Distributed Timestamping with Smart Cards using Efficient Overlay Routing

Guenther Starnberger, Lorenz Froihofer and Karl M. Goeschka
Vienna University of Technology
Distributed Systems Group (DSG), Information Systems Institute
Argentinierstrasse 8/184-1
1040 Vienna, Austria
{*guenther.starnberger, lorenz.froihofer, karl.goeschka*}*@tuwien.ac.at*

Abstract

A secure digital timestamp can assert the existence of a particular document at a specific point in time. While centralized timestamp servers show dependability drawbacks with respect to node or link failures, e.g., outages of the Internet connectivity, distributed timestamping protocols don't have these drawbacks, but are nowadays rarely seen in practice because of the distributed trust requirements and applicability issues.

In this paper, we contribute with: (i) A distributed timestamp protocol addressing practical applicability issues with an efficient overlay routing architecture able to minimize the effects of node churn and connection establishment delays, at the cost of higher impacts of hop-to-hop latencies, (ii) Smart card integration to introduce a distributed web of trust and hence increase the security of applied timestamps, and (iii) An evaluation using a prototype implementation and network simulation that shows the performance gains of our protocol in comparison to the state-of-the-art.

The properties of our protocol allow for its application in scenarios where distributed timestamping protocols have not been an option so far, for example, because of mutually distrusting users. Furthermore, while many existing distributed timestamping protocols are only theoretically evaluated, we show the feasibility of our protocol with a prototype implementation.

1. Introduction

Timestamping protocols allow to certify that a particular document existed at a particular point in time. In practice, central timestamping servers operated by trusted third parties are typically used. While such central timestamp servers work fine in application scenarios where availability is only secondary, they exhibit a lower dependability than distributed timestamping approaches as server or network outages or overload can lead to a loss of service or to inaccurate timestamps.

While distributed timestamping protocols are a well researched topic, centralized protocols are primarily used in practice. Security issues in today's distributed protocols are one of the reasons, as each participant needs to trust that a high percentage of other participants will cooperate. In addition, communication between nodes is a problem. Usually, distributed timestamping protocols are built on the assumption that each node can directly request a timestamp from every other node. However, NAT (Network Address Translation) traversal is often required for communication between nodes on the Internet, leading to increased connection setup times, and thus decreased timestamp accuracy. In addition, some of the nodes might not be active due to node churn. In a traditional distributed timestamping protocol such inactive nodes will only be detected *after* trying to send a request to these nodes, which can lead to inaccurate timestamps.

In this paper we present our distributed timestamping protocol. To increase the trustworthiness of the applied timestamps, we augment user's terminals with trusted smart cards. Untrusted terminals are used for network communication and as interface to the user, while smart cards are responsible for timestamping, cryptographic aspects, and application level routing decisions. Thus, even an attacker with full control over a user's terminal has only limited options to attack our protocol. In addition, we provide an efficient overlay routing protocol, which trades simplicity in connection setup and maintenance for a lower latency in timestamp transmission. While pre-establishment of outgoing connections leads to higher bandwidth and computational requirements, it allows to efficiently route requests along pre-existing connections, without connection establishment or node lookup delays. However, this comes at the cost of a higher impact of hop-to-hop latencies, as mes-

sages now have to be routed along several hops. Finally, we evaluate our approach with a prototype implementation and simulation studies.

First, Section 2 discusses our timestamping protocol with a focus on our overlay routing approach and efficient connection establishment. Based on this, Section 3 continues with the evaluation using our prototype implementation and a protocol simulation using the PeerSim [12] network simulator. Afterwards, Section 4 discusses related work. Finally, Section 5 concludes the paper with an outlook on future work.

2. Timestamping protocol

This section discusses our distributed timestamping protocol that enables accurate and secure distributed timestamping in real world application scenarios. In comparison to the state-of-the-art, our protocol allows for minimum latency when deployed on the public Internet. In addition, we increase the security by restricting which nodes are allowed to timestamp particular messages.

In comparison to traditional "k among n" based protocols [3]—where k timestamps are obtained from a set of n nodes—our protocol benefits from the following characteristics:

- **Deterministic calculations without global state**

  We deterministically assign nodes responsible to timestamp a particular document allowing to subsequently verify if a given document has been timestamped by the correct nodes. Only an agreement on the *approximate* size of the network is required, instead of a global agreement on the set n.

- **Small latencies between hosts**

  Our protocol takes real-world network conditions into account and is optimized for the fact that connection establishment can be expensive due to NAT (Network Address Translation) traversal. As we pre-establish connections, connection setup times do not influence the accuracy of timestamping. Furthermore, pre-established connections allow to use keep-alive messages as simple failure detector to detect unavailable nodes.

- **Anonymity**

  While full anonymity is not a goal of our protocol, we strive for semi-anonymity to make it more *difficult* to assert the originator of a timestamping request. This prevents individual participants from, e.g., deliberately delaying requests of certain other participants.

- **Smart card based security**

  Existing distributed timestamping protocols typically assume semi-trusted nodes for the application of each other's timestamps, which is often not a reasonable assumption. Therefore, we decrease trust requirements by combining the user's terminal with a smart card responsible for security-critical functionality.

To provide these benefits, we take the following trade-offs in comparison with a traditional "k among n" approach into account:

- **Increased hop count**

  In comparison to "k among n" approaches, which do not route messages along multiple hops, we increase the overall length of the routing path and thereby the influence of latency on the timestamp accuracy. However, we keep hop-to-hop latencies low by pre-establishing connections. Furthermore, nodes can detect slow neighbors with our keep-alive mechanism, allowing to replace them with faster nodes.

- **Higher overhead**

  There is a higher total overhead due to the pre-established connections and the DHT (Distributed Hash Table). However, this overhead does not affect the accuracy of timestamping requests and is a deliberate trade-off.

In the following subsections we first give a protocol overview, followed by an examination of our Node IDs and how values derived from these IDs are calculated. We then continue by discussing our network structure, message routing, and connection establishment techniques.

2.1. Protocol overview

In our protocol each node is represented by the user's computer under full control of the user, as well as the trusted smart card emitted by a trusted third party. While the main protocol tasks are executed by the user's terminal, the smart card is responsible for the timestamping itself. In addition, the smart card is also able to restrict routing decisions by the terminal. Apart from users, nodes can also be operated by trusted third parties to enhance the security of the protocol by providing more trustworthy timestamps.

Our protocol is implemented as overlay network, with timestamp routing performed on this overlay network. The network is partitioned into sets of address ranges, where each node belongs to exactly one particular address range. To mitigate node churn, address ranges can be reassigned (see Section 2.2) when the total number of nodes changes, so that the number of nodes within an address range is relatively constant. Each outgoing connection of a node has a unique index number and is established to one node within a deterministically identified address range. The respective address range to which a connection is established is determined by factors such as the index number of the outgoing

connection, the local node's own address range, and the current time. Thus, a node is restricted in the choice of outgoing connections. Open outgoing connections are regularly recycled to increase diversity in routing paths.

When a timestamp request arrives, the hash of the request determines the routing path, which corresponds to the index numbers at the respective nodes. Thus, timestamp requests can be efficiently routed along the already pre-established connections, without needing to establish new connections. Due to the deterministic routing decisions, external nodes can verify if timestamp requests have been routed along the correct path.

2.2. Node IDs

A fundamental security feature of our protocol is to restrict the nodes that are allowed to apply a timestamp for a given document. Typically, such restrictions are implemented with "k among n" schemes [7], where k nodes are required that can be chosen from a set of n nodes. In our case we determine the set n from factors such as the hash of the local node and the current time. We cannot directly use IP addresses in this set, as these addresses are sparse and unevenly assigned, which makes it difficult to define a function that returns an address range with a predefined number of active nodes. Definition of such a function would only be possible with the knowledge of all active node's IDs.

Instead, we use an overlay network and perform the operations on overlay node IDs, which are assigned in a circular address space. IDs are derived by hashing the user's public key and are thus randomly distributed. As the public key pair of a node is generated directly on the smart-card while the card is still under physical control of the trusted third party, the user cannot use brute-force attacks to force a node ID in a chosen ID range.

Furthermore, each node ID is member of exactly one address range, with the whole address space partitioned into r different ranges. While for each particular point in time there is a deterministic mapping between node IDs and address ranges, r can change over time, thereby affecting the mapping of node IDs to address ranges. In practice, r will be adapted to the total number of nodes, leading to a relatively stable amount of active nodes within each particular address range. Depending on the concrete application scenarios there are different approaches to estimate the total amount of nodes. For example, if the Kademlia DHT [10] is used, trusted nodes can estimate the amount of total nodes by looking at the Kademlia bucket utilization and then cryptographically sign and broadcast this information via the overlay.

2.3. Derived values

In our protocol we use different *derived* values to determine values such as address ranges. These derived values are used in place of *random* values (e.g., to determine the routing path). Because of the deterministic calculation, dishonest nodes cannot choose values at will, as external observers can verify if the correct rules have been used to obtain the values. In this section we specify how we map these input values to their respective output values.

Derived values are the result of a hash functions with the source element from which a value is derived combined with an index used as the input. The index serves as distinguishing element when the same input element is used to produce multiple output elements. For example, to derive an address range for the fifth outgoing connection from the node's own address range we calculate $new\_range = hash(local\_range + index)$, where new_range identifies the resulting address range, $local\_range$ identifies the node's own address range, and $index$ is set to 5 to represent the fifth outgoing connection.

In related work sometimes pseudo-random number generators are used for a similar purpose in "k among n" schemes [7]. However, using a hash function allows for non-sequential access to the individual values, for example, if the smart card is used to verify the value of a particular outgoing connection. Furthermore, on smart cards the respective optimized native implementations can be used for hash functions, while the use of pseudo-random number generators would require access to the generator's seed which is often not possible.

2.4. Network structure

Our network is built as an overlay network between nodes. At each point in time each node has exactly c_o outgoing and approximately c_i incoming connections. When a node sends a timestamp request to an outgoing connection, the request arrives at the incoming connection of another node. While communication channels are bidirectional, requests only travel along outgoing connections, while replies travel in the reverse directions.

When a new connection is established, this step is split into two tasks: First, we use a mapping function that takes (i) the particular index of the outgoing connection, (ii) the node's own address range, and (iii) the time as input. The output of of this function is the address range to which the connection will be established. Afterwards, the terminal looks up one of the nodes within this address range and establishes an outgoing connection. For even better handling of node churn, redundant connections into the same address range can be established for immediate failover. This approach is explained in detail in Section 2.6.

Each of the outgoing connections is assigned an index from 0 to $c_o - 1$. The routing path is calculated by the source node and included within the message. For example, a source node may specify that a message should travel along the edges 1–2. The source node then relays the message to its outgoing connection with the index 1, where the

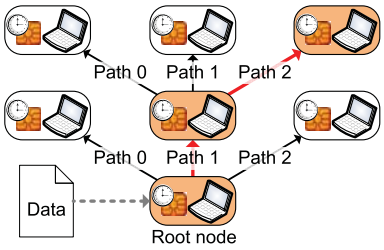

Figure 1. Path selection: Path 1-2

node receiving the message continues relaying the message to it's own outgoing connection with the index 2.

The exact lookup mechanism used by the protocol to find nodes within particular address ranges is not fixed and can vary across implementations. For example, if a DHT (Distributed Hash Table) such as Kademlia is used, a *range query* can be used to find suitable candidates, while in the case of JXTA [1] advertisements stored in a SRDI (Shared Resource Distributed Index) would be used.

2.5. Message routing

This section describes how a message for a particular timestamp is routed along the nodes. First, the node requesting a timestamp uses the hash of the document to calculate the routing path of the message. For the routing path each hop is iteratively calculated with the formula $hop_{next} = hash(document\_hash + hop\_cnt)$, where hop_cnt indicates the index of the particular hop. The routing path is then embedded within the message that is to be timestamped. Before sending a message, the originating node removes the first element from the routing path, and forwards the message on the outgoing connection that is identified by the element. Each node receiving the message in sequence removes the first element from the routing path and forwards the message to the respective outgoing connection. Once the routing path in the message is empty the message is not forwarded any further. An example of this routing strategy is given in Figure 1 where the root node calculates the routing path "1-2".

To mitigate message loops where a message is relayed back to a node that has already received this message in a previous step, the first node originating the message validates that each address range is only traversed once. To do so, the node simulates the message path, as it can locally calculate to which address range an outgoing connection of another node will connect. If a potential loop is detected, because the message is relayed back to an address range from which it has previously been forwarded, the hop_cnt used in the formula above is incremented, thus shifting the hop_cnt of the affected element and all remaining elements by one. This algorithm is deterministic and only depends on the partitioning of the system into the different address ranges. Therefore, it allows to externally verify that the correct routing path was used. This verification depends on the cryptographically signed address range partitioning parameter r (Section 2.2) that is embedded within the timestamp.

When a node sends or relays a message, the message itself is generated and signed by the smart card. This step is required to prevent malicious nodes from *injecting* bogus messages into the network. Therefore, the message also needs to contain the concrete node ID of the next node, as otherwise the user's terminal would be able to inject a single timestamping message to multiple nodes within the same address range. The signatures applied by smart cards are only used on a hop-by-hop basis. Thus, removal of an element from the routing list does not affect signatures applied by a preceding hop.

When a node receives a message from a neighboring node it first verifies if the message has indeed been signed by its neighbor and if the existing timestamp(s) in the message are still current. Additionally, it verifies if the destination field of the message matches its own local node ID. Before forwarding the message, the node locally stores a timestamp for this message and the node ID from the node from which the message was received, but does not include this timestamp in the relayed message. When the last node on the routing list receives the message it includes a timestamp and relays the message back to the node from which it has been received. Each node then includes its previously stored timestamp and continues relaying the message in the reverse direction. When the message arrives at the initial originator of the request, the message contains the timestamps assigned by all the nodes along the routing path.

2.6. Connection establishment

For regular connection establishment we use an approach that continuously renews existing connections to provide diversity in the possible routing paths. With each step only one single connection is renewed, thereby minimizing the impact on the network setup. In addition, we adapt our routing approach to prevent routing paths over outgoing connections that will be renewed soon, as this would prevent the sender from uniquely determining the routing path, which is required for loop mitigation.

For example, consider that each node has 10 outgoing connections and that within a 10 minute interval each outgoing connection should reconnect to a new address range. When updating connections continuously, we can reconnect a single connection once each minute. To avoid any routing problems due to reconnecting nodes, a node can establish a connection to the new address range in advance, and then just switch connections when the interval of the current connection ends. If the node is still waiting for a reply message that was initially forwarded over the old connection, it can additionally leave the old connection open until the reply message has been received.

While the renewal of a connection does not directly affect messages in transit, this renewal can lead to routing loops, as the original mitigation strategy presented in Section 2.5 only considers the network configuration at the time of message creation. To mitigate such loops, we adapt the routing mechanism not to use connections that will be renewed *soon*, e.g., to never use the next connection that will be renewed. In such a case a similar strategy as for loop mitigation is used: hop_cnt is incremented, causing the algorithm to calculate a different route. Of course, such an approach requires that all nodes have a synchronized time. This is not a problem in our case, as such a synchronized time is already required for timestamping purposes.

To account for node churn, e.g., active nodes that change their status to inactive, we use regular keep-alives to detect stale connections. In this case a node will re-establish the connection to another node within the same address range. Such a connection establishment cannot lead to a routing loop, as the address range of the old connection and the new connection is the same. While node churn leads to overhead due to higher reconnect rates, it does not affect latency, as messages are routed only along pre-existing connections.

3. Evaluation

The evaluation of our protocol was conducted in two distinct steps. First, we used a prototype implementation to show the feasibility of our approach and for performance tests with low amounts of nodes. This implementation is based on JXTA [1]. The *Netim* Linux kernel module is used to accurately simulate network latency. Second, we used the PeerSim [12] network simulator to examine the effects of latency and churn. Significant parts of the prototype implementation's code have been reused in the simulation, ensuring that the simulation's behavior matches the prototype. We also validated if the results of the simulation match the results obtained in the prototype implementation.

The Figure 2 shows a comparison between the average latency in our overlay routing approach and an alternative approach that requires connection establishment for forwarding. To model node-churn, we assume a node-failure rate of 5%. If a message is transmitted to a *failed* node, it cannot be processed, but is re-transmitted to another node.

In the results we observe that the latency without an overlay network is roughly twice as high as the latency in our overlay approach. While in time synchronization protocols latency can be mitigated by assuming synchronous network delays, this is not possible in timestamping. Therefore, the latency directly affects the timestamps assigned to individual documents. While in some application scenarios timeliness of timestamps is only secondary—e.g., when only an approximate date is required—in other application scenarios such as deadline-based online auctions timeliness is a fundamental property [5].

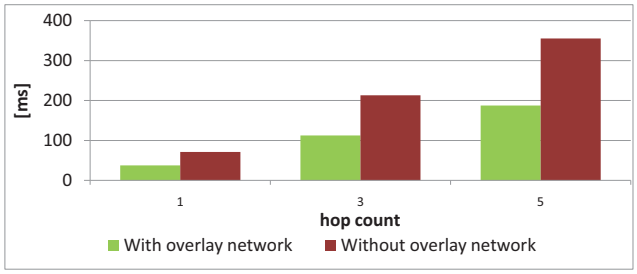

Figure 2. Hop latency

In addition, our simulation has shown that the results also depend on the assumed application scenario: In application scenarios without node churn where UDP (User Datagram Protocol) communication without NAT traversal can be used between nodes, the differences between a traditional approach and our approach will be smaller, while in application scenarios with considerable node churn the differences will be higher. For example, if we decrease the node failure rate from 5% to 0% in the scenario *without* overlay network, this decreases the average delay at the fifth hop from 355 ms to 338 ms, which is only a marginal improvement. On the other hand, if we increase the node failure rate to 30% this increases the average delay to 482 ms. However, even in the first example with lower differences our protocol allows more efficient routing without requiring nodes to know all other nodes, while a traditional protocol would require nodes to store information about other nodes, if nodes are to be selected according to the document's hash.

To examine the effects of node churn we simulated our protocol with different median session times and tracked the number of failed request. A request fails, if a node sending a request does not receive the corresponding response, because an intermediate node cannot forward the request as no outgoing connections to the respective node range are available. For our tests we assumed a hop count of 5 hops as well as 3 redundant connections into each node range. The results in Figure 3 show that request failure rates are an issue for median session times below approximately 40 minutes. While failure rates are high for very low session times, they decrease to less than 5% for median session times of 20 minutes or more and to less than 1% for median session times of 40 minutes or more.

4. Related work

In this section we examine publications related to our distributed timestamping protocol. Distributed timestamping approaches are discussed in [3, 6–9, 13, 14], but only Nishikawa and Matsuoka [13] seem to actually have implemented their approach and also consider lower level issues such as the protocol for timestamp transmission. As the protocol requires individual TCP connections, it leads to higher latency than our overlay approach. [3] presents a "k

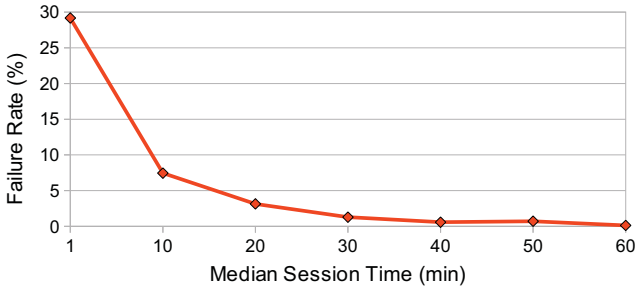

Figure 3. Median Session Time vs. Failure Rate

among n" timestamping scheme with redundancy to mitigate failed servers. Unlike our approach it requires synchronization between servers and does not mitigate delays during connection establishment. [14] suggests a timestamping scheme with better scalability than classical treebased linking scheme. In their work client latency depends mainly on the transmission delay which can be reduced with our approach. While [6] uses onion routing over an overlay network for timestamping, it differs from our approach as the path of the onion route is not dependent on the input document and hence does not allow to specify and verify whether correct timestamping nodes were chosen.

5. Conclusion

In this paper we presented our protocol that enables secure distributed timestamping in a smart card based environment. The initial motivation for the protocol was to provide a secure low-latency mechanism for document-dependent routing and to decrease information required at individual nodes to facilitate implementation on smart cards. As a solution approach we have decoupled the setup of the overlay network from the underlying routing mechanism. This allows for variable routing paths, while at the same time allowing the system to route all messages along pre-existing connections, which leads to a higher overhead for connection setup and connection maintenance. While hop-to-hop latencies are amplified by our overlay approach, this can be mitigated with techniques such as Pharos [4] that allow to find nodes with low latencies. Our evaluation has shown that routing messages along existing connections on an overlay network provides for considerably better performance than state-of-the-art protocols.

While in some application scenarios timeliness of timestamping is not a considerable factor, other application scenarios benefit from performance increases in the milliseconds range. An example are online auctions [5] and distributed computer games traditionally using the lockstep protocol [2], where better performance can be provided if an accurate timestamping mechanism is available [11].

6. Acknowledgments

The authors would like to thank Markus Jung for the implementation of the work described in this paper. This work has been partially funded by the Austrian Federal Ministry of Transport, Innovation and Technology under the FIT-IT project TRADE (Trustworthy Adaptive Quality Balancing through Temporal Decoupling, contract 816143, `http://www.dedisys.org/trade/`).

7. References

[1] JXTA Project. `https://jxta.dev.java.net/`. (Access date: 29 June, 2010).

[2] N. E. Baughman, M. Liberatore, and B. N. Levine. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Trans. Netw.*, 15(1):1–13, 2007.

[3] A. Bonnecaze, P. Liardet, A. Gabillon, and K. Blibech. Secure time-stamping schemes: A distributed point of view. In *Annals of Telecommunications*, volume 61, pages 662–681. Institut Télécom, 2006.

[4] Y. Chen, Y. Xiong, X. Shi, B. Deng, and X. Li. Pharos: A decentralized and hierarchical network coordinate system for internet distance prediction. In *GLOBECOM*, pages 421–426. IEEE, 2007.

[5] L. Froihofer and K. M. Goeschka. Balancing of dependability and security in online auctions. In *Dependable Systems and Networks, 2008. DSN '08. Int. Conf. on*, 2008.

[6] M. Gogolewski, M. Kutyłowski, and T. Łuczak. Distributed Time-Stamping with Boomerang Onions. *Tatra Mountains Mathematical Publications*, 33:31–40, 2006.

[7] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, Jan. 1991.

[8] D.-P. Le, A. Bonnecaze, and A. Gabillon. A secure round-based timestamping scheme with absolute timestamps (short paper). In *ICISS*, volume 5352 of *Lecture Notes in Comp. Sci.*, pages 116–123. Springer, 2008.

[9] H. Massias, X. S. Avila, and J.-J. Quisquater. Timestamps: Main issues on their use and implementation. In *WETICE*, pages 178–183. IEEE Computer Society, 1999.

[10] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *IPTPS*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2002.

[11] S. Mogaki, M. Kamada, T. Yonekura, S. Okamoto, Y. Ohtaki, and M. B. I. Reaz. Time-stamp service makes real-time gaming cheat-free. In *NETGAMES*, pages 135–138. ACM, 2007.

[12] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *Peer-to-Peer Comp.*, pages 99–100. IEEE, 2009.

[13] T. Nishikawa and S. Matsuoka. Time-stamping authority grid. In *CCGRID*, pages 98–105. IEEE Comp. Soc., 2008.

[14] D. Tulone. A scalable and intrusion-tolerant digital time-stamping system. In *Communications, 2006. ICC '06. IEEE Int. Conf. on*, volume 5, pages 2357 –2363, 11-15 2006.