Università di Pisa
Dipartimento di Informatica

Technical Report: TR-04-12

A Secure and Scalable Digital Time–stamping Service

Daniela Tulone
University of Pisa

June 7, 2004

Address: via F. Buonarroti 2, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

A Secure and Scalable Digital Time–stamping Service

Daniela Tulone
University of Pisa

June 7, 2004

Abstract

We propose a distributed digital time–stamping service (TSS) based on absolute timestamps and resilient to Byzantine failures. In contrast with previous solutions based on linking schemes, our TSS provides *high scalability*, *better response time*, and *fine granularity*. It works in asynchronous systems, and is built on top of Byzantine quorum system techniques, a secure robust proactive threshold signature scheme, and a novel *robust time service*. This service has independent interest for solving a problem that is more general than the clock synchronization problem and suitable for large and dynamic populations, and for being more efficient than previous protocols. We compare the complexity of our TSS scheme with the linking scheme, and show that our TSS is particularly suitable for applications requiring fine–grained timestamps and good response time (i.e. temporal access control, stock–trading, e–commerce transactions).

1 Introduction

More than ever, securities firms, banking institutions, on–line services rely on electronic communications to conduct their business. It is crucial for these applications to guarantee the trustworthiness of their electronic records, as well as to ensure the time at which they were created or modified. Time–stamping services (TSS) address this problem: they bind *authentically* a document to a certain point in time, and provide the ability to prove that. Notice that time–stamping does not concern only secure archives, but more generally applications that require to link authentically a *serializable event* to a point in time.

The vast majority of well–known TSS systems [1, 2, 3, 7, 8, 11, 5, 13] are based on *linking schemes* which link documents together by means of a collision–resistant hash function, and therefore provide a *relative temporal order*. Unfortunately, this approach presents some drawbacks: it is not scalable, clients experience delay in getting a timestamp, and it has high time complexity and coarse granularity. Moreover, a large majority of TSS systems rely on a centralized secure timestamp authority and storage medium to publish timestamps, thus showing a *single point of failure*. A different approach based on *distributed trust*, digital signatures and a secure pseudo–random generator, was proposed by Haber and Stornetta [1]. As pointed out in [6], this scheme does not seem to be practical for relying on a large set of nodes available for time-stamping requests. Ansper et al. address in [13] the availability of the TSS by using multiple servers, however their proposals assume only crash failures and cannot be applied to Byzantine failures. Moreover, these proposals [1, 13] are not scalable to high volume of requests or to large populations. The only work addressing Byzantine failures is by Maniatis et al [7], and proposes an architecture for a distributed TSS system which is based on linking schemes. However, as pointed out by the authors, this work is at an early stage and in fact does not provide correctness proofs. The straightforward solution of hashing and signing a document and the current *real time*, has been disregarded [11, 6] for requiring *unconditional trust* of the timestamp authority, and because of the unreliability of old timestamps after a leakage of the private signature key.

We propose here a different approach, which is alternative to the linking scheme and has some attractive features, hard to achieve with previous schemes: high scalability and availability, Byzantine–tolerance, good response time, fine granularity (tunable by the application), and simplicity in the design. Our TSS is based on *absolute timestamps*, and is resilient to Byzantine server and client failures. It works out the problems of the *hash–and–sign* scheme mentioned above, mainly for being distributed and based on a novel *robust time service*. This primitive has independent interest for solving a problem that is *more general* than the well–known clock synchronization problem that maintains a *static set* of distributed clocks synchronized within a known constant factor. Clearly, clock synchronization has a prohibitive cost when used in very large and dynamic populations since it requires periodic communications among all processes. In addition, several applications running in a malicious environment (i.e. Internet–based applications) do not require a *continuous and secure* clock synchronization at the client side during their lifetime but rather *on–demand*, to perform some critical steps. Furthermore, it might not be convenient for applications with stringent physical requirements, such as mobile applications, to run a clock synchronization protocol. For all these cases it is more significant to study the problem of providing a generic process dispersed in a WAN with a *close approximation* of the real time, based on some fixed infrastructure. This problem has less stringent requirements than the previous one, and therefore lower complexity.

The security of our TSS relies on a *secure robust proactive* (b, n) *threshold signature* scheme. It is based on the threshold signature scheme proposed by Boldyreva [15], which works in any Gap Diffie–Helmann group where the Computational Diffie–Helmann problem is hard but the Decisional Diffie–Helmann problem is easy, and the efficient asynchronous protocol proposed by Cachin et al. in [16] for proactively refreshing discrete logarithm–based shares of the secret key. These choices are motivated by their properties particularly attractive for our TSS (i.e. for the robustness, simplicity, efficiency, optimal resiliency of [15] and for not requiring interactions among servers). Each server refreshes periodically sensitive information stored locally (i.e. authentication keys), thus allowing our TSS to tolerate *any number* of faults provided a limited number of intrusions during each *time window*. In case of *key revocation*, the TSS resolves disputes and guarantees the validity of the timestamps issued before the revocation time.

Our protocols are built on top of Byzantine quorum systems [29] which improve the system load balancing, reduce the access cost per operation, and make the system highly scalable. The high scalability of our TSS is achieved also by fixed–size timestamps and constant verification costs, derived by the threshold signature scheme, and by avoiding interactions among servers. We show in Section 5.3.1 and Section 5.2.1 that distributed TSS based on linking schemes [7] are much more complex and less efficient than our approach. The TSS features can be summarized as follows:

- Byzantine–tolerance and high availability;
- Absolute timestamps more expressive than relative ones, and fine granularity;
- Better response time, constant verification cost with no extra communications;
- High scalability, and better load balancing;
- Dispute resolution in case of key revocation, and no trusted dealer;
- Simplicity and more flexibility in the design, and modular implementation.

The good response time, fine granularity and simplicity makes our TSS particularly suitable for a number of applications that require digital timestamp as a crucial step to carry out sensitive tasks (i.e temporal access control, e–commerce transactions, electronic stock–trading).
The paper is structured as follows: Section 2 discusses related works and Section 3 describes the system model and techniques used by our TSS. Section 4 presents a novel distributed time

service resilient to Byzantine failures and shows its correctness. In Section 5 we describe our TTS service built on top of the time service presented in Section 4, analyze it and compare the complexity of our approach with the linking scheme. Section 6 discusses some applications.

2 Related works

Several linking schemes [1, 3, 5, 11, 12] have been proposed, ranging from *linear linking* [1] to *tree–like linking*. Tree–like linking schemes, proposed first by Benaloh and de Mare [3] and then by Haber and Stornetta [2, 5], improve linear linking in terms of storage space, verification costs, and are more robust since their security relies on the lifetime of collision-resistant hash functions that is longer than digital signatures [6]. This scheme has been employed by current digital timestamp systems, such as Surety Technology [9], and also by the TIMESEC project [10]. The time–stamping procedure in tree–like schemes runs in *rounds*. At the end of each round the TSS computes a Merkle hash tree whose leaves represent the requests submitted at that round. These requests appear to occur simultaneously since their timestamp is the cumulative hash of tree root value and of the timestamp at the previous round [5]. The timestamp returned to the client contains information necessary to rebuild the tree branch correspondent to that request in order to verify its validity. As a result, this scheme does not scale well to high volume of requests since the size of the timestamp and the cost to verify it grow with the number of requests submitted at that round. Massias et al [6] employ different levels of secure timestamp authority to improve scalability, however this does not provide a full answer to the problem. As discussed in Section 5, our TSS is scalable to high volume of requests and large populations.

Notice that the choice of the duration of the round is crucial in tree–like linking schemes and shows an inherent trade–off between efficiency (computational cost, memory), and timestamp granularity and response time. Since a malicious TSS could subvert the order of the timestamps, the TSS usually publishes occasionally *round values* in a trusted and public medium (i.e. newspaper). This process occurs rarely (i.e. daily) since it is time consuming and costly, while the duration of rounds is kept short (i.e. every minute) because it determines the timestamp granularity. Clearly, relative temporal order does not suit well applications involving comparisons across different TSS systems even if timestamps are endowed with time values, not only for the time accuracy problem (i.e. NTP has a weaker failure model), but also for different trustworthiness of the TSS. Massias et al [6] address this issue by distinguishing documents for internal and external use, and grounding documents for external use. However, this strategy is very costly and not flexible. Clearly, our TSS does not present all these issues for its different approach much simpler than the previous ones. Its simplicity leads also to a better performance.

Since clients need guarantees of *accountability*, the TSS has to take extra steps to convince them that it cannot cheat (i.e. back– or post–date requests) by subverting the order of timestamps between grounding. There have been several attempts to minimize the trustworthiness of the TSS, based on trusted digital notary [14, 12, 9], or on binary linking [11], or on authentication graphs [12]. However, all these solutions are costly and vulnerable to a *single point of failure*. We have already discussed in the Introduction TSS based on multiple processes, [1, 13, 7]. The distributed TSS proposed in [7] and based on linking schemes relies on a randomized Byzantine agreement, run periodically to guarantee the consistency of the server's linking structures. In contrast with our proposal, TSS based on linking schemes are not suitable for applications requiring fine granularity and good response time.

As mentioned in the Introduction, our TSS is built on top of a novel *robust* time service. This primitive has independent interest for studying a problem that is *more general* than the clock synchronization problem, well–known in the literature, and suitable for applications involving large dynamic populations. The service has optimal accuracy, is more efficient than previous protocols (i.e. [26, 20, 25]) and is highly scalable for using Byzantine quorum systems. Previous clock synchronization protocols suitable for large populations in a WAN, are based on a hierarchical network structure. They are highly scalable but assume a weaker abstract fault model (i.e . NTP [27] and CesiumSpray [24]). Our solution is novel for addressing at the same time efficiency, scalability and tolerating up to one third of Byzantine server failures.

3 Preliminaries

Our system model has a hierarchical structure that allows a practical and modular implementation. It consists of the following processes, each denoted by a unique identification number:

- Level 0: a set of servers attached to a GPS receiver [28] providing an accurate estimation of the real time with constant error bound;
- Level 1: a set of n timestamp servers $S_1, \ldots, S_n$ responsible for time–stamping documents submitted by clients, and called *TSS servers*;
- Level 2: a *dynamic* set of clients submitting requests to a subset of TSS servers.

In this hierarchical structure, processes communicate to a subset of processes at the previous level providing a service (time service at Level 0, and digital timestamp at Level 1). Clearly, this classification is functional (i.e. Level 0 and Level 1 can be combined). Notice that Level 0 consists of a set of servers to tolerate crash failures, intrusions, and occasionally discontinuities of the GPS receiver (e.g. due to satellite positioning). We denote by b the maximum number of Byzantine server failures.

Our protocols are built on top of a b–masking Byzantine quorum system, a collection of subsets of servers (*quorums*) such that (1) any two subsets overlap in at least $2b + 1$ servers (*consistency property*), and (2) for any failure configuration there is a quorum consisting of correct servers (*availability property*). It tolerates $b < \frac{n}{4}$ Byzantine failures; see [29] for details.

Messages can be arbitrarily delayed but not indefinitely. This assumption is realistic since it is reasonable to assume that network instabilities and DoS attacks are reparable. We assume that communications are authenticated. A process is *correct* if it follows its specifications. Our distributed TSS employs *proactive security*: each server periodically reboots and refreshes sensitive data stored locally (i.e. secret authentication keys and secret shares). This allows the system to tolerate *any number* of faults provided that fewer than one fourth of the servers become faulty within a *time window* (time period between two consecutive refreshes). A *malicious adversary* can compromise up to one fourth of servers within each *time window* (though in some case $b < \frac{n}{3}$), eavesdrop the network, delay messages though not indefinitely, and collude with clients. We assume that the adversary (and the faulty nodes it controls) is computationally bound so that with very high probability it is unable to forge digital signatures, or find collision of one–way hash functions. Clients can fail by crash and be malicious.

3.1 Clocks

Each process has access to a local hardware clock whose function is denoted by $H(t)$. Because of impressions of the oscillator, temperature changes, and aging, a hardware clock can drift apart from real time. It is common practice to assume its drift rate within a *narrow envelope* of the real time. If ρ is the maximum drift rate of H, we say that H is *correct* if for any real time $t_1 > t_2$ it measures the passage of time in $[t_1, t_2]$ with an error of at most $\rho(t_2 - t_1)$. We employ Cristian probabilistic reading method [21] to estimate a remote clock, for its simplicity and applicability to asynchronous systems. It relies on the round trip delay elapsed between the sending of the time request and the arrival of the reply, and computed by the sender.

Each TSS server maintains a *virtual clock* C which differs from H by an offset γ. To bound the clock error, each server adjusts periodically C to a more accurate time source by resetting its offset γ. Since the drift rate might vary between synchronization, each process maintains an *error function* $E(t) = \varepsilon + \rho\Delta t$ with ε error inherited at the last synchronization and Δt

time elapsed since the last adjustment. According to [23], the *time interval* $I(t)$ is defined as $[C(t) - E(t), C(t) + E(t)]$. It is *correct* at real time t if it contains t.

3.2 TSS specifications

The target of a TSS is to *authentically* bind a timestamp to a request so that the time and its authenticity can be respectively measured and verified at some later time. A digital time–stamping system consists of a triple of protocols:

1. A time–stamping protocol *timestamp* that on a input x produces a timestamp T_x and a proof of its authenticity σ_x, that is $\langle T_x, \sigma_x \rangle$.
2. A verification protocol $verify$ that given $\langle x, T_x, \sigma_x \rangle$, verifies the authenticity of the timestamp associated to x. If successful, the measure of time associated to x is accepted.
3. An audit protocol used to verify whether the TSS carries out its duty.

The basic requirements of a digital timestamp are the followings:

- Correctness: *timestamp* binds authentically a request x to a timestamp T_x representing the time, or order, at which the request has been issued, and provides a proof σ_x for that; $verify$ succeeds only if T_x is correct.
- Data Integrity: an adversary cannot tamper the timestamp by back– or post–dating it, or modify the request associated to it, or insert an old timestamp in the list of timestamps previously issued.
- Availability: *timestamp* and $verify$ must be available despite processes failures.

3.3 Cryptographic tools

One–way hash functions A user submits the message digest of his request to guarantee data privacy and improve the network bandwidth.

Proactive threshold signature Our protocol uses a secure robust proactive (b, n) threshold signatures scheme which is intended not only to enhance the lifetime of the digital signatures, but also to guarantee fixed–size timestamps, thus providing scalability. It also leads to a better system load balancing since a quorum of servers (whose size can be as small as $2b + 1$) is asked to produce a partial signature. We assume a TSS private and public key; the TSS private key is secretly shared at the init time so that $b + 1$ shares are sufficient to compute it but b shares do not provide any information to a polynomial–time adversary. The TSS *certified* public key is available to clients. We employ for its efficiency the asynchronous proactive refresh protocol proposed by Cachin et al. [16], and the threshold signature scheme proposed by Boldyreva [15]. We choose this scheme [15] for being robust, simpler and more efficient than previous schemes, and for the optimal resiliency (less than half) relevant to our protocol for using quorum sets of different sizes. In addition, the signature generation does not involve interactions among servers, and the signature shares are very short, of length approximately 160 bits. The proactive refresh protocol proposed by Cachin et al. [16] builds on their verifiable secret sharing protocol and an efficient randomized asynchronous multi–valued Byzantine agreement, and has expected message complexity $O(n^3)$. Liveness is guaranteed to the extent that the adversary does not delay messages of the refresh protocol for longer than the time window. Otherwise, the secret key may become unaccessible, and a new TSS key pair has to be generated.

Puzzle techniques Our TSS service uses client puzzles to contain the effects of DoS attacks: a server engages in a resource consuming task upon client request, only after the client has paid a tax in terms of a computational effort. We use the client puzzle suggested by Juels and Brainard [17] for its simplicity.

time():

1) while ($true$)
2) $\quad t_0 \leftarrow H(now)$
3) $\quad \{I_j\}_{j \in Q} \xleftarrow{Q} \text{time}()$
4) $\quad$ if $(H(now) - t_0)(1 - \rho) \leq M$
5) $\quad\quad T \leftarrow Aver(\{Exp(I_j)\}_{j \in Q})$
6) $\quad\quad$ return T
7) end while

Figure 1: The time service.

4 A robust distributed time service

In this section we propose a distributed time service $time()$ resilient to Byzantine failures, that provides an accurate estimate of the real time based on some fixed infrastructure (servers). As mentioned in the Introduction, this protocol solves a problem that is more general than the clock synchronization problem, broadly analyzed in the literature. It relies on a set of n servers that have access to an accurate time source, and whose time value deviates from the real time by at most a constant factor Λ. There are several ways to enforce that: servers can be (all or partially) attached to a GPS receivers as in [20], or periodically synchronized to more accurate sources thus creating a hierarchical structure. Our time service satisfies the following properties in the presence of an adversary: (1) *correctness*: it returns a time value that is within a constant error ϵ from the real time, and (2) *availability*: it is available despite b failures.

4.1 Protocol overview

The protocol underlying the distributed time service is very simple. It follows the interval paradigm proposed in [23], and is built on top of Byzantine quorum systems. It applies Cristian probabilistic clock reading [21] and the Marzullo fault-tolerant average algorithm [22]. We denote by $\xleftarrow{Q} f$ a remote call invoking f on quorum Q; if the client does not receive a reply within a certain timeout, it contacts another quorum set. The minimum one–way message delay is denoted by m, and by M the maximum *round–trip* delay acceptable by the application. As pointed out by Cristian [21], M shows an inherent *trade–off* between accuracy and efficiency, and therefore should be tuned by the application. We follow the notation used in Section 3.1.

Figure 1 illustrates the client side of the protocol. Clients simply pool time information from a quorum Q, line 1:3. If the round trip delay is *acceptable*, line 1:4, the client computes for each time interval returned by the server, its approximation at the current time. This is done by means of a translation and expansion of the time interval. More precisely, given the server interval $I = [l, L]$, the client computes interval $Exp(I) := [l + m(1 - \rho),\ L + (1 + \rho)^2(D - m)]$ where D is the round trip delay. Procedure $Aver()$ computes the Marzullo fault–tolerant average algorithm [22] on the set of the *expanded intervals*, and returns its midpoint. The Marzullo average interval $[l, L]$ is such that l is the maximum left side interval contained in $n - b$ intervals and L is the minimum right side contained in $n - b$ intervals.

4.2 Analysis of the protocol

The following Lemmas derive simple properties based on the correctness of H and its maximum drift rate ρ.

Lemma 1 *If ρ is a valid upper bound of the hardware drift rate and the periodic synchronization are correct, then $I(t) = [C(t) - E(t), C(t) + E(t)]$ is a correct time interval at real time t.*

Lemma 2 *If $I = I(t)$ is a correct time interval sent by a server at real time t, then $Exp(I)$ is a correct interval at the real time t' at which the client received I, and $m(1-\rho) \leq t' - t \leq (1+\rho)^2(D-m)$ with D its round trip delay.*

The following Marzullo's theorem [22] provides an upper bound for the size of the fault–tolerant average interval. This is important to establish the degree of accuracy of our time service.

Theorem 1 *(Marzullo) Let $\mathcal{I}$ be a set of q intervals and $b < \lfloor \frac{q}{2} \rfloor$, then the fault-tolerant average interval is smaller than the $2b+1$ smallest interval in $\mathcal{I}$.*

Lemma 3 *If each server clock deviates from the real time at most by Λ time units, the value returned by time is within a constant error ϵ from the real time. More precisely, ϵ is bounded by $\Lambda + \frac{M-2m+\rho M}{2}$ with M maximum acceptable delay.*

Proof The quorum size of any b–masking quorum system is greater than $3b$. Let us suppose by contradiction that there exists a b–masking quorum system $\mathcal{Q}$ and a quorum set S in $\mathcal{Q}$ such that $|S| \leq 3b$, and that b servers in S fail leaving only $x \leq 2b$ servers in S that are operational. Then, because of the consistency property of Byzantine quorum systems, every quorum would contain at least one faulty server, thus contradicting the availability property. Therefore, for Lemma 1, the client receives at least $2b+1$ correct intervals from each quorum. The correctness of the average time interval derives from Lemma 2 and the Marzullo algorithm [22]. The time accuracy is inferred by Theorem 1 that provides an upper bound on the size of the average interval, smaller than the largest *correct expanded* interval. Qvd

The computational cost at the client side is $O(q \log q)$ with q quorum size [22]. Notice that, when applied to synchronize a known set of clocks, this protocol is more efficient than previous ones (i.e. [26, 20, 25]) for relying on a subset of servers.

Resiliency If the quorum set consists of $2b+1$ servers, Theorem 1 does not provide any guarantee on the size of the average interval, since b malicious servers could send a very large interval, thus compromising correctness. However, it is reasonable to assume that clients know the degree of accuracy Λ of the time servers, and discard intervals larger than 2Λ. In this way, clients can communicate with only $2b+1$ servers, thus improving the access cost per operation and increasing the resiliency b to one third of servers.

Denial of service attacks launched by malicious clients against the servers are feasible. However, this does not depend on the protocol itself since servers perform only a constant look–up, but rather on the openness of the system, and can be addressed by well–known techniques.

5 A scalable and efficient TSS

In this section we describe our distributed TSS built on top of the time service proposed in Section 4. The computational load is distributed quite evenly among clients and servers to detect malicious client/server behavior. The idea underlying the time–stamping protocol is very simple: the protocol is invoked by a client on input F, and consists of two phases. The first phase, called $getTime$, is an adaptation of our time service: the client sends a request for time information attaching the message digest x of F to have an accurate estimate T_x of the real time. In the second phase, called *sign*, the client asks a quorum to compute a *partial signature* of $\langle x, T_x \rangle$, an then gathers $2b+1$ partial signatures to compute the digital signature σ_x of $\langle x, T_x \rangle$. According to Figure 2, T_x approximates the invocation real time of $getTime$ however, it can be easily adjusted to reflect the invocation real time of the *sign* phase. The first choice is particularly advantageous for applications in which *timeliness* is crucial (i.e.

stock–trading, real time services), while the second one for applications in which the timestamp should reflect the real time at which an authorization or resource has been granted. This makes the protocol more *flexible* and easily adaptable to different application requirements. Notice that, the order imposed on the timestamps is a *refinement* of the order provided by the client invocation/response time of the whole protocol. This reduces the degree of concurrency among client requests thus providing a more accurate order over the timestamps, and reducing the degree of freedom of an adversary.

5.1 TSS building blocks

Time service The time–stamping protocol requests in the first phase of the protocol time information, mainly for three reasons. 1) The $getTime$ phase allows the TSS servers to validate the timestamp before signing it, thus preventing a malicious client from back– or post–dating it during the execution of the protocol for more than $M - 2m + 2\Lambda$ time units, with M *maximum admissible* round trip delay, and $2m$ *minimum feasible* round trip delay. 2) The $getTime$ phase dismisses clients from having their clock synchronized to an external reference source. This can be advantageous for applications involving a large number of transient clients. 3) It allows to filter out requests semantically incorrect, and provides *some sort* of protection from DoS attacks against servers, by means of puzzle techniques. In Section 5.2.1 we discuss the option of a one–phase protocol. Notice that, rejecting $getTime$ replies with round trip delay longer than a fixed constant M as done in Fig 1:4, might result in a client vulnerability since an adversary could launch a DoS attack against him by simply delaying $getTime$ messages. This could be a serious thread for applications in which timeliness is crucial (i.e. stock–trade, bidding). However, a closer look at our protocol shows that this is not a problem since M is intended to bound the timestamp accuracy. It can be render dynamic by computing the round trip delay M_x for each request x, and having the TSS servers sign $\langle x, T_x, M_x \rangle$ after validating it. Clearly, the smaller M_x, the more accurate the timestamp T_x. We follow this approach since it is more flexible. We omit here details necessary to adjust our protocol for timely applications since they are technical. Subscriptions of M_x, T_x are omitted whenever it is clear their context.

Puzzle techniques We employ puzzle techniques to contain DoS attacks against the servers, carried out by malicious clients. Our puzzle mechanism is a local mechanism controlled by the server and activated by it *only* in response of higher server load. When activated, the server replies to $getTime$ by computing a puzzle for the client to solve. The difficulty of this puzzle (number of bits to guess) is tunable and depends on the server load, though it is bounded to protect clients against attacks launched by Byzantine servers. The idea of using a tunable puzzle was employed in [18], though in a different context.

Key revocation In case the TSS secret key has been compromised, our TSS resolves disputes by means of *server logs*. After computing a partial signature, each server inserts *relevant data* into a local hash table L_w relative to the current time window, (i.e. the w^{th} time window since the system init time). At the end of each time window and before rebooting, the server stores L_w on a *read–only* removable secondary storage. If the TSS private key is revoked at time T_{rev}, a process can verify the validity of a timestamp issued before T_{rev} by sending a request with $\langle x, T_x, \sigma_x \rangle$ to all servers. Each server verifies σ_x, gets the correspondent log and looks for that entry. If T_x is contained in at least $b + 1$ server logs then it is authentic since the adversary can compromise at most b servers during each time window. If this timestamp is still in use, servers can sign it with the share of the new TSS private key, or cache it. However, this is unlikely to happen in our target applications that rely on *short–lived* timestamps.

5.2 The time–stamping protocol

We assume that the clock of the TSS server is synchronized and deviates from the real time by at most Λ time units. The client clock is not necessary synchronized and its maximum drift rate is equal to ρ. Figure 2 illustrates the client side of the time–stamping protocol, and Figure 3 the server side. Protocol *timestamp* is invoked by the client on a message F. The $getTime$ phase is an adaptation of the time service proposed in Section 4: the client sends a time request to a quorum Q attaching x, message digest of F, Fig 2:4. Upon receiving a $getTime$ request,

timestamp(F):

1) $x \leftarrow hash(F)$
2) $t_0 \leftarrow H(now)$
3) $\{\langle I_j, p_j\rangle\}_{j\in Q} \overset{Q}{\longleftarrow}$ **getTime**(x)
4) $M \leftarrow H(now) - t_0$
5) $T \leftarrow Aver(\{Exp(I_j)\}_{i\in Q}) - M$
6) $\{y_j\}_{j\in A} \leftarrow$**solvePuzzle**$(\{p_j\}_{j\in Q})$
7) $PS \leftarrow \perp;\;\; S \leftarrow A$
8) while $((|PS| < 2b+1) \wedge (H(now) - t_0 < \eta))$
9) $\quad \{\sigma_j\}_{j\in S'} \overset{S}{\longleftarrow}$ **sign**(x, T, M, y_j)
10) $\quad PS \leftarrow PS \cup \{\sigma_j\}_{j\in S'}$
11) $\quad$ if $(|PS| \geq 2b+1)$
12) $\quad\quad \sigma \leftarrow$**assemble**$(PS)$
13) $\quad\quad$ return $\langle x, T, M, \sigma\rangle$
14) $\quad$ else $S \leftarrow S \setminus S'$
15) end while

Figure 2: The client side.

Server S_j:

getTime(x) :
1) $t \leftarrow C(now)$
2) if $(highLoad)$
3) $\quad p_j \leftarrow$ **puzzle**(x, t, y_j)
4) $\quad R \leftarrow R \cup \langle x, t, y_j\rangle$
5) else
6) $\quad p_j \leftarrow \perp$
7) return $\langle I(now), p_j\rangle$

sign(x, T, M, y_j) :
1) if $(\langle x, y_j\rangle \in R \;\wedge\; M > 2m)$
2) $\quad$ if $m - \Lambda < t - T < M - m + \Lambda$
3) $\quad\quad \sigma_j \leftarrow$**psign**$(x, T, M)$
4) $\quad\quad R \leftarrow R \setminus \langle x, t, y_j\rangle$
5) $\quad\quad L_w \leftarrow L_w \cup \langle x, T, M\rangle$
6) $\quad\quad$ return σ_j
7) else return $\perp$

Figure 3: The server side.

each server in Q performs a local clock reading $C(now)$, in case of high server load it computes a fresh puzzle p_j based on $x, C(now)$ and a random value, and inserts $\langle x, C(now), y_j\rangle$ in a temporary hash table R, with y_j solution of puzzle p_j, Fig 3 getTime:1-4. This is done to filter out requests that do not follow the semantic of the protocol. Entries in R older than τ time units are removed. The timeout τ is a parameter tuned by the application, and is greater than the time estimated to solve the most difficult puzzles.

Upon receiving a quorum of responses, the client computes a close approximation of the current real time as done in our time service, Fig 2:5. The time value T is the timestamp associated to x and approximates the real time at which the client invoked $getTime$, with an additional error ρM that is negligible for our discussion. Then, the client calls $solvePuzzle$, Fig 2:6, which spawns one thread for each puzzle to solve, and returns a subset of at least $2b+1$ puzzle solutions, including empty solutions. Notice that in condition of stability the client *do not* solve *any* puzzle, since at least $2b+1$ servers sent empty puzzles, and therefore proceeds without additional computations to the second phase of the protocol. To prevent clients from wasting time in solving unsolvable puzzles sent by Byzantine servers, $2b+1$ solutions are sufficient to proceed to the *sign* phase. Several strategies can be employed by $solvePuzzle$ in case the servers are overloaded, depending on the application requirements: it can try to solve as many puzzles as possible to improve the resiliency of the *sign* phase, or only a minimum number, or combine both approaches by having a timeout. We denote by A the subset of servers in Q whose puzzle has been solved, by S the servers in A to contact at line Fig 2:9 and S' the servers that replied within the timeout, and by PS the set of partial signatures.

In the second phase, the client contacts servers whose puzzle has been solved, and ask them to sign $\langle x, T, M\rangle$, Fig 2:9. If the client receives $2b+1$ partial signature then, it calls $assemble$, Fig 2:11-12, that produces the digital signature of $\langle x, T, M\rangle$ by choosing uniformly at random the partial signatures to verify. Notice that $b+1$ correct partial signatures are sufficient to produce a valid digital signature, but are not sufficient to guarantee the validity of the timestamp in case of key revocation (though this might not be relevant for short–lived timestamps). Therefore, in order to resolve disputes the client waits for $2b + 1$ partial signatures. This complicates a bit the protocol, since the client might not receive a server reply because of process crash, or message delay longer than the timeout, or because a server is under DoS attack. This justifies the while loop at line Fig 2:8, which terminates as soon as the client receives $2b + 1$ partial signatures or after $\eta = (1 - \rho)(m + \tau - \Lambda)$ timeout. More precisely, the client re–submits a $getTime$ request after η time units elapsed since its last $getTime$ request. Different strategies can be employed if the client does not receive $2b+1$ partial signatures (i.e. the client re–contacts the remaining servers Fig 2:14). Another strategy that improves the response time, attempt to produce a digital signature from at least $b + 1$ partial signatures, and if successful return T while contacting in the background the remaining servers to guarantee a quorum of $2b + 1$ acknowledgments. Upon receiving a *sign* request, the server verifies it and validates timestamp T and y_j if $p_j \neq \perp$ Fig 3 sign:1-2. It computes the partial signature, and logs this event in L_w, Fig 3 sign:3-5.

5.2.1 TSS discussion

Two–phase protocol vs. one–phase The $getTime$ phase of the protocol can be omitted by strengthening system assumptions. For instance, in an asynchronous system (i.e. in a LAN) in which clients have limited access and their clock are securely synchronized to an external reference source, the client could set the timestamp to the current value of its virtual clock $T = C(now)$, and send a sign request to a quorum. Correct servers sign it only after validating T to avoid back– or post–date.

Distributed linking vs. absolute time Besides the drawbacks pointed out in the Introduction and Section 2, linking schemes seem not to be suitable for asynchronous distributed TSS resilient to Byzantine failures because they introduce additional complexity. Clearly, different servers can receive client requests in a different order due concurrency and message delays. Therefore, in order to provide a global order of the client requests, servers need to synchronize their local linking chains thus increasing the communication and computational complexity of the TSS. Similarly, the secure public storage medium should be always available and resilient to intrusions, thus increasing the complexity of the system. Our scheme is simpler and also more flexible then the previous schemes. It involves neither coordination among servers, since they exchange messages only at the beginning of each time window to refresh secret shares, nor a public storage medium. Such simplicity leads not only to a better performance but also makes the TSS less vulnerable to software bugs, much easier to occur in complex systems.

5.3 Correctness analysis of the TSS

We can say that the timestamp T_x associate to x and issued by our TSS, is *authentic* if *at least* one correct server has computed a partial signature of $\langle x, T_x, M_x \rangle$, where T_x approximates within a constant error ϵ the real time at which the client invoked $getTime$. However, since our TSS should resolve disputes in case of key revocation, we strengthen this condition and say that T_x is *authentic* if *at least* $b + 1$ correct servers have computed the partial signature.

Lemma 4 *If the TSS secret key has not been compromised, then any timestamp T_x issued by our TSS and provided with a valid signature σ_x is authentic. An adversary cannot tamper T_x by back– or post–dating it by more than $\varphi = M_x - 2m + 2\Lambda$ time units, or modify the request associated to it, or insert an old timestamp in the list of timestamps previously delivered. If the TSS private key has been revoked at time T_{rev}, correct timestamps issued before $T_{rev} - \varphi$ are still valid.*

Proof If the TSS secret key has not been compromised, then for the security of the underlying threshold signature scheme [15] no polynomial–time adversary that corrupts b servers can learn any information about the TSS secret key or forge a valid signature on a new message of its choice. Therefore, according to the above definition, the timestamp T_x issued by our TSS is authentic, provided a valid signature σ_x. Since servers validate the timestamp before signing, it approximates within a constant error the real time at which the client issued the $getTime$ request, and this error is defined by Lemma 3. Because of the timestamp validation, a malicious client cannot back– or post–date it by more than $M_x - 2m + 2\Lambda$ time units, size of the time uncertainty $[t - (M_x - m) - \Lambda, t - m + \Lambda]$ of T_x, Fig 3 sign:2. Since each client request is associated to the message digest x of the request, tampering a request would imply finding a collision of the hash function. If the key is revoked at time T_{rev}, any timestamp issued before $T_{rev} - \varphi$ can be verified by means of server logs. If a timestamp is contained in $b+1$ server logs then it is valid, since at most b servers can be compromised within a time window. Otherwise, it can be either illegal or submitted by a client that crashed during the $sign$ phase. Qvd

Lemma 5 *Protocol timestamp terminates if the network eventually becomes stable.*

Proof Let us consider the worst case in which each server in a quorum Q replies to a $getTime$ request with a puzzle, and the client solves s puzzles with $2b + 1 \leq s < |Q|$. A client might not receive a partial signature from a server within a timeout because a server is faulty, or under DoS attack, or the message is delayed more than the timeout. Let us suppose that the client does not receive $2b+1$ partial signatures. If the missing replies are from faulty servers, then the client is able to solve the remaining puzzles since the difficulty of puzzles is bounded, and gets $2b + 1$ partial signatures the next time. Otherwise, if the network is instable the client executes the while loop until it gets $2b+1$ replies, or η time units elapse. The robustness of the underlying threshold signature scheme ensures that the signature generation protocol terminates despite b faulty parties. Qvd

Lemma 6 *The TSS is available despite b malicious failures and crash failures.*

Proof The verification procedure is always available since it does not involve communications with any process. The time–stamping protocol is also available despite b failures because it relies on Byzantine quorum systems (availability property) and it requires only $2b + 1$ partial signatures from a quorum set Q containing more than $3b$ servers (see Lemma 3). Lemma 5 guarantees protocol termination if the network eventually gets stabilized. Qvd

5.3.1 Complexity analysis of the TSS schemes

In this subsection we discuss the computational and communication complexity of our TSS protocol and compare it with general linking schemes.

Tree–like linking schemes: the cost of time–stamping depends on the number N of requests submitted at a round. In fact, at the end of each round the TSS computes the timestamp of the requests submitted at that round, and for each request its authentication path. The computational cost of time–stamping is $O(N \lg N)$ since the TSS computes the Merkle hash tree whose leafs are the requests (in some TSS are ordered), and for each request its corresponding tree branch to allow the client to verify the timestamp. Notice that since N is hardly predictable, the hash tree is not usually complete and this implies a higher cost. Clearly, the cost is noticeably increased when the TSS has to publish the round values. The cost of verifying a timestamp at the client side is dominated by the cost of rebuilding the tree branch of the hash Merkle tree, that is $O(\lg N)$. In some system the verification involves also the interaction of on–line server, with the aim of reducing the TSS trust. Notice that the size of the timestamp is $O(d \log N)$ with d size of the message digest, and does not scale well to high volume of requests. In a distributed

TSS based on linking schemes the communication cost would be $O(n \log N)$ with n number od servers. Notice also, that the response time is *subordinated* to the completion of the round.

Our distributed TSS: the computational cost of time–stamping at the server side, is dominated by the computation of the partial digital signature that is equal, according to Boldyreva's scheme [15], to $\sigma_j = \langle x, T_x, M_x \rangle^{s_j}$ at server S_j, with s_j secret share. As mentioned before, the TSS can implement different strategies according to the application requirements, and require only $2b+1$ servers (half of the universe) to sign it. At the client side, the cost of time–stamping is given by $C + P + sV_\sigma + S_\sigma$ where $C = O(q \lg q)$ is the cost of computing timestamp T with q quorum size, P is the cost of computing the puzzles if necessary (in normal condition it is null), sV_σ is the cost of verifying s partial signatures with $b+1 \leq s \leq 2b+1$, and S_σ is the cost of producing a digital signature σ. In Boldyreva scheme [15] the verification of partial signature implies the decisional Diffie–Helman problem, and the computation of the digital signature involves multiplications of the valid partial signatures. Clearly, the computational cost depends on the underlying threshold signature scheme and is independent from the TSS scheme. Notice that since the number of TSS is usually kept small, the client has to verify very few partial signatures. In addition, the client is not forced to verify the validity of the digital signatures since $2b+1$ replies provide a proof of the validity of the timestamp. The verification task could be delegated to some proxy (not necessary trusted), thus reducing the cost to $C + P + V$ with V cost of verifying the digital signature σ returned by the proxy. Notice that in contrast with some TSS based on on–line server, the verification procedure does not involve any additional communication, and its cost V is constant. Notice that the size of the timestamp is constant and is much shorter for using short digital signatures (i.e. 160 bits). The communication cost is $O(q)$, thus improving the network bandwidth and system load balancing, and the client does not experience additional delays besides message delays.

6 Applications

The applicability of digital time–stamping services is broad and goes beyond secure archives or intellectual property. Its importance becomes clear in the presence of legal use of electronic documents (i.e. e–payment, on–line banking) and sensitive resources. We discuss here only those applications involving large populations and requiring fine granularity and good response time, for which our TSS is particularly suitable in contrast with previous works.

Fine granularity and good response time play an important role in *temporal access control* in which an authorization is automatically granted in a specified interval and revoked when such interval expire [30]. Our TSS can suit well temporal access control mechanisms and be employed as a fundamental building block. It could be applied to the management of sensitive resources such as data bases, or resources whose access should be charged according to the duration. Potential applications concern also cooperative work–flow, or *information portals* [31] serving as main providers to gather information from *distributed* data sources. This can be relevant for financial services [1], such as quoting services, requiring *data freshness* to provide a real time snapshot of the market (with new high and low) for which pricing delays should be kept short.

Fine granularity, timeliness, and high scalability play an important role also in electronic stock–trading. The fastest growing stock exchange NASDAQ is wholly electronic with transactions conducted over computer networks across an international area. Important requirements for stock market transactions are timeliness, efficiency in the orders to buy/sell stocks, and the ability to detect delays in the transaction executions. We think that our TSS can suit well these applications we are currently working in this direction.

[1] For instance, Kiplinger.com http://www.kiplinger.com

7 Conclusion

We have proposed a distributed digital timestamp that employs a novel approach than previous proposals and that offers promising features. It is highly available and tolerates intrusions. In contrast with previous solutions, it has fine granularity, good response time, and is highly scalable. Our TSS is built on top of a novel time service resilient to Byzantine failures that is highly scalable and efficient. This primitive has independent interest for studying a more general problem than the well–known clock synchronization problem, and for its simplicity and wide applicability.

Acknowledgments: The author would like to thank Mike Reiter for helpful discussions at early stages of this work, and Lavinia Egidi for useful discussions on threshold signature schemes.

References

[1] S. Haber and W. S. Stornetta *How to Time-Stamp a Digital Document* J. Cryptology, 3(2), pp. 99-111 (1991).

[2] D. Bayer, S. Haber, *Improving the efficiency and reliability of digital time-stamping.* In Sequences II: Methods in Communication, Security, and Computer Science, 329-334, 1993.

[3] J. Benaloh, M. de Mare *Efficient Broadcast Time-Stamping.* Tech. Rep. 1, Clarkson University, Dept. of Mathematics and Computer Science, Aug 1991.

[4] J. Benaloh, M. de Mare *One-Way Accumulators: A Decentralized Alternative to Digital Sinatures.* Advances in Cryptology - EUROCRYPT '93, pp. 274-285, May 1993.

[5] S. Haber and W. S. Stornetta *Secure Names for Bit–Strings* In Proc. of the 4th Conference on Computer and Communications Security, pp. 28–35, Apr 1997.

[6] H. Massias, X. Serret Avila, J.-J. Quisquater *Timestamps: Main issues on their use and implementation.* In Proc. of the 8th Workshop on Enabling Technologies (WETICE '99), pp.178-183, 1999.

[7] Petros Maniatis, T.J. Giuli, and Mary Baker, *Building Trusted Distributed Services Across Administrative Domains*, Technical Report cs.DC/0106058, http://www.arxiv.org/abs/cs.DC/0106058, 2001.

[8] P. Maniatis, M. Baker, *Enabling the Long-Term Archival of Signed Documents through Time Stamping.* In Proc. of the USENIX Conference on File and Storage Technologies, pp. 31-45, Jan 2002.

[9] Surety Inc. *Secure time/date stamping in a public key infrastructure.* Available at http://www.surety.com

[10] B.Preneel, B. Van Rompay, J.-J. Quisquater, H.Massias, J. Serret Avila *Design of a timestamping system.* WP3 Technical Report, 1998, http://www.dice.ucl.ac.be/crypto/TIMESEC.

[11] A. Buldas, P. Laud, H. Lipmaa and J. Willemson *Time-Stamping with Binary Linking Schemes.* Advances in Cryptology - CRYPTO '98, pages 486–501, Aug 1998.

[12] A. Buldas, H. Lipmaa, B. Schoenmakers, *Optimally efficient accountable time-stamping.* In Proc. 3rd International Workshop on Practice and Theory in Public Key Cryptography (PKC '00), pp. 293-305, Jan 2000.

[13] A. Ansper, A. Buldas, M. Saarepera and J. Willemson *Improving the Availability of Time-stamping Services*, In Proc. of the 6th Australian Conference (ACISP '01), 360-375, Jul 2001.

[14] A. Ansper, A. Buldas, M. Roos, J. Willemson, *Efficient long-term validation of digital signatures.* In Proc. of the 4th International Workshop on Practice and Theory in Public Key Cryptography (PKC '01), pp 402-415, Feb 2001.

[15] A. Boldyreva *Efficient threshold signatures, multisignatures and blind signatures based on the Gap-Diffie-Hellman-group signature scheme.* In Proc. 6th International Workshop on Practice and Theory in Public Key Cryptography (PKC '03), pp. 31-46, Jan 2003.

[16] C. Cachin, K. Kursawe, A. Lysyanskaya and R. Strobl *Asynchronous verifiable secret sharing and proactive cryptosystems.* In Proc. of the 9th ACM Conference on Computer and Communications Security (CCS '02), pp. 88-97, Nov 2002.

[17] A. Juels, J. Brainard *Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks.* In Proc. of Networks and Distributed Security Systems (NDSS '99), pp. 151-165, 1999.

[18] X. Wang, M. K. Reiter *Defending against denial-of-service attacks with puzzle auctions.* In Proc. of the IEEE Symp. on Security and Privacy, pp. 7892, May 2003

[19] M. Castro, B. Liskov *Practical byzantine fault tolerance and proactive recovery.* ACM Trans. Computer Systems 20(4): 398-461 (2002).

[20] C. Fetzer and F.Cristian. *Integrating external and internal clock synchronization.* Journal of Real-Time Systems, 12(2), pp. 123-171, Mar 1997.

[21] F.Cristian. *Probabilistic clock synchronization.* Distributed Computing 3(3), pp. 146-158, 1989.

[22] K.Marzullo. *Tolerating Failures of Continuous-Valued Sensors.* ACM Trans. on Computer Systems, 8(4), pp. 284-304, Nov 1990.

[23] K.Marzullo, S.Owicki. *Maintaining the Time in a Distributed System.* In Proc. of the 2nd ACM Symp. on Principles of Distributed Computing (PODC '83) pp. 295-305, Aug 1983.

[24] P. Verissimo, L. Rodrigues, A.Casimiro. *Cesiumspray: a precise and accurate global clock service for large-scale systems.* Real-Time Systems, 12(3), pp. 243-294, May 1997.

[25] K.Schossmaier, B.Weiss. *An Algorithm for Fault–Tolerant Clock State&Rate Synchronization.* Proc. of the 18th IEEE Symp. on Reliable Distributed Systems (SRDS '99), Lausanne, pp. 36-47, Oct 1999.

[26] B.Barak, S.Halevi, A.Herzberg, D.Naor. *Clock synchronization with faults and recoveries.* In Proc. of the 9th ACM Symp. on Principles of Distributed Computing (PODC '00), pp. 133-142, Jul 2000.

[27] D.L. Mills *Improved algorithms for synchronizing computer network clocks.* IEEE/ACM Trans. Networks 3(3), pp. 245-254, June 1995.

[28] *The Science of Time-keeping.* Hewlett Packard, application Note 1289.

[29] D.Malkhi, M.Reiter, A.Wool *The load and availability of Byzantine Quorum Systems.* SIAM J.Computing 29(6), pp. 1889-1906, (2000).

[30] E. Bertino, C. Bettini, E. Ferrari, P. Samarati *An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning.* ACM Trans. Database Syst. 23(3): 231-285 (1998).

[31] V. Atluri, A. Gal*An authorization model for temporal and derived data: securing information portals.* ACM Trans. on Information and System Security (TISSEC), 5(1), 2002.