Journal of Computer and System Sciences 80 (2014) 994–1007

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

# A security framework in G-Hadoop for big data computing across distributed Cloud data centres

Jiaqi Zhao [a], Lizhe Wang [b,*], Jie Tao [c], Jinjun Chen [d,*], Weiye Sun [c], Rajiv Ranjan [e], Joanna Kołodziej [f], Achim Streit [c], Dimitrios Georgakopoulos [e]

[a] School of Basic Sciences, Changchun University of Technology, PR China
[b] Institute of Remote Sensing and Digital Earth, Chinese Academy of Sciences, PR China
[c] Steinbuch Center for Computing, Karlsruhe Institute of Technology, Germany
[d] Faculty of Engineering and IT, University of Technology Sydney (UTS), Australia
[e] Information Engineering Laboratory, CSIRO ICT Centre, Australia
[f] Institute of Computer Science, Cracow University of Technology, Poland

ARTICLE INFO

*Article history:*
Received 30 September 2012
Received in revised form 15 March 2013
Accepted 27 August 2013
Available online 12 February 2014

*Keywords:*
Cloud computing
Massive data processing
Data-intensive computing
Security model
MapReduce

ABSTRACT

MapReduce is regarded as an adequate programming model for large-scale data-intensive applications. The Hadoop framework is a well-known MapReduce implementation that runs the MapReduce tasks on a cluster system. G-Hadoop is an extension of the Hadoop MapReduce framework with the functionality of allowing the MapReduce tasks to run on multiple clusters. However, G-Hadoop simply reuses the user authentication and job submission mechanism of Hadoop, which is designed for a single cluster. This work proposes a new security model for G-Hadoop. The security model is based on several security solutions such as public key cryptography and the SSL protocol, and is dedicatedly designed for distributed environments. This security framework simplifies the users authentication and job submission process of the current G-Hadoop implementation with a single-sign-on approach. In addition, the designed security framework provides a number of different security mechanisms to protect the G-Hadoop system from traditional attacks.



## 1. Introduction

Today, data explosion is commonly observed in various scientific and social domains, such as GeoScience, Life Science, High Energy and Nuclear Physics, as well as Materials and Chemistry. Modern scientific instruments, the Web, and simulation facilities are producing huge data in the range of several petabytes. Currently, MapReduce [3] is commonly used for processing such big data. With a Map and a Reduce function, MapReduce provides simple semantics for users to program data analysis tasks in the code. Additionally, the parallelism in MapReduce is automatically done by a runtime framework, which is especially friendly for application developers.

The Apache Hadoop [1] is one of the well-known MapReduce implementations [3,18,7,8,4,11,16] and has been used/extended by scientists as the base of their own research work [15,19,10,6]. Hadoop's MapReduce architecture is based on a master/slave communication model with one JobTracker as the master and multiple TaskTrackers acting in the role of

* Corresponding authors.
*E-mail addresses:* Lizhe.Wang@gmail.com (L. Wang), jinjun.chen@gmail.com (J. Chen).

http://dx.doi.org/10.1016/j.jcss.2014.02.006
0022-0000/

slaves. Hadoop uses its own file system, the Hadoop Distributed File System (HDFS), to manage the input/output data of the MapReduce applications.

G-Hadoop [25] is a MapReduce implementation targeting on a distributed system with multiple clusters, such as a Grid infrastructures [9,12,22,26,20], a Cloud [13,27,24,21], distributed virtual machines [23], or a multi-data centre infrastructure [28,17]. In order to share data across multiple administrative domains, G-Hadoop replaces Hadoop's native distributed file system with the Gfarm Grid file system [19]. MapReduce applications in G-Hadoop are scheduled across multiple clusters using a hierarchical scheduling approach. The MapReduce tasks are firstly scheduled among the clusters using Hadoop's data-aware scheduling policy and then among compute nodes using the existing cluster scheduler on the target clusters. G-Hadoop maintains the master/slave model of Hadoop, where the slave nodes are simple workers, while the master node accepts the jobs submitted by the user, splits them into smaller tasks, and finally distributes the tasks to the slave nodes.

The current G-Hadoop system reuses the Hadoop mechanisms for user authentication and job submission, which is actually designed for single cluster environments. The mechanism applies the Secure Shell (SSH) protocol to establish a secure connection between a user and the target cluster. This kind of mechanism is not suitable for a distributed environment, like Grid, that contains several large-scale clusters. In G-Hadoop, for example, an individual SSH connection has to be built between the user and each single cluster. This approach does not handle the system as a whole; rather it works separately with the systems components. In addition, with the Hadoop security approach a user must log on to each cluster in order to be authenticated before being capable of using its resources for running MapReduce tasks. This is undoubtedly a tedious task for the users. Therefore, a more general, system-wide solution is required to hide the architecture details of the system as well as to free the user from the burden.

In this work, we designed a new security model for the G-Hadoop framework to meet the above challenges. The security framework is designed with the following properties:

- A single-sign-on process with a user name and password: A user logs on to G-Hadoop once simply with his user name and password. In the following, all resources of the underlying system are available for the user and can be accessed by the user without additional self-participating security processes. The procedure of being authenticated by the different clusters of the underlying system is automatically performed by the security framework in the background.
- Privacy of user information: The user information, such as authentication information, is invisible at the side of the slave nodes. Slave nodes take tasks, which are assigned by the master node, without being aware of user information, including the user name and password.
- Access control: The security framework protects resources of the whole system from misuse or abuse of non-professional users. Users of the system only have the right to access the resource of a cluster that he can access with an SSH connection.
- Scalability: A cluster can be easily integrated or removed from the execution environment without any change of the code on the slave nodes or any modification of the security framework itself.
- Immutability: The security framework does not change the existing security mechanism of the clusters. Users of a cluster can still rely on their own authentication profiles to get access to the clusters.
- Protection against attacks: The security framework protects the system from different common attacks and guarantees the privacy and security by exchanging sensitive information, such as information of authentication and encryption. It is also capable of detecting the fraudulent party of a fake entity to avoid abusing or illegal access to the resources by an attacker.

The remainder of the paper is organized as following. Section 2 gives a short introduction to the G-Hadoop architecture. Section 3 describes the designed security framework in detail. Section 4 demonstrates how the proposed security framework guarantees the security of the underlying system and analyzes its security performance against common attacks. An initial implementation of the framework is described in Section 5. Finally, the paper concludes in Section 6 with a brief summary and several future directions.

## 2. The G-Hadoop architecture

G-Hadoop maintains the Hadoop's MapReduce architecture with a master/slave communication model. The JobTracker is the master server in the MapReduce framework and represents a central service that is responsible for managing the control flow of running MapReduce jobs. The JobTracker receives new jobs from its clients and splits the jobs into smaller tasks. A built-in scheduler dispatches the individual Map and Reduce tasks, and coordinates the order of the map phase and the reduce phase. The JobTracker is also responsible for monitoring the health status of all running tasks and detecting failures and re-scheduling the failing until all tasks of a job are finished.

TaskTrackers are the workhorses of the JobTracker. Each TaskTracker has a specific number of slots denoting how many tasks the TaskTracker is configured to run at the same time. The TaskTrackers periodically report their health status to the JobTracker using a heartbeat message. This message includes the current progress of all running tasks and the number of idle slots available on the TaskTracker node. In response, the JobTracker sends new instructions to the TaskTracker. These instructions include the assignment of a new task or the termination of a running task. To sandbox each running task the

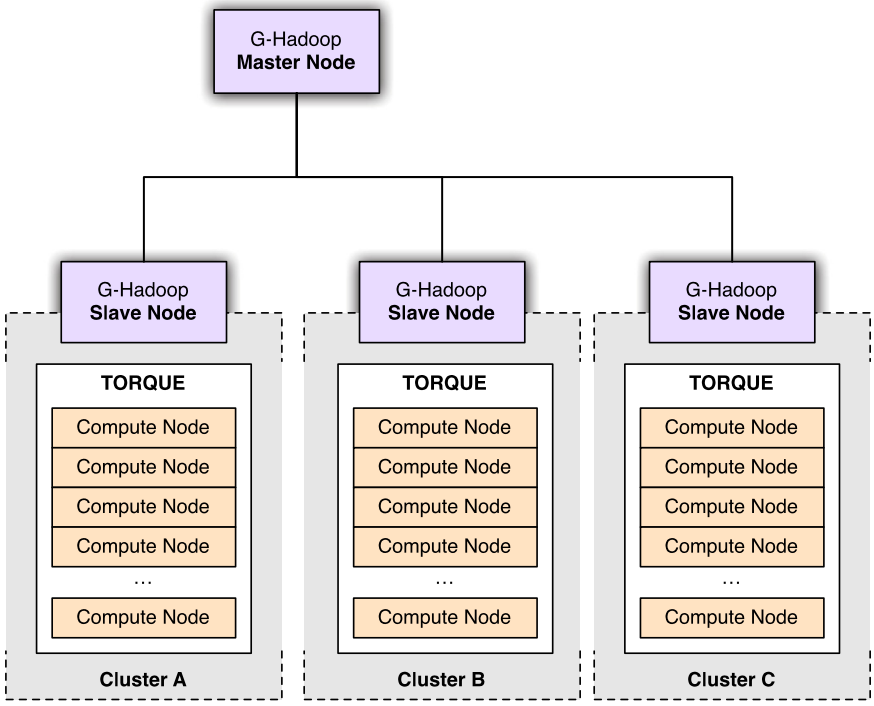


**Fig. 1.** G-Hadoop software architecture.

TaskTracker runs a separate JVM per slot. It is the TaskTracker's responsibility to monitor the health status of the active task, managing its log files, detecting failures, and reporting back to the JobTracker.

Based on the Hadoop MapReduce implementation, we developed the G-Hadoop framework for MapReduce executions in a wider area. The goal of G-Hadoop is to enable large-scale distributed computing across multiple clusters. To share data sets across multiple administrative domains, G-Hadoop replaces HDFS, the Hadoop's native distributed file system, with the Gfarm file system.

Fig. 1 shows an overview of the G-Hadoop high-level architecture and its basic components. As mentioned, the proposed architecture of G-Hadoop follows a master/slave communication model. The master node is the central entity in the G-Hadoop architecture. It is responsible for accepting jobs submitted by the user, splitting the jobs into smaller tasks, and distributing these tasks among its slave nodes. It is also responsible for managing the metadata of all files available in the system. The master node contains a metadata server that manages files distributed among multiple clusters, and a JobTracker that is responsible for splitting jobs into smaller tasks and scheduling these tasks among the participating clusters for execution. Tasks are submitted to the queue of the cluster scheduler (e.g. Torque) using a standard interface. The slave node of G-Hadoop enables the execution of MapReduce tasks on the computing nodes of the participating clusters. Its main component is a TaskTracker that is in charge of accepting and executing tasks from the JobTracker of the master node.

## 3. The security framework

The G-Hadoop prototype currently uses the Secure Shell (SSH) protocol to establish a secure connection between a user and the target cluster. This approach requires a single connection to each participating cluster, and users have to log on to each cluster for being authenticated. Therefore, we designed a novel security framework for G-Hadoop.

### *3.1. The concept*

There are several successful security solutions, including the SSL protocol [14], the Globus Security Infrastructure [5] and the Java security solutions. The cryptographic protocol Secure Sockets Layer (SSL) and its successor Transport Layer Security (TLS) are widely used to provide a secure communication over the public networks. It takes two phases to set up an SSL connection. The first phase uses digital signatures and asymmetric cryptography for authentication, while the second phase is for data transmission. The Globus Security Infrastructure (GSI) is a standard of Gird security and provides the single-sign-on process in the Grid environment as well as authenticated communication by using asymmetric cryptography as the base for its functionality. As a security standard, GSI also meets different security requirements. The Java platform not only supports the cross-platform capabilities, but also provides a variety of security features and solutions. Most of these security features and solutions are encapsulated in three standard Java extensions: Java Cryptography Extension (JCE),

Java Secure Sockets Extension (JSSE), and Java Authentication and Authorization Service (JAAS). Each of these extensions contributes certain technologies to the security features and solutions.

Our security model follows the authentication concept of the Globus Security Infrastructure (GSI) [2,5], while using SSL for the communication between the master node and the CA server. GSI is a standard for Gird security. It provides a single-sign-on process and an authenticated communication by using asymmetric cryptography as the base for its functionality. As a security standard, GSI adopts several techniques to meet different security requirements. This includes the authentication mechanisms for all entities in a Grid, integrity of the messages that are sent within a Grid, and delegation of the authority from an entity to another. A certificate, which contains the identity of users or services, is the key of the GSI authentication approach. In this work, we use similar certificate for the authentication between the master node and slave nodes of G-Hadoop, where the master node is in charge of the single-sign-on process. The user needs only provide his user name and password or simply log on to the master node; jobs can then be submitted to the clusters without requesting any other resources. A secure connection between the master node and slave nodes is established by the security framework using a mechanism that imitates the SSL handshaking phase.

### 3.2. Applied terms

The following terms are used in the description of the novel security model:

- User instance: A user instance is an object that is created by the master node for a user who has active jobs – the job is initialized or in execution. The user instance is deleted by the master node, when the user does not have active jobs any more. A user instance is an important but dynamic element of the security framework. It is related with other information and data of the corresponding user as well as the authentication process on slave nodes. User instances are independent between any two different users. Another property of user instance is sharing. If there are more than one active jobs of a user, these jobs share a common user instance. In case that the user is logged out but there are still active jobs for him, the user instance will not be deleted, thus to implement an "off-line execution" mode by using the user instance approach.
- Proxy credential and slave credential: Proxy credentials and slave credentials are used by our security framework to describe two different certificates. They are issued by the same CA (Certification Authority).
  The proxy credential is designed for slave nodes to authenticate the master node, actually the user instances maintained on the master node. For each user instance a proxy credential is issued. After a user instance is authenticated by a slave node, a secure communication is established and the proxy credential is not required any more. Therefore, proxy credentials are designed as one-time-usage and have short life cycle to protect the authentication process from MITM attacks and replay attacks. A proxy credential contains the following information: the CA identity, the expiration time, a public key of the master node as well as its life cycle, and a random message that is generated by the CA.
  In contrast to a proxy credential, a slave credential has a long life cycle and is owned by a slave node. Slave credentials are used by the master node to authenticate the slave nodes during the procedure of establishing a communication channel between the master node and a slave node. A slave credential contains information about the CA identity and the public key of the corresponding slave node.
- User session: The term *session* is typically used to describe the interactive information interchange between two communication parties. In this security framework, we use *session* to specify the information of a user instance, and hence call it *user session*. It serves as the identity of a user instance for the slave nodes after the user instance has been authenticated. Therefore, a user session has the same life cycle as the corresponding user instance.

### 3.3. The security model

As mentioned above, G-Hadoop was developed for a computing environment with large-scale distributed multiple clusters. The system components, including the CA server and the master node of G-Hadoop, may be distributed over the world and connected via a public network, for example the Internet, where various security threats exist. Hence, one component can trust another only when the authentication process is successfully conducted. However, the CA server has to be trusted by all components of the system. This is the basis condition of using certificates for authentication.

In our security framework, the CA server is an important component that issues proxy credentials and slave credentials. Since the Gfarm file system, which is the underlying file system of G-Hadoop, applies the GSI security mechanism and has already a CA server, we simply reuse this server as the CA server of the proposed security framework. In addition, the slave nodes in G-Hadoop have their own GSI certificate for the Gfarm file system. This GSI certificate is also simply used as the slave credential.

Fig. 2 shows the architecture model of the proposed security framework. The new designed security framework is an extension of the current G-Hadoop system without changing its master–slave architecture. The additional component in this architecture is the CA server, which is needed to issue proxy credentials as well as slave credentials.

The proposed security framework extends G-Hadoop in the phase of submitting jobs from a user and the phase of job termination, where additional steps are performed to authenticate the communication parties and to establish a secure connection before executing jobs/tasks.

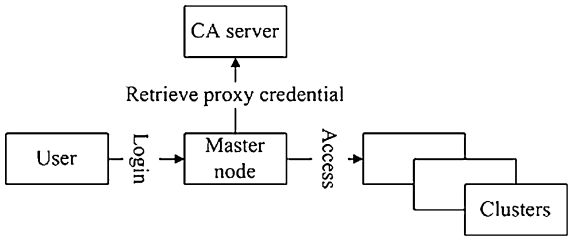


**Fig. 2.** The G-Hadoop security architecture.

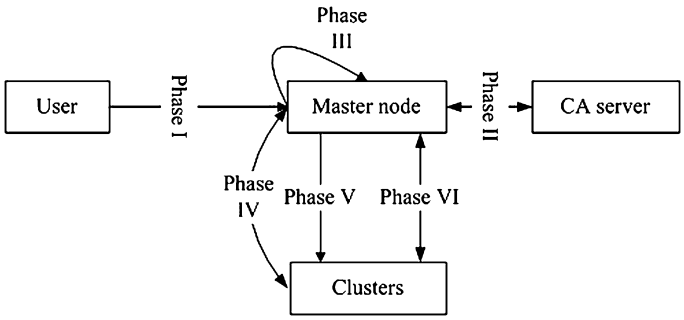


**Fig. 3.** The work flow of the authentication procedure.

Fig. 3 demonstrates the authentication procedure and the interaction between the components of the designed security framework. The whole work flow consists of the following main phases: user authentication, proxy credential assignment, preparing authentication information on the master node, authentication of the master node and slave nodes, as well as job execution, termination, and disconnection.

We apply a public key cryptographic algorithm in this work, similar to the one used by Digital Signature. To simplify the description, we use *PBA* to represent this public key algorithm. Its encryption and decryption can be described with the following formats:

$$PBA.encrypt(m, k)$$

for encrypting a message $m$ with the public key $k$, and

$$PBA.decrypt(m, k)$$

for decrypting a message $m$ with the private key $k$.

**Phase I: User authentication.** From the view of users the way to submit a job to the G-Hadoop system with the security framework is not different from the submission process of the initial G-Hadoop implementation. However, with the security model a user does not need log on to each slave node. Instead, a single log-on to the master node with his user name and password pair $\{U, P\}$ is sufficient.

After receiving the user authentication information $\{U, P\}$ from a user, the master node searches its database, which stores the user authentication information, to check if the user name exists, if the password is correct, and if the user has right to use the G-Hadoop resources. If one of these checks does not pass through, the master node delivers an error message as feedback to the user. Otherwise, the user is authenticated by the master node.

Being authenticated by the master node, the user can now submit jobs to G-Hadoop. In the following, the master node accepts the job, initializes it, creates a user instance for it, and finally contacts the CA server to apply a proxy credential for the user instance. It is possible that the user instance already exists, meaning that the off-line execution is activated. In this case, the master node only shows the current status of the active jobs to the user, and no further operation is performed.

**Phase II: Applying and issuing a proxy credential.** To apply and issue a proxy credential, the first step, Phase II(a) in Fig. 3, is to set up a secure connection between the master node and the CA server. The SSL protocol [14] is applied in this phase to provide the secure connection. During the setup of an SSL connection, the master node acts as the client of SSL and the CA server as the server side. According to the flow of the SSL protocol with client authentication, the master node and the CA server are authenticated by each other before any data transmission. After the authentication via the handshaking process of SSL, a secure connection with encryption between them is established.

In the next step, the master node sends a request to apply a proxy credential for the user instance, as depicted by Phase II(b) in Fig. 3. Since a proxy credential is the authentication information of a user instance on the slave nodes, it must be unique with each use case to avoid abusing, even for the same user instance but different jobs. For this purpose,

the master node generates a random key pair {*MN_Pub*, *MN_Prv*}. This key pair is used later during the authentication phase between the master node and slave nodes. The master node retains the private key *MN_Prv* and sends the public key *MN_Pub* to the CA server.

After receiving the randomly generated public key *MN_Pub* from the master node, the CA server generates a random message *CA_Rand* and specifies the life cycle for the public key *MN_Pub* of the master node. Specifying the life cycle of *MN_Pub* also regulates the life cycle of the corresponding user session that will be generated in the next phase. The CA server signs then a proxy credential for the current request of the master node by using the technique of Digital Signature. The proxy credential contains the identity of the CA, its expiration time, the public key *MN_Pub* of the master node, the life cycle of *MN_Pub*, and the randomly generated message *CA_Rand*. The new generated proxy credential is then sent to the master node, together with the random message *CA_Rand*.

With the randomly generated public key *MN_Pub* for the master node as well as the random message *CA_Rand* of the CA server, the uniqueness can be ensured. This is an important property of proxy credentials. The random message *CA_Rand* is also designed with the purpose for authentication of the master node and slave nodes by each other, the same functionality as the random message that used in the GSI scheme. Therefore, the master node should know about the random message. Out of this reason the random message *CA_Rand* is sent to the master node as well. After receiving the proxy credential as well as the random message *CA_Rand*, the master node disconnects the SSL connection. This phase is then terminated.

**Phase III: Generating user session and preparing the authentication for slave nodes.** In this step the master node generates a user session of the current user instance for later communication and interaction with the slave nodes. The user session *U_Session* simply uses the MD5 hash value of the user name and the random message to guarantee the uniqueness of the user session and to protect the user privacy from the slave nodes:

$$U\_Session = MD5(U + CA\_Rand)$$

The user session has the same life cycle as the public key *MN_Pub*, whose life period is specified by the CA server and signed in the proxy credential.

According to the user accessing right to the slave nodes, the master node creates a list of all the available slave nodes for the current user instance. In the following operations, the master node will only distribute jobs/tasks to the slave nodes in the list. This mechanism ensures that only the slave nodes that can be accessed by the user with SSH are provided to the user, so that abusing or misusing of the resources in G-Hadoop can be avoided.

Similar to the master node, the slave nodes must also make sure that the master node is not a fake but the real master node that is authenticated by the CA server. Therefore, the authentication information of the master node has to be prepared as well.

Since G-Hadoop deals with a large-scale distributed system connected with public networks, transmitting sensitive information such as authentication information in plain text is insecure and inadvisable. A secure connection is required. The SSL connection is an option. However, using this protocol an additional SSL certificate, others than the proxy credential, is needed for the authentication process during the handshaking process to set up the SSL connection. In addition, a storage space on the master node is required to save the identities of the slave nodes. Furthermore, if a new slave node is added or an existing one is removed, the master node must also renew its database for the identities to maintain the synchronization. Due to these reasons, we propose a secure connection between the master node and slave nodes using the proxy credential, without the necessity of storing the identities of slave nodes on the master node. The details will be given in the following subsection, together with the description of the next phase.

Unlike the SSL handshaking process, the master node does not send a Hello message to the slave nodes. Instead, it sends the proxy credential, the random message *CA_Rand* from the CA server, and the user session. The proxy credential is sent directly, while the random message *CA_Rand* and the user session are encrypted with the private key *MN_Prv* that was generated in Phase II:

$$MN\_Rand = PBA.encrypt(CA\_Rand, MN\_Prv)$$

$$MN\_U\_Session = PBA.encrypt(U\_Session, MN\_Prv)$$

By now, all necessary information from the view of the master node is readily prepared. The information, including the authentication information *MN_Rand*, encrypted session user name *MN_U_Session*, and the proxy credential issued by the CA, is sent to the slave node, and the authentication process goes to the next phase.

**Phase IV: Handshaking between the master node and slave nodes.** This phase is similar to the handshaking phase of the SSL connection. In this phase, the master node and slave nodes must be authenticated by each other, and a secure communication for further data transmission is set up. The operations performed in this phase are the same for all slave nodes.

As depicted in Fig. 4, the handshaking phase is divided into three steps. In the first handshaking step, the master node sends the messages that are prepared in the last phase to the slave node. The slave node tries to authenticate the master node with the proxy credential and the authentication information *MN_Rand* from the master node. It first decrypts the proxy credential with the public key of the CA server for the digital signature. If the proxy credential is the real one that

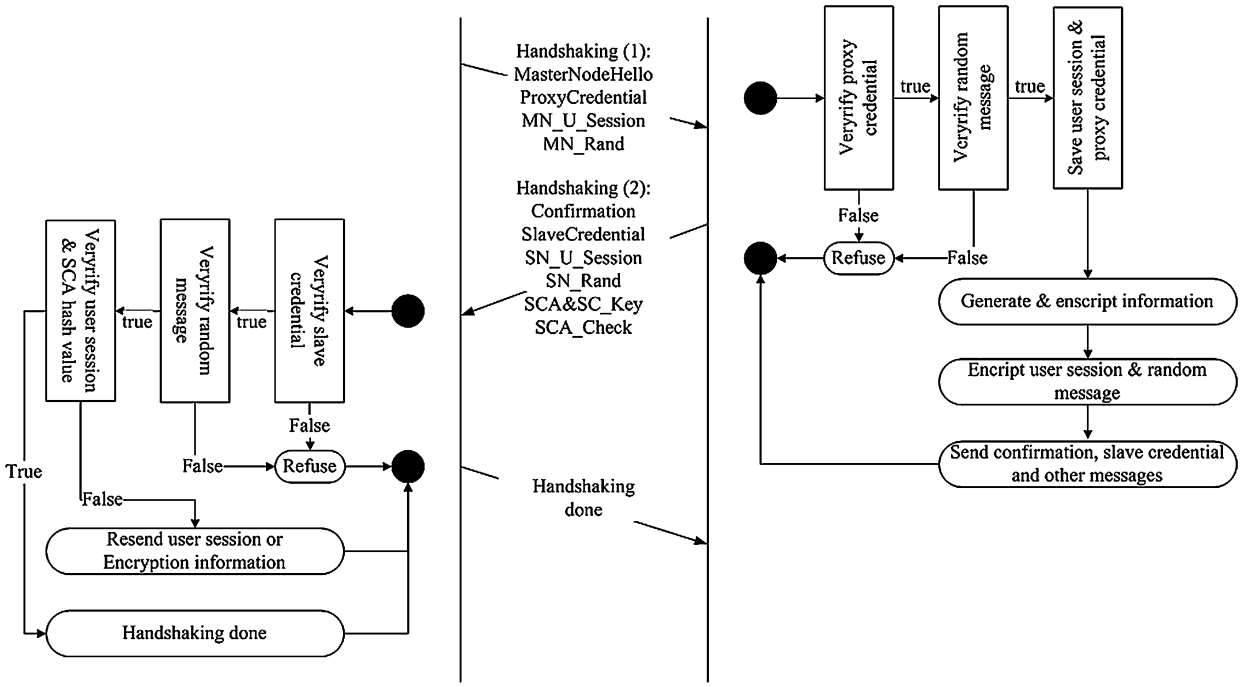


**Fig. 4.** Flow diagram of the handshaking phase.

was signed by the CA server, the slave node should have all the information that is encapsulated in the proxy credential. It then checks whether the identity of the CA server in the proxy credential is correct. If not, the master node is not authenticated. Otherwise, the slave node takes the public key *MN_Pub* of the master node from the proxy credential, and decrypts the message *MN_Rand* with the key *MN_Pub* to get the message *CA_Rand* that is generated by the CA server:

$$CA\_Rand' = PBA.decrypt(MN\_Rand, MN\_Pub)$$

It then compares the decrypted message $CA\_Rand'$ with the original message that is encapsulated in the proxy credential. The expiration time of the proxy credential as well as the expiration time of the public key *MN_Pub* are also checked. If one of these checks results false, the master node is not authenticated as well. In this case, a refusal message with reasons is sent back to the master node. The authentication phase is then terminated.

Otherwise, if all these checks are passed through meaning that the master node is authenticated, the slave node sends a confirmation message as feedback to the master node and then acquires the user session information by decrypting the message *MN_U_Session* with the public key *MN_Pub*:

$$U\_Session = PBA.decrypt(MN\_Rand, MN\_Pub)$$

The user session *U_Session* and the life cycle of the public key *MN_Pub* are stored on the slave node. *U_Session* will be used later as authentication information of the current user instance for task assignment and execution on the slave node. The proxy credential is stored on the slave node as well to protect the slave node from replay attacks. All information is maintained on the slave node, until a job-finished message is received from the master node. In this case, the slave node deletes the information.

The first handshaking aims to authenticate the master node by the slave node. In the second handshaking step, the slave node is authenticated by the master node. In addition, the encryption scheme and the corresponding key for later communication are also determined in this step.

The slave node encrypts the random message of the CA server with its own private key *SN_Prv* to acquire the encrypted message *SN_Rand*:

$$SN\_Rand = PBA.encrypt(CA\_Rand, SN\_Prv)$$

This encrypted message is used to prove that the slave node is the actual owner of the slave credential that is received by the master node. It has the same functionality as the message *MN_Rand*. The user session *U_Session* is also encrypted with the private key *SN_Prv* of the slave node:

$$SN\_U\_Session = PBA.encrypt(U\_Session, SN\_Prv)$$

The purpose of encrypting the user session is to make sure that the user session, received by the slave node, is the same one that the master node has sent. At the same time, the slave node selects a symmetric cryptographic algorithm *SCA*, generates

a corresponding key *SC_Key* for this algorithm, and saves the pair {*SCA*, *SC_key*} together with the user session. Another copy of the name of the chosen cryptographic algorithm and its key *SC_Key* is encrypted with the public key *MN_Pub* of the master node. This algorithm and the corresponding key *SC_Key* will be used to encrypt all the data transmissions between the master node and this slave node after this phase. Here, it must be noticed that the key to encrypt the random message *CA_Rand* as well as the user session is the private key *SN_Prv* of the slave node, while the algorithm's name *SCA* and the key *SC_Key* is encrypted with the public key *MN_Pub* of the master node. To avoid modification of the encryption information {*SCA*, *SC_key*} by an attacker, the hash value of this encryption information is computed by the slave node. The hash value is encrypted with the private key of the slave node and sent then to the master node:

$$SCA\_Check = PBA.encrypt(SCA, SN\_Prv)$$

In addition, the slave credential of the slave node is also sent to the master node without encryption.

The master node receives a feedback message from the slave node, which can be either a refusal or a confirmation message. Normally, the real master node will only receive a confirmation, but the fraudulent party of a fake master node could receive a refusal.

The slave credential of the slave node, the encrypted message *SN_Rand*, the encrypted user session *SN_U_Session*, and the message of encryption information should be acquired by the master node in the second handshaking. After receiving these messages, the master node tries to verify the slave credential with the public key of the CA server and decrypts the message *SN_Rand* with the public key of the slave node that is extracted from the slave credential. If the slave credential is unacceptable and the decrypted message from *SN_Rand* is different to the original message *CA_Rand*, the master node treats the slave node as a fake node and terminates the communication with it. Otherwise, the slave node is authenticated by the master node. The master node then decrypts the encrypted user session with the public key *SN_Pub* of the slave node and the encrypted message with its private key *MN_Prv* to acquire the user session, cryptographic algorithms *SCA*, and the corresponding key *SC_Key*. If the user session from the slave node is not identical to the original user session *U_Session*, the master node sends its encrypted user session *MN_U_Session* again to the slave node and asks the slave node to confirm it with *SN_U_Session*. The similar operation is performed to the encryption information {*SCA*, *SC_key*} as well. The master node computes the hash value of the encryption information {*SCA*, *SC_key*} and compares it with the value that is decrypted with the public key *SN_Pub* of the slave node from the message *SCA_Check*. If the checking operation results a false value, the master node asks the slave node to generate and send back a new encryption information. These two checking operations aim to make sure that the user session as well as the encryption information is not altered by a third party, who could be able to perform an MITM attack to change the information and use this information to access the resource on the slave node.

In the third handshaking step, the master node simply sends a confirmation message to the slave node, which is encrypted with the symmetric cryptographic algorithm *SCA* and the key *SC_Key*.

By now, the master node and the slave node have been authenticated by each other. A cryptographic algorithm *SCA* with the corresponding key *SC_Key*, that will be used to encrypt all the data transmissions in further communications, has been determined by both the master node and the slave node. A third party of the communication is also excluded. A secure connection between the master node and the slave node is established. The work in this phase is done.

**Phase V: Job execution.** In this phase, the G-Hadoop system executes the jobs that the user has submitted. The execution of the job is similar to G-Hadoop without the designed security framework. One difference is that the master node must assign a task to a slave node with the user session *U_Session* as an authentication information, and correspondingly, the slave node checks the user session for expiration time before executing tasks. If the user session is expired, the slave node asks the master node to update the life cycle of the user session. In this case, the master node simply applies a new proxy credential and sends it directly to the slave node. By receiving the new proxy credential, the slave node updates the expiration time of the user session as well as the proxy credential.

**Phase VI: Terminating a job.** The active job is terminated when all the tasks of the job on the slave nodes are finished, or a fatal error occurred during the execution so that the tasks cannot be executed further more. The master node sends the result or some information about the fatal error to the user. It also sends a job-finished message with the user session to all the participating slave nodes. The latter clears all information related to the user session, including the proxy credential and the encryption information {*SCA*, *SC_Key*}. If the job-finished message is caused by a fatal error, the slave node terminates the current task of the job. The slave node is now disconnected from the master node. The master node deletes the proxy credential for the user instance and all other information related to this user instance. The connection of the master node to the slave node is also cut down. The last phase of the security process as well as the user job is completed.

## 4. Security analysis

This section analyzes the security mechanisms of the designed security framework and discusses how they protect the G-Hadoop system against different attacks, abusing, or misusing.

### 4.1. User control on the master node

One of the primary goal of this security framework is to supply users with the single-sign-on process to submit a job to G-Hadoop. It has to be noted that G-Hadoop covers a distributed system and its components are physically distributed across resource centers connected with a public network. A user may not have any accessing right to one or more slave nodes. If a simple user database that only stores the user name and password is used, the master node could not be aware of the accessing right, which will cause the break of rights control on the slave nodes. Therefore, in our security framework, the user database also stores the accessing right of slave nodes for each user, in addition to the user login information. During the execution of a user's jobs, the master node only tries to connect with the slave nodes that can be found in the corresponding user's item in the database. In this case, the master node is able to avoid abusing of the resources in G-Hadoop.

In addition, the single-sign-on process exposes the whole G-Hadoop to attackers, if an attacker has obtained the data in the user database. Therefore, keeping the database secure is also one of the most important missions.

### 4.2. Security analysis with applying and issuing a proxy credential

As described above in the security work flow, the second authentication phase is to apply and issue a proxy credential. Normally, the master node and the CA server are distributed at different locations and connected via a public network, which is insecure to transmit sensitive information. On the other hand, a proxy credential functions as the identity of the user instance that is given by the CA server. It is used to authenticate the master node with the related user instance by slave nodes, so that the user instance can access or use the resources of G-Hadoop. Therefore, a secure and trustful communication between the master node and the CA is required. In this work, the SSL protocol is chosen to guarantee the security of the communication. According to the features of the SSL protocol, SSL has the ability to avoid the man-in-the-middle (MITM) attack, version rollback attack, delay attack, and replay attack. In addition, it provides an extension solution of client-authenticated handshaking, which performs the authentication of both communication parties before the secure connection is set up. This feature meets the requirement of our security framework, where the master node, which acts as the client in the SSL connection, is authenticated by the CA server and a fraudulent party as the fake master node is refused by the CA server.

The SSL protocol ensures the authentication of the master node and the CA server with an encryption of the communication. However, like all other security solutions, SSL cannot guarantee the absolute security. For example, an attacker can listen to the communication, capture the message, and obtain the key to encrypt the communication with the help of cryptographic analysis. In this situation, the attacker could obtain the proxy credential. Considering this situation, a random message *CA_Rand*, which is generated by the CA server, is introduced for the proposed security framework. This random message is used to authenticate the entities of G-Hadoop as the fingerprint of them. The security analysis of this fingerprint will be discussed in the following subsections.

### 4.3. Secure communication between master and slave

The security framework is designed to realize the single-sign-on process from the user's sight. After a user has been authenticated by the master node, the master node tries to log on to the slave nodes, where the user has right to access. Authenticating the master node by slave nodes is the most important step of the security framework.

Therefore, preventing this step from attacking or abusing is one of the most important security missions of our work. By designing this security framework, several attacking methods has been taken into account, and the security framework is designed to have the ability of keeping the authentication phase secure from these attacks.

#### 4.3.1. Protection against MITM attacks

The random message *CA_Rand* generated by the CA server is the second protection next to credentials. If an attacker has performed an MITM attack during the phase of applying and issuing the proxy credential or in the first handshaking step of Phase IV, the attacker may be able to have obtained the public key *MN_Pub* of the master node, the proxy credential, as well as the random message *CA_Rand*. However, the random message *CA_Rand* is encrypted with the private key *MN_Prv* of the master node. The attacker is unable to get the private key *MN_Prv* of the master node, which is randomly generated by the master node and never transmitted over the public networks. Therefore, the attacker cannot encrypt the random message *CA_Rand* and a failure will be caused during the authentication step by the slave nodes.

The only way to get the private key of the master node is to crack the cryptographic algorithm. However it needs time, potentially more than one month. At this moment, the master node must have already been connected with the slave nodes. Additionally, the use of randomly generated key pair $\{MN\_Pub, MN\_Prv\}$ of the master node also increases the difficulty to crack the public key cryptographic algorithm, because different key pairs are used for different user instances.

In another situation, if an MITM attack occurs between the master node and a slave node during the second handshaking step in Phase IV, the attacker has already captured all the messages. However, since the attacker does not own the private key *MN_Prv* of the master node, he cannot decrypt the message that contains the encryption information $\{SCA, SC\_Key\}$ for the further communication. The attacker may also attempt to carry out an MITM attack with a fake slave node in this

handshaking step. Nevertheless, he will not be authenticated by the master node, because he does not have the private key of the slave node to encrypt the random message *CA_Rand*, which acts as the fingerprint of the slave nodes.

Therefore, with the mechanisms of using the public key scheme on the randomly generated message as the fingerprint, the abuse of proxy credentials by a third party as well as a fake master or slave node of a fraudulent party is prevented. The MITM attack against Phase II and Phase IV is affectless.

#### 4.3.2. Protection against delay attacks and replay attacks

Beside the mechanisms against MITM attacks, the security framework is designed to be able of preventing delay attacks and replay attacks as well. A proxy credential is combined with a digitally signed expiration time, which is only valid for a short time. If a delay attack is performed, the proxy credential will be expired and refused by the slave nodes.

Since a proxy credential of an active user instance is stored on the slave node, the slave node can detect the replay attack easily. Another request of authentication with a proxy credential that is already saved on the slave node will be treated as a replay attack, because proxy credentials are designed as one-time-usage.

#### 4.3.3. Protection for the integrity of user session

In the first handshaking of Phase IV, the user session is encrypted with the private key *MN_Prv* of the master node. An attacker, who has listened to the communication between the master node and the CA server or has obtained the public key of the CA server, may be able to decrypt the user session. However, after Phase IV the user session is encrypted with the encryption information {*SCA*, *SC_Key*} determined by the slave node. Since the attacker could not have the knowledge of the encryption information, he is hence not able to encrypt the user session to perform an attack. On the other hand, if the attacker has altered the user session, the verification of the user session in the second handshaking step of Phase IV cannot pass through. The modification can be easily detected by the master node. Therefore, the first and second handshaking with encrypted user session ensure the integrity of the user session.

#### 4.3.4. Secure connection for data transmission

The connection between the master node and slave nodes are encrypted by a symmetric cryptographic algorithm, which is determined by the slave node, while the master node is noticed by encrypting the message with the public key *MN_Pub* of the master node. It is already pointed out that an attacker cannot have any knowledge about the symmetric cryptographic algorithm *SCA* and the corresponding key *SC_Key*. In case of an MITM attack, an attacker may modify the encryption information {*SCA*, *SC_Key*} from the slave node by using the public key of the master node to encrypt the encryption information that is generated by the attacker. But with the hash value *SCA_Check*, the master node can verify the validity and integrity of the encryption information. Since the hash value *SCA_Check* is encrypted with the private key *SN_Prv* of the salve node, it is impossible for the attacker to encrypt the encryption information correctly. The attack is hence avoided.

To increase the difficulty of cryptographic analysis, which an attack may perform to crack the symmetric cryptographic algorithms, the algorithms and the key are chosen by the slave node, rather than the master node. In this case, each connection between the master node and a slave node can be encrypted in different ways, even for the connection to the same slave node with different user instances.

In addition, the reason to choose a symmetric cryptographic algorithm by the slave node is also due to the cost of computing. Since the encryption with a public key cryptographic algorithm is computationally expensive in contrast to the symmetric cryptography, it is no more necessary to use the public key cryptography to encrypt the message after a secure communication between the master node and a slave node has been established. A symmetric cryptography such as AES can provide sufficient security for the communication.

## 5. Prototypical implementation

The proposed G-Hadoop security framework was implemented as a single plug-in of the Hadoop MapReduce implementation and can be started with the G-Hadoop/Hadoop SHELL facility. The plug-in, called Single-Sign-ON (SSON), is flexibly designed that can be integrated or removed from the Hadoop system as the user's expect.

### 5.1. Architecture overview

The security framework is composed of six major modules, which are the Common Module, Security Module, User Management Module, CA Node Module, Master Node Module and the Slave Node Module. Fig. 5 illustrates the interaction between these components. Each module provides special functionalities, where the

- Common Module is the library of the security framework, providing the basic interfaces, such as event handlers and listeners, as well as random message generators, for the requirement of other modules;
- Security Module defines and implements the security mechanisms applied in this security framework. This also includes the message protocols that are used in the phase of distributing proxy credential and hand-shaking;
- User Management Module implements all user related operations and functionalities, including user profiles, user session and user instance. In addition, this module also provides the functionality of centrally managing the user information as well as the application interface for job submission;

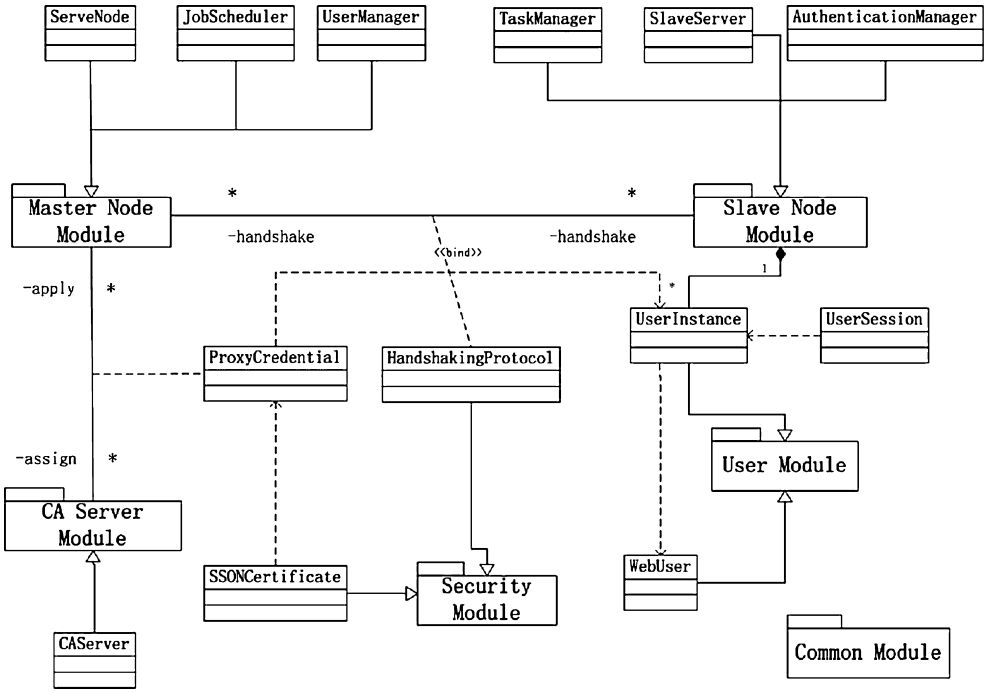


**Fig. 5.** Software architecture of the prototypical implementation.

- CA Node Module is in charge of issues with proxy credentials, such as request and distribution;
- Master Node Module provides the functionality of the master node, including HTTP Service, the centralization of user and job management, as well as maintaining the authenticated user list and running the job scheduler. An additional functionality of this module is to apply proxy credentials, accomplish the authentication phase with slave nodes, and set up a secure connection to slave nodes for further job execution;
- Slave Node Module first performs the authentication procedure with the master node and then executes the Map/Reduce tasks that are assigned to them. It also maintains the proxy credential list to manage the authentication information for tasks.

### *5.2. Implementation details*

The most important module in this security solution is the Security Module, which contains the implementation of functionalities to sign and verify an SSON certificate in a digital signature message as proxy credentials. In this module, the messages for requesting and distributing a proxy credential as well as the three-way handshake messages between the master node and the slave nodes are encapsulated as specified protocols, which are implemented as serializable objects. In this way, the messages between the communication parties can be easily managed without taking care of the packet loss in the network.

```
public final class ApplicationMessage
              implements Serializable {...}
public final class DistributionMessage
              implements Serializable {...}
```

Another advantage of using such serializable protocols is the transformation of message to byte stream, which is the single way for data transmission through network. The SSON certificate and proxy credential in this module are the core of the whole security framework. The information of an SSON certificate is visible to the certificate owner and the authenticator but these information must not be modified after it has been assigned by the CA server in order to meet the non-modification feature of a certificate.

The User Management Module provides the user management application interfaces and the implementation of user related information. These information contains the three important terminologies of this security framework: user profile, user instance and user session. For the purpose of single-sign-on and the management of users in G-Hadoop with Web page based operations, a class named '*WebUser*' is defined to present a user substance. When a user has logged on to the G-Hadoop system with a correct user name and password pair through the authentication Web page that is provided by this security framework, the user is authenticated and a user substance of type *WebUser* is created in the background of

the master node to hold resources and information of the user. After that the user substance is inserted into the active user list that is maintained by the user management component in the Master Node Module. The resources that a user substance occupies contain the basic user information and the user profile. When a user substance is created, the user profile is created as well, if there is not any off-line jobs in execution or the waiting status. Otherwise, the user profile that has been previously created is allocated to the new created WebUser substance.

If a user submits a job, the job is added to the job list of the user profile and the job exists in the list till the execution is completed. The result list of the user profile stores the results of all finished jobs during the life period of the user profile. Another list holds the slave nodes, whose resources cannot be accessed by the user. With such mechanisms, the job scheduler in the Master Node Module does not try to set up the connection to the slave nodes that are in this list, so that the resources in G-Hadoop are protected from misusing.

The CA Node Module implements the functionality of the CA Server, i.e., handling the application of proxy credentials from the master node as well as generating a corresponding SSON certificate and distributing it as a proxy credential to the master node. This module has a single class named *CAServer*, which provides all the functionality for the mission of this component. As depicted in its name, a *CAServer* substance is a server, which is implemented as an SSL socket server. This SSL socket server is configured to request client authentication, so that both communication parties must be authenticated by each other before a secure connection is set up. With such mechanisms, the CA server can ensure that the proxy credential is transferred to the real master node safely.

For being able of handling concurrent multiple requests from the master node the CA server adopts the technique of *multitasking*. After setting up the SSL connection with the master node, the CA server creates a new thread to deal with the further requests from the master node. By finishing the reading of the information in the request message, the CA server generates a random message and species the life period for the public key of the master node as well as the expiration time of the SSON certificate in GMT. After that the CA server calls the signature method of ProxyCredential in the Security Module to create and digitally sign the key.

The Master Node Module is the kernel of the entire security solution. It provides the HTTP service for the interaction between the users and the whole G-Hadoop system. It also takes the responsibility of centrally managing the users and their jobs. The most important role of the Master Node Module is to apply proxy credentials, authenticate the slave nodes and set up a secure connection with the slave nodes.

As depicted in the architecture overview in Fig. 5, this module contains a ServerNode, a JobScheduler and a UserManager. The ServerNode acts as the control center of the whole security framework, where the Jetty server is launched to run the HTTP service to provide the log in/out, job submission and other Web-based user interfaces for using the G-Hadoop resources. Besides the Jetty server, the ServerNode also runs other important components to support the requirements of the security framework.

The JobScheduler component is designed to manage and schedule all the user jobs that are not executed or are being executed. As mentioned, the security framework enables the feature of off-line execution, which is not supported in the original G-Hadoop system. Therefore, the job scheduler component has the ability of managing and maintaining the job information as long as the user has submitted it. For this three hash table structures are implemented in the job scheduler component to hold these information. When a user submits a job, the job is first added into the waiting list, in case that the job is accepted by the security framework. When the job is ready for execution, the job scheduler checks if the user instance of the job owner exists in the active instance list. If it exists, the existing user instance is directly used to set up the connection with the slave nodes to execute the job; otherwise, the job scheduler creates a separated thread to apply a proxy credential. This thread tries to set up an SSL connection to the CA server. According to the SSL protocol, the CA server and the master node must be authenticated with each other during the process to set up the SSL connection. After the connection is set up, the master node encapsulates the random generated public key and the corresponding algorithms into the application message and sends it to the CA server. On another hand, the master node creates a user instance and saves the private key as well as the public key algorithms into the user instance. The application process at the master node's side is completed.

The Slave Node Module contains a task component, an authentication management component and a slave node handshaking component. The task component is implemented as the subclass of the original Hadoop task to perform similar work as the original one but with the extensions for adapting to the mechanisms of this security framework. The extension is implemented to enable the update of the proxy credential if it is expired. After receiving the new proxy credential from the master node, the slave node updates the expiration time of the user session as well as the proxy credential itself and puts the task into execution. The authentication management component provides the methods to check the validity of the authentication information from master node and manages these information, including the proxy credentials and the token of user session, which are related to the tasks. In addition, the authentication manager has a special list to store the authentication information that is asked to be deleted after all jobs of a user have finished its execution.

### 5.3. Initial evaluation

We evaluated the implemented security framework by verifying the functionality of individual modules and the framework as a whole. For the experiments we set up a master node, a CA server node and several slave nodes. We conducted a number of different tests.

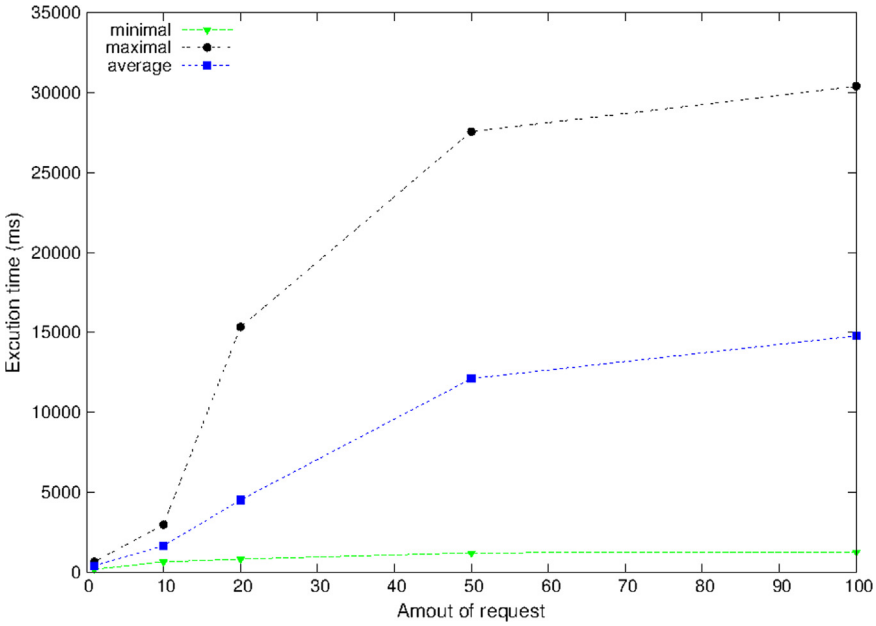


**Fig. 6.** Execution time for requesting proxy credentials on the master node.

The first test was performed to evaluate the ability of the master node in handling the user requests for job submission. As described in the previous section, the users submit a job through a JSP page, which runs a separated thread for each user. This means that multiple threads will run on the same node in case that several users access the submission page and submit the job at the same time. We modeled this scenario and found that the more users try to submit a job at the same time, the more time is needed to perform the submission process. For example, when the number of requests is increased from one to one hundred, the minimal time needed for handling a request raises in five folds. A solution is to deploy separated servers to run the HTTP service for guaranteeing the performance of master node. This is one of the future work.

The second test was done to evaluate the performance of requesting proxy credentials. We created different numbers of requests and for each case we measured the minimal and maximal time for handling a single request. We performed the same test 20 times and calculated the middle value of the 20 tests. Fig. 6 shows the experimental results, where the $x$-axis depicts the amount of the requests and the $y$-axis shows the minimal, maximal and average time to process a single request. The time is computed from the time point of generating the request message to the time point of the master node receiving the distribution message, meaning that the CA server has received the request message and sent the distribution message to the master node.

As shown in the figure, the performance of applying and distributing proxy credentials is strongly influenced by the amount of requests under the threshold of 50. However, when the number of requests is larger than 50, the influence is getting smaller. This could be caused by the network condition, while the request message as well as the distribution message are transmitted much more frequently, if there is a lot of requests for proxy credentials. Therefore, the configuration of the threshold for the maximal amount of application requests should be configured to a suitable value. If it is too small, the efficiency of the security framework is limited.

## 6. Conclusions

This work designed and implemented a security framework for running *MapReduce* tasks across different clusters in a distributed environment. The security framework provides users with a single-sign-on process to submit jobs to G-Hadoop. In addition, it applies various security mechanisms to protect the G-Hadoop system from attacks as well as abusing or misusing. These security mechanisms are based on some current security solutions, for example SSL and cryptographic algorithm, or the concepts of other security solutions such as GSI. Some concepts, for example, proxy credentials, user session, and user instance, are applied in this security framework as well to provide the functionalities of the framework.

With these security mechanisms the designed security framework has the ability to prevent the most common attacks, such as MITM attack, replay attack, and delay attack, and ensures a secure communication of G-Hadoop over public networks. In addition, it adopts different mechanisms to protect the resources of G-Hadoop from abusing or misusing. On the whole, it provides a trustful and complete solution of the single-sign-on process for the user to access G-Hadoop. For further improvement, job execution in Phase V as well as the encryption algorithms and keys will be designed as changeable to increase the difficulty of cryptographic analysis by an attacker.

## References


[1] Apache Hadoop project, Web Page http://hadoop.apache.org/.
[2] A. Chakrabarti, Grid Computing Security, Springer, 2007.
[3] Jeffrey Dean, Sanjay Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (January 2008) 107–113.
[4] Adam Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, Ville H. Tuulos, Misco: a MapReduce framework for mobile systems, in: Proceedings of the 3rd International Conference on Pervasive Technologies Related to Assistive Environments, PETRA'10, ACM, New York, NY, USA, 2010, pp. 32:1–32:8.
[5] Globus. Overview of the Grid Security Infrastructure, Available at: http://toolkit.globus.org/toolkit/docs/3.2/gsi/key/index.html, 2014.
[6] Sijie Guo, Jin Xiong, Weiping Wang, Rubao Lee Mastiff, A MapReduce-based system for time-based big data analytics, in: 2012 IEEE International Conference on Cluster Computing, 2012, pp. 72–80.
[7] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, Tuyong Wang, Mars: a MapReduce framework on graphics processors, in: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT'08, ACM, New York, NY, USA, 2008, pp. 260–269.
[8] Shadi Ibrahim, Hai Jin, Bin Cheng, Haijun Cao, Song Wu, Li Qi, CLOUDLET: towards MapReduce implementation on virtual machines, in: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, HPDC'09, ACM, New York, NY, USA, 2009, pp. 65–66.
[9] Joanna Kolodziej, Fatos Xhafa, Integration of task abortion and security requirements in GA-based meta-heuristics for independent batch Grid scheduling, Comput. Math. Appl. 63 (2) (2012) 350–364.
[10] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, Zhe Zhang, MOON: MapReduce On Opportunistic eNvironments, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC'10, ACM, New York, NY, USA, 2010, pp. 95–106.
[11] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis, Evaluating MapReduce for multi-core and multiprocessor systems, in: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, IEEE Computer Society, Washington, DC, USA, 2007, pp. 13–24.
[12] Rajiv Ranjan, Aaron Harwood, Rajkumar Buyya, Coordinated load management in peer-to-peer coupled federated grid systems, J. Supercomput. 61 (2) (2012) 292–316.
[13] Rajiv Ranjan, Karan Mitra, Dimitrios Georgakopoulos, MediaWise cloud content orchestrator, J. Internet Serv. Appl. 4 (1) (2013) 1–14.
[14] Eric Rescorla, SSL and TLS Designing and Building Secure Systems, Addison–Wesley, 2002.
[15] Robert Schaefer, Aleksander Byrski, Joanna Kolodziej, Maciej Smolka, An agent-based model of hierarchic genetic search, Comput. Math. Appl. 64 (12) (2012) 3763–3776.
[16] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, Huazhong Yang, FPMR: MapReduce framework on FPGA, in: Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA'10, ACM, New York, NY, USA, 2010, pp. 93–102.
[17] Weijing Song, Shasha Yue, Lizhe Wang, Wanfeng Zhang, Dingsheng Liu, Task scheduling of massive spatial data processing across distributed data centers: What's new?, in: IEEE 17th International Conference on Parallel and Distributed Systems, 2011, pp. 976–981.
[18] Bing Tang, Mircea Moca, Stephane Chevalier, Haiwu He, Gilles Fedak, Towards MapReduce for Desktop Grid computing, in: Proceedings of the 2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC'10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 193–200.
[19] Osamu Tatebe, Kohei Hiraga, Noriyuki Soda, Gfarm Grid file system, New Gener. Comput. 28 (3) (2010) 257–275.
[20] Lizhe Wang, Dan Chen, Ze Deng, Fang Huang, Review: Large scale distributed visualization on computational Grids: A review, Comput. Electr. Eng. 37 (4) (July 2011) 403–416.
[21] Lizhe Wang, Dan Chen, Yangyang Hu, Yan Ma, Jian Wang, Towards enabling cyberinfrastructure as a service in clouds, Comput. Electr. Eng. 39 (1) (January 2013) 3–14.
[22] Lizhe Wang, Dan Chen, Fang Huang, Virtual workflow system for distributed collaborative scientific applications on Grids, Comput. Electr. Eng. 37 (3) (2011) 300–310.
[23] Lizhe Wang, Dan Chen, Jiaqi Zhao, Jie Tao, Resource management of distributed virtual machines, Int. J. Ad Hoc Ubiq. Comput. 10 (2) (July 2012) 96–111.
[24] Lizhe Wang, Marcel Kunze, Jie Tao, Gregor von Laszewski, Towards building a cloud for scientific applications, Adv. Eng. Softw. 42 (9) (2011) 714–722.
[25] Lizhe Wang, Jie Tao, Rajiv Ranjan, Holger Marten, Achim Streit, Jingying Chen, Dan Chen, G-Hadoop: MapReduce across distributed data centers for data-intensive computing, Future Gener. Comput. Syst. 29 (3) (March 2013) 739–750.
[26] Lizhe Wang, Gregor von Laszewski, Jie Tao, Marcel Kunze, Virtual data system on distributed virtual machines in computational Grids, Int. J. Ad Hoc Ubiq. Comput. 6 (4) (2010) 194–204.
[27] Lizhe Wang, Gregor von Laszewski, Andrew J. Younge, Xi He, Marcel Kunze, Jie Tao, Cheng Fu, Cloud computing: a perspective study, New Gener. Comput. 28 (2) (2010) 137–146.
[28] Wanfeng Zhang, Lizhe Wang, Weijing Song, Yan Ma, Dingsheng Liu, Peng Liu, Dan Chen, Towards building a multi-datacenter infrastructure for massive remote sensing image processing, Concurr. Comput.: Pract. Exper. 25 (12) (2013) 1798–1812.