used; therefore, the bigger the token size is, the fewer tokens are used, and fewer bytes are added.

GSSAPI allows enabling, if wanted, mechanisms to secure the communications using different algorithms. With the used configuration, securing communications implies encrypting and signing the content of each message; thus, the actual payload of each token is reduced according to the encryption and signing algorithms (in our tests, 60 bytes). Since the length of each token (4 bytes) cannot be exposed, it becomes a part of the cipher and signature input; tokens have a fixed size. This overhead has a significant weight on the message/payload ratio for really small tokens (25% for 256-bytes tokens) and increases the number of tokens (with their respective length descriptor) required to send a message. For a 10,000,000-bytes transfer, the overhead reaches up to a 33.35% when using 256-bytes tokens. Conversely, using large tokens reduces the number of tokens to send large messages and, therefore, the additional bytes (0.43% when transmitting 10,000,000 bytes in 16,384-bytes tokens). On the other hand, messages are to be padded to match the token size; when using very large token sizes to send small messages, the pad has a significant weight on the length of the transmitted message (6,452.60% increase for a 250-bytes message in a 16,384-bytes token).

Comm. Type	**Token Size**	**Message size (bytes)**					
		250	**2,500**	**10,000**	**100,000**	**1,000,000**	**10,000,000**
non-secure	**256**	74	75	107	193	745	6,734
	512	74	74	85	127	508	3,806
	1,024	73	73	78	127	367	2,146
	2,048	73	73	72	123	327	1,918
	4,096	73	73	75	123	323	1,512
	8,192	73	73	75	121	307	1,446
	16,384	73	73	74	107	278	1,384
secure	**256**	84	90	128	314	1,808	16,265
	512	84	88	96	251	1,068	9,384
	1,024	84	86	93	186	780	6,530
	2,048	83	83	92	199	604	5,001
	4,096	86	86	95	158	574	4,668
	8,192	87	94	99	152	540	4,376
	16,384	96	103	107	162	500	4,074

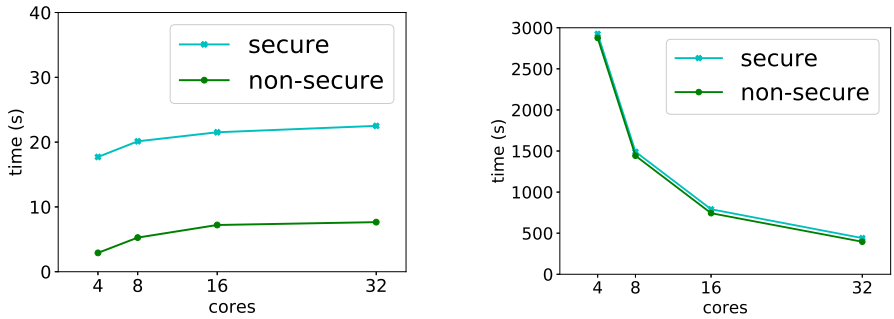
Table 3 Time elapsed (ms) to transfer 250, 2,500, 10,000, 100,000, 1,000,000 and 10,000,000 bytes by token size (256, 512, 1,024, 2,048, 4,096, 8,192, 16,384)

As shown by the time spent on data transfers, security adds an overhead on the communications. Part of the origin of this overhead is the establishment of the security protocol and mutual authentication; and part, for encrypting/decrypting the messages.

In our second test, we aimed to evaluate the impact of using secure communications on COMPSs-Mobile applications. For that purpose, we run HeatSweeper on the mobile, offloading parts of the computation to a cluster composed by eight quad-core VMs in the same OpenNebula cloud. HeatSweeper is an intensive search

algorithm that optimizes the location of 1-to-N heat sources to minimize the time to warm-up the whole surface to a certain temperature. For this purpose, the algorithm runs a set of heat transfer simulations, each one encapsulated in a *simulate* task that generates a report (smaller than 10,000 bytes) describing the result of the simulation. In a second phase, the algorithm selects the best performing configuration by comparing pairs of reports in *getBest* tasks. Given the small size of the data and the number of commands, we fixed the size of the tokens on 2,048 bytes.

We run two different configurations to optimize the placement of up to two heat sources. The low-resolution scenario, representing short-lasting applications, has 9 possible locations and short simulations (up to 50 iterations each). It creates 45 tasks that in total takes 71 seconds to run on the mobile. The high-resolution configuration emulates large computations and considers 25 different spots (325 tasks) on the surface and long simulations (up to 10,000 iterations each) that takes 99,641 seconds (more than 27 hours). Charts 6(a) and 6(b) show the respective evolution of their execution times when increasing the number of available cores from 4 (1 cloud VM) to 32 (8 VMs) comparing the behavior of original COMPSs (no endpoint authentication nor message encryption) against the version with secured communications.



(a) Low-resolution (b) High-resolution

Fig. 6 Elapsed time comparison of HeatSweeper executions using clear and secure communications

In both cases, executions using secure and non-secure communication behave alike; however, security adds a delay of 15 seconds (500-200% overhead depending on the number of cores) for the low-resolution scenario and 50 seconds (2-10% overhead depending on the number of cores) for the high-resolution as shown by Figures 6(a) and 6(b), respectively. This delay appears for two reasons. The first one is the application-level protocol that COMPSs-Mobile follows to execute one task. On a first stage, the master requests to the network the execution of one task. Then a worker would submit a subscription to the network to be noticed when the input data is created (message are within the cluster, so it does not use security). Upon the creation of all the input data, the task becomes dependency-free and requests the transmission of the data value to one of the sources (if the source is the master, it secures the communication). For the simulation tasks of

the application, this protocol enforces the submission of three secured message (the submission of the task, and the request and value transfer of the simulation parameters as an object from the mobile to the worker – request and value transfer happen on the same connection –), which explains 770 ms of this delay according to the results presented before.

The data-sharing protocol also explains the lack of scalability shown in the low-resolution scenario, even when using non-secure communications. The more nodes on the peer-to-peer network, the less likely that data sources information for a specific value are cached on the local worker, and the more messages and hops per message are needed to notice a value creation and query the sources. Subscribing for a data value existence, notifying its existence, querying for the sources to the corresponding node of the DHT and sending them back to the querier host delays up to 4.7 additional seconds the execution of the first task; and thus, the whole application. Unlike the high-resolution scenario, tasks last too short (35 ms) to overlap the fetching of input values for one task with the computation of others; and that blocks the improvement of the application performance when the number of resources is increased.

The second reason for the delay is the number of threads created by a COMPSs-Mobile node. All nodes have one thread dedicated to the reception and submission of TCP packets, a second thread for the application-level management – i.e. token handling and responding to the received COMPSs-Mobile commands– and several additional threads to fulfill with its specific duties of its role in the infrastructure. A single thread for handling all the communications allows a concurrent establishment of several connections and message transmission; however, it serializes the computations related to the application-level message, the GSSAPI negotiations and the wrapping/unwrapping of the received tokens. The low-resolution scenario submits up to 89 tasks in parallel; the high-resolution, 649. This increase on the communication parallelism incurs a growth on the delay in command submissions.

6 Conclusions

This paper describes an extension to secure communications on applications built using COMPSs-Mobile, a framework for parallel applications on mobile cloud computing (MCC) environments. The framework allows applications to run on a mobile phone and offload tasks to more powerful resources; our extension implements the security necessary to communicate across untrusted networks.

Security is implemented wholly within the communication layer, and its benefits are transparently exploited by the application without requiring any modification. Since the extension is encapsulated within the lowest layer of the software stack, the extension is not bound to COMPSs-Mobile but can be adopted by other Java applications using the same communications library.

In the COMPSs-Mobile model, the master node (the mobile device) initially connects to the worker node, so the master needs to have the user credential, and worker nodes must have a host credential. For testing purposes, we use Kerberos credentials as user credentials, and keytabs for host credentials; however, these are obtained outside of COMPSs-Mobile. Other security methods could equally well be used without changing a line of code. Nevertheless, some work is still needed to ensure that credentials are in place prior the application starts offloading tasks.

As discussed in section 5.3, enabling security adds a performance overhead because of the exchange of messages to establish the security context (strongly influenced by the network latency and the number of connections being established concurrently) and the padding bytes used for matching the pre-configured token sizes. In practice, it may be necessary to fine tune the token sizes for obtaining the best performance for each application.

7 Future Directions

In continuation of the work described in this paper, we have identified some potentially interesting future directions. Regarding the performance of our current solution, there is room for improvement if we consider mechanisms to automatically tune the size of the token according to the data to transfer (e.g., submitting a COMPSs-Mobile internal command could use small tokens while data objects use larger token sizes; or transfers could start using small tokens and slowly increase the size of the token as the transfer progresses, ...). Also, it would make sense to reuse an open connection due to the overhead of establishing the connection and secure the context.

As previously mentioned, implementing a cache or interface node between the master and the rest of the workers would be useful, so fewer nodes need to be open to the outside world through the firewall; and thus, reducing the number of needed host credentials.

Related to security and cache, future work could look into supporting delegation, not just through GSSAPI but also delegate onward to other worker nodes so that they can access resources with credentials delegated on behalf of the user.

We also plan to migrate our developments to Federated Identity Management (FIM). As GSSAPI is not a web-based technology , we propose to support FIM via Moonshot [21]. Kerberos allows federations only through cross-realm authentication while Moonshot would allow us to extend it to a full federation using Moonshot.

Like every other computational device, mobile phones and other portable devices such as tablets are getting more powerful, with multi-core CPUs and GPUs. Enabling the support for these devices in COMPSs-Mobile is on-going work.

Acknowledgements This work has been supported by the Spanish Government (contracts TIN2012-34557, TIN2015-65316-P and grants BES-2013-067167, EEBB-I-15-09808 of the Research Training Program and SEV-2011-00067 of Severo Ochoa Program), by Generalitat de Catalunya (contract 2014-SGR-1051) and by the European Commission (ASCETiC project, FP7-ICT-2013.1.2 contract 610874). The second author was partially supported by the European Commission's Horizon2020 programme under grant agreement 653965 (AARC).

References

1. Heterogeneous Secure Multi-level Remote Acceleration Service for Low-Power Integrated Systems and Devices (RAPID). http://rapid-project.eu/
2. Java programming assistant (javassist). http://www.javassist.org

3. MIT Kerberos Consortium. http://www.kerberos.org/software/index.html
4. OASIS Web Services Business Process Execution Language. http://www.oasis-open.org/committees/wsbpel/
5. The treacherous 12: Cloud computing top threats in 2016 (2016)
6. Globus toolkit. http://toolkit.globus.org/toolkit/ (2017)
7. Allen, G., Others: The grid application toolkit: toward generic and easy application programming interfaces for the grid. Proceedings of the IEEE **93**(3), 534–550 (2005)
8. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Others: Above the clouds: A berkeley view of cloud computing (2009)
9. Astalos, J., Cecchini, R., Coghlan, B.A., Cowles, R., U. Epting, T.G., Gomes, J., Groep, D.L., Gug, M., Andrew Hanushevsky, M.H., Jensen, J., Kanellopoulos, C., Kelsey, D.P., R. Marco, I.N., Nicoud, S., O'Callaghan, D., Darcy Quesnel, I.S., Shamardin, L., Skow, D., Sova, M., Wäänänen, A., Wolniewicz, P., Xing, W.: International grid ca interworking and peer review, policy management through the european datagrid certification authority coordination group. Lecture Notes in Computer Science (2005)
10. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation computer systems **25**(6), 599–616 (2009)
11. Chen, H., Zhu, X., Qiu, D., Liu, L., Du, Z.: Scheduling for workflows with security-sensitive intermediate data by selective tasks duplication in clouds. IEEE Transactions on Parallel and Distributed Systems (2017)
12. 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 '11, pp. 301–314. ACM, New York, NY, USA (2011). DOI 10.1145/1966445.1966473. URL http://doi.acm.org/10.1145/1966445.1966473
13. Chun, B.G., Maniatis, P.: Augmented smartphone applications through clone cloud execution. In: Proceedings of the 12th Conference on Hot Topics in Operating Systems, HotOS'09, pp. 8–8. USENIX Association, Berkeley, CA, USA (2009). URL http://dl.acm.org/citation.cfm?id=1855568.1855576
14. Cuervo, E., Balasubramanian, A., Cho, D.k., Wolman, A., Saroiu, S., Chandra, R., Bahl, P.: Maui: Making smartphones last longer with code offload. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys '10, pp. 49–62. ACM, New York, NY, USA (2010). DOI 10.1145/1814433.1814441. URL http://doi.acm.org/10.1145/1814433.1814441
15. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI'04, pp. 10–10. USENIX Association, Berkeley, CA, USA (2004). URL http://dl.acm.org/citation.cfm?id=1251254.1251264
16. Dhinesh Babu, L.D., Venkata Krishna, P.: Honey bee behavior inspired load balancing of tasks in cloud computing environments. Applied Soft Computing Journal **13**(5), 2292–2303 (2013). DOI 10.1016/j.asoc.2013.01.025
17. Fernando, N., Loke, S.W., Rahayu, W.: Mobile cloud computing: A survey. Future generation computer systems **29**(1), 84–106 (2013)
18. Galante, G., Erpen De Bona, L.C., Mury, A.R., Schulze, B., da Rosa Righi, R.: An Analysis of Public Clouds Elasticity in the Execution of Scientific Applications: a Survey. Journal of Grid Computing **14**(2), 193–216 (2016). DOI 10.1007/s10723-016-9361-3. URL http://dx.doi.org/10.1007/s10723-016-9361-3
19. Gupta, M.K.: Akka Essentials. Packt Publishing (2012)
20. Hardt, M., Kannelopoulos, C.e.: Djra1.2 blueprint architectures. https://aarc-project.eu/documents/deliverables/. (April 2017)
21. Howlett, J., Hartman, S., Tschofenig, H., Schaad, J.: Application bridging for federated access beyond web (abfab), architecture. IETF (2016)
22. Humphrey, M., Thompson, M.: Security implications of typical grid computing usage scenarios. https://www.ogf.org/documents/GFD.12.pdf (2000)
23. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev. **41**(3), 59–72 (2007). DOI 10.1145/1272998.1273005. URL http://doi.acm.org/10.1145/1272998.1273005
24. Kemp, R., Palmer, N., Kielmann, T., Bal, H.E.: Cuckoo: A computation offloading framework for smartphones. In: Gris and 0001 [24], pp. 59–79. URL http://dblp.uni-trier.de/db/conf/mobicase/mobicase2010.html#KempPKB10

25. Khan, A.N., Kiah, M.M., Khan, S.U., Madani, S.A.: Towards secure mobile cloud computing: A survey. Future Generation Computer Systems **29**(5), 1278–1299 (2013)
26. Kosta, S., Aucinas, A., Hui, P., Mortier, R., Zhang, X.: Unleashing the power of mobile cloud computing using thinkair. CoRR **abs/1105.3232** (2011). URL http://dblp.uni-trier.de/db/journals/corr/corr1105.html#abs-1105-3232
27. 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: INFOCOM, 2012 Proceedings IEEE, pp. 945–953. IEEE (2012)
28. Lineback, R.: Cellphone ic sales will top total personal computing in 2017. http://www.icinsights.com/data/articles/documents/987.pdf
29. Linn, J.: Generic security service application programming interface, version2, update 1. IETF (2000)
30. Lordan, F., Badia, R.M.: COMPSs-Mobile: parallel programming for mobile cloud computing. Journal of Grid Computing (2017). DOI 10.1007/s10723-017-9409-z. (to appear)
31. Lordan, F., Tejedor, E., Ejarque, J., Rafanell, R., lvarez, 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). DOI 10.1007/s10723-013-9272-5. URL http://dx.doi.org/10.1007/s10723-013-9272-5
32. Missier, P., Soiland-Reyes, S., Owen, S., Tan, W., Nenadic, A., Dunlop, I., Williams, A., Oinn, T., Goble, C.: Taverna, reloaded. In: M. Gertz, T. Hey, B. Ludaescher (eds.) SSDBM 2010. Heidelberg, Germany (2010). URL http://www.taverna.org.uk/pages/wp-content/uploads/2010/04/T2Architecture.pdf
33. Montesi, F., Guidi, C., Lucchi, R., Zavattaro, G.: Jolie: a java orchestration language interpreter engine. Electronic Notes in Theoretical Computer Science **181**, 19–33 (2007)
34. Pashalidis, A., Mitchell, C.: A taxonomy of single sign-on systems. In: Information security and privacy, pp. 219–219. Springer (2003)
35. Rescorla, E.: HTTP Over TLS. RFC 2818 (Informational) (2000). URL http://www.ietf.org/rfc/rfc2818.txt. Updated by RFCs 5785, 7230
36. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of secure computation **4**(11), 169–180 (1978)
37. Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N.: The case for vm-based cloudlets in mobile computing. Pervasive Computing, IEEE **8**(4), 14–23 (2009). DOI 10.1109/MPRV.2009.82
38. Singh, S., Chana, I.: A Survey on Resource Scheduling in Cloud Computing: Issues and Challenges (2016). DOI 10.1007/s10723-015-9359-2
39. Solagna, P., Kannelopoulos, C., Liampotis, N., Hardt, M., Sallé, M., Paetow, S., Malavolti, M., Van Dijk, N., Jensen, J., Liabotis, I., Jankowski, M., Memon, S., Prochazka, M., Oshrin, B., Monticini, B., Short, H., Stevanovich, U.: Existing AAI and available technologies for federated access. AARC Project (2015). URL https://aarc-project.eu/wp-content/uploads/2016/01/MJRA1.1-Existing-AAI-and-available-technologies.pdf
40. Tilevich, E., Smaragdakis, Y.: J-orchestra: Automatic Java application partitioning. Ecoop pp. 178–204 (2002). DOI 10.1145/1555392.1555394. URL http://link.springer.com/chapter/10.1007/3-540-47993-7_8%5Cnhttp://dl.acm.org/citation.cfm?id=680022
41. Tuecke, S., Welch, V., Engert, D., Pearlman, L., Thompson, M.: Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile. RFC 3820 (Proposed Standard) (2004). URL http://www.ietf.org/rfc/rfc3820.txt
42. Vecchiola, C., Chu, X., Buyya, R.: Aneka: A software platform for .net-based cloud computing. CoRR **abs/0907.4622** (2009)
43. Virding, R., Wikström, C., Williams, M.: Concurrent Programming in ERLANG (2Nd Ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1996)
44. Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: A language for distributed parallel scripting. Parallel Comput. **37**(9), 633–652 (2011). DOI 10.1016/j.parco.2011.05.005. URL http://dx.doi.org/10.1016/j.parco.2011.05.005
45. Williams, N., Johansson, L., Hartman, S., Josefsson, S.: Generic security service application programming interface naming extensions (2012)
46. Xie, T., Qin, X.: Security-aware resource allocation for real-time parallel jobs on homogeneous and heterogeneous clusters. IEEE trans. on par. and dist. systems pp. 682–697 (2008). DOI 10.1109/TPDS.2007.70776
47. Zhan, Z.H., Liu, X.F., Gong, Y.J., Zhang, J., Chung, H.S.H., Li, Y.: Cloud Computing Resource Scheduling and a Survey of Its Evolutionary Approaches.

ACM Computing Surveys **47**(4), 1–33 (2015). DOI 10.1145/2788397. URL http://dl.acm.org/citation.cfm?doid=2775083.2788397
48. Zhu, J., Leach, P., Jaganathan, K., Ingersoll, W.: The simple and protected generic security service application programming interface negotiation mechanism (2005)