Data and Code Integrity in Grid Environments

RACHEL AKIMANA AND OLIVIER MARKOWITCH
Département d'Informatique
Université Libre de Bruxelles
Bd. du Triomphe – CP212, 1050 Bruxelles
BELGIUM
{rakimana, omarkow}@ulb.ac.be

Abstract: - In a large distributed system such as the Grid, ensuring data integrity is of particular importance. Since in a same network honest users and possible malicious entities live together, the risks of unauthorized alterations of data and information cannot be ignored. This concern on data integrity has two faces. On the one hand, insurance has to be given that data has not been altered by unauthorized hands. This is called integrity of passive data. On the other hand, users may want to have the guarantee that the jobs they submit on the Grid are executed in the right way with the proper input data, and result on reliable output data. This second flavor of integrity is called integrity of active data. In this paper, we consider these integrity concerns, identify the needs when considering privacy aspects of passive data and a Grid's adapted framework for integrity of active data.

Key-Words: - Grid security, integrity, privacy

1 Introduction

A Grid is a (widely) distributed system composed of resources of many computing systems. It is usually used to resolve scientific or technical problems that require a large amount of resources. Grids perform heavy computations on large amount of data, by breaking them down into many smaller pieces, or provide the ability to process many computations in parallel. Therefore, a Grid is a parallel and distributed system that allows to share and aggregate geographically distributed resources.

Since a Grid is usually a huge system, a lot of different users are using its resources. Some of these users may be malicious entities. Therefore, the risks of unauthorized alterations of data and information that are stored or processed on Grid resources, or even that are traveling on the Grid's network, cannot be disregarded.

Large amount of data are stored on Grid's resources. These data are used as input for distributed executions and/or are the results of these executions. It is crucial that these data are not illegitimately altered. Therefore, we have to ensure the integrity of these data. We are dealing here with the *integrity of passive data*. On another hand, the users need to have the guarantee that the asked executions are correctly processed. The jobs submitted on a Grid have to be executed in the right way with the proper input data. And in consequence, the resulting output data have to be reliable. This is also a kind of integrity that we call *integrity of active data*.

2 Integrity of passive data

When considering the context of the Grid, passive data may refer to data resulting from experiments and simulations. These data are generally organized in databases accessible to Grid users. Grid users want to get the assurance that the consulted data has not been altered by unauthorized hands. Usually, hashing functions and/or digital signatures are used to ensure data integrity. For example, keyed hash functions (MAC) may be used on database contents when the corresponding secret keys are securely shared. However, the secret key management in a large distributed system like a Grid is not straightforward. Digital signature schemes can be used to guarantee the integrity of data whose owner is known (to allow the public key-based signature verification). However, using digital signatures in a classical way are not an appropriate tool when we deal with the integrity of anonymous data. For example, we may consider a medical database accessible to patients and their physicians in which the patients' privacy is ensured by replacing their name by a code number. Therefore, a patient cannot sign his own data in order to guarantee their authenticity without breaking its privacy at the time of the signature verification. Even, if a physician signs these data, the patient's identity could be possibly established by considering the set of patients of the related physician. On the basis of this situation, we propose, in the next subsection, a protocol that ensures the integrity of a database while preserving the privacy of the related

entities concerned by the stored information.

2.1 Integrity and privacy

When looking at privacy concerns in addition to the integrity service, we may consider that each entity authorized to access to an information and possibly to change it must be able to sign the new version of this information in such a way that its identity has to be indistinguishable from the identities of all the user entitled to access the database in writing.

Group signature schemes [3] allow each entity that belongs to a group to sign an information in such a way that at the time of a signature's verification it appears that the signature comes from the group without indicating which member of the group actually generated the signature. Moreover, in case of problem (when an authorized entity makes an dishonest modification of the database for example) group signature schemes allow a designated group authority to reveal the identity of the signer.

Another kind of group signatures is ring signatures [8], that have the advantage, in comparison to classical group signatures, to allow a member of a group to sign an information knowing only its own signature secret key and the verification public key of all the other members of the group. Therefore, there is no group setup nor any need for a group manager.

However, ring signatures do not offer, in case of problem, a mechanism to reveal the identity of the entity who generated a signature. In group signatures the management of the keys is sometimes heavy. Moreover, in our Grid's framework it may be problematic that the access control authority is not able to check if the identity of the entity who gained access to an information is the same that the identity of the entity who, afterwards, made a modification and signed it.

Therefore, we propose here a protocol that allows genuine users to make modifications on anonymous data in such a way that the identity of the corresponding data owners as well as the identity of the user who makes the modifications remain secret. The protocol uses a trusted third party (TTP), associated to the database(s) in which these anonymous data is stored, that realizes the access control and that ensures indirectly the integrity of the database. When considering our previous medical context, the data owner is a patient and the entity allowed to modify these data (the user) is his physician.

2.1.1 The protocol

We consider data stored on Grid resources that have to be accessible and modifiable by authorized users in an anonymous way. We present a protocol that takes into consideration the privacy of the entities that may be related to the stored data, while ensuring the integrity of these data.

Since the data are stored on Grid resources managed by a database administrator that realizes the access control, we can use this particular framework to propose a protocol based on the existence of a trusted third party (TTP), that may be the database administrator, in order to issue integrity tokens. When an information is modified, the TTP delivers the corresponding integrity token, based on its digital signature, at the place of the authorized user that made the modification. Using a TTP allows to be exempted from the management of a group and from the corresponding group signature key. Moreover, the protocol allows the TTP to reveal the identity of an authorized user that made a dishonest modification in the database.

We use the following notations: $sign_{user}(m)$ means that the user signs the hash of the message m with his private key; $E_{TTP}(x)$ is the asymmetric encryption of the information x with the TTP's public key; $E_k(y)$ is the symmetric encryption of the information y with the secret key k.

At the first step of the protocol, a user that wishes to make a modification on an information stored in a Grid resource provides to the corresponding database administrator his credentials, which prove that he is entitled to access and modify the database. We suppose here that the TTP and the database administrator are a unique entity. The user also sends the current date (and time), a randomly chosen session key k as well as his digital signature on these information. All these information are sent to the TTP ciphered with the TTP's public key. User $\rightarrow$ TTP: $E_{TTP}(user's\ credentials, date, k, sign_{user}(date, k))$.

If the access is granted by the TTP (thanks to appropriate credentials and date), at the second step, the user transmits to the TTP the ciphered description of the modifications that have to be made on the database and his digital signature on this description. The description of the modifications are the position in the database where the modifications have to be made and the updated data that have to replace those that appear in the indicated position. Since the description of the modifications (and more precisely

the updated data) may be of important size, in order to be efficient, it is ciphered symmetrically with the secret session key k provided at the first step of the protocol. User $\rightarrow$ TTP: $E_k(modifications, date, sign_{user}(modifications, date))$.

The TTP deciphers the received message and verifies (1) if the date indicated in the first and second step are the same, (2) if the two steps were both made in a time close to that date, (3) the user's digital signature on the description of the modification. If these checks are correct, the TTP makes the expected modifications in the database, produces its signature on the modified data ($sign_{TTP}(updated\ data)$) and stores it in the database and stores in a private place the evidence that proves that the user asked for the modifications that were done: $sign_{user}(date, k))$ and $sign_{user}(modifications, date)$.

Integrity is ensured due to the presence of the TTP's digital signature on the data. Any entity that accesses the database is then able to check whether these stored data were not modified in an unauthorized way. Only the TTP's signatures appear in the database. Therefore, if the stored data are anonymized, no information about the identity of the entities concerned by the data may be inferred from the data or from the integrity token that are the digital signatures of the TTP. However, if a modification made in the database is litigious, the TTP may be asked to reveal the identity of the user that made the given modification. If the TTP considers the revelation request as legitimate, it discloses the user's identity, by publishing $sign_{user}(date, k))$ and $sign_{user}(modifications, date)$.

3 Integrity of active data

In this section, we investigate mechanisms that allow to detect whether a job has been executed correctly or whether its code has been modified by a malicious hand.

Usually, it is hard to prevent such modifications since a malicious system manager is always able to reach and act on a job that is executed on his node. Providing digitally signed information about the job to be executed allows to check if the information about the job were not altered during its travel on the network but does not prevent the target node owner to execute something else.

Traditionally, the mechanisms that allow to ensure a correct distant execution of jobs are related to fault-tolerant distributed systems.

3.1 Terminology

Distributed systems are made up of processes, located on one or more sites, that communicate with one another to offer services to upper layer applications [5]. The term *fault* is usually used to name a defect at the lowest level of abstraction. A fault may cause an error that leads to a system failure. There are three fault models according to the system behaviors that they induce. The *crash failure model* in which processors simply stop executing at a specific point in time; the *fail-stop model* where a processor crashes in such a way that its neighbors can detect it; and, finally, the *byzantine fault model* in which processors may behave arbitrarily, even in a malevolent way. The alteration of code enters in the category of byzantine faults. When processors can experience byzantine failures, a set of processors implementing a t-fault-tolerant state machine must have at least $2t + 1$ replicas and the output of the set is the output produced by the majority of the replicas. If processors experience only fail-stop failures, then a set containing $t + 1$ replicas suffice and the output of the set can be the outputs produced by any of its members. A system correctness is always proved with respect to a specific fault model. *Fault-tolerance* is the ability of a system to behave in a well-defined manner once faults occur.

3.2 Previous works

Many existing solutions to fault-tolerant distributed systems impose that (part of) the jobs are executed many times. In case of such jobs replication, the strategy used by the user to find good results among the set of results that he has received is called *voting*. This is done under the assumption that, among nodes that have executed one job, there is at least one honest node that has returned a good result.

Server replication, also known as state machine approach has been used up now as a popular mechanism for building fault-tolerant distributed services. A state machine consists of state variables that represent the different states in which the machine can be as well as the commands allowing to change from one state to another possible one. A t-fault-tolerant version of a state machine can be implemented by replicating that state machine and running a replica on each of t processors in a distributed system. It is assumed that replicas being run by non-faulty processors start in the same initial state and execute the same requests in the same order, so

each replica will perform the same operations and produce the same output [10]. Replication is also used as a solution for improving the scalability of a distributed service.

In [5], an overview on fundamental techniques that implement replicated services is presented. This work emphasizes on the relationship between replication techniques and group communication and considers that the correctness criterion is the linearizability that gives the illusion of non-replicated servers. Two fundamental replication techniques ensure linearizability: (1) the *primary-backup replication* where one *primary* replica plays a special role of interacting directly with clients who address requests, whereas the other replicas are *backups* that interact only with the primary (in case of the primary fails, one of the backups can become the primary); (2) *active replication*, also called state machine approach, gives to all replicas the same role without the centralized control of the primary-backup technique. Invocations are sent to all replicas that process the invocations. The client waits until it receives the first response or a majority of identical responses.

In [10], Schneider presents a detailed model of the state machine approach for implementing fault-tolerant services. The paper discusses fault-tolerance in the framework of the byzantine fault and fail-stop models. System reconfiguration techniques for removing faulty components and integrating repaired components are also considered.

In [1], a new programming abstraction called *resilient object* is introduced. Each resilient object provides some services to a set of sites where it is represented by components to which requests can be issued using remote procedure calls in the way of the primary-backups replication system. The resulting distributed system gives behavior indistinguishable from a single-site instantiation of the original specification.

In [7], Reiter proposes protocols to facilitate the development of high-integrity services that retain their availability and correctness despite the malicious penetration of some component servers by an attacker. These protocols were developed to facilitate reliable communications between a given number of servers that implement the same service. The paper emphasized on the replication of some critical services like authentication services or certification authorities. Four main protocols are developed : (1) the group membership protocol, that supposes the existence of a group of servers that implement a given service and provide the abstraction of a group of operational servers. The group members may change to reflect the perceived failure, recovery of servers as well as the addition of new servers. The membership protocol ensures that if sufficiently many members of a group request that a member be removed, then that server will eventually be removed from the group; (2) the reliable group multicast protocol, that provides an interface through which group members can multicast messages to other group members. The protocol assumes that fewer than one third of servers in a group are faulty; (3) the atomic multicast protocol, that is similar to the reliable group multicast protocol. Moreover, it offers an additional functionality that determines the order in which messages are delivered to group servers; (4) the outvoting protocol, that ensures that the replies delivered to clients are only those sent by correct servers.

In [6], Krishnamurthy et al. evoke the server replication approach in order to tolerate timing faults. Only the first response received for a request is delivered to the client. Thus, a timing failure occurs only if no response was received from any of the replicas within t time units after the request was sent.

In [2], the authors investigate the case of a state machine replication system that tolerates byzantine faults which can be caused by malicious attacks or software errors. This approach emphasizes on faulty replicas recovery by refreshing state automatically. Because of the recovery, the system can tolerate any number of faults over the lifetime of the system.

In [9], the framework of redundancy and voting is presented in the area of Volunteer Computing. The authors show how voting and redundancy systems are inefficient to reduce the error rate in accepted results when the ratio of faulty hosts on average of all hosts is not small in a given system. The mechanism of spot-checking is proposed as a solution. Spot-checking consists in that the master node gives to worker nodes jobs whose results are known in advance or can be easily verified afterwards. The concept of credibility is also introduced (a worker's creditability depends, between others, on its answers to the submitted spots-checks or on the comparisons of results received for the same work from different workers). The author showed how the combination of voting, spot-checking and credibility can be used to shrink the error rate in accepted results.

In [4], the authors introduce on the Grid the concept of nodes' reputation, which is not too different from the concept of worker's credibility introduced in [9].

In our work, the state machine approach has been also used in the purpose of ensuring data integrity. The differences of our approach and the others in the framework of server replication are the strategy used in tasks' distribution on computing nodes, the voting strategy as well as the subsequent actions like the removal of faulty nodes and their re-integration.

The primary-backups system of [5] and [1] is not suitable with byzantine faults since the primary may be a malicious node. In our work, we rather use the concept of active replications of [5] enriched with hints that guide user's decisions in case where all the replicas answers are different. Our work considers byzantine faults in Grid computing environment. The removal of a faulty node is based on the faulty replies it returns on clients' requests. However, contrary to [10], we distinguish faulty replies induced by malicious actions from these induced by involuntary failure in the system. The group membership protocol proposed in [7] considers fail-stop failures since it is assumed that when one of the replica services is faulty the other replica services can detect it and request the removal of the faulty service. In Grid computing, the different nodes that execute the same task ignore each other, unless they are dishonest members of a coalition. In [6] it is only considered that right replies are those returned in time. In our work, we consider also faulty nodes providing a wrong result in the required time. We cannot either proceed like in [2] where an automatic state refreshing is considered, in our case the re-integration of a node that was faulty is done after a given number of successful tests.

3.3 Active data integrity on the Grid

The k-resilient scheme that we propose, that fits the framework of the Grid, uses replications to achieve active data integrity but tries to reduce this redundancy by using spot-checkings. The degree of replication depends on the credibility c ($0 \leq c \leq 1$) that each user gives to the Grid (the more a user is confident in the Grid, the higher c will be and the smaller the redundancy will be). The protocol considers that there is no coalition (in order to organize the wrong execution of a task) of more than $k-1$ hosts (k-resilient).

Let us consider a user who has to launch a job J composed by n tasks t_i: $J = (t_1, \ldots, t_n)$. At the first round of execution, the resource broker launches the n tasks t_i. Tasks are distributed randomly over the computing nodes according to the number of nodes offered by the Grid for the job and their corresponding computing power (some nodes may receive many different tasks).

At the end of the first round, the user assumes that $n_p = c \cdot n$ tasks have been executed correctly. Consequently, $n' = n - n_p$ tasks have to be executed again. The user does not wait the end of the first round to execute again n' tasks. At the beginning of the first round, the user chooses randomly the n' tasks that will have to be replicated k times and launches them.

For each task t_i, after a delay d_i the user considers that the corresponding results have to be available. Therefore, he makes the following checks:

- if t_i was planned to be executed only once and if results are obtained for it, then the user stores the results and considers that the host that has executed t_i behaved correctly.
- if t_i was planned to be executed only once and if no results are obtained for it, then the user launches again t_i k times. We assume that the probability that a same task is assigned on the same host for two different rounds of execution is arbitrary small. The user considers that the node that has executed t_i did not behave correctly and contacts the TTP to record a complaint about this node.
- if t_i was executed more than once and if at most $k' < k$ obtained results, acquired over all already executed rounds, are the same, then the user launches again t_i k times (assuming that the probability that a same task is assigned on the same host for two different rounds of execution is arbitrary small).
- if t_i was executed more than once and if at least k obtained results, acquired over all already executed rounds, are the same, the user stores these results, considers that the hosts that produced these identical results behave correctly and considers that all the other hosts that provided different results for this task t_i did not behave correctly. He contacts the TTP to record a complaint about these hosts.

We assume the use of a TTP that will manage a list, called *banned list*, containing a reference to the hosts that do not behave correctly. When a user launches a task, the resource broker selects a node to execute it that is not listed in the banned list. A computing node is

said to behave incorrectly if the TTP attests that the results that the node provides, after the execution of given tasks, are erroneous or are not made available in the expected time.

The TTP records all the complaints provided by the users about the hosts. The TTP begins to secretly spot-check these nodes by sending ε tasks execution requests for which the TTP knows the correct corresponding answers. The computing nodes have to ignore that they are spot-checked (for example, the TTP may use the identity of an arbitrary Grid user and may launch tasks that are indistinguishable from real usual tasks). If several spot-checked nodes provide, even once, an identical incorrect answer, they are supposed to be colluding malicious nodes, and all of them are registered on the banned list. If a checked node is not considered as a colluding node, then if it answers incorrectly more than once or does not answer, it is supposed to be an independent malicious node or to be a node that experiences a failure that is not yet corrected. In both cases the node is registered on the banned list. Otherwise, if the spot-checked node answers incorrectly only once, it is supposed to be an honest node that experienced a temporary failure. In that case, the node is not registered in the banned list. A node does not remain on the banned list forever, the TTP spot-checks regularly the banned nodes and if a node provides correct answers ε times in a row, the TTP removes it from the banned list.

Note that if a user is fully confident in the behavior of all the hosts, $c = 1$, no redundancy appears in the execution of the tasks. In contrary, if the user does not trust any of the hosts, $c = 0$, all the tasks composing his jobs will be replicated. Between these two extreme views, the user may dynamically (since the user may change the value of c, for example, on the basis of the current content of the banned list) parametrize the number of tasks that have to be replicated. Moreover, each user can also choose an appropriate value k depending on the supposed maximum size of the possible coalitions of dishonest nodes.

Acknowledgement

The authors express their gratitude to Nicolás González-Deleito for the help he gave to accomplish this work.

References

[1] K. Birman. Replication and fault-tolerance in the isis system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 79–86, Dec. 1985.

[2] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 273–288, Oct. 2000.

[3] D. Chaum and E. van Heijst. Group signatures. In *Proceedings of Advances in Cryptology – Eurocrypt '91*, volume 547 of *Lecture Notes in Computer Science*, pages 257–265. Springer-Verlag, 1992.

[4] A. Gilbert, A. Abraham, and M. Paprzycki. A system for ensuring data integrity in grid environments. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)*, volume 1, pages 435–439. IEEE Computer Society, Apr. 2004.

[5] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer – Special Issue on Fault Tolerance*, 30(4):68–74, Apr. 1997.

[6] S. Krishnamurthy, W. H. Sanders, and M. Cukier. A dynamic replica selection algorithm for tolerating timing faults. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2001)*, pages 107–116, July 2001.

[7] M. K. Reiter. The rampart toolkit for building high-integrity services. In *Proceedings of Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 99–110. Springer-Verlag, 1995.

[8] R. Rivest, A. Shamir, and Y. Tauman. How to leak a secret. In *Proceedings of Advances in Cryptology – Asiacrypt 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 552–565. Springer-Verlag, 2001.

[9] L. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid*, May 2001.

[10] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.