A Scalable and Intrusion–tolerant Digital Time–stamping System

Daniela Tulone
University of Pisa, Italy and MIT CSAIL, USA
tulone@csail.mit.edu

***Abstract*— Secure digital time–stamps play a crucial role in many applications that rely on the correctness of time–sensitive information. Well–known time–stamping systems are based on *linking schemes* which provide a relative temporal order by linking requests together. Unfortunately, these schemes do not scale well to *large volume of requests*, and have *coarse granularity* and *high latency*. Therefore, they are unsuitable for very large and dynamic systems, or applications requiring fine–grained and short–lived timestamps, or timeliness, such as stock trading, e–auctions, financial applications, aggregation of real–time sensitive information, and temporal access control.**

We propose a different scheme, based on real–time timestamps, which overcomes those drawbacks and leads to a performance enhancement. Our time–stamping system is intrusion–tolerant and it is based on a novel *robust time service* suitable for very large populations, on a robust threshold signature scheme, and on quorum system techniques. We prove its correctness and liveness, and compare its computational and communication complexity to the complexity of tree–based linking scheme. Finally, we show how the fine–granularity, improved scalability and efficiency of our scheme make it particularly suitable to those applications mentioned above, and also to mobile e–commerce.

I. Introduction

More than ever, security 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 and give proof of the time at which they were created or modified. A digital time–stamping system (TSS) addresses this problem: it binds *authentically* a request to a certain point in time, and provides the ability to prove that. It usually consists of a time–stamping server that issues, for each client request, a timestamp and a non–malleable proof of its validity. Such a proof allows any process to validate the timestamp associated to it. Clearly, the TSS has to guarantee the correctness of the timestamps issued in the presence of an adversary: the timestamp must represent the time at which it was issued, and an adversary must not back– or post–date it, or insert an old timestamp among those previously issued.

There are many applications that require such a time–binding both in wired and wireless networks: e–commerce, financial applications, stock trading, aggregation of real–time sensitive information. The vast majority of well–known TSS systems [14], [4], [6], [18], [19], [8], [15], [1], including commercial ones such as Surety Technology [26], are based on *linking schemes* that link requests together by means of a collision–resistant hash function, thus providing a *relative temporal order* over the set of requests. They work well for applications involving long–term timestamps, such as e–archives and intellectual property, but are unsuitable for applications which involve very large systems or require *fine–grained* and *short–term* timestamps to carry out sensitive tasks (i.e., temporal access control, financial transactions, stock market, e–commerce, and aggregation of real–time information). It is well–known that tree–based linking TSSs scale poorly to large volume of requests, and have coarse timestamp granularity and high client latency. These drawbacks are noticeable in applications where low latency and system performance play a crucial role, such as in those mentioned before. We discuss deeply these pitfalls in Section II.

Contributions. In contrast with previous works, we focus mainly on real–time applications requiring short–lived and fine–grained digital time–stamps, and involving large and dynamic systems. We propose a robust and scalable time–stamping system (RSTS) for asynchronous systems, which is resilient to Byzantine client and server failures, and relies only on real–time digital timestamps. The RSTS is based on a novel *robust time service* (RT), and on a robust threshold signature scheme [7]. Notice that our RT primitive solves a problem that is *more general* than the well–studied clock synchronization problem since it provides a generic process (even mobile) with an accurate time estimate based on some fixed infrastructure. This problem has lower complexity than the clock synchronization problem which requires *periodic* clock adjustments among *all* processes, and whose cost becomes prohibitive in very large systems. Notice that clock synchronization might not be convenient in mobile applications, due to energy constraints, and that the client side of most applications (i.e., Internet–based) do not require to be synchronized all the time, but rather *on–demand* to perform some critical steps. The RT service derives an estimate of the reference time from a *subset* of servers. Such an estimate is accurate in the presence of malicious intrusions. Notice that the RT service makes the RSTS system highly flexible: we show in Section V how the timestamps can be tuned according to the application needs. Both the RT service and the RSTS system are built on top of Byzantine quorum systems [17] to improve the load balancing and the scalability of the system, and reduce the access cost per operation. Since in a real setting an adversary can compromise more than a threshold of servers during a sufficient large interval, and the private RSTS can become compromised, a practical TSS should also employ proactive security, and resolve disputes in case of key revocation. As discussed in Section V, our RSTS can address both issues, though these features do not represent the core of our proposal. In Section V-E we discuss in details the following RSTS features which are difficult to achieve with linking schemes:

- Robustness: intrusion–tolerance and availability, dispute resolution in case of key revocation, no trusted dealer, and reduced vulnerability to DoS attacks;
- System performance: high scalability, better load balancing and response time;
- Fine–grained timestamps consisting of real–time values, more expressive and portable than relative timestamps, and flexibility to the application needs.

The paper is structured as follows: Section II discusses related works, and Section III describes our system model and the techniques used. In Section IV we present our RT service, and describe our RSTS in Section V, and prove its correctness. We also analyze its complexity and compare it with tree–like linking schemes. In Section VI we briefly show how to apply the RSTS to a stock–market, an e–action, aggregation of real–time sensitive information, and mobile e–commerce.

II. Related works

Several linking schemes [14], [6], [15], [8], [9] have been proposed, ranging from *linear linking* [14] to *tree–like linking*. The tree–like linking scheme, proposed first by Benaloh and de Mare [6] and then by Haber and Stornetta [4], [15], improves linear linking in terms of storage space, verification costs, and it is more robust for relying on the lifetime of collision–resistant hash functions that is longer than in digital signatures [22]. These features make this scheme the mostly used linking scheme, also by commercial TSS [26]. In a tree–like linking TSS the time–stamping procedure runs in *rounds*: the server collects requests submitted by the clients during a round, and computes the Merkle hash tree whose leaves are the requests. The root value of the tree is concatenated with the value obtained from the preceding round, and hashed again to obtain the actual round value [22]. Clearly, requests submitted during the same round appear to occur *simultaneously*. The timestamp contains information necessary to rebuild the tree branch of the correspondent request in the Merkle tree to let the client verify its validity. Since a malicious server could subvert the order of the timestamps, the server periodically publishes the round values to a trusted and public medium (i.e. newspaper). This occurs rarely (i.e. hourly) since it is time consuming and costly, while the round duration is kept short (i.e. every minute) since it determines the timestamp granularity. Tree–like schemes present the following drawbacks, noticeable in real–time and e–commerce applications, and not present in our RSTS for its different approach.

- As pointed out in [22], tree–based TSS do not scale well to high volume of requests since the cost of time–stamping and verifying, and the size of the timestamp grow with the number of requests submitted per round. This makes the TSS particularly vulnerable to DoS attacks. Massias et al [22] propose different levels of timestamp authority to improve scalability, however this does not provide a full answer to the problem. They also address the *portability* of timestamps issued by different TSS systems by distinguishing documents for internal and external use. However, this strategy is very costly and not flexible.
- A client has to wait for the completion of the round in order to get its timestamp.
- The granularity of the timestamps depends on the round duration, which is usually 1 minute. Notice that there is an inherent trade–off between granularity, and system performance (computational cost, memory). Moreover, since the round duration is a–priori fixed, it is unlike to fit the volume of client requests which is usually unpredictable.
- Since clients need guarantees of *accountability*, the server has to take extra steps to convince them that it cannot cheat by subverting the order of timestamps between publishing. There have been several attempts to minimize the trustworthiness of the TSS in the past, based on trusted digital notary [2], [9], [26], or on binary linking [8], or on authentication graphs [9]. However, all these solutions are costly and rely on a centralized timestamp authority and trusted storage medium. Ansper et al. [1] address TSS availability in case of crash failures by using multiple servers. However, their work cannot be extended to Byzantine failures, and does not scale well to high volume of requests and large populations. Note that an intrusion–tolerant TSS based on tree–like linking scheme has high complexity, since servers have to maintain a consistent view, with additional computational and communication costs. Maniatis et al [18] proposed an architecture for an intrusion–tolerant TSS based on linking scheme which relies on randomized agreement. The public storage medium should also be robust to crash and intrusions.

A different approach based on *distributed trust* was proposed by Haber and Stornetta [14]. It relies on digital signatures, and a secure pseudo–random generator to select the nodes to contact. As pointed out in [22], this scheme is not practical since it requires a very large set of nodes available for time–stamping. Little attention has been paid to TSS based on real–time timestamps. The straightforward solution of *hashing–and–signing* a request has been disregarded [8], [22] for requiring *unconditional trust* of the timestamp authority, and because of the unreliability of old timestamps after a leakage of the private signature key. RSTS overcomes these problems by using our RT service and a robust threshold signature scheme. The RT service has independent interest for studying a problem that is *more general* than the well–studied clock synchronization problem, and for addressing both scalability and robustness. In fact, clock synchronization protocols suitable for large systems in a WAN, are based on a hierarchical network structure, but assume a weaker abstract fault model [23], [28]. The RT has optimal time accuracy, and improved performance than previous solutions [3], [13], [25] for using quorum systems.

III. Preliminaries

A. System model

Our system model consists of a set of n servers $S_1, \ldots, S_n$ responsible for time–stamping requests, and a *dynamic* set of clients that submit requests to a *subset* of servers. Messages can be *arbitrarily delayed* but not indefinitely. This assumption is realistic since it is reasonable to assume that network instabilities and denial of service (DoS) attacks are reparable. In addition, we assume that communications are authenticated, and that each server has an additional read–only storage medium where it logs the significant events necessary to resolve disputes in case of key revocations. Clearly, if servers employ proactive security they must satisfy the standard assumptions [11] (i.e. be equipped with a secure co–processor, a watchdog timer that periodically interrupts processing and hands control to a recovery monitor, and a read–only memory).

Adversary model. Similarly to [11], [29], our RSTS system employs *proactive security*: each server periodically reboots and refreshes sensitive data stored locally (i.e. authentication keys and secret shares). Therefore, it tolerates *any number* of faults provided that fewer than one fourth of the servers become faulty within a *time window* that is the time elapsed between consecutive refreshes. A *malicious adversary* can compromise b servers during a time window, eavesdrop the network, delay messages though not indefinitely, and collude with clients. We assume a computationally bounded adversary so that with very high probability it is unable to forge digital signatures, or find collisions of one–way hash functions. Clients can fail by crash and be malicious.

Byzantine quorum system. Our protocols are built on top of a b–masking Byzantine quorum system [17], 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*). Notice that $b < \frac{n}{4}$.

Clock model. Each process has a hardware clock $H(t)$ with maximum drift rate ρ, with $\rho \approx 10^{-5}$. Therefore, for any real time $t_2 > t_1$ it measures the passage of time in $[t_1, t_2]$ with maximum error $\rho(t_2 - t_1)$. Each server maintains a virtual clock $C(t)$ which is periodically synchronized to a more accurate time source, so that it always deviates from the reference time by at most Λ time units.

We employ the interval–based paradigm introduced by Marzullo and Owicki [21]: each server maintains a time interval $I(t) = [C(t) - E(t), C(t) + E(t)]$, with $E(t)$ maximum error for $C(t)$. Clearly, $I(t)$ is *correct* at time t if it contains t. If ρ is a valid drift bound, then $I(t)$ is a correct time interval [21]. We apply Cristian's probabilistic reading method [12] to estimate a remote clock for its simplicity. A process estimates a remote clock based on the clock value returned by the remote process, and the round trip delay elapsed between its sending and arrival time. The method assumes a minimum round trip delay m, and a maximum *acceptable* delay M, tunable by the application. If the round trip delay exceeds M, the sender discards the message and retries.

B. 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 $tstamp(x)$ that on an input x produces $\langle T_x, \sigma_x \rangle$ with T_x timestamp of x, and σ_x proof of its authenticity.
2) A verification protocol $verify(x, T_x, \sigma_x)$ that given $\langle x, T_x, \sigma_x \rangle$ verifies the authenticity of the timestamp T_x associated to x.
3) An audit protocol that verifies whether the TSS carries out its duty.

The usual requirements for a digital timestamp can be summarized as followings:

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

C. Cryptographic tools

Threshold signature scheme. Our RSTS relies on a robust (b, n) threshold signatures scheme. We assume the existence of a RSTS private and public key: the RSTS *certified* public key is available to clients, and the correspondent private key is initially secretly shared among the servers, so that $b+1$ *secret shares* are sufficient to compute it, but b shares do not provide any information to our adversary model. Upon receiving a sign request a server S_j generates a partial signature σ_j based on its secret share. The validity of a signature share can be verified, and $b+1$ valid partial signatures can produce a digital signature verifiable with the RSTS public key. In a secure threshold signature scheme, it is infeasible for a computationally bounded adversary to produce $b+1$ valid signature shares on a forged message to be combined and form a valid signature. We apply the threshold signature scheme proposed by Boldyreva [7] for its optimal resiliency relevant to our discussion, and for being robust and more efficient than other schemes (i.e., it does not involve server interactions, and produces short partial signatures, approximately 160 bits). However, other schemes can be applied.

Proactive security. Since an adversary might compromise more than b servers in a sufficiently large time interval, servers periodically refresh the shares of the private RSTS key. We apply the asynchronous proactive refresh protocol proposed by Cachin et al.[10] for its efficiency. It guarantees liveness if the adversary does not delay messages during the refresh protocol for longer than a time window. If this occurs, the private RSTS key may become unaccessible, and a new key pair has to be generated.

One–way hash functions. A user submits the message digest of its request (i.e. it applies MD5 one–way hash function) to guarantee data privacy and improve the network bandwidth.

Puzzle techniques. Our RSTS system uses client puzzle [16] to contain the effects of DoS attacks. In case of high server load, a server asks a client to pay a tax in terms of a computational effort, in order to sign the client request.

IV. A building block: the RT service

In this section we present a simple and robust time service (RT) built on top of quorum systems and resilient to Byzantine failures. The RT service returns an estimate of the current reference time which is accurate up to a constant error ϵ, despite b Byzantine server failures. Clearly, the presence of malicious failures, the uncertainty of the message delays, and the use of quorum systems increase the complexity of computing constant ϵ. However, the application of the Marzullo fault–tolerant average algorithm [20], and an adaptation of the Cristian probabilistic clock reading [12] to estimate the server time interval at the current time, simplifies noticeably this problem. In Lemma 2 we show how it is possible to bound the error by means of properties of the b–masking quorum systems and of the Marzullo algorithm. We denote by $\{r_i\}_{i \in Q} \overset{Q}{\longleftarrow} f$ a remote call invoking f on a quorum Q, and receiving the set of replies $\{r_i\}_{i \in Q}$ from servers in Q. The minimum one–way message delay is m, and the maximum round–trip delay *acceptable* by the application is M.

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

Fig. 1. The time service.

Figure 1 illustrates the client side of the protocol: the client pools time information from quorum Q, line 1:3. If the round trip delay is less than M, it approximates the server time interval at the current time. More precisely, if $I = [l, L]$ is the interval sent by a server, the client computes interval $Exp(I) := [l + m(1-\rho),\ L + (1+\rho)^2(D-m)]$ where D is the round trip delay computed by the client. We call $Exp(I)$ *expanded interval* of I. Function $Aver()$ returns the midpoint of the Marzullo fault–tolerant average interval [20] derived from the set of the expanded intervals. The Marzullo average interval of q intervals among which at most b can be faulty, is the interval with left side equal to the maximum left side contained in $q-b$ intervals, and right side equal to the minimum right side contained in $q-b$ intervals. If the round–trip delay exceeds M, the client contacts another quorum. The following lemmas prove the correctness of the RT service. Lemma 1 follows from the fact that ρ is a maximum drift rate of the hardware clock.

Lemma 1: Let I be a correct time interval sent by a server at time t, then $Exp(I)$ is a correct interval at the time t' at which it was received, with $m(1-\rho) \leq t' - t \leq (1+\rho)^2(D-m)$ and D round trip delay.

Theorem 1: (Marzullo [20]) 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 2: If server clocks deviate from the real time by at most Λ time units, the value returned by RT is accurate with maximum error $\Lambda + \frac{M-2m+\rho M}{2}$.

Proof: The size of a quorum of a b–masking quorum system is greater than $3b$. In fact, if by contradiction there exists a quorum Q of size $3b$, and b servers in Q fail leaving $2b$ correct servers in Q, then each quorum contains at least one faulty server because of the consistency property, thus contradicting the availability property. Therefore, a client invoking RT receives at least $2b+1$ correct time intervals. The correctness and accuracy of the resulting time interval follows from Lemma 1 and Theorem 1. ■

The computational cost of RT at the client side is $O(q \log q)$ with q quorum size [20]. Notice that, when applied to synchronize a known set of clocks, RT leads to a performance improvement with respect to previous protocols [3], [13], [25] since it reduces the access cost, and improves the system load balancing and scalability.

Resiliency. Theorem 1 does not provide a bound on the size of the average interval if $q = 2b+1$, since b malicious servers could send a very large interval. However, if a client knows the accuracy Λ of server clocks (reasonable assumption), it can discard intervals larger than 2Λ. In this way, it communicates with only $2b+1$ servers, thus improving the access cost per operation and increasing the resiliency to $b < \frac{n}{3}$.

Notice that assuming a maximum delay M could make the client vulnerable to DoS attacks (i.e., if the adversary consistently delay messages). We show in Section V-A how to overcome this problem by making M a variable.

V. The RSTS system

In this section we describe the RSTS system which produces real–time timestamps. It is built on top of our RT service and uses a robust (b, n) threshold signature scheme. Note that the fact of relying only on real–time timestamps when ordering requests introduces new issues compared to linking schemes since the timestamp has to represent trustworthily the real time at which it was issued and it must not be back– or post–dated by malicious clients or servers, and because of the delay uncertainty (it should not be assumed to be bounded to reduce DoS vulnerabilities). The definition of the time associate to a request and of the party that is in charge of computing it, is crucial in the design of an intrusion–tolerant TSS based on real–time timestamps. We illustrate our solution to these problems in Section V-A.

A. Protocol overview

The high–level idea underlying the RSTS is very simple: the protocol is invoked by a client on message digest x of its request, and consists of two phases (Section V-E shows also a one–phase protocol). In the first phase called $getTime$, the client computes the value T_x of the timestamp of x, by running the RT service. In the second phase called $sign$, it asks a subset of $2b+1$ servers (half of the servers) to produce a partial signature σ_x of $\langle x, T_x \rangle$, and then gathers $b+1$ correct partial signatures to produce a signature σ_x of $\langle x, T_x \rangle$ verifiable with the RSTS public key. The reason for such a design is threefold:

1) Since both client and servers can be malicious and collude, they can validate each other: servers validate timestamp T_x 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. It allows to filter out requests semantically incorrect, and provide some protection from DoS attacks against servers by means of client puzzles.
2) It makes the RSTS more flexible since timestamp T_x can be tuned according to the application needs (see Section VI). The following three options are available:

   - T_x can be the time at which the client sends a getTime request (error ρM is negligible to our discussion). This choice is suitable for applications in which clients compete for a resource (i.e., e–auction). Our presentation follows this approach.
   - T_x can be the earliest time at which the client completes RSTS. This choice is useful for temporal access control where T_x establishes the access time to a sensitive resource.
   - T_x can approximate the time at which the servers receive the getTime request and send some information (i.e., stock market, quoting).

3) The $getTime$ phase dismisses clients from having their clock securely synchronized to an external reference source. This can be advantageous for applications involving a very large number of transient clients, such as Internet–based or mobile applications.

Round trip delay. Because of the Cristian remote clock reading, getTime requests with a round trip delay larger than M are discarded. This might result in a client vulnerability since an adversary could launch a DoS attack by simply delaying getTime requests, and could be a serious threat for applications in which timeliness is crucial (i.e. e–bidding). This problem is overcome if the client transmits the round trip delay M_x of its getTime request, and asks servers to sign it along with x, T_x, provided $M_x \geq 2m$. Clearly, the smaller M_x, the more accurate the timestamp T_x.

Puzzle techniques. Since in the real setting an adversary can launch a DoS attack, we provide some defence against it. In case of high server load, the server replies to a getTime request by asking the client to solve a puzzle before sending a sign request. In normal conditions servers do not send *any* puzzle. We call a server that sends no puzzle, *ready* to sign. The difficulty of the puzzle [16] (number of bits to guess) is tuned by the server according to its load, however it is bounded to protect clients against attacks launched by Byzantine servers. Below is the high–level description of the RSTS system.

```
Client side:
    send getTime request $\langle x \rangle$ to quorum $Q$
    compute round trip delay $M_x$
    compute $T_x$
    if $\nexists 2b+1$ servers ready in $Q$
        solve necessary puzzles $sp_j$
    send sign request $\langle x, T_x, M_x \rangle$ to $2b+1$ servers in $Q$
    assemble $b+1$ correct partial signatures

Server side:
    getTime($x$)
        if high load
            computes puzzle $p_j$
        send $\langle I(t) \rangle$ and optionally $p_j$

    sign($x, T_x, M_x$)
        if valid request
            compute partial signature $\sigma_j$
            write to removable log $L_w$
            send $\langle \sigma_j \rangle$
```

Key revocation. In our target applications involving short–lived timestamps, key revocation does not play a crucial role. However, for completeness we provide a *basic* mechanism to resolve disputes in case of key revocation. If the RSTS secret key has been compromised, the RSTS resolves disputes by means of *server logs*. Upon computing a partial signature for $\langle x, T_x, M_x \rangle$, the server inserts it into a local hash table L_w relative to the current time window. At the end of each time window and before rebooting, the server stores L_w on a *read–only removable* secondary storage. A timestamp T_x issued before the key revocation time and still *in use*, can be validated by sending a request for $\langle x, T_x, M_x, \sigma_x \rangle$ to all servers. Each server verifies the signature σ_x, and looks for entry $\langle x, T_x, M_x \rangle$ in its correspondent log (secondary storage for timestamps with lifetime *longer* than the time window, and local hash table otherwise). If T_x is contained in at least $b+1$ server logs, then it is authentic.

B. The time–stamping protocol

We describe here the RSTS protocol in more details, Figures 2 and 3 illustrate its client and server side. Subscriptions of M_x, T_x are omitted whenever it is clear their context.

Upon receiving a getTime request, a server returns its current time interval $I(now)$, and in case of high server load computes a *fresh puzzle* p_j for request x. It inserts x, the current time t, and if necessary the solution y_j for puzzle p_j, into a temporary hash table R, (Fig 3 getTime:1-6). This allows the server to validate later T, filter out requests that do not follow the semantic of the protocol, and contain DoS attacks. Entries in R older than τ time units are periodically removed, with τ parameter greater than the time estimated to solve the most difficult puzzle and transmit it. Upon receiving a quorum of responses, the client computes the round trip delay M and timestamp T, (Fig 2:3-4). In the normal case the client detects $2b+1$ *servers ready* and asks them to sign $\langle x, M, T \rangle$. Otherwise, it has to solve some puzzle and calls $solvePuzzle$ which spawns one thread for each puzzle to solve. Clearly, other strategies can be employed by $solvePuzzle$ depending on the application requirements: it can try to solve as many puzzles as possible to improve the resiliency of the $sign$ phase, or use also a timeout. We denote by S the subset of $2b+1$ servers of Q which are ready or whose puzzle has been solved.

In the second phase, the client asks servers in S to sign $\langle x, T, M \rangle$, Fig 2:9. Upon receiving a $sign$ request, the server verifies its correctness and validates T, and y_j if needed, Fig 3 sign:1-2. Then, it computes a partial signature σ_j for

```
tstamp(x):

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

Fig. 2. The client side.

```
getTime($x$) :
1) $t \leftarrow C(now)$
2) if $(highLoad)$
3)     $p_j \leftarrow$ puzzle($x, t, y_j$)
4) else
5)     $p_j \leftarrow \perp$
6) $R \leftarrow R \cup \langle x, t, y_j\rangle$
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)     if $m - \Lambda < t - T < M - m + \Lambda$
3)         $\sigma_j \leftarrow$psign($x, T, M$)
4)         $R \leftarrow R \setminus \langle x, t, y_j\rangle$
5)         $L_w \leftarrow L_w \cup \langle x, T, M\rangle$
6)         return $\sigma_j$
7) else return $\perp$
```

Fig. 3. Server S_j.

$\langle x, T, M\rangle$, and logs this event in L_w, Fig 3 sign:3-5. As soon as the client receives $2b+1$ partial signatures, it calls $assemble()$ which verifies the validity of $b+1$ partial signatures (chosen uniformly at random) and produces a signature σ_x, Fig 2:11-12. 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). However, in order to resolve disputes the client waits for $2b+1$ partial signatures. This justifies the while loop at line Fig 2:8, which terminates as soon as the client receives $2b+1$ partial signatures or after $\eta = (1-\rho)(m+\tau-\Lambda)$ time units elapsed since the getTime invocation. In fact, a client might not receive soon a server reply because of process crash, or longer message delay. Different strategies can be employed if the client does not receive immediately $2b+1$ partial signatures: the client can re–contact the remaining servers in Q (Fig 2:15), or attempt to produce σ_x from at least $b+1$ partial signatures to improve the response time, and if successful return T while contacting in the background the remaining servers to guarantee a quorum of $2b+1$ acknowledgments.

C. Correctness analysis of the RSTS

We say that timestamp T_x associate to x and issued by the RSTS, 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 ϵ the real time at which the client invoked $getTime$. If we want to resolve disputes in case of key revocation, we have to strengthen this condition and say that T_x is *authentic* if *at least* $b+1$ correct servers have computed the partial signature.

Theorem 2: (Correctness and data integrity) If the RSTS secret key has not been compromised, then any timestamp T_x issued by the RSTS and equipped 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 RSTS private key was revoked at time T_{rev}, timestamps issued before $T_{rev} - \varphi$ remain valid.

Proof: If the RSTS secret key has not been compromised, a timestamp T_x endowed with a valid signature σ_x, is authentic because of the security of the underlying threshold signature scheme [7]. Since at least $b+1$ correct servers validate T_x, it approximates the getTime invocation time (we choose this option) with a constant error. Therefore, a malicious client cannot back– or post–date it by more than $M_x - 2m + 2\Lambda$ time units. Moreover, the client request with message digest x cannot be tampered by a computationally–bounded adversary. Any timestamp issued before the key revocation time can be verified by means of server logs, since at most b servers can be compromised within a time window. ■

Theorem 3: (Availability) The RSTS system is always available if the network eventually stabilize.

Proof: Each remote call in $tstamp$ completes because of the quorum availability, and since at most b servers can be faulty during a time window. In fact, each remote quorum call returns despite b failures in the quorum. The client succeeds in sending a sign request to a quorum of $2b+1$ servers because the difficulty of the correct puzzles is bounded. Therefore, $tstamp$ terminates if the network eventually stabilize, since the client requires only $2b+1$ partial signatures from a quorum set Q containing at least $3b$ servers, and because of the robustness of the underlying threshold signature scheme that terminates despite half failures. Clearly the verification procedure is always available since it is local. ■

D. Performance analysis

	Tree–linking TSS	RSTS
Comp. cost (server)	$O(N \lg N)$	C_P
Comp. cost (client)	$O(\lg N)$	$O(q \lg q) + C_V$
Comm. cost	$O(n \lg N)$	$O(q)$

Computational cost of RSTS. The computational cost of time–stamping is dominated by the computation of the partial signature σ_j, and depends on the threshold scheme used (e.g., in Boldyreva's scheme [7] $\sigma_j = \langle x, T_x, M_x\rangle^{s_j}$ with s_j secret share). In normal conditions, the client cost is given by $C_T + sC_{PV} + C_S$ where $C_T = O(q \lg q)$ is the cost of computing T_x and q the quorum size, sC_{PV} is the cost of verifying s partial signatures with $b+1 \leq s \leq 2b+1$, and C_S is the cost of producing signature σ_x verifiable by the RSTS public key. In case of high system load the client cost is increased by the cost of solving puzzles. We show in Section VI that a low–powered client can delegate a proxy (not necessary trusted) to verify the partial signatures and assemble them, thus reducing the cost to $C_T + C_V$, where C_V is the cost of verifying σ_x.

Communication cost of RSTS. Notice that the size of T_x and σ_x is constant, and that the size of the signatures produced by applying [7] is much shorter than usual signatures (i.e. 160 bits), thus improving the network bandwidth. In addition, servers do not communicate among them during time–stamping, and the client communicates with a *subset* of servers: a fraction of $\frac{3}{4}$ servers is involved during the getTime phase, and $\frac{1}{2}$ during the sign phase.

Performance comparison with tree–like linking TSS. The cost of time–stamping grows with the number N of requests submitted at a round, since it is given by the cost of computing the Merkle hash tree, and the authenticated path (timestamp) for each request. In case of a complete tree, it is $O(c\, N \lg N)$ with c (constant) cost of computing the hash function. This makes the TSS more vulnerable to DoS attacks. In addition, the server has to periodically publish the round values, and this is a costly procedure. The cost of verifying a timestamp at the client side is given by the cost of rebuilding the tree branch of the hash Merkle tree, that is $O(c \lg N)$. Note that in some TSS, the client has to interact also with an *on–line server* to verify a timestamp. In any cases, it has to wait at least for the completion of the round. Since the size of the timestamp is $O(d \log N)$ with d size of the message digest, the communication complexity of an intrusion–tolerant TSS based on tree–like linking scheme with n servers [18] is $O(nd \log N)$.

E. RSTS features

We motivate here those RSTS features briefly mentioned in the Introduction following the same order.

Robustness. We proved in Section V-C that the RSTS system is intrusion–tolerant and always available if the network eventually stabilize. It does not require a trusted dealer or/and a public storage medium. In fact, the accountability of the system is based on the assumption that no more than b servers are compromised within a time window. Our scheme makes both servers and clients less vulnerable to DoS attacks with respect to linking schemes as discussed in Section V-D for not grouping requests into *rounds*, and also for considering unbounded delays and using puzzle techniques as discussed in Section V-A. Clearly, brute force DoS attacks are still possible. In case the RSTS private key is revoked, the RSTS system resolves disputes and guarantees the validity of timestamps issued before the revocation time. It is important to note that the order imposed by RSTS on the client requests is a *refinement* of client invocation/response time for RSTS. This reduces the *degree of concurrency* among requests, thus providing more accurate time–stamps and reducing the degree of freedom of an adversary. As mentioned in the Introduction, linking schemes seem not to be suitable for intrusion–tolerant TSS because of their high complexity [18].

System performance. Our scheme is more scalable to high volume of requests then linking schemes since both time–stamping and verification costs are constant and timestamps are fixed–size, as shown in Section V-D. In addition, the use of quorum systems improve the system load balancing and its scalability: only *half of the servers* are asked to compute a partial signature. Moreover, in our scheme the client latency depends mainly on the transmission delay which is usually few order less than the duration round of linking schemes. Clearly, the scalability of RSTS has to be also evaluated by experimental results. However, because of the simplicity of the RSTS and its dependance on the package used for threshold signature, it does not seem much instructive to run experiments on its simple implementation, but rather on a more interesting application (see Section VI), as we plan to do. Also, currently it is not possible to compare the response time of the RSTS with an intrusion–tolerant linking TSS (see Section II).

Fine–granularity and flexibility. Our RSTS produces fine–grained real–time timestamps, which are more expressive and easy to compare across different system. As shown before, RSTS is highly flexible and tunable to the application needs.

One–phase RSTS. The getTime phase can be skipped if the message delays are bounded (i.e. in a LAN), and client clocks are securely synchronized to a reference source. In fact, the client can set timestamp T to its clock value, and servers can validate T since message delays are bounded.

VI. RSTS APPLICATIONS

We discuss here some applications involving very large systems, and requiring fine granularity and good response time, for which RSTS is particularly suitable.

E–stock market. Fine granularity, timeliness, and high scalability play an important role in stock markets such as the NASDAQ, the fastest growing electronic stock exchange with transactions conducted over computer networks across an international area. A stock market transaction involves investors (buyers or sellers), and the stock exchange: sellers/buyers wish to sell/buy shares of a certain company, and the stock exchange ensures that the transactions are committed in a proper and timely manner. The RSTS suits well buy/sell transactions of the investors, especially *market orders* where stocks are bought/sold at the *current price*. It also takes into account transmission delays. More generally, RSTS is applicable to any financial service that provides quoting services and real time snapshots of the market.

E–auctions. A sealed–bid auction is an electronic system in which secret bids are issued for an advertised item, and once the bidding period closes, the bids are opened and the winner is determined according to some publicly known rule. It is used in the sale of artwork or real estate, or in auctioning of government procurement contracts, or by Internet services such as EBay. Note that time–stamping plays a crucial role not only during the *bidding period* to guarantee the correctness of the bidders, but also to protect the auction system from misbehavior of insiders in charge of executing and overseeing the auction (i.e. to avoid manipulations of the closing time). Our RSTS suits well e–auctions and applications where the users compete for a resource, for its fine–granularity, good response time, and for being non–repudiated.

Mobile e–commerce. Mobile e–payments and mobile Internet services are becoming more popular. Clearly, time–stamping is crucial in any financial transaction and e–payment. The RSTS is particularly suitable for mobile e–commerce because it takes into account *energy conservation* and *low bandwidth*. In fact, since $2b+1$ partial signatures are sufficient to produce a correct digital signature, a low–powered client might delegate a proxy (not necessary trusted) to assemble the partial signatures. In a cellular network, the *base station* could act as a proxy and assemble the partial signatures on behalf of the mobile node. In this way, the client has to verify the correctness of the digital signature based on its certified RSTS public key.

Information aggregation. RSTS could be used by news agencies (i.e., Reuters, government agencies), or press/newscaster centers that gather information from different sources, and require *fresh* and *reliable* information (i.e., in case of breaking news). RSTS can be also applied to provide financial information to private customers equipped with RSS alerts.

VII. Conclusion

We have proposed and analyzed an intrusion–tolerant TSS based on a novel and robust time service. In contrast with previous solutions based on linking schemes, it has fine granularity, better response time, and it is highly scalable. Our RSTS can have a noticeable impact on those applications, both in wired and wireless networks, where linking schemes seem unsuitable. We have discussed some of these applications, such as e–stock market, e–auctions, mobile e–commerce, information aggregation, and quoting services.

References

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

[2] 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, pp 402-415, Feb 2001.

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

[4] 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.

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

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

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

[8] 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.

[9] A. Buldas, H. Lipmaa, B. Schoenmakers, *Optimally efficient accountable time-stamping.* In Proc. 3rd Intl. Work. on Practice and Theory in Public Key Cryptography, pp. 293-305, 2000.

[10] C. Cachin, K. Kursawe, A. Lysyanskaya, R. Strobl *Asynchronous verifiable secret sharing and proactive cryptosystems.* In Proc. of the 9th Conf. on Computer and Communications Security, pp. 88-97, Nov 2002.

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

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

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

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

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

[16] A. Juels, J. Brainard *Client puzzles: a cryptographic countermeasure against connection depletion attacks.* In Proc. of Networks and Distributed Security Systems, pp. 151-165, 1999.

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

[18] P. Maniatis, T.J. Giuli, M. Baker, *Building Trusted Distributed Services Across Administrative Domains*, Tech. Rep. cs.DC/0106058, http://www.arxiv.org/abs/cs.DC/0106058, 2001.

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

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

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

[22] 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, pp.178-183, 1999.

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

[24] B. Preneel, B. Van Rompay, J.-J. Quisquater, H.Massias, J. Serret Avila *Design of a timestamping system.* WP3 Tech. Rep., 1998.

[25] K. Schossmaier, B. Weiss. *An algorithm for fault–tolerant clock state&rate synchronization.* Proc. of the 18th IEEE Symp. on Reliable Distributed Systems, pp. 36-47, Oct 1999.

[26] Surety Inc. http://www.surety.com

[27] D. Tulone *How Efficiently and Accurately Can a Process Get the Reference Time?* Brief Announcement Intl. Symp. of Distributed Computing, pp. 25-32, October 2003.

[28] 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.

[29] L. Zhou, F. B. Schneider, R. Van Renesse, *COCA: A secure distributed online certification authority*, ACM Transactions on Computer Systems (TOCS), Volume 20, Issue 4, 329 - 368, 2002.