Integrity-Preserving Replica Coordination for Byzantine Fault Tolerant Systems

Wenbing Zhao
Department of Electrical and Computer Engineering
Cleveland State University, 2121 Euclid Ave., Cleveland, OH 44115
wenbing@ieee.org

Abstract

The use of good random numbers is essential to the integrity of many mission-critical systems. However, when such systems are replicated for Byzantine fault tolerance, a serious issue arises, i.e., how do we preserve the integrity of the systems while ensuring strong replica consistency? Despite the fact that there exists a large body of work on how to render replicas deterministic under the benign fault model, the solutions regarding the random number control are often overly simplistic without regard to the security requirement, and hence, they are not suitable for practical Byzantine fault tolerance. In this paper, we present a novel integrity-preserving replica coordination algorithm for Byzantine fault tolerant systems. The central idea behind this algorithm is that all random numbers to be used by the replicas are collectively determined, based on the contributions made by a quorum of replicas, at least one of which is correct. We have implemented the algorithm in Java and conducted extensive experiments, in both a LAN testbed and an emulated WAN environment. We show that our algorithm is particularly suited for Byzantine fault tolerant systems operating in the LAN environment, or where replicas are connected by high-speed low-latency networks.

Keywords: System Integrity, Replica Consistency, Byzantine Fault Tolerance, Security, Random Numbers

1. Introduction

Many mission-critical systems rely on the availability of random bits for their security-related operations such as choosing cryptographic keys, random nonces, and session ids [6, 9]. The consequences of using weak random bits and the exploitation of such vulnerabilities have been extensively analyzed and reported in many books and research papers (*e.g.,* [6, 9, 20, 22]).

For concreteness, consider an e-commerce system that relies on the use of session-id for stateful interactions between the server and its clients, which is not uncommon for many practical systems using the Java Servlet technology. As pointed out in [6], if an adversary can predict the session-id of an active session, she can hijack the client's session, and consequently, she can obtain the client's private user data (such as names and addresses), view the client's profile data (such as previous orders), or possibly order merchandise without the consent of the actual client (*e.g.,* if the client has enabled the "one-click" option on Amazon.com). The predication of a session-id can be done by searching limited entropy space if weak random bits are used in a system. For example, [6] reverse-engineered a version of Tomcat (a popular Java Servlet Engine) and the related operations in a Window's based Java Virtual Machine. They could attack the system by performing about 2^{51} searches in finding an active session-id.

Therefore, when replicating these systems for high availability and fault tolerance, it is essential not to weaken the strength of the random bits essential for the integrity of their operations. For a sound replication coordination algorithm, it is essential to enable each replica to access its own entropy source and maintain its independence in such operations. However, what makes this objective very hard to achieve is that state machine replication [15] requires the replicas to be deterministic or rendered deterministic to maintain strong replica consistency. The conflicting requirements for security and replication must be reconciled, otherwise, we may end up with only two choices: (1) we favor security over high availability by not performing state machine replication of the systems, or (2) we trade security for high availability by removing the randomness of the systems in order to perform state machine replication. Obviously, neither choice is acceptable.

There has been a large body of work on how to render replicas deterministic in the presence of replica nondeterminism and most of them are designed to work under benign faults (*e.g.,* [3, 4, 13, 18]). Unfortunately, previous solutions proposed to handle operations related to random numbers are overly simplistic (*e.g.,* by seeding the random number generator with a deterministic value) *without* regard

to the security requirement, and hence, they are not suitable for use to build practical Byzantine fault tolerant systems.

In this paper, we present a novel replica coordination algorithm, referred to as the Collective-Determination algorithm, or CD-algorithm in short, towards the reconciliation of the conflicting requirements for security and for strongly consistent replication. The central idea behind this algorithm is that *all random numbers to be used by the replicas are collectively determined, and furthermore, the determination is based on the contributions made by a quorum of replicas, at least one of which is correct.*

In the CD-algorithm, the replicas first reach a Byzantine agreement on the set of contributions from replicas, and then apply a deterministic algorithm (for all practical purposes, the bitwise exclusive-or operation [22]) to compute the final random value. The freshness of the random numbers generated is application dependent. Our approach neither enhance nor reduce the freshness of the random numbers. If a pseudo-random number generator is used, it should be periodically reseeded from a good entropy source.

This paper makes the following research contributions:

- We point out the danger and pitfalls of controlling replica randomness for the purpose of ensuring replica consistency. Removing randomness from replica operations when it is needed could seriously compromise the system integrity.
- We propose the use of collective determination of random numbers contributed from individual replicas, as a practical way to reconcile the requirements of strong replica consistency and systems security.
- We present a lightweight, Byzantine agreement based algorithm to carry out the collective determination. The CD-algorithm only introduces two additional communication steps because the Byzantine agreement for the collective determination of random numbers can be integrated into that for message total ordering, as needed by the state-machine replication.
- We conduct extensive experiments, in both a LAN testbed and an emulated WAN environment[1], to thoroughly characterize the performance of our approach. We show that the CD-algorithm is particularly suited for Byzantine fault tolerant systems operating in the LAN environment, or where replicas are connected by high-speed low-latency networks.

This paper is structured as follows. Section 2 introduces the system model and provides a brief overview of the Byzantine fault tolerance algorithm developed by by Castro and Liskov [3]. Section 3 enumerates the pitfalls in controlling replica randomness and presents the rationale for our approach. Section 4 describe our collective-determination algorithm in detail. Section 5 presents a sketch of proof of correctness for our algorithm. Section 6 discusses a number of issues related to the practical usage of our algorithm. Section 7 provides performance measurement results. Section 8 describes related work, and Section 9 concludes the paper.

[1]As much as we would like to carry out experiments in a real WAN testbed such as the PlanetLab, we do not have access to any such testbed. The authors' institution is not yet a member of the PlanetLab Consortium.

2. Byzantine Fault Tolerance

In this section, we introduce the system model for our work, and the practical Byzantine fault tolerance algorithm (BFT algorithm, for short) developed by Castro and Liskov [3] as necessary background information.

Byzantine fault tolerance refers to the capability of a system to tolerate Byzantine faults. It can be achieved by replicating the server and by ensuring that all server replicas reach an agreement on the total ordering of clients' requests despite the existence of Byzantine faulty replicas and clients. Such an agreement is often referred to as Byzantine agreement [12].

Recently, a number of efficient Byzantine agreement algorithms [3, 10, 21] have been proposed. In this work, we focus on the BFT algorithm and use the same system model as that in [3]. The BFT algorithm operates in an asynchronous distributed environment. The safety property of the algorithm, *i.e.*, all correct replicas agree on the total ordering of requests, is ensured without any assumption of synchrony. However, to guarantee liveness, *i.e.,* for the algorithm to make progress towards the Byzantine agreement, certain synchrony is needed. Basically, it is assumed that the message transmission and processing delay has an asymptotic upper bound. This bound is dynamically explored in the algorithm in that each time a view change occurs, the timeout for the new view is doubled.

The BFT algorithm is executed by a set of $3f + 1$ replicas to tolerate up to f Byzantine faulty replicas. One of the replicas is designated as the primary while the rest are backups. Each replica is assigned a unique id i, where i varies from 0 to $3f$. For view v, the replica whose id i satisfies $i = v \; mod \; (3f + 1)$ would serve as the primary. The view starts from 0. For each view change, the view number is increased by one and a new primary is selected.

The normal operation of the BFT algorithm involves three phases. During the pre-prepare phase, the primary multicasts a pre-prepare message containing the client's request, the current view and a sequence number assigned to the request to all backups. A backup verifies the request and the ordering information. If the backup accepts the pre-prepare message, it multicasts a prepare message containing the ordering information and the digest of the request being

ordered. This starts the prepare phase. A replica waits until it has collected $2f$ matching prepare messages from different replicas, and the pre-prepare message, before it multicasts a commit message to other replicas, which starts the commit phase. The commit phase ends when a replica has collected $2f + 1$ matching commit messages from different replicas (possibly including the one sent or would have been sent by itself). At this point, the request message has been totally ordered and it is ready to be delivered to the server application once all previous requests have been delivered.

All messages exchanged among the replicas, and those between the replicas and the clients are protected by an authenticator [3] (for multicast messages), or by a message authentication code (MAC) (for point-to-point communications). An authenticator is formed by a number of MACs, one for each target of the multicast. We assume that the replicas and the clients each has a public/private key pair, and the public keys are known to everyone. These keys are used to generate symmetric keys needed to produce/verify authenticators and MACs. To ensure freshness, the symmetric keys are periodically refreshed by the mechanism described in [3]. We assume that the adversaries have limited computing power so that they cannot break the security mechanisms described above.

A compromised replica may replace a high entropy source to which it uses to seed its random number generator with a deterministic algorithm, and convey such an algorithm and the random numbers collectively determined via a covert channel to its colluding clients (we assume that a faulty replica cannot explicitly piggyback such information with a reply through the normal communication channel, which could be ensured by using an application-level gateway, or a privacy firewall as described by Yin et al.[21]). However, we assume that the bandwidth of the covert channel is very limited such that a faulty replica cannot transmit confidential state information to its colluding clients *in real time*, more specifically, given the size of the secret data S, the refresh period T, and the covert channel bandwidth B, we assume $T < SB$. For example, if the random number is of size 128 bits, and the covert channel bandwidth is 0.1 bit/second, the refresh period should be made less than 1280 seconds. How to limit the bandwidth of covert channels is still under intense research (for example [8, 16]), and it is out of the scope of this paper.

3. Pitfalls in Controlling Replica Randomness

In this section, we analyze a few well-known approaches that can possibly be used to ensure replica consistency in the presence of replica randomness. We show that they are not robust against Byzantine faulty replicas and clients. We reiterate here the importance of using random bits for security-related operations. Any attempt to weaken or remove such randomness from the system for replication will compromise the system integrity. Furthermore, for Byzantine fault tolerance, it is essential *not* to place undue trust to any single replica, such as the primary replica, because it is virtually impossible to verify if the random number proposed by a replica is truly random, and consequently, if that particular replica is compromised, the integrity of the entire replicated system will be lost.

For systems that use a pseudo-random number generator, their replicas can be easily rendered deterministic by ensuring that they use the same seed value to initialize the generator. One might attempt to use the sequence number assigned to the request as the seed. Even though this approach is perhaps the most economical way to render replicas deterministic (since no extra communication step is needed and no extra information is to be included in the control messages for total ordering of requests), it takes the randomness away from the system. Consequently, a Byzantine faulty client can easily guess the seed and predict the random numbers. A seemingly more robust approach is to use the timestamp as the seed to the pseudo-random number generator. As shown in [20, 22], the use of timestamp does not offer more robustness to the system because it can also be guessed by Byzantine faulty clients.

The only option remaining seems to be the use of a truly random number to seed a strong pseudo-random number generator (or to obtain random numbers entirely from a high entropy source). We note that the elegant mechanism described in [3] cannot be used in this case because backups have no means to verify whether the number proposed by the primary is taken from a strong random number generator seeded periodically using a high-entropy source, or is generated according to a deterministic algorithm. If the latter is the case, the Byzantine faulty primary could continue colluding with Byzantine faulty clients without being detected. A slight improvement over this scheme is to proactively rotate the primary. However, this improved scheme does not solve the problem: while the faulty replica serves as the primary, it can use a predictable seed and/or a weak random number generator without being detected, and hence, the system integrity cannot be guaranteed during this period.

Therefore, we believe the most effective way in countering such threats is to collectively determine the random number, based on the contributions from a quorum of replicas so that Byzantine faulty replicas cannot influence the final outcome.

4. The Collective-Determination Algorithm

The normal operation of the CD-algorithm is illustrated in Figure 1. As can be seen, the collective-determination mechanism is seamlessly integrated into the original BFT algorithm. On ordering a request, the primary determines

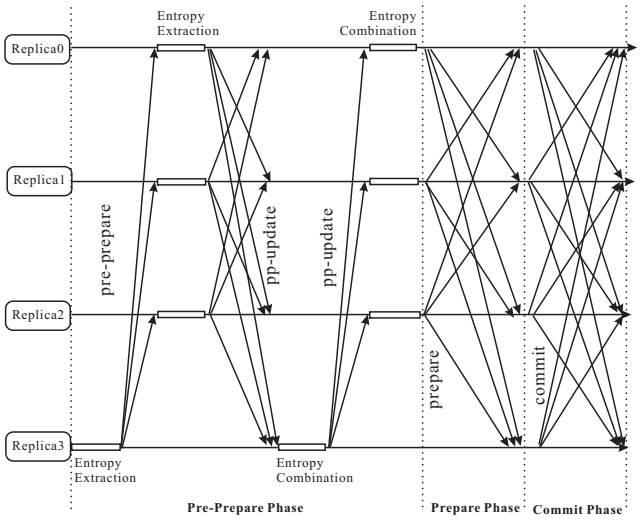

Figure 1. Normal operation of the Collective-Determination Algorithm.

the order of the request (*i.e.*, assigns a sequence number to the request), and queries the application for the type of operation associated with the request. If the operation involves with a random number as input, the primary activates the mechanism for the CD-algorithm. The primary then obtains its share of random number by extracting from its own entropy source, and piggybacks the share with the pre-prepare message multicast to all backups. The pre-prepare message has the form $<$PRE-PREPARE,$v, n, d, R_p>\alpha_p$, where v is the view number, n is the sequence number assigned to the request, d is the digest of the request, R_p is the random number generated by the primary, and α_p is the authenticator for the message.

On receiving the pre-prepare message, a backup performs the usual chores such as the verification of the authenticator before it accepts the message. It also checks if the request will indeed trigger a randomized operation, to prevent a faulty primary from putting unnecessary loads on correct replicas (which could lead to a denial of service attack). If the pre-prepare message is acceptable, the replica creates a pre-prepare certificate for storing the relevant information, generates a share of random number from its entropy source, and multicasts to all replicas a pp-update message, in the form $<$PP-UPDATE,$v, n, i, R_i, d>\alpha_i$, where i is the sending replica identifier, R_i is the random number contributed by replica i.

When the primary has collected $2f$ pp-update messages, it combines the random numbers received according to a deterministic algorithm (referred to as the entropy combination step in Figure 1), and builds a pp-update message with slightly different content than those sent by backups. In the pp-update message sent by the primary, the R_i component is replaced by a set of $2f + 1$ tuples containing the random numbers contributed by replicas (possibly including its own share), S_R. Each tuple has the form $<R_i, i>$. The replica identifier is included in the tuple to ease the verification of the set at backups.

On receiving a pp-update message, a backup accepts the message and stores the message in its data structure provided that the message has a correct authenticator, it is in view v and it has accepted a pre-prepare message to order the request with the digest d and sequence number n. A backup proceeds to the entropy combination step only if (1) it has accepted a pp-update message from the primary, and (2) $2f$ pp-update messages sent by the replicas referenced in the set S_R. The backup requests a retransmission from the primary for any missing pp-update message.

When it finishes the entropy combination, a backup multicasts a prepare message. The message has the form $<$PREPARE$v, n, i, d'>\alpha_i$, where d' is the digest of the request concatenated by the combined random number. The replica then waits for $2f$ valid prepare messages from different replicas (possibly including the message sent or would have been sent by itself), after which, it multicasts to all replicas a commit message in the form $<$COMMIT$v, n, i, d'>\alpha_i$. When a replica receives $2f + 1$ valid commit messages, it decides on the sequence number and the collectively determined random number. At the time of delivery to the application, both the request and the random number are passed to the application.

In Figure 1, the duration of the entropy extraction and combination steps have been intentionally exaggerated for clarify. In practice, the entropy combination can be achieved by applying a bitwise exclusive-or operation on the set of random numbers collected, which is very fast. The cost of entropy extraction depends on the scheme used. Some schemes, such as the TrueRand method [11], allows very prompt entropy extraction. TrueRand works by gathering the underlying randomness from a computer by measuring the drift between the system clock and the interrupts-generation rate on the processor.

5. Proof of Correctness

In this section, we provide a sketch of the proof for the correctness of the CD-algorithm. The correctness criteria for the algorithm are:

C1 [Consistency Property] All correct replicas deliver the same random number to the application together with the associated request.

C2 [Security Property] The random number is secure (*i.e.*, it is truly random) in the presence of up to f Byzantine faulty replicas.

C1 is guaranteed by the use of Byzantine agreement algorithm. C2 is ensured by the collection of $2f + 1$ shares contributed by different replicas, and by a sound entropy combination algorithm. By collecting $2f + 1$ contributions, it is guaranteed that at least $f + 1$ of them are from correct replicas, so faulty replicas cannot completely control the set. (The use of $f + 1$ shares are all that needed for this purpose. However, collecting more shares is more robust in cases when some correct replicas use low-entropy sources. This is analogous to the benefit of Shoup's threshold signature scheme [17].) The entropy combination algorithm ensures that the combined random number is secure as long as at least one share is secure. The bitwise exclusive-or operation could be used to combine the set and it is provably secure for this purpose [22]. Therefore, the CD-algorithm satisfies both C1 and C2.

6. Discussion

For clarity, when describing the CD-algorithm, we have assumed that each remote method invocation involves only a single random number. In practice, an invocation might need to access a number of random numbers. Our algorithms can be trivially modified to accommodate this need. Let n be the number of random numbers needed for an operation. In the CD-algorithm, during the pre-prepare phase, each replica obtains n random numbers from its entropy source and includes them in the pre-prepare and pp-update messages. The entropy combination step would now combine n random numbers instead of 1. The remaining steps are similar to those in the original algorithm.

If the entropy consumption rate exceeds the entropy generation rate of the entropy source, which is the case for most practical systems, a pseudo-random number generator is normally used. If such systems are replicated for Byzantine fault tolerance, unless the pseudo-random number generator is frequently reseeded, it is not wise to collectively determine the seed to the random number generator so that the sequence of pseudo-random numbers can be generated deterministically without replica coordination.

This is because if a compromised replica leaks the collectively-determined seed to some faulty clients (an adversary can take her time to leak the seed number through a very low-bandwidth covert channel, if the reseeding is not carried out frequently), they can predict all the future random numbers (for example, session-ids) produced by the generator, which would compromise the integrity of the system (*i.e.*, the backward security will be lost). For better security, the replicas should still collectively determine *every* random number (with the tradeoff of higher communication cost). It is also advised that each replica uses a different (strong) pseudo-random number generator for enhanced security. In fact, even if all correct replicas use the same relatively weak random number generator that can be broken by searching 2^E space, our algorithm can expand the search space significantly to $2^{(f+1)E}$ (because the collectively determined random number is based on contributions from at least $f + 1$ correct replicas).

7. Performance Characterization

The CD-algorithm has been implemented and incorporated into a Java-based BFT framework. The Java-based BFT framework is developed in house and it is ported from the C++ based BFT framework of Castro and Liskov [3]. Due to space limitation, the details of the framework implementation is omitted.

The development and test platform consists of a group of Dell SC440 servers each is equipped with a PentiumD processor of 2.8GHz and 1GB of RAM running SuSE 10.2 Linux. The nodes are connected via a 100Mbps LAN. As we noted earlier, the WAN experiments are emulated by introducing artificial delays in communication, without injecting message loss.

To characterize the cost of the CD-algorithm, we use an echo application with fixed 1 KB-long requests and replies. The server is replicated at four nodes, and hence, $f = 1$ in all our measurements. Up to 12 concurrent clients are launched across the remaining nodes (at most one client per node). Each client issues consecutive requests without any think time. For all measurements, the end-to-end latency is measured at the client and the throughput is measured at the replicas. The Java `System.nanoTime()` API is used for all timing-related measurements.

7.1. Cost of Cryptographic Operations

We first report the mean execution latency of basic cryptographic operations involved in the CD-algorithm and present an analysis of their contributions to the end-to-end latency because such information is beneficial to the understanding of the behaviors we observe. The latency cost is obtained when running a single client and four server replicas in the LAN testbed. The results are summarized in Table 1.

Without any optimization (and without fault), an end-to-end remote call from a client to the replicated server using the original BFT algorithm involves a total of four authenticator generation operations (A_g), five authenticator verification operations (A_v) (one does not need to verify the message sent by itself), one MAC generation operation (M_g) and two MAC verification operations (M_v) on the critical execution path (*i.e.*, $A_g + A_v$ for request sending and receiving, $A_g + A_v$ for the pre-prepare phase, $A_g + A_v$ for the prepare phase, $A_g + 2A_v$ for the commit phase, and

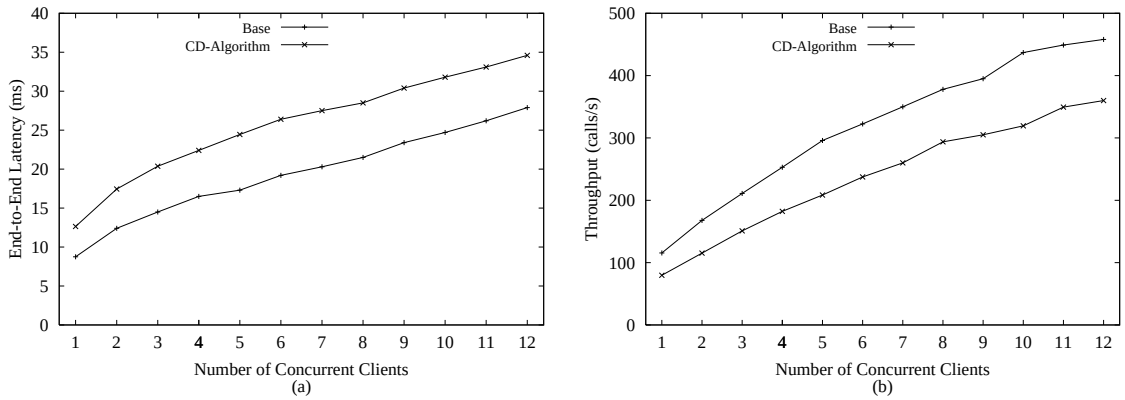

Figure 2. LAN measurement results. (a) End-to-end latency, and (b) the system throughput, in the presence of different number of concurrent clients.

Operation Type	Signing /Generation	Verification /Combination
MAC	24.1 μs	237.3 μs
Authenticator	80.2μs	892.0μs

Table 1. Execution time for basic cryptographic operations involved with the CD-algorithm.

$M_g + 2M_v$ for the reply sending and receiving). The CD-algorithm introduces two additional communication steps and two authenticator generation operations and three authenticator verification operations on the critical path.

From this analysis, the minimum end-to-end latency achievable using the CD-algorithm is $L_{BA}^{min} = 6A_g + 8A_v + M_g + 2M_v$ (a replica can proceed to the next step as soon as it receives one valid prepare message from other replica in the prepare phase, and two valid commit messages from other replicas in the commit phase, and the client and proceed to deliver the reply as soon as it has gotten two consistent replies). Plugging in our experimental data, $L_{BA}^{min} = 8115.9\mu s$.

7.2. LAN Experimental Results

Figure 2 shows the summary of the experimental results obtained in the LAN testbed. Figure 2(a) shows the end-to-end latency as a function of the load on the system in the presence of concurrent clients. As a reference, the latency for the BFT system without the additional mechanisms described in this paper is shown as "Base". In the figure, the result for the CD-algorithm is shown as "BA". As can be seen, the end-to-end latency is dominated by the cryptographic operations rather than the communication cost in the LAN environment. For example, in the presence of a single client, the end-to-end latency for the CD-algorithm is about $12.6ms$, of which $8.1ms$ is due to the cryptographic operations. Comparing with the base, the CD-algorithm incurs an runtime overhead of about $4-7ms$ in the end-to-end latency, which is rather moderate. The throughput measurement results are shown in Figure 2(b) and they are consistent with the trend observed in the latency measurements.

7.3. WAN Experimental Results

The experimental results obtained in an emulated WAN environment are shown in Figure 3. The observed metrics and the parameters used are identical to those in the LAN experiments. As can be seen in Figure 3(a), the end-to-end latency for the CD-algorithm is significantly higher than that of the base setup, especially when the load is high, due to the extra communication steps involved. However, the throughput reduction in the CD-algorithm remains moderate.

8. Related Work

How to ensure strong replica consistency in the presence of replica nondeterminism has been of research interest for a long time, especially for fault tolerant systems using the benign fault model [3, 4, 13, 18]. However, while the importance of the use of good random numbers has long been recognized in building secure systems [20], we have yet to see substantial research work on how to preserve the randomized operations necessary to ensure the system integrity in a fault tolerant system. For the type of systems where the use of random numbers is crucial to their service integrity, the benign fault model is obviously inadequate and

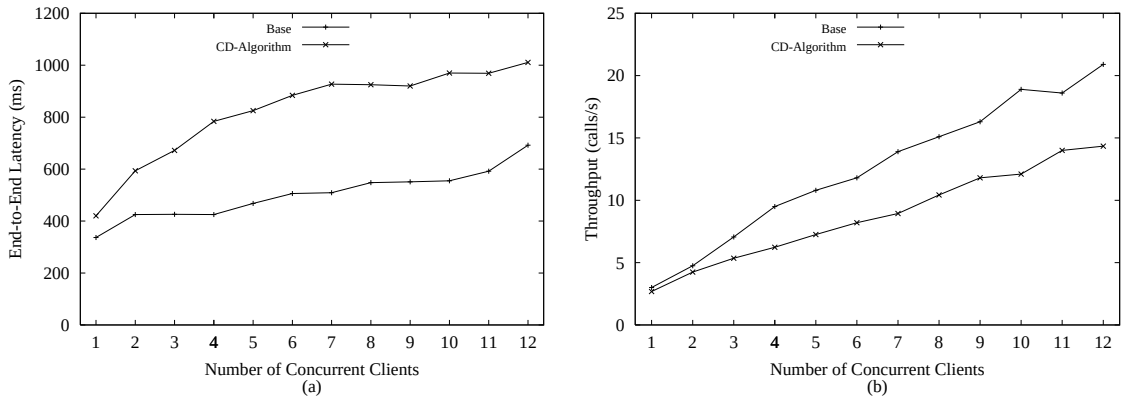

Figure 3. Emulated WAN measurement results. (a) End-to-end latency, and (b) the system throughput, in the presence of different number of concurrent clients.

the Byzantine fault model must be employed if fault tolerance is required.

In the recent several years, significant progress has been made towards building practical Byzantine fault tolerant systems, as shown in the series of seminal papers such as [3, 4, 10, 21]. This makes it possible to address the problem of reconciliation of the requirement of strong replica consistency and the preservation of each replica's randomness for real-world applications that requires both high availability and high degree of security. We believe the work presented in this paper is an important step towards solving this challenging problem.

We should note that some form of replica nondeterminism (in particular, replica nondeterminism related to timestamp operations) has been studied in the context Byzantine fault tolerant systems [3, 4]. However, we have argued in previous sections that the existing approach is vulnerable to the presence of colluding Byzantine faulty replicas and clients.

The main idea of this work, *i.e.*, collective determination of random values based on the contributions made by the replicas, is borrowed from the design principles for secure communication protocols [19]. However, the application of this principle to solving the strong replica consistency problem is novel.

Finally, one may argue that threshold cryptography [1, 2, 5, 7, 14, 17, 23] can be readily used to perform many secure operations across several server replicas without ever exposing the security key, and hence, it is more robust against malicious attacks (it works fine in the presence of high-bandwidth covert channels between a compromised server replica and its colluding clients). However, it might not be appropriate for all applications due to the following limitations:

- It is extremely computationally expensive. We have experimented with a Java-based threshold cryptography framework[2] on our testbed. For a key of 1024 bit-long, the key shares generation operation takes over $1s$, and the key shares combination operation takes over $2s$. This would be too expensive for most Web-based applications, which require soft realtime responses.
- To achieve proactive threshold cryptography, the key shares must be frequently refreshed, which involves not only expensive computations, but substantial message exchanges among the replicas as well.
- Threshold cryptography assumes the availability of a trusted dealer to distribute the initial shares. Our CD-algorithm does not impose such a strong requirement.

Therefore, our CD-algorithm offers a very lightweight solution for applications that prefer to use Byzantine replication for fault tolerance.

9. Conclusion

In this paper, we presented our work on reconciling the requirements of strong replica consistency and the desire of maintaining each replica's individual randomness. Based on the central idea of collective determination of random values needed by the applications for their service integrity, we designed and implemented the CD-algorithm. The CD-algorithm is based on reaching a Byzantine agreement on a quorum of random number shares provided by $2f + 1$ replicas. We thoroughly characterized the performance of the CD-algorithm in both a LAN testbed and an emulated WAN environment. We show that the overhead incurred

[2]Threshsig: Java threshold signatures. An implementation of Shoup's practical threshold signatures algorithm [17]. Available at http://threshsig.sourceforge.net/

by the CD-algorithm with respect to the base BFT system is relatively small in the LAN environment, making it possible for practical use.

References

[1] A. Boldyreva. Efficient threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. *Lecture Notes in Computer Science*, 2567:31–46, 2003.

[2] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18:219–246, 2005.

[3] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[4] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21(3):236–269, 2003.

[5] Y. Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–457, 1994.

[6] L. Dorrendorf, Z. Gutterman, and B. Pinkas. Cryptanalysis of the random number generator of the windows operating system. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 476–483, Alexandria, VA, 2007.

[7] Y. Frankel, P. Gemmal, P. MacKenzie, and M. Yung. Proactive RSA. In *Proceedings of the 17th Annual International Cryptology Conference*, Santa Barbara, CA, August 1997.

[8] S. Gianvecchio and H. Wang. Detecting covert timing channels: an entropy-based approach. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 307–316, Alexandria, VA, 2007.

[9] Z. Gutterman and D. Malkhi. Hold your sessions: An attack on java session-id generation. *Lecture Notes in Computer Science*, 3376:44–57, 2005.

[10] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, 2007.

[11] J. Lacy, D. Mitchell, and W. Schell. Cryptolib: Cryptography in software. In *Proceedings of the 4th USENIX Security Symposium*, pages 1–17, 1993.

[12] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[13] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[14] T. Rabin. A simplified approach to threshold and proactive RSA. In *Proceedings of the 18th Annual International Cryptology Conference*, Santa Barbara, CA, August 1998.

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

[16] G. Shah, A. Molina, and M. Blaze. Keyboards and covert channels. In *Proceedings of the 15th USENIX Security Symposium*, pages 59–75, Vancouver, B.C., Canada, July 31 - August 4 2006.

[17] V. Shoup. Practical threshold signatures. *Lecture Notes in Computer Science*, 1097:207–220, 2000.

[18] J. Slember and P. Narasimhan. Living with nondeterminism in replicated middleware applications. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference*, pages 81–100, Melbourne, Australia, 2006.

[19] A. Tanenbaum. *Computer Networks*. Prentice Hall, 4th edition, 2003.

[20] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[21] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 253–267, Bolton Landing, NY, 2003.

[22] A. Young and M. Yung. *Malicious Cryptography: Exposing Cryptovirology*. Wiley, 2004.

[23] L. Zhou, F. B. Schneider, and R. van Renesse. APSS: Proactive secret sharing in asynchronous systems. *ACM Transactions on Information and System Security*, 8(3):259–286, August 2005.