USENIX Association

Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems

Lihue, Hawaii, USA
May 18–21, 2003

Secure Data Replication over Untrusted Hosts

Bogdan C. Popescu
bpopescu@cs.vu.nl

Bruno Crispo
crispo@cs.vu.nl

Andrew S. Tanenbaum
ast@cs.vu.nl

Department of Computer Science,
Vrije Universiteit, Amsterdam, The Netherlands

Abstract

Data replication is a widely used technique for achieving fault tolerance and improved performance. With the advent of content delivery networks, it is becoming more and more frequent that data content is placed on hosts that are not directly controlled by the content owner, and because of this, security mechanisms to protect data integrity are necessary. In this paper we present a system architecture that allows arbitrary queries to be supported on data content replicated on untrusted servers. To prevent these servers from returning erroneous answers to client queries, we make use of a small number of trusted hosts that randomly check these answers and take corrective action whenever necessary. Additionally, our system employs an audit mechanism that guarantees that any untrusted server acting maliciously will eventually be detected and excluded from the system.

1 Introduction

Secure data replication on untrusted hosts has received a considerable amount of attention in the past few years. There are two generic mechanisms to handle this problem: state signing and state machine replication [16]. Solutions based on state signing [7, 2, 6, 11, 13, 3] can only support semi-static data content and restrictive, pre-defined types of queries. Furthermore, all these systems, except for the one described in [11], require that state updates are executed on trusted servers. On the other hand, systems based on state machine replication [4, 15, 10] allow untrusted servers handle the updates and support random queries, but require any particular operation to be executed multiple times (on different hosts), which greatly increases the amount of computing resources needed.

In this paper we present a system architecture that allows dynamic data replication with support for random queries, while avoiding much of the overhead associated with state machine replication. We are able to achieve this by providing only statistical guarantees on the correctness of any given query, combined with a background audit mechanism that detects false responses with a high degree of probability so corrective action can be taken. Our system is configurable, so it can easily provide 100% correctness and/or 100% false response detection, at the expense of operational performance.

Allowing erroneous behavior and taking corrective action only **after** an error has occured may seem a strange policy; however, our model is based on the assumption that byzantine failures from untrusted components of the system are rare, so the system can be optimized to give best performance in common case, which is when everything works correctly.

This paper is organized as follows: Section 2 introduces our system model, Section 3 describes the algorithms used to handle read and write operations on replicated data, Section 4 discusses several variants of our basic algorithms, and the operational scenarios where such variants may be appropriate, Section 5 reviews the related work in this area, and Section 6 concludes.

2 System Model

In this paper we consider a system model consisting of the following elements:

- The **data content**; this can be a database, the contents of a large Web site, or a file system. The data content needs to support both read and write operations; however, in our model we expect the number of reads to be at least an order of magnitude larger than the number of writes. The read operations can be very complex; they can request parts of the data content, but also the results of applying aggregation functions on this content. Taking the example of a file system, it should not only support operations of the type *read FileName*, but also operations of the type *grep Expression Path*.

- The **content owner**; this is one individual or organization which administers the content, and is in charge of setting an access control policy for it. For the purpose of this paper, we assume that data secrecy is not an issue, so the access control policy is only concerned with operations that modify the content.

- The **content key**; this is a public/private key pair associated with the data content. The content private key is known only by the content owner, while the content public key needs to be known by every client that wants to use the data. The latter can be accomplished by making this key part of the content identifier, as suggested in [5].

- The **master servers**; these are trusted hosts directly controlled by the content owner, each of them holding a copy of the data content. All the master servers in the system form the **master set**. There is a public/private key pair associated with each master server. The master servers' public keys are certified through digital certificates issued by the content owner (and signed with the content key). These certificates bind each server's contact address (IP address and port number) to its public key, and are stored in a public directory, indexed by content public key. Thus, by knowing the content public key and the address of the directory, any client can securely get the addresses and public keys of all the master servers replicating that content.

- The **slave servers**; they hold copies of the data content but are not directly controlled by the content owner, and because of this, they are only marginally trusted. They can be part of a content delivery network run by a separate organization, or managed by a number of cooperating, but mutually-suspicious institutions. There is a public/private key pair associated with each slave, and each master keeps track of the contact addresses and public keys of the slaves it has been assigned.

- The **clients**; they perform read/write operations on the data content. For a client to use the system, it first has to go through a setup phase, when it connects to exactly one master and one slave. First, the client queries the directory and selects one master (the closest one for example) to which it establishes a secure connection (using the master's certified public key). The master then sends the client the address and public key of one of its slaves (the one closest to the client for example) to which the client also establishes a secure connection. This concludes the setup phase; at this point the client can start issuing read/write requests - by sending them to either the master or the slave - according to the algorithm described in the next section.

3 Algorithm

In this section we present the algorithm used to handle read/write requests from clients. This algorithm guarantees that write requests are always processed correctly in some sequential order. However, in order to allow fast processing of read requests, only statistical guarantees are given for their correctness; this is based on our original assumption that byzantine failures from untrusted (slave) servers are infrequent.

Our algorithm requires the masters to be fully connected to each other through secure (e.g. cryptographically) communication links, and implement a reliable, total-ordering, broadcast protocol that can tolerate benign (non-malicious) server failures. The broadcast protocol itself is outside the scope of this paper; a good choice could be for example the protocol described in [8].

Using this reliable broadcast protocol, master servers ensure they always agree on the same sequential ordering for write requests. Through the same broadcast protocol, the masters also elect one of them to function as an **auditor**. The auditor checks (in the background) the correctness of computations performed by slaves, and takes corrective action when any of them is found acting maliciously.

Each master server is responsible for updating the servers in its slave set when the data content changes due to writes. This updating occurs only **after** the masters have comitted the write. The masters also periodically broadcast their slave list to the master set, so in the event of a master crash, the remaining ones will divide its slave set. This also entails that all the clients connected to the crashed server will have to go through the setup process again.

It is important to notice that a slave receives a state update only **after** that update has been comitted. The reason we have chosen this "lazy" state update algorithm, as opposed to having masters **and** slaves participate in the total ordering broadcast, is performance. Since only masters are trusted, a total ordering broadcast protocol including the slaves would have to be resistant to byzantine failures, and implementing such an algorithm over a WAN is extremely expensive. "Lazy" state updates make the write protocol much more efficient, but also weaken the consistency model since a client cannot be guaranteed that once his write is comitted it will be seen in all subsequent reads. We tackle this problem by introducing a special parameter, dubbed $max\_latency$ that bounds the inconsistency window for a given write operation: a client is guaranteed that once $max\_latency$

time has elapsed since comitting a write, no other client will accept a read that is not dependent on that write. It is worth stressing out that we do not guarantee that a write will propagate to all slaves in $max\_latency$ (this will violate the asynchronous nature of the WAN environment); there may be slaves for which it takes longer that $max\_latency$ to get a state update, but if they behave correctly they should stop handling user requests until they are back in sync (later we will show how to handle malicious slaves).

3.1 Write Protocol

Writes are executed only on trusted (master) servers. When a client wants to perform a write, it sends the request to its assigned master, which first checks whether the client is allowed to invoke such a request, and if this is the case, it broadcasts the request to the other servers in the master set. Upon the successful completion of this broadcast, each master executes the request and increments a special variable, dubbed here $content\_version$ (which is initialized zero when the content is created). In the end, all master servers hold updated, but still identical copies of the data content (because they have executed the same write) and have the same value for the $content\_version$ variable.

At this point, the slaves are updated: each master sends the update together with the signed and time-stamped and new value for the $content\_version$ variable to all its subordinates through a secure broadcast. In order to prevent race conditions, two write operations cannot be, time-wise, closer than $max\_latency$ to each other. This ensures that any reads on which the second write depends will take into account the first write. This obviously limits the number of write operations that can be executed in a given time, which is why we advocate our architecture only for applications where there is a high reads to writes ratio.

In order to satisfy the latency constraint, the master servers have to periodically broadcast "keep-alive" packets consisting of the signed and time-stamped value of the $content\_version$ variable to their server set, even when no writes occur. A slave can handle client requests only if the most recently receive "keep-alive" packet is less than $max\_latency$ old.

3.2 Basic Read Protocol

As mentioned, the replicated data content needs to support flexible query operations (reads); calculating the result of such a query can be a computationally very intensive task if it requires applying an aggregation function on the entire data content (a complex join for a database, or a *grep Expression Path* request in the case of a file system). Therefore, in order decrease the workload on the master servers and improve performance, read requests are handled by the slaves.

When a client wants to perform a read, it sends the request to its assigned slave server. The slave executes the request, and constructs a "pledge" packet which contains a copy of the request, the secure hash (SHA-1 [1]) of the result, and the latest time-stamped $content\_version$ value received from the master. After signing this "pledge" packet, the slave sends it to the client, together with the result of the query.

At the other end, the client first computes the secure hash of the query result, and makes sure it matches the hash in the "pledge" packet. Then, the client verifies the slave's signature on the "pledge" packet. Finally, the client makes sure the time-stamp is not older than $max\_latency$. If all these conditions are met, the client accepts the answer, otherwise it rejects it.

Because of the asynchronous nature of the network connection between the client and the slave, it is possible that a result that was "fresh" when sent by the slave, becomes stale by the time it reaches the client. In such a situation, the client has to drop the answer and try the query again. By carefully selecting the value for $max\_latency$, and the frequency masters send "keep-alive" packets, the probability of such events occuring can be reduced. However, clients with very slow or unreliable network connections may never be able to get fresh-enough responses. One possible way to accommodate such clients is to relax the consistency model and allow the $max\_latency$ to be set by the clients themselves. In this case, clients with fast network connections can set high "freshness" requirements, while clients with slow connections can settle with more modest expectations.

The main vulnerability of this basic read protocol is that a malicious slave can return wrong answers to client requests. To protect against such malicious slaves, we employ two techniques - probabilistic checking and auditing which will be described in the next two sections.

3.3 Probabilistic Checking

For each read request, there is a certain probability that a client will send the same request to a master and compare the slave and master results. This "double-check" probability is a system parameter - it should be small enough so it does not excessively increase the workload on the masters, but large enough so it guarantees that a malicious slave is caught "red-handed" quickly.

As described in the previous section, for every read it handles, a slave has to sign a "pledge" packet that contains a copy of the request, the content version time-stamped by the master and the secure hash of the result (as computed by the slave). Should the slave act maliciously and return an incorrect answer, the "pledge" packet becomes an irrefutable proof of its dishonesty. Once a slave server is proven malicious it can be excluded from the system so it can do no further harm.

It is important to notice that the way the "pledge" packets are constructed makes impossible for a client to "frame" an innocent slave server - unless that client is able to fake the slave's digital signature. The only harm a client can do is to abuse its "double-check" quota (by double-checking all the slave responses instead of a small fraction of them). However, by keeping track on the number of double-check requests it receives from each of its clients, a master can identify statistically anomalous client behavior, which most likely indicates a "greedy" client. The master can then enforce fair play by simply ignoring a large fraction of the double-check requests coming from clients suspected to be greedy.

3.4 Auditing

Besides relying on the probabilistic checker to detect wrong answers, our system also employs an auditing mechanism that ensures that even if a malicious slave manages to return an erroneous result to a client, that slave will eventually get caught and excluded from the system.

The auditing mechanism works as follows: after the client accepts the result from the slave, and given that the client decides not to double-check that result, it still forwards the slave's "pledge" packet to the special auditor server.

The auditor is the only trusted server that does not have a slave set, and therefore does not handle any double-checking requests from clients; its only duty is to check the validity of "pledge" packets, by re-executing the read request in the packet and comparing the secure hash of the result to the hash in the packet. The auditor is allowed to lag behind when executing write requests; it executes a write only after it has audited all the read requests for the $content\_version$ that precedes that write. In fact, in order to take into account possible network delays, the auditor can move to a new $content\_version$ only after a sufficiently large time interval (more than $max\_latency$) has elapsed since the rest of the trusted servers have moved to that same $content\_version$. This ensures that at that point no client will accept any more read results for the previous $content\_version$. Here we assume clients accept read results only after they have forwarded the corresponding pledges to the auditor.

The auditor has several advantages over the slaves it has to verify, which allow it to achieve a much higher throughput when (re)executing read operations: first the auditor does not have to produce digital signatures (slaves on the other hand have to digitally sign a "pledge" packet for every client request they execute). Second, the auditor does not have to send any answers back to the clients. Third, since the auditor knows in advance all the operations it has to re-execute, it can, for certain types of applications (for example databases), employ query optimization mechanisms (cache results in the simplest case). Finally, because the auditor needs not to worry about client latency, it can spread its work so it minimizes idle time. Assuming that read requests show daily peak patterns (few requests at 3AM in the night for example), it is possible that the auditor will seriously lag behind during peak hours, but catch up during the night. However, it is essential that in the long run the auditor is able to keep up with the amount of reads it has to verify. If the auditor is over-used, the solution is to either add extra auditors, or weaken the security guarantees by verifying only a randomly chosen fraction of all reads.

3.5 Taking Corrective Action

In this section we discuss what happens when one of the slave servers is caught "red-handed," either as a result of a client double-checking a read result with a master (immediate discovery), or during the audit process (delayed discovery).

In the case of immediate discovery, the client forwards the incriminating "pledge" packet to the master. At this point, the master contacts all the clients connected to the (now provably malicious) slave, informs them the slave has been excluded from the system, and assigns each of them to a new slave server. Finally, the client that has made the discovery connects to its newly assigned slave and issues the same read request again.

In the case of delayed discovery, the situation is more complex, since at least one client has already accepted an incorrect answer. In some applications, the harm may be undone, by rolling back the client to the state before that particular read. In any case, the malicious slave needs to be excluded from the system so it can do no further damage. To this extent, the auditor sends the incriminating "pledge" packet to the master in charge of the slave that has signed it. The master then contacts all the interested clients (clients connected to the malicious slave) informing them of the problem and assigns them to new slave servers.

What happens after a malicious server has been excluded from the system is dependent on the administrative relations between that server and the content owner. If a formal content hosting contract exists, the matter can be further taken to courts (given that the incriminating "pledge" packet can be used as evidence). Another possibility is that the slave server is not inherently malicious, but is has been the victim of an attack (the trusted servers are supposedly administered much more carefully, so they cannot be easily hacked), in which case, after recovering it to a safe state, it can be brought back to use.

4 Discussion

The algorithm described is based on the optimistic assumption that byzantine failures are rare, allowing the system to be optimized for the common case, where untrusted components work correctly, even if there is a danger that incorrect results may be passed to clients. The probabilistic checking and auditing mechanisms guarantee that components acting maliciously will eventually be identified and excluded from the system.

There may be situations when stronger correctness guarantees are required. We outline a number of variants of our algorithm to handle this situation:

One variant is to give clients the option to differentiate between "normal" and "security sensitive" reads. The latter ones are then to be executed only by the trusted servers (which guarantees that clients always get correct results), while normal (non-sensitive) operations are carried out as described above. This variant allows us to provide 100% correctness guarantees for sensitive operations, at the expense of putting extra load on the trusted components. A further refinement of this scheme assigns even more security levels for read operations and sets the double-check probability based on the read's security level. Thus for the least sensitive operations the probability can be set very low, while for the very sensitive it can be set to 1 (which means "execute only on trusted hosts").

Another possibility is to send the same read request to more than one untrusted server. If all the answers are identical, the client proceeds as in the original algorithm - double-check with the master (with a small probability) and send the "pledge" packets, to the auditor. If not all answers match, the client automatically double-checks, since at least one of the slaves has to be malicious. This approach is similar to what is proposed in [4], and has the advantage that a number of malicious slaves would have to collude in order to pass an incorrect answer. The disadvantage is that more computing resources are needed in order to handle the same request, but these resources need not be trusted, and may therefore be easier to come by.

5 Related Work

There are two generic mechanisms for securely replicating data over untrusted hosts: state signing and state machine replication.

With state signing, the data content is divided into small (disjunct) subsets which are signed with a content private key. Clients then retrieve data from untrusted storage and verify its integrity using the content public key (assumed to be known a-priori). In order to minimize the number of digital signatures, some form of hash-tree authentication [12] is normally used in this context.

Work that falls into this category includes [7, 11, 13, 3] which apply this technique to distributed file systems, [2] which applies it to free software distribution, [6] which applies it to signing XML documents, [14] which applies it to digital certificate revocation and [9] which applies it on building a trusted database on untrusted storage. The main limitation for all these systems is that dynamic queries on the data need to be executed on trusted hosts. This requires the trusted host to first retrieve all data relevant to the query from untrusted storage, verify it, and then perform the operation.

With state machine replication [16], the idea is to execute the same operation on a number of untrusted hosts (quorum), and accept the result only when a majority of these hosts agree upon it. In this way, malicious hosts need to collude in order to pass an erroneous result; by requiring a large quorum size, the system can offer very strong security guarantees. Work in this area includes [4, 15, 10, 17]. The problem with this approach is that it greatly increases the amount of computing resources needed for handling a given request. Additionaly, the request latency is dictated by the slowest server in the quorum group.

The scheme we describe in this paper allows dynamic queries to be handled by untrusted servers, while avoiding most of the overhead associated with state machine replication. We are able to achieve this by providing only statistical guarantees on the correctness of any given query. However, our background auditing mechanism ensures that malicious servers are eventually detected, so they can be handled appropiately (either brought back to a safe state, or removed from the system).

6 Limitations and Future Work

One of the possible usage scenarios for the system architecture described in this paper is in the area of content delivery networks (CDNs), used for replicating semi-static Web content such as product catalogues for e-commerce, or academic, medical and legal databases. One possibility is having the organization that owns the data content to provide the master servers, while the CDN provides the slaves. Yet another possibility is to have the CDN itself divide its servers in a trusted core and a much larger set of outsourced and thus less trusted support servers. This scenario seems particularly realistic given the fact that most CDNs physically host most of their servers with Internet service providers and only remotely administer them.

The work presented in this paper is based on the fundamental assumption that byzantine failures are rare events, so applications can be optimized to work efficiently in the common case - when everything works correctly. This assumption is also the major limitation of our approach as it cannot be used (or at least is not efficiently) in scenarios when 100% security guarantees are required. However, looking at the current state of the Internet (the vast majority of WWW traffic is not encrypted, and even secure DNS is slow in gaining acceptance) it seems there are numerous applications where people can do well even without strong security guarantees.

The other limitation of our approach is that there is a certain latency for propagating writes, and in order to avoid race conditions we need to limit the frequency of such operations. As a result, the architecture described in this paper is appropriate for applications with a high reads to writes ratio. CDNs used for replicating slowly changing Web content, as well as academic, legal or medical databases clearly fall in this category. On the other hand, it would be impractical to use this architecture for disseminating data that changes rapidly and requires tight freshness guarantees, such as live stock quotes.

References

[1] Secure Hash Standard. FIPS 180-1, Secure Hash Standard, NIST, US Dept. of Commerce, Washington D. C. April 1995.

[2] A. Bakker and M. van Steen and A. Tanenbaum. A law-abiding peer-to-peer network for free-software distribution. In *Proc. Int'l Symp. on Network Computing and Applications*, Cambridge, MA, Feb. 2002. IEEE.

[3] M. Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[5] D. Mazieres and F. Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *In Proc. 8th ACM SIGOPS European Workshop*, pages 118–125. ACM, 1998.

[6] P. T. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine. Flexible authentication of XML documents. In *ACM Conf. on Computer and Communications Security*, pages 136–145, 2001.

[7] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *Computer Systems*, 20(1):1–24, 2002.

[8] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, 1989.

[9] U. Maheshwari, R. Vingralek, and B. Shapiro. How to build a trusted database system on untrusted storage. In *OSDI: 4th Symposium on Operating Systems Design and Implementation*, pages 135–150. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 2000.

[10] D. Malkhi and M. K. Reiter. Secure and scalable replication in Phalanx. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, pages 51–58. IEEE, 1998.

[11] D. Mazieres and D. Shasha. Building secure file systems out of byzantine storage. In *Proc. of the 21st Annual Symposium on Principles of Distributed Computing*, pages 108–117, Monterey, CA, 2002. ACM.

[12] R. C. Merkle. A certified digital signature. In *Proc. Crypto '89*, LNCS 435, pages 234–246. Springer-Verlag, 1989.

[13] E. L. Miller, W. E. Freeman, D. D. E. Long, and B. C. Reed. Strong security for network-attached storage. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, Jan. 2002.

[14] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proceedings 7th USENIX Security Symposium*, Jan 1998.

[15] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Lecture Notes in Computer Science*, volume 938, pages 99–110. Springer-Verlag, Berlin Germany, 1995.

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

[17] L. Zhou, F. Schneider, and R. van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.