Keyless Signatures' Infrastructure: How to Build Global Distributed Hash-Trees

Ahto Buldas[1,2], Andres Kroonmaa[1], and Risto Laanoja[1,2]

[1] Guardtime AS, Tammsaare tee 60, 11316 Tallinn, Estonia.
[2] Tallinn University of Technology, Raja 15, 12618 Tallinn, Estonia.

Abstract. Keyless Signatures Infrastructure (KSI) is a globally distributed system for providing time-stamping and server-supported digital signature services. Global per-second hash trees are created and their root hash values published. We discuss some service quality issues that arise in practical implementation of the service and present solutions for avoiding single points of failure and guaranteeing a service with reasonable and stable delay. Guardtime AS has been operating a KSI Infrastructure for 5 years. We summarize how the KSI Infrastructure is built—and the lessons learned during the operational period of the service.

1 Introduction

Keyless signatures are an alternative solution to traditional PKI signatures. The word *keyless* does not mean that no cryptographic keys are used during the signature creation. Keys are still necessary for authentication, but *the signatures can be reliably verified without assuming continued secrecy of the keys*. Keyless signatures are not vulnerable to key compromise and thus provide a solution to the problem of long-term validity of digital signatures. The traditional PKI signatures may be protected by timestamps, but as long as the time-stamping technology itself is PKI-based, the problem of key compromise is still not solved.

In a keyless signature system, the functions of signer identification—and of evidence integrity protection—are separated and delegated to cryptographic tools suitable for those functions. For example, signer identification may still be done by using asymmetric cryptography but the integrity of the signature is protected by using keyless cryptography—the so-called *one-way collision-free hash functions*, which are public standard transformations that do not involve any secret keys.

Keyless signatures are implemented in practice as *multi-signatures*, i.e. many documents are signed at a time. The signing process involves the steps of:

1. *Hashing*: The documents to be signed are hashed and the hash values are used to represent the documents in the rest of the process.
2. *Aggregation*: A global temporary per-round hash tree is created to represent all documents signed during a round. The duration of rounds may vary; it is fixed to one second in the described implementation.
3. *Publication*: The top[3] hash values of the per-round aggregation trees are collected into a perpetual hash tree (so-called *hash calendar*) and the top hash value of that tree is published as a trust anchor.

To use such signatures in practice, one needs a suitable *Keyless Signatures' Infrastructure (KSI)*—analogous to PKI for traditional signature solutions. Such an infrastructure consists of a hierarchy of aggregation servers that, in co-operation, create the per-round global hash trees. First layer aggregation servers—*gateways*—are responsible for collecting requests directly from clients; every aggregation server receives requests from a set of lower level servers, hashes them together into a hash tree and sends the top hash value of the tree as a request to higher-level servers. The server then waits for the response from a higher-level server and—by combining the received response with suitable hash chains from its own hash tree—creates and delivers responses for each lower-level server.

In this paper, we discuss some service quality and availability issues that may arise and describe solutions. The implementation avoids single points of failure and guarantees reasonably low and stable service latency.

[3] For intuitivity, by *top* we denote root of the 'upside down' tree-shaped data structure.

Guardtime AS has been operating a KSI Infrastructure for 5 years—long enough to draw some conclusions about availability and scalability; practical lessons were learned during the operational phase. This paper describes KSI Infrastructure main components and operational principles with a brief overview of design decisions that minimize security risks.

2 Hash Trees and Hash Calendars

Hash Trees: Hash-tree aggregation technique was first proposed by Merkle [6] and first used for digital time-stamping by Haber et al [5]. Hash-tree time-stamping uses a one-way hash function to convert a list of documents into a fixed length digest that is associated with time. User sends a hash of a document to the service and receives a *signature token*—proof that the data existed at the given time and that the request was received through a specific access point. All received requests are *aggregated* together into a large hash tree; and the top of the tree is fixed and retained for each second (Fig. 1). Signature tokens contain data for reconstructing a path through the hash tree—starting from a signed hash value (a leaf) to the top hash value. For example, to verify a token y in the place of x_2 (Fig. 1), we first concatenate y with x_1 (retained as a part of the signature token) and compute a hash value $y_2 = h(x_1 \mid y)$ that is used as the input of the next hash step, until we reach the top hash value, i.e. $y_3 = h(y_2 \mid x_{34})$ in the example case. If $y_3 = x_{top}$ then it is safe to believe that y was in the original hash tree.

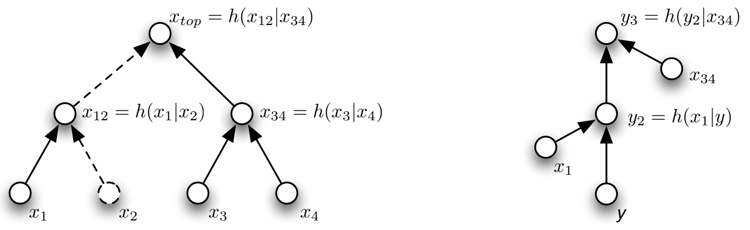

Fig. 1. Computation of a hash tree (left), and verification of y at the position of x_2.

Hash Calendar: Top hash values for each second are *linked* together in a globally unique hash tree called a *hash calendar*—so that new leaves are added only to one side of the tree. Time value is encoded as the *shape* of the calendar—the modification of which would be evident to other users. The top hash of the calendar is periodically published in widely witnessed media.

There is a deterministic algorithm to compute top of the linking hash tree, giving a distinct top level hash value at each second. Also there is an algorithm to extract time value from the shape of the linking hash tree for each second, giving a hard-to-modify time value for each issued token.

Security Against Back-Dating: malicious servers can not add new requests to already-published hash trees. It has been shown [4, 3, 2] the scheme is indeed secure if the hash function is secure in ordinary terms (one-wayness, collision-resistance, etc.) and the aggregation tree is of limited size.

3 Service Architecture

3.1 Design Principles

The system provides a global scale time-stamping and server-assisted digital signature service; it is *provably secure* and has minimal trust requirements. Secondary goals are easy auditability and accreditation, efficiency and robustness.

Underlying data structures guarantee that it is not possible to issue fake, backdated or otherwise misleading signature tokens—even where rogue client and rogue service provider collaborate. Committing into globally unique and public Hash Calendar makes tampering with the system, especially with the clock value, highly visible to all users. The system security does not depend on the long-term secrecy of the private keys as it is not possible to prove that the keys were not actually leaked. Underlying cryptographic primitives may be easily changed, e.g. in case of apparent weakening of the algorithms. There may be occassions when the infrastructure must be stopped—if the system integrity or clock accuracy is in doubt. The signature token itself is independently verifiable by third parties using only public information and algorithms; verification must be possible even after the service provider ceases the operations.

In order to provide highly available service single points of failure are eliminated. The requirements on system reliability are different: a globally unique core cluster must be operated by the best trust authority practices, but the service delivery network may use commodity virtual servers without much requirements on operating environment, like a reliable "wall clock" or persistent storage. Privacy and confidentiality risks are minimal, because the infrastructure handles only aggregate hashes.

3.2 Hash Tree in Action

In spite of the conceptual simplicity of a Merkle hash tree, assembling a global-scale network of servers for building such a tree complex. Simple tree-shaped service architectures are undesirable because every node is a potential single point of failure. So, the aggregation tree must be built by redundant clusters of servers. We must also have redundancy in handling of the unique top of the tree, thus we face potentially different top hash values, cluster partitions, etc.

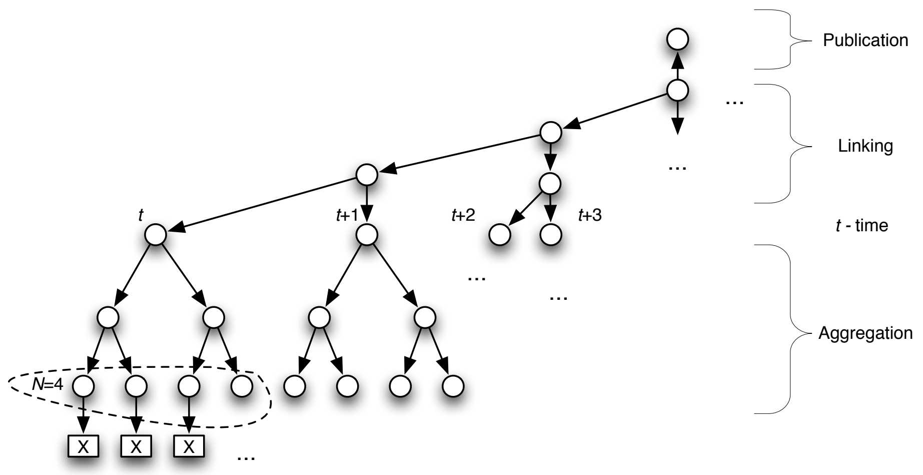

Fig. 2. Binary hash tree based time-stamping.

At each aggregation period the top of the aggregation tree is stored in a *calendar database*. In order to irrevocably commit to the ordered sequence of values in calendar database they are *linked* into a binary hash tree—so that the leaves are added to one side only—like the ticks on the time-line (see t at Fig. 2). This database is globally unique and public—open for all third parties for auditing, archiving or verification. Aggregation is handled by the distributed *aggregator* network. Linking, based on the contents of the calendar database, can be performed consistently and locally. The shape of the path from any leaf to the top of the linking tree encodes UTC time of aggregation period, when this leaf was produced. Tampering with the time-value of an issued signature token is as hard as rearranging the nodes of public, fixed-top Merkle tree.

There is deterministic algorithm to compute the top of the linking hash tree, giving a distinct top level hash value covering all leaves—at each aggregation period. This value is periodically fixed by *publishing* it in hard-to-modify and widely witnessed media. Currently, newspapers and popular micro-blogging platform are used monthly; there is also an electronically distributed and digitally signed publications list for automated use. It is possible to move to different publishing pattern to suit changing circumstances.

Fig. 2 shows how the system operates in rounds (t, $t+1$, $t+2$, $\ldots$), producing one aggregation tree per round. Capacity of the system (N) is determined by the maximum tree size $N = 2^l$, where l denotes the depth of the aggregation tree.

The limited size of the tree (N) is important for the provable security of the scheme, as referred in Section 2.

4 Infrastructure Overview

High-level system architecture is depicted in Fig. 3. Document hashing happens in the application that forms a signing (or time-stamping) request—using provided software tools. The request is sent to a *gateway*—the system component which delivers the service to end-users. The gateway performs initial aggregation of the requests received in its aggregation cycle—then sends its aggregate request to upstream aggregation cluster. Requests are aggregated through multiple layers of aggregator servers—and a globally unique top hash value is chosen by the core cluster. A response is sent back immediately through the aggregation layers.

Top hash values for each aggregation period are collected in the "calendar database" and distributed through the "calendar cache" layer to the extender service, usually co-located with the gateway host. Client applications use the extender service for the verification purposes.

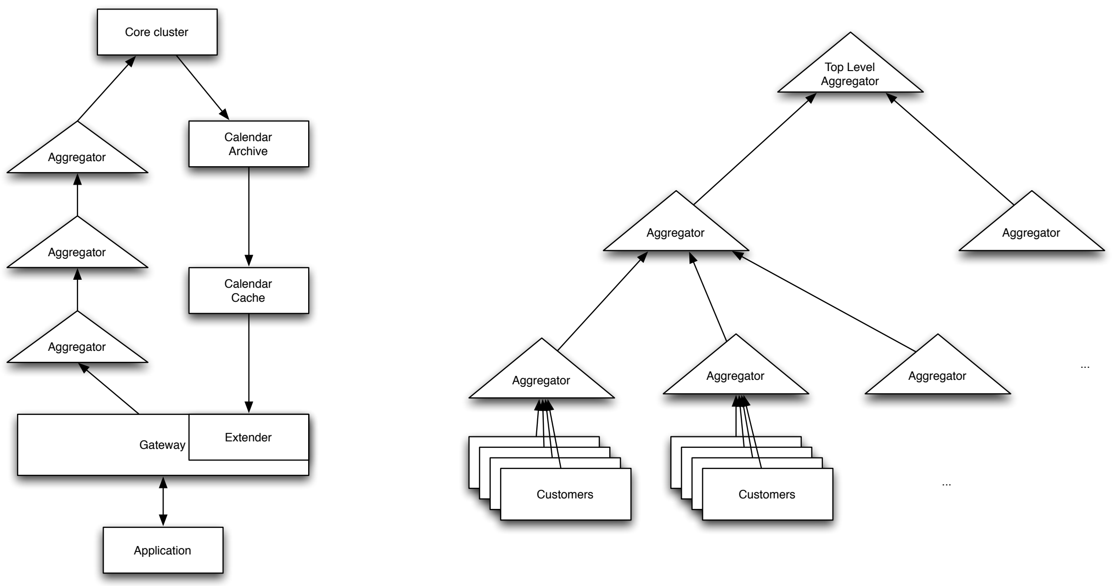

Fig. 3. High-level system architecture and the aggregation network.

4.1 Aggregation Network

An *aggregator* is a system component that builds hash trees from all incoming requests and passes top hash values to upstream system components. Aggregators work in rounds of equal duration. The requests received during a round are aggregated into the same hash tree. After receiving a response from an upstream component, an aggregator immediately delivers the response to all child aggregators together with hash paths of its own tree. Subsequent responses from upstream components for the same round are ignored and thus it is possible to run multiple aggregators in parallel for higher availability.

The aggregation tree is horizontally split in four layers, and an infrastructure is built so that the top layer is adjacent to the core cluster; two intermediate layers provide geographic scale. The bottom layer is bundled with gateways and hosted close to end users. Each aggregator labels its downstream clients by hashing their names into the hash trees. These forward secure labels form a hierarchical namespace used to identify distinct service end-points.

The aggregation tree scales easily; in order to double the system capacity we have to add only one hash value to signature tokens. Current hash-tree depth is fixed at 50 steps, giving a theoretical maximum capacity of $2^{50} \approx 10^{15}$ signatures per second. This initial configuration is believed to cover possible signature needs for the foreseeable future. Each gateway and aggregation server generates *constant* upstream network traffic which does not depend on load. This isolates the customers, does not leak information about the actual service usage and provides denial-of-service attack isolation. Service scales up linearly.

4.2 Core Cluster

The aggregation network is topped by a *core* cluster. Core is a distributed synchronized machine responsible for achieving consensus about the top value for each aggregation period, durably storing it in the Calendar database and returning it to the aggregation network. The core maintains accurate clock value, which is presented to customers as issuing time of each signature token. The calendar database is distributed using "dumb" caching layers to extender servers for signature token verification.

4.3 Gateway

The gateway works as a protocol adapter, accepting requests in application specific formats (RFC3161, OpenKSI) and forwarding them to the designated aggregator(s). In practice, the first level of aggregation happens at a gateway host—giving low and predictable communication bandwidth between the aggregators and gateway.

The gateways also host an *extender* (also called *verifier*) service—a signature verification assistant. Using its fresh copy of the calendar database, the extender service returns missing hash values necessary for building full hash chain from signed data to the latest published hash value. These missing hash values were not yet known at the signing moment. The hash chains created with the help of the extender are validated by client APIs. The token validity is decided at the application layer and gateway must not be treated as trusted party. Client applications may store tokens with full information for re-creation of the hash-chain by creating so-called 'extended' tokens.

5 Availability and Service Quality

5.1 Availability of the Aggregation Network

To increase the *availability* of the service, single points of failure must be avoided and we use redundancy everywhere in the system. Every aggregation server is replaced with a *geographically dispersed cluster* of aggregation servers that work in parallel, so that the lower-level server sends requests to every member of the cluster and uses the first received valid reply. If the availability coefficient of a single aggregation server is assumed to be 0.99 (i.e. approximate downtime is about 3.5 days per year), then a cluster with two servers has availability about 0.9999—assuming total independence of downtime events. Note that clusters can be enlarged and reconfigured without any downtime.

5.2 Core Availability

The uppermost aggregation steps are also performed by a cluster of redundant servers. It is necessary to ensure the uniqueness of the global hash calendar, thus there has to be some kind of agreement protocol between the servers of the uppermost cluster (called the *core*). Possible situations which must be handled include: (1) some requests may not reach all servers of the core, (2) the core serves may receive requests in different order and (3) some sites may be disconnected, shut down, maintained etc. In principle, it may be that all core servers will have different top hash values. A solution to the problem is that the servers try to propagate their top hash values to all core nodes, by using a multi-party protocol where by the end of each potentially overlapping round *majority of core servers have identical sets* of top hash values and the *servers with different sets will know that they do not belong to the majority.* Only the core servers that belong to the majority fill in values in calendar database based on the top hash values in the set they have agreed on, and answer their requests. The protocol will work whenever a quorum of servers can communicate with each other. Servers cut out of the core because of the connectivity or local hosting problems will recover automatically.

The average agreement time is 1.5 times the round-trip between (connected) core servers. This is achievable because we do not assume the malicious security model and do not need to solve the complete Byzantine Agreement problem. Every message is broadcasted using different routes in the full mesh connection model.

5.3 Optimizing the Service Quality

Below we describe how we optimized the latency and eliminated the 'long tail' of KSI service response time.

Simplified Approach The aggregation network is redundant—it has a cluster of m aggregators instead of one. Every aggregator has a certain aggregation period d (in time units). Large aggregation periods create large service delays because a request received in random time will be aggregated (i.e. the Merkle tree built, the top hash calculated and sent to the parent cluster) approximately after $d/2$ units of time. This means that every aggregator in the path from a client to the core-cluster adds $d/2$ time units of service delay.

If an aggregation round begins at 0 and ends at d, then a request that arrives at t (in $[0 \ldots d]$) will have service delay $d - t$, i.e. the larger t is, the smaller will be service delay. The requests that arrive later (just before the round is closed) have smaller service delays.

We adjust the round schedules of the aggregators in the same cluster so that the average delay of requests will be minimal; if we have two aggregators in the cluster both with round length d (in time units) and the round of the second aggregator begins at time $d/2$ (instead of 0), then (as every request is sent to both aggregators), the average service delay is $d/4$ instead of $d/2$. This is because the delay for a request received at t is now the following function $\delta(t) = \frac{d}{2}(1 + \lfloor 2t/d \rfloor) - t = \begin{cases} d/2 - t & \text{if } t \in [0 \ldots d/2] \\ d - t & \text{if } t \in [d/2 \ldots d] \end{cases}$ and the average value of this function in $[0 \ldots d]$ is $d/4$. In general, if we have m aggregators in the cluster, and the round of the i-th aggregator in the cluster begins at time i/m, then $\delta(t) = \frac{d}{m}(1 + \lfloor mt/d \rfloor) - t$ and the average delay is $\frac{d}{2m}$.

In general, if we have m aggregators in the cluster, and the round of the i-th aggregator in the cluster begins at time i/m, then the average service delay will be $d/(2m)$.

We reduce the service delay by interleaving the aggregation rounds in a cluster. The simplified approach is useful only if the delay is almost completely random, with large standard deviation comparable to the duration of the aggregation round. Such extreme conditions are very rare in practice.

Practical Approach A network delay between a child aggregator C and a parent aggregator P consists of several components:

– *Propagation delay* caused by the basic physics and depends on the length of wires between C and P. This delay cannot be eliminated.

– *Serialization delay* caused by global cloud of network routers that choose the paths in the network that are used to send data from C to P.
– *Jitter*. Mostly caused by varying utilization which creates processing queues and causes retransmissions.

All these component-delays create a probability distribution that is not uniform but a rather sharp bell-curve. If we know that 95 per cent of the requests (of C to P) have delays between $25 - 40$ms (milliseconds), then we can adjust the round schedules of C and P, so that their rounds (if they are of equal duration d) begin at t and $t + 40$ms, respectively. This means that 95 per cent of the requests sent by P to C have an additional delay of less than 40ms. In practice, the delay is much smaller than d/m, where m is the number of aggregators in a cluster and d is the aggregation period.

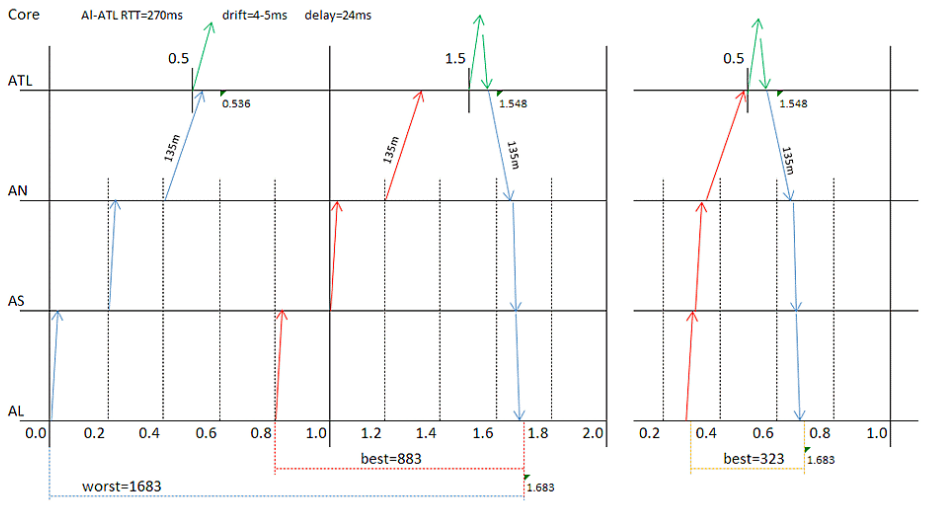

Fig. 4. Message flow through the aggregation layers (AL → AS → AN → ATL → core and back). Horizontal axis represents the flow of time in seconds.

The message flow between the aggregation layers is shown in Fig. 4. The vertical axis represents layers of the aggregation tree, and the horizontal axis represents time-flow in seconds. The left hand drawing illustrates two unsynchronized requests. The first request is the worst case scenario and second is the best. As the request travels upstream it waits for end of the aggregation round at each layer. The first request narrowly misses the end of 1-second top level aggregation cycle; the second request arrives just before the end and the response for both requests arrives at the same time. The right hand drawing depicts the ideal case with synchronized layers.

6 Empirical Results

6.1 Test Setup

The test was performed during the service expansion to Japan, topologically very distant location from the core cluster which is distributed between the different jurisdictions in Europe. The Service was tested extensively in the earlier laboratory environment; its performance in non-ideal network conditions was already mapped. The service latency, when operated within a single continent, had been proven to be satisfactory. We rented physical and virtual servers from five service providers based in Tokyo and Nagano.

Load was generated remotely and measurements were performed at the gateway host—so that client application to gateway connection did not impact the measurements. Tests were run for 24 hour periods, for a minimum of three consecutive days. Worst case scenarios were considered.

As secondary objectives we looked for service providers with independent resources, especially ISP peering and tested the service quality provided by different sizes of clusters of physical and virtual servers. We also measured the effect of system parameters like aggregation periods and provided data to assist the drafting of the service level agreements.

6.2 Results

The main goal was to improve the service quality—provide minimal and deterministic latency of the signing service to the end users; the secondary target was finding a cost effective setup which provides reasonable availability. The progress is presented with the before and after response timing histograms in Fig. 5 and Fig. 6. The first graph depicts the initial real-life signing response timing distribution. Note that there are no failed requests because of the redundancy and automatic retry mechanism on all aggregation layers.

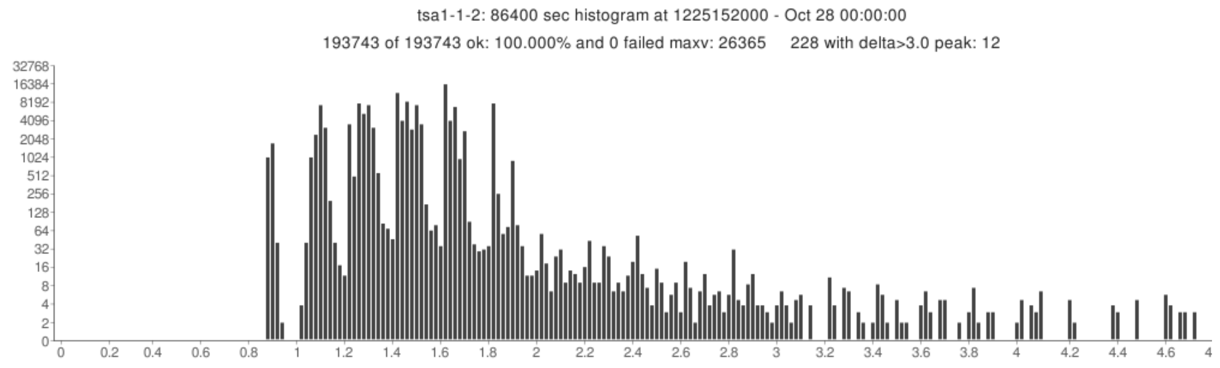

Fig. 5. Real-life response time histogram before the optimizations. Vertical logarithmic scale is number of samples, horizontal scale is response latency in seconds.

Figure 6 histogram illistrates results after the synchronization of the aggregation layers and IP network optimizations. Latency is mostly dictated by the underlying network; the round-trip delay from AN to ATL is approximately 270ms and other network delays are much smaller. Clock drift at all layers is less than 4ms and core protocol voting time is 48ms.

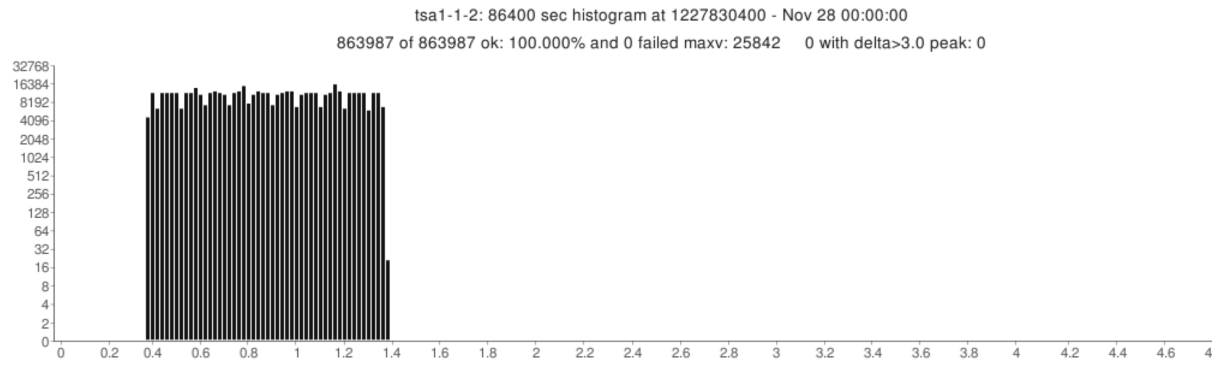

Fig. 6. Real-life response time histogram after the optimizations.

Final optimizations and findings included:

– It is beneficial to synchronize the aggregation layers, so that aggregation periods start in a staggered way.
– The lowest latency was achieved by the aggregation period of 200...400ms. We started with 200ms and later reverted to 400ms for less data traffic.
– In redundant clusters, virtual servers are reasonably good. Three virtual servers cost less and provide better service than 2 dedicated physical servers.
– Virtualized servers can have choppy time flow; it helps to keep local disk IO minimal. We found it necessary to set up network logging.
– Although easier to use, the TCP based network protocol had some unwanted quirks, especially the "*TCP slow start after idle*" feature.

6.3 Practical Implications

We set up a reasonably good configuration of internet-based NTP time synchronization and configured optimal timing offsets—based on measured RTT between the aggregation layers at each aggregator site; two or three aggregation servers in a cluster provided satisfactory results. The core cluster did not need any modifications as its clock ticks were already synchronized to the UTC—using atomic clocks and quality time-sources.

The overall availability of the system over 5-year measuring period is 99.952%; the downtime incidents happened during the first years of operation and were caused by system administration errors and a historical feature in the aggregation protocol that propagated an error in the experimental client API code to the core. Lessons must be learned from the incidents because seemingly impossible things do happen: (1) Do not allow the SysOps touch more than one cluster member during a day, (2) Network protocols must have a versioning mechanism to support the partial rollout of a new version, (3) Testing must be performed on a fully independent system.

7 Conclusions and Future Work

The idea of staggered aggregator layer synchronization suggests the need for an algorithm that tunes the aggregation network to best possible service timing automatically. So far, the possible service improvement have not justified the increased complexity and potential instability.

This is a glimpse of the core cluster operations. There are number of interesting topics to discuss in the follow-up papers.

References

1. Bayer, D., Haber, S., Stornetta, W.-S.: Improving the efficiency and reliability of digital timestamping. In: Sequences II: Methods in Communication, Security, and Computer Sci., pp. 329–334. Springer, Heidelberg (1993)
2. Buldas, A., Laanoja, R.: Security proofs for hash tree time-stamping using hash functions with small output size. In: Boyd, C., Simpson, L. (Eds.): ACISP 2013, LNCS 7959, pp. 235–250, 2013. Springer, Heidelberg (2013)
3. Buldas, A., Niitsoo, M.: Optimally tight security proofs for hash-then-publish time-stamping. In: Steinfeld, R., Hawkes, P. (eds.): ACISP 2010. LNCS 6168, pp. 318–335. Springer, Heidelberg (2010)
4. Buldas, A., Saarepera, M.: On provably secure time-stamping schemes. In: Lee, P.J. (Ed.): ASIACRYPT 2004. LNCS 3329, pp. 500–514. Springer, Heidelberg (2004)
5. Haber, S., Stornetta, W.-S.: How to time-stamp a digital document. Journal of Cryptology 3(2), 99–111 (1991)
6. Merkle, R.C.: Protocols for public-key cryptosystems. In: Proceedings of the 1980 IEEE Symposium on Security and Privacy, pp. 122–134 (1980)