ASCENT: a Provably-Terminating Decentralized Logging Service

Xavier Bonnaire, Rudyar Cortés, Fabrice Kordon, Olivier Marin

ASCENT: a Provably-Terminating Decentralized Logging Service*†

Xavier Bonnaire,
Department of Computer Science,
Universidad Técnica Federico Santa María, Valparaíso, Chile
Rudyar Cortés, Fabrice Kordon
Sorbonne Universités, UPMC Univ. Paris 06,
LIP6 CNRS UMR 7606, F-75005 Paris, France
Olivier Marin
NYU Shanghai, Shanghai, China

September 30, 2017

Abstract

Building a certification authority that is both decentralized and fully reliable is impossible. However, the limitation thus imposed on scalability is unacceptable for many types of information systems, such as e-government services, which require an infrastructure able to support heavy computation loads while remaining highly reliable. Our scalable approach opts for the next best thing to full reliability: a certification authority with a probability of arbitrary failure so low that, in practice, false positives should never occur.
Keywords : Large Scale, Certification, Reliability, Formal Verification

1 Introduction

Imposing a limitation on scalability is unacceptable for many types of large scale information systems such as e-government services [43]. These services require an infrastructure able to support a heavy load while remaining highly reliable. A typical example is income declaration, also known in the US as tax return, where an authority must attest that a form has been filled in time by the corresponding actors. Solutions based on the well-known client-server paradigm do not easily scale on demand, especially when load peaks occur. Building a fault tolerant and semi-scalable service on a cluster based architecture can imply a very high cost in terms of hardware and Internet connection bandwidth that small digital actors like startups cannot afford.

In this context, Peer-to-Peer (P2P) solutions based on a Distributed Hash Table (DHT) such as Pastry [37] or Chord [39] provide a basis for scalable applications as opposed to those involving centralized servers. However, DHT-based applications rely on a centralized trusted entity to prevent the compromission of a disproportionate share of the system by a small number of entities [17]. Even if it provides fault tolerance using redundancy and a high availability, a DHT alone provides no guarantees regarding trustworthiness. In this sense, it is not a satisfactory solution for attesting that a timely action has been performed (for example validating income declarations). Using trusted servers in the DHT does not solve the problem: it limits scalability and requires a central administration of these servers in order to be trusted.

*Contact: Xavier.Bonnaire@inf.utfsm.cl, Rudyar.Cortes@lip6.fr, Fabrice.Kordon@lip6.fr, Olivier.Marin@nyu.edu
†This paper is the HAL version of https://doi.org/10.1093/comjnl/bxx076

In this paper, we present ASCENT (A Scalable Certification through Endorsement by Nodes that are Trustworthy), a quasi-certification authority above a DHT that may involve malicious nodes. The goal of ASCENT is to deliver a certificate attesting that a transaction has occurred between a client A and a service S around time T. Our solution provides such a certificate by means of a distributed logging service which guarantees that A can only append items to a log it has initiated with S; proofs of further requests/replies associated with the same transaction cannot be inserted anywhere else. Section 2 defines the system model and the threat model, and gives an overview of some building blocks that we use to make ASCENT as reliable as possible. We explain the structure of ASCENT in section 3 as well as the details of the associated protocol. In order to prove that our protocol does not have deadlocks and only terminates with known states, we give in section 4 a formal verification of the protocol using Petri Nets.

We call our solution a *quasi-certification authority* as it can fail to attest that a transaction has occurred, or produce a false certificate. However, the probability of failure is so low (as low as 10^{-13}) that we consider that ASCENT can theoretically fail, but that it is practically infeasible. We give in section 5 a probabilistic study of ASCENT, as well as the cost of the protocol in terms of the number of messages it generates.

A general discussion about ASCENT is presented in section 6, including its scope of applicability, its reliability in practical terms, some security issues, and related works. Finally, section 7 draws conclusions and future works.

The contributions of this papers are:

- A quasi-certification authority for Distributed Hash Tables
- A formal evaluation of our solution in order to prove our protocol
- A probabilistic evaluation of our proposal to validate our approach and its reliability

2 System Model

This section presents the theoretical and practical backbone of our work. It states the assumptions we make about the system and the properties we desire of our solution so that it can be deployed successfully. It also describes the bodies of work we draw upon to build our solution.

2.1 Assumptions

We assume a distributed system that is large-scale and dynamic. It is large-scale in the sense that there is no limit on the number of nodes in the system, and dynamic because nodes are expected to join and leave the system arbitrarily; thus the topology of the system varies over time. Nodes communicate via messages.

We make no assumption regarding the communication channels: messages can be duplicated, delayed, lost, and modified arbitrarily. Nodes themselves may fail arbitrarily, and display byzantine behaviors where they act maliciously against the system. However, for the system to be able to function at all, no more than 30% of all system nodes may behave incorrectly at any given time. Essential building blocks of our solution, namely DHT-based overlays and reputation systems, are designed to withstand failure within this proportion. Beyond this limit, they will collapse anyway.

We assume that any node X in the system has a pair of public and private keys (X^{pub}, X^{priv}) it can use with an asymmetrical encryption algorithm [38] known by all the nodes. For scalability considerations, we suppose that every node self-generates its own pair of keys, and that the size of the keys used in the whole system is both fixed and large enough to make it computationally infeasible to decrypt an information without knowing the corresponding private keys (see section 3.2).

We also assume that the underlying operating system of every node is running the Network Time Protocol. Thus, we consider that every correct node has a good global time, with a small deviation from the Universal Time Coordinated (UTC).

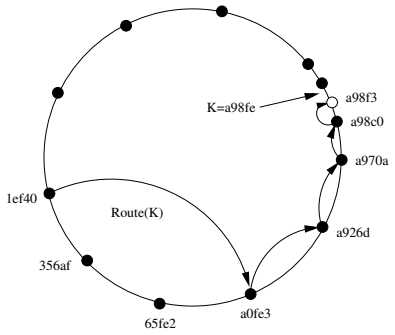

Figure 1: Routing in Pastry

2.2 Design Goals

Our solution aims to provide availability, to show extremely high resilience to corruption, and to be efficient when a client requests a service transaction and a timestamped certificate of said transaction. Every certificate entails a high probability (see section 5) that the associated transaction occurred within a specified timeframe.

In terms of performance, our solution must not hinder scalability and availability. We intend our protocol to be formally verifiable so as to support the claim that it is sound, in other words that only safe situations are reachable from the initial configuration..

2.3 Distributed Hash Table

In order to ensure scalability, the primary building block of our solution is a distributed hash-table (DHT). DHT overlays are autonomous, self-organized, and highly scalable systems that have the potential to grow up to millions of nodes. DHT nodes share their resources, collaborate and interact directly to provide efficient key lookups, high data availability, and persistence. DHT networks are systems that maintain a strong topology, for example a ring in the case of Pastry [37]. For the sake of clarity we assume the use of Pastry throughout this work, but replacing it with any other DHT is straightforward. Please refer to section 6 for more details about the portability of our proposition to other DHTs.

Every node in Pastry is assigned a unique nodeID in a space of identifiers of 128-bits. The nodeID determines the position of the node in the circular namespace of the Pastry ring. A key that indexes information is a value in the same namespace as the nodeIDs. Both nodeIDs and keys are generated using a cryptographic hash [25] that guarantees uniform distribution, thus enforcing scalability and load balancing.

Pastry uses a prefix based algorithm to route every message to the node that is numerically closest to a given key k. Each node X maintains a routing table: the n^{th} row stores the IP addresses of nodes whose nodeID shares the first n digits with that of X. The algorithm forwards the messages to a node from its routing table that shares at least one more digit with the key k than the current node. If no such node can be found, then the current node is the closest and the routing ends. This algorithm allows to route a message to a given key k in O(log(N)) hops, where N is the network size. Figure 1 shows an example of the routing process. Upon each hop, the message with key $a98fe$ reaches a node whose nodeID is closer by at least one more digit. The routing stops at node $a98f3$ because there is currently no other node in the DHT whose nodeID is closer to $a98fe$.

Every node in Pastry references its neighbors in a *leafset* that contains the addresses of nodes whose nodeIDs in the ring are numerically closest to its own. Let L be the standard notation for a leafset, and $|L|$ for the size of this leafset. A typical value for $|L|$ is 16: with 8 references to the numerically closest nodes clockwise, and 8 to the numerically closest nodes counter-clockwise. Every node is in charge of maintaining its own leafset: it polls its neighbors periodically to check for missing nodes, and starts an update procedure to repair its leafset upon a node failure. A very important property of the leafset is its density: the average

distance between all nodeIDs in the leafset. The uniform distribution of nodeIDs induces that all leafsets have the same density. Among other applications, density can be used to detect counterfeit leafsets [11]. Another important property is that nodes in the same leafset are very unlikely to be close within the network topology. This is a direct consequence of generating nodeIDs with a hash function that guarantees uniform distribution.

2.4 Reputation System

The goal of a reputation system [9] is to discriminate malicious nodes from honest ones. Most existing reputation systems use a recommendation mechanism. Every time two nodes carry out a transaction, they produce feedback on each other's behavior during the transaction. Let T be a transaction where a node X requests a service from another node Z. At the end of T, X evaluates the service provided by Z. X emits a positive recommendation about Z if T was successful, a negative recommendation otherwise. A reputation system uses the list of recommendations emitted for node Z to compute its reputation.

To manage trust, our solution requires a reputation system that outputs a reputation value $R(X) \in [0..1]$ for every node X. $R(X)$ represents an assessment of the behavior of node X; its value is the perceived probability for node X to be honest. $R(X) = 0$ denotes a node perceived to be fully malicious, while $R(X) = 1$ designates a node perceived to be fully honest. It is important to keep in mind that these are subjective values: a reputation system can never fully decide whether a node is malicious or honest. A node with a high reputation is a node with a higher probability of behaving honestly. We also need the reputation to be persistent: in other words, any node Z can obtain the reputation $R(X)$ of node X at any time.

In the following, we use the WTR (Worth The Risk) [10] reputation system, as it has demonstrated very good performance, and is also resistant to most existing attacks that are specific to reputations systems. Nevertheless, our proposition can be implemented over any reputation system that fulfills the above mentioned requirements.

2.5 Community of Trusted Nodes

To enhance the trustworthiness of our quasi certification authority, we allow for the integration of a membership algorithm like CORPS [36], which builds a group G of trusted nodes such that:

$$\forall X \wedge R(X) \geq \rho \, then \, X \in G$$

where ρ is a threshold determined by the application that allows to discriminate trusted nodes (with a sufficiently high reputation) from the others (malicious ones, or not yet trusted). Simulations performed with CORPS have shown that $\rho = 0.85$ is a good threshold (see [36] for more details).

Similarly to Pastry, the nodes in G are organized in a ring. Each node in G and in the DHT maintains a trustset $TS = \{ts_1, .., ts_{|L|}\}$ where $|L|$ is the size of the DHT leafset. Hence a node X references $|L|/2$ nodes clockwise, and $|L|/2$ counterclockwise in its trustset.

After a start-up phase, typically after a sufficient amount of transactions, CORPS converges to a unique ring where all the honest nodes belong to the ring. We call G a Trusted Ring. Therefore, any node can easily find a trusted node using its trustset. Figure 2 shows an example of CORPS ring over Pastry with $|L| = 4$. In particular, it shows two examples of trustset for trusted nodes (*e.g.* N4) and for normal ones (*e.g.* N10).

The membership algorithm of CORPS ensures that:

- When a new node enters or leaves the trusted ring G, its neighbors update their respective trustset to maintain the same number of nodes, the same way Pastry does for the LeafSet.
- When a node $X \in G$ becomes such that $R(X) < \rho - \alpha$, where α is a tolerance parameter (to avoid a yo-yo effect), then X must leave the trusted group. If it does not, the CORPS protocol ensures that every adjacent node of X will remove X from its trustset, implicitly pushing X out of the trusted ring. For more details about the CORPS maintenance protocol, please refer to [36].

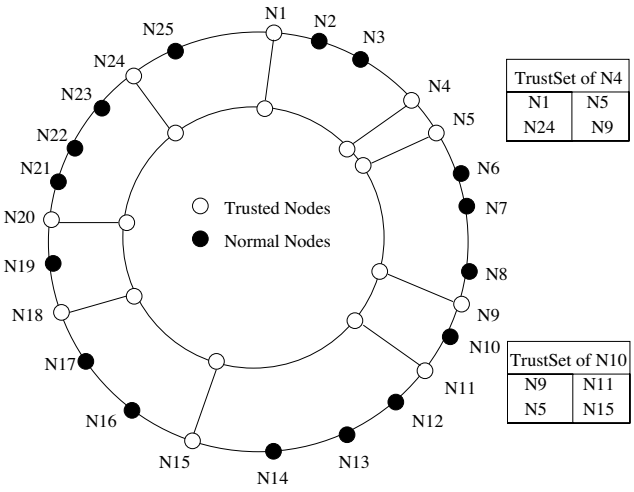

Figure 2: A CORPS ring above Pastry

Because the reputation system may fail with a probability of ε when classifying a node, there is still a probability ε for a node $X \in G$ to be malicious. Implementation of a Trusted Routing similar to that of Pastry on top of the CORPS mitigates this issue. As demonstrated in [36], the expected number of attempts required to reach the destination node via Trusted Routing at the CORPS level is much lower than using the Pastry routing at the DHT level.

3 ASCENT, a Quasi-Certification Authority

This section details the structure and the protocol of ASCENT, a quasi certification authority (CA) for DHTs. ASCENT is a "quasi" CA, as opposed to a full CA, since [17] shows that a small number of malicious nodes can compromise a disproportionate share of a DHT if there is no trusted centralized entity. Hence the goal of ASCENT is to certificate, with a very low probability of failure, a transaction between a client node *A* and a service *S*.

Figure 3 shows the general structure of ASCENT. The *DHT layer* provides access to the P2P network on the Internet. The goal of the *certification layer* is to generate certificates as evidence that a transaction has occurred between a client *A* and a service *S*. This layer also maintains a distributed log of the certificates for further requests (proofs of past transactions). The *trust layer* provides mechanisms to discriminate honest nodes from malicious ones. Note that our certification protocol can work directly above the DHT, and thus does not require a trust layer. In fact, forgoing the trust layer reduces the cost in terms of network load. However, the use of trust mechanisms is strongly recommended as it significantly reduces the probability of

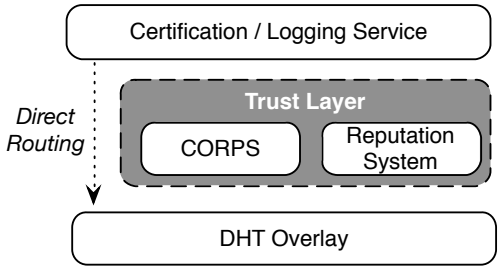

Figure 3: General architecture of ASCENT

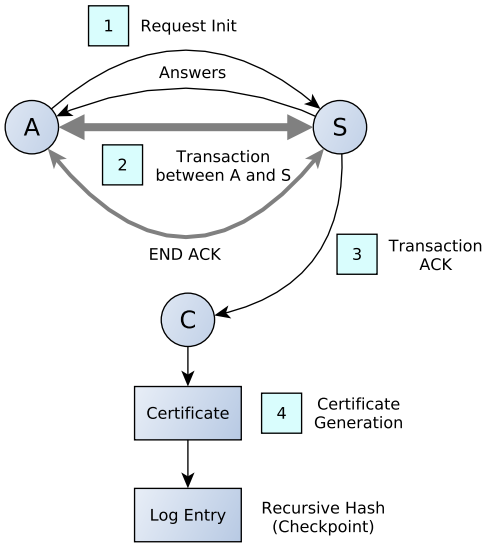

Figure 4: Certification Scheme

failure of the certification. See section 5 for more details about the impact of the trust layer.

We assume in the following that every node participates to the CORPS membership algorithm mentioned in 2.5, and thus that every node supports a combination of a DHT with a reputation system. We also assume that the system has reached a stable state where every node of the DHT can access a CORPS trusted ring.

Let A be a node of the DHT, and $S = \{S_1, S_2, \cdots, S_n\}$ be a set of n nodes that belong to the CORPS ring and cooperate to provide a service. Let *NameofS* be the name of service S; every node that needs to use the service knows *NameofS*. $K_s = SHA(NameofS)$ is the key that identifies S in the trusted ring, and $S_{root} = TrustedRoute(K_s)$ is the node closest to K_s in the trusted ring. It follows that $S = \{S_1, S_2, \cdots, S_n\}$ is composed of node S_{root} and of all the nodes in its trustset. Hence every existing service in the trusted ring is associated with a different set of trusted nodes which handle the service. This induces natural load balancing among services.

Let C be a certification authority such that $C = \{C_1, C_2, \cdots, C_n\}$ is a group of n nodes belonging to the CORPS trusted ring. Figure 4 shows the general interaction scheme between A, S, and C.

We suppose that the number of nodes that compose a service S is fixed and does not change over time. We also suppose that the same stands for any quasi certification authority C.

Finally, we assume that two identical requests (same version number, same client identifier, same content) from a client A to a server S are in fact duplicates of the same request. For instance if a student repeatedly submits electronic versions of her essay, two exact same submissions of the essay will be considered by the server as duplicates of the same request. If the essay has been modified in-between two submissions then these two messages will be considered as different requests.

3.1 Securing a reliable trustset

The first step to requesting a certification/service from trusted nodes is to acquire the set of nodes that compose the trustset for a given root key K. The most efficient way to acquire this trustset is to request it directly from the node K_{ROOT} associated with K in the Trusted Ring of the CORPS. However, even though CORPS nodes are more trustworthy, it is still possible that K_{ROOT} may be malicious. K_{ROOT} might lie about its trustset and provide references to other malicious nodes in order to collude against the requester, or it might deny any service by remaining silent. Therefore, a simple request to K_{ROOT} may not suffice to acquire a reliable trustset associated with K.

On the other hand, it is possible to build the trustset associated with K from scratch. But such an operation is costly, and it would be a mistake to carry it out systematically on a large scale.

Our approach towards mitigating this issue starts with a validation process that verifies whether the trustset returned by node K_{ROOT} is reliable. If the validation fails, an alternative protocol takes over and builds the target trustset from scratch.

Trustset validation

Let TS be the trustset returned by K_{ROOT}. The validation of a trustset TS consists in two successive tests. The first test checks the density of TS by comparing the distribution of the node identifiers in TS with the distribution of identifiers in the nodeId space. The second test checks the consistency of TS by comparing the respective trustsets held by each node in TS.

Our first test, the density check, uses the *Routing Failure Test* proposed in [11]. It assumes that the generation of node identifiers uses a cryptographic hash function with a global uniform distribution like Secure Hash Function (SHA-1). This test computes the average distance between identifiers in the neighborhood of the client node, and then compares it with the average distance between identifiers in TS.

Tampering with a trustset density requires control of an entire portion of the namespace in the trusted ring. This starts with corrupting a set of controlled nodes in that portion, and forcing the secure routing algorithm to converge on the corrupted set. Given the inherent dynamic property of DHTs, the attacker must also block any other node from entering both the associated portion of the namespace and the possible routes to these nodes. Previous work [36] showed that the probability of corrupting this many nodes can be kept below 10^{-20}.

Formally, a density check works as follows. Let ρ be the average distance between identifiers in TS, and let α be the average distance between identifiers in the neighborhood of the client node. The density check only succeeds if $\rho \leq \alpha\delta$, where δ is a parameter for minimizing false positives and false negatives. This test detects a node that tampers with its own trustset; however, it can generate false positives.

Our second test, the consistency check, dismisses false positives by requesting a confirmation from the nodes of TS, the trustset of K_{root}. The client node sends its own image of TS to all the nodes in this image and requests a confirmation value in $\{0,1\}$. Upon reception of the request from the client node, each callee compares the received set of nodes with its own trustset. The callee returns 1 if the nodes in the received trustset match its own partial view. If there is a mismatch, the callee returns 0. The client can consider its own image of TS as valid if it receives a 1 from at least $\lfloor \frac{|L|}{4} + 1 \rfloor$ nodes on the left side of K_{root}, and $\lfloor \frac{|L|}{4} + 1 \rfloor$ nodes on the right side. If the client node does not receive a majority of validations both from the left side and from the right side of K_{root}, then it considers that K_{root} has illegally manipulated its own trustset.

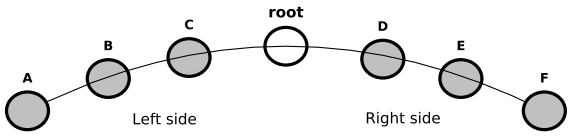

Figure 5: Trustset of the node K_{root}

Figure 5 gives a logical representation of an example trustset of size $|L| = 6$, while table 1 shows the partial views that each node of the example trustset must share with K_{root} for the sets to be consistent. Note that nodes A, B, and C should share a partial view of the left side (A,B,C) of the trustset of K_{root}, whereas nodes D, E, and F should share a partial view of its right side (D,E,F). As an example, if K_{root} delivers a fake trustset by replacing node A with some colluding node X that is a neighbor of A, nodes on the right side will validate the trustset, but nodes on the left side (B and C) will not because X does not belong to their partial view. As soon as the client node gets a majority of invalidations from nodes on the left side of K_{root},

Node	Partial View
A	$\{A,B,C\}$
B	$\{A,B,C,D\}$
C	$\{A,B,C,D,E\}$
D	$\{B,C,D,E,F\}$
E	$\{C,D,E,F\}$
F	$\{D,E,F\}$

Table 1: Partial views of the trustset of K_{root} in the example of figure 5

it rejects the trustset.

Algorithm 1 Initial acquisition of the trustset associated with key K

```
GetTrustset(K)
{
m.label = GET_TRUSTSET
m.source = myIP@
route(m,K)
receive(<m>) from <K_root>
TS = m.content
if Test(TS) == False then
    TS = BuildSubstituteTrustset(K)
end if
return TS
}
```

Algorithm 1 routes a message all the way to node K_{root} and tests the returned trustset. If TS passes both the density check and the consistency check, then the client considers it valid. If any of these tests fails, `BuildSubstituteTrustset()` triggers an alternative protocol that builds the target trustset from scratch.

Secure construction of a substitute trustset

If K_{root} lies about its own trustset or fails to reply, the client node can build its own target trustset as a cross section of the union of validated trustsets from trusted nodes that are in the vicinity of K_{root}.

Towards this purpose we alter the routing protocol so that it follows a diversity routing scheme like [11, 15]. Diversity routing uses different nodes as starting points in order to reach trusted nodes in the vicinity of node K_{root}. The diversity of the routing tables among different nodes guarantees a very high probability of reaching K_{root} via different paths, and therefore of entering the true trustset of K_{root} via different nodes.

Algorithm 2 Secure routing of a message

```
DiverseTrustedRoute(m,k)
{
if (m.label == BUILD_TRUSTSET) and
(k ∈ [minID(leafset), maxID(leafset)]) then
    //key K is within the range of my leafset
    send(myTrustset) to m.source
else
    next = DiversitySelect(myTrustset);
    DiverseTrustedRoute(m,k) via next;
end if
}
```

Algorithm 2 details our alterations of the routing protocol: it enforces diversity routing, it channels the routing via trusted nodes only, and it provides an option for acquiring the trustsets of trusted nodes in the

vicinity of K_{root}. `DiversitySelect()` enables diversity routing by picking a node in the given set according to the scheme chosen for the implementation. The client node can opt to target a vicinity instead of the key itself by labelling its request message m with $m.label = BUILD\_TRUSTSET$ before routing it. Instead of forwarding such a message until it reaches the node whose identifier is *the* closest to key K, a receiving node checks whether K belongs to its own leafset L, ie. K is both smaller than the largest node identifier in L and greater than the smallest node identifier in L ($minID(L) \leq K \leq maxID(L)$). If that is the case, the receiver returns its own trustset directly to the client node that initiated the request.

On its own, our alternative routing doesn't guarantee a functional trustset. Firstly, it is crucial to carry out a validation test on the returned trustset because the alternative node that responded may also be lying. Secondly, since the $BUILD\_TRUSTSET$ option is on, the returned trustset may not be centered on key K, so it might not provide enough trusted nodes on both sides of K. Remember that a functional trustset references $|L|/2$ nodes on each side of K_{root}. Our solution is to acquire trusted nodes from valid trustsets that overlap key K, and poll these nodes for their own trustsets. Thus, as we extract further trusted node references in the vicinity of K, we build a functional trustset by increments.

Algorithm 3 Construction of an alternative trustset associated with key K

```
BuildSubstituteTrustset(K)
{
//PHASE 1
//Acquire a first valid trustset via secure routing
VisitedNodes = ∅
repeat
    m.label = BUILD_TRUSTSET
    m.source = myIP@
    DiverseTrustedRoute(m, K)
    receive(<m>) from <node>
    VisitedNodes = VisitedNodes ⋃ {node}
    TS = m.content
until Test(TS) == True
//PHASE 2
//Pull nodes from valid trustsets
//until TS is functional
NodesToQuery = TS \ VisitedNodes
while Functional(TS) == False do
    m.label = GET_TRUSTSET
    m.source = myIP@
    //Recenter() reduces the number of polled
    //nodes to at most |L|/2 on either side of K
    NodesToQuery = Recenter(K, NodesToQuery)
    ∀ X ∈ NodesToQuery, send(m) to X
    ExpectedReplies = |NodesToQuery|
    repeat
        receive(<m>) from <node>
        ExpectedReplies = ExpectedReplies - 1
        VisitedNodes = VisitedNodes ⋃ {node}
        TS' = m.content
        if Test(TS') == True then
            TS = Recenter(K, TS ⋃ TS')
            NodesToQuery = NodesToQuery ⋃ TS
        end if
    until ((Functional(TS) == True) or
          (ExpectedReplies == 0))
    NodesToQuery = NodesToQuery \ VisitedNodes
end while
Return TS
}
```

Algorithm 3 starts with the acquisition of a valid trustset via successive secure routing attempts. Once it

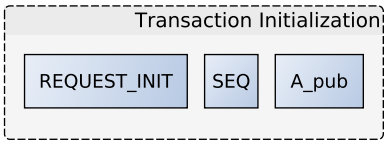

Figure 6: RequestInit message format

acquires a valid trustset TS that overlaps key K, the client node checks whether it is functional. If it isn't, the client node retrieves each trustset TS' from the nodes in TS, checks the validity of TS', computes the union of TS and TS', and then filters the result. `Recenter()` retains the $|L|/2$ node identifiers that are closest to K on either side, and discards the others. If the resulting set references exactly $|L|/2$ trusted nodes on either side of K, then it is functional. If such is not the case, all new (ie. not yet queried) nodes are stored for an ulterior round of polling.

In order to ilustrate our algorithm consider the trustset from figure 5 and the partial views of table 1. If the root node returns an invalid trustset, the client node routes a *BUILD_TRUSTSET* message. Let us assume that the message reaches node A. Once it has verified the trustset returned by A with the *density* and *consistency* tests, the client node can poll nodes B and C directly. The latter return their own trustsets, and the client node now has a partial view $V = \{A,B,C,D,E\}$. The next round of polling pulls a reference to node F from either D or E, and thus completes a functional trustset $T_{K_{ROOT}} = \{A,B,C,D,E,F\}$.

3.2 Service Request

To find the nodes associated with a given service, node A computes the hash of the service name: $K_s = SHA(ServiceName)$. Let S_{root} be the node that is closest to K_s in the CORPS Trusted Ring, that is $NodeID_{S_{root}} = GetTrustset(K_s)$ (see algorithm 1). A thus obtains access to the set S of nodes that process requests for service *ServiceName*. Since $|L|$ is the number of nodes in a trustset, $card(S) = |L|+1$.

Once A has acquired S, it proceeds to the inception of its service request by sending a *RequestInit* message to every node in S. A *RequestInit* message contains the public key A^{pub} of client A and a sequence number SEQ used as a transaction identifier (see section 3.4.1). Figure 6 represents the RequestInit message format. In reply to such a message, every node S_i sends its own public key S_i^{pub}. In order to avoid a man-in-the-middle attack, all the ensuing communications between A and nodes that belong to S will use a public/private key encryption scheme to ensure that no other node can read the content of the messages. Figure 7 sums up the RequestInit phase.

If A receives at least $\frac{|L|}{2}+1$ answers from the nodes in S, then A considers that the transaction will proceed. Although S is composed of nodes from the trusted ring, there remains a probability for some of these nodes to be malicious. However, as explained in our evaluation of the quasi certification protocol

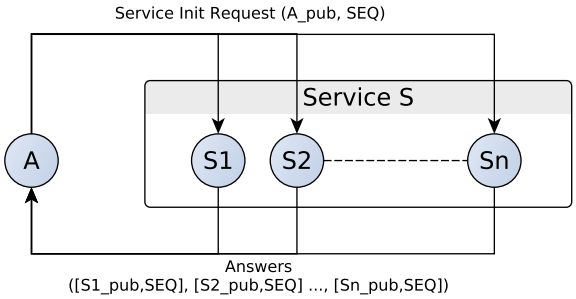

Figure 7: Request Init Phase

Algorithm 4 Request Init (Client Side)

Input : name *//Service name*
Ouput : Boolean
$K_s = SHA(name)$
S_{root} = TrustedRoute(K_s)
trustset = Gettrustset(S_{root})
$SEQ++$ //Static variable
for S_i IN trustset **do**
SendMessage(REQUEST_INIT, SEQ, A^{pub}, S_i)
end for
SetTimer($\triangle t$)
while Not received $\frac{|L|}{2}+1$ identical answers **do**
WaitAnswer()
end while
if TimedOut **then**
return FAILED
else
return OK
end if

(Section 5), a situation where S contains more than $\frac{|L|}{2}$ malicious nodes is highly improbable.

Algorithms 4 and 5 detail the pseudo-codes executed respectively by A and by every node S_i of S. Note that variable SEQ is computed to guarantee that each transaction message is distinct and globally unique, as explained in 3.4.1.

Algorithm 5 Request Receive (Service Side)

request=ReceiveRequest()
pub_key=ExtractKey(request)
SEQ=ExtractSequence(request)
message=BuildMessage(ACK)
Encrypt(message, SEQ, pub_key)
Reply(request, message)

3.3 Service Progress

After the initialization phase, A and S proceed to a service progress phase. All exchanged messages include the identities of the communicating parties. A sends all its messages to every node of S and expects to receive at least $\frac{|L|}{2}+1$ identical answers for each message. A signs every message to the nodes of S with its private key A^{priv}, and then encrypts every message using the public key of the recipient. Conversely, all messages sent to A by a node S_i of S are signed using its private key S_i^{priv} and encrypted using the public key A^{pub} of A. Figure 8 shows the format of a message sent by A to S_i.

Upon receiving the last transactional message from A, every node of S waits for an ACK message from A

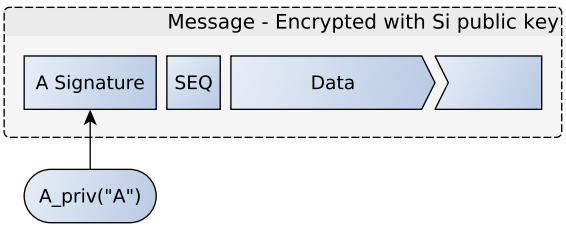

Figure 8: Transaction progress message format

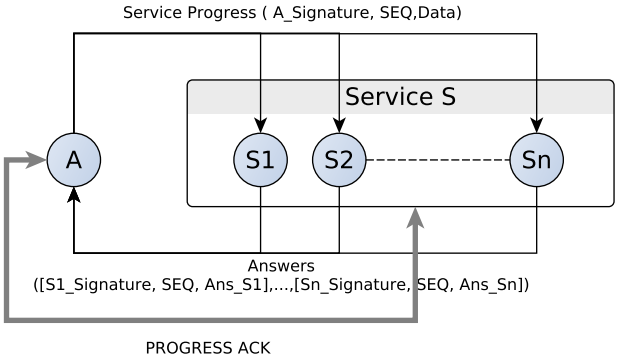

Figure 9: Transaction Progress

and responds with an ACK to A. If A does not receive at least $\frac{|L|}{2}+1$ ACKs, it considers that the transaction has failed.

Similarly to the service request, A may fail to receive at least $\frac{|L|}{2}+1$ answers because of malicious behaviours or because of network partitions. In such cases, A considers that the transaction has failed and aborts after a time $\triangle t$.

Figure 9 represents the exchange of messages between A and S during the service progress phase. Algorithm 6 details the protocol between A and S.

Algorithm 6 Transaction Progress Protocol for node A

Input : trustedset *// trustset of S*
Input : SEQ *//Sequence Number*
for i IN Number_of_messages **do**
  for S_i IN trustedset **do**
    message = Message[i]
    Sign(message)
    Encrypt(message, SEQ, S_i^{pub})
    SendMessage(message, S_i)
  end for
  SetTimer($\triangle t$)
  WaitForIdenticalAnswers() *//At least* $\frac{|L|}{2}+1$
  if TimedOut) **then**
    Abort() *//Transaction failed*
  end if
end for
for S_i IN trustedset **do**
  message=BuildMessage(ACK)
  Encrypt(message, SEQ, S_i^{pub});
  SendMessage(message, SEQ, S_i);
end for
SetTimer($\triangle t$)
WaitForIdenticalACKs() //At least $\frac{|L|}{2}+1$
if TimedOut **then**
  Abort()
else
  TransactionOK()
end if

In order to identify request duplicates, every node S_i of S computes a recursive hash signature $hash_{SIGN}$ for the transaction with A. Equation 1 shows the computation of the hash signature for the transaction with A, where $||$ denotes the concatenation operation.

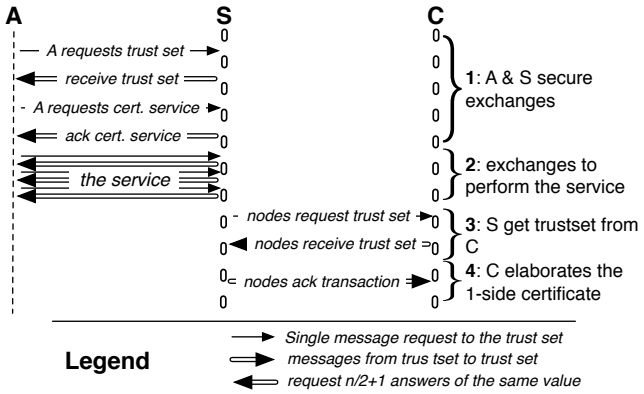

Figure 10: Structure of the protocol

$$\begin{cases} hash^0_{SIGN} = SHA(SEQ||FirstData) \\ hash^i_{SIGN} = SHA(hash^{i-1}_{SIGN}||SEQ||CurrentData) \end{cases} \quad (1)$$

Hence, two sets of messages from A that have the same hash signature value correspond to the same transaction. This allows to determine whether a request is a duplicate. If A uses the same sequence number and the same data content for two transaction requests, then nodes in S will consider these requests as duplicates and issue a single certificate. In any other case, S will consider the messages as two different transaction requests and will issue two different certificates.

Figure 10 presents the overall structure of the protocol. **A** is the client which initiates the service with **S**, and **C** is the quasi-certification entity. S gets executed on a set of trusted servers, and so does C on a different set. As the figure shows, the protocol comprises four phases: phase 1 includes all preliminary exchanges between A and S to prepare the transaction, phase 2 includes all exchanges between A and S as they carry out the transaction, phase 3 includes all preliminary exchanges between S and C towards the certification, and finally phase 4 consists in the delivery of the certificate by C.

3.4 Certification

Concurrently to carrying out a transaction for a client A, S proceeds to obtain its certification from another set C of nodes. At the end of the transaction, the objective is to return a certificate to A and to store the certificate on the nodes that belong to C.

3.4.1 Certificate Generation

Upon reception of a *RequestInit* message from a client A, every node in S starts looking for the set C of nodes that will constitute the quasi certification authority for the transaction. To determine which nodes belong to C, S computes its key K such that $K = SHA(NodeID_A||ServiceName)$. $C_{root} = TrustedRoute(K)$ is the node closest to K in the Trusted Ring, and the quasi certification authority $C = \{C_1, C_2, \cdots, C_{|L|+1}\}$ for the transaction between A and S is composed of C_{root} and its trustset.

To prevent man-in-the-middle attacks, we use an asymmetric key encryption scheme for the communications between S and C. This induces that every node in S must exchange public keys with every node in C. The initial cost expressed in number of messages is $2 \times |L|^2 + 4 \times |L| + 2$, but this exchange only occurs the first time a client A requests a service from S. Every node in S maintains a local cache with the public keys of C, and key exchanges only occur again if a new node replaces one of the original nodes from C. Since the nodes with the highest reputations are nodes that remain in the system for long periods of time, the Trusted Ring is bound to have a low churn rate. This in turn sustains a very low cost with respect to public key retrievals.

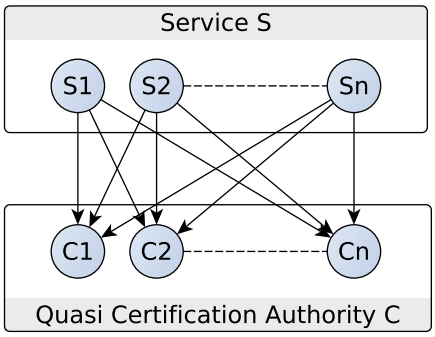

Figure 11: Notifications from service S to quasi authority C

Upon reception of the ACK that denotes the end of a transaction, each node of S sends a notification to every node C_i in C, encrypted with the public key of the recipient. Figure 11 shows the exchange of messages between S and C. The notification message contains the hash signature $hash_{SIGN}$ of the transaction, computed by and signed with the private key of the sender in order to prevent a man-in-the-middle attack.

When a node C_i of C receives at least $\frac{|L|}{2}+1$ identical notifications from S regarding a transaction with A (same hash signature), then C_i assumes that a transaction between A and S has occurred indeed, emits a certificate, and logs the latter locally in an append-only file. C_i uses the key $K = SHA(NodeID_A \| ServiceName)$ to identify the log. It follows that every log file is associated with a specific service and a specific client.

Since a client can submit different requests to the same service, we define an order relation to sort transactions in the same log.

Let $T_x = [SEQ_x, hash^x_{SIGN}]$ identify a transaction. For two transaction identifiers T_a and T_b :

$$\begin{cases} T_a < T_b \quad if \begin{cases} SEQ_a < SEQ_b \\ \quad \textbf{OR} \\ SEQ_a = SEQ_b \quad \wedge hash^a_{SIGN} < hash^b_{SIGN} \end{cases} \\ T_a = T_b \quad otherwise \end{cases}$$

If two transactions are identical, C considers them as duplicates and stores only one certificate into the corresponding log.

Every node in C maintains the same log concurrently, which means that at any given time several versions of the same log may coexist. However, although some certification nodes may lie when exchanging log updates, it is not necessary to impose total ordering among nodes when handling the certification process. In order to avoid such a strong constraint, we designed a lazy consistency protocol that we describe in Subsection 3.4.2. The protocol relies on the concept of log entries. Every node in C arranges log entries in the log file so that, if $E_y(T_x)$ is the log entry of index y associated with transaction T_x:

$$\forall a,b \quad \forall i,j \quad E_i(T_a)\, appears\, before\, E_j(T_b) \quad if\, T_a < T_b$$

Every log entry follows the format:

$$E_i(T) = \{SEQ_T, hash^T_{SIGN}, hash^K_i, t\}$$

where

- SEQ_T is the sequence number of the transaction provided by the client,
- $hash^T_{SIGN}$ is the transaction signature,
- $hash^K_i$ is the recursive hash of the log entry as defined below,

- t is a timestamp generated with the local UTC time of the node.

In order to distinguish successive versions of a log, we use a recursive hashing method computed as follows:

$$\begin{cases} hash_0^K = 0 \\ hash_{i+1}^K = SHA(SEQ_{i+1} \| hash_{SIGN}^i \| hash_i^K) \end{cases}$$

This recursive hashing method is our secure alternative to Bloom filters for sifting through successive versions of a certificate log. In other words, it provides a secure identification for the state of the log on each node of C. It allows to determine with absolute certainty whether a node is indeed in possession of a given version, and induces neither false positives nor false negatives. A single missing entry prevents a node from pretending it stores a valid log. It also guarantees that the recursive hash of the last entry of the log corresponds to the recursive hash of the whole log, and allows to compare two versions efficiently.

3.4.2 Lazy Log Consistency Maintenance

Insertions, reputation assessments, and departures of nodes in the DHT cause modifications of the trustsets, and it is therefore crucial to maintain the consistency of the certificates logs on every node. A node gets removed from a trustset either because it simply leaves the DHT of its own will, or because the *CORPS* maintenance protocol no longer considers this node as trustworthy when it fails to verify equation 1. A node can enter the trustset of another node either when its reputation has increased above the required application threshold enforced by the *CORPS*, or when its reputation is already high enough and a node removal calls for a replacement.

Upon entering a trustset, a node X must update its own view of the certificates logs. It starts by following the steps below to find which logs must be updated.

1. X constructs the set of nodes $N = \{X_{-|L|/2}, X_{-|L|/2+1}, \ldots, X, \ldots, X_{|L|/2-1}, X_{|L|/2}\}$ composed of its trustset, and of the $\frac{|L|}{2}+1$ nodes both on the right and on the left of its trustset. Figure 12 gives an example for a trustset of size $L = 2$. In order to build this set of nodes, X can ask the farthest nodes $X_{-|L|/2}$ and $X_{|L|/2}$ of its trustset for their own respective trustsets. If one of these nodes fails to answer in a timely manner, X can ask the second to farthest node, and so on until its immediate neighbors in the trustset. Having $\frac{|L|}{2}+1$ malicious neighbors on both sides is highly improbable, and practically infeasible: for more details about this statement, please refer to our probabilistic evaluation in section (5)).

2. X computes d_{LEFT}, d_{RIGHT} and finally interval I such that

$$d_{LEFT} = \frac{\left|X_{-|L|/2} - X_{-|L|/2+1}\right|}{2}$$
$$d_{RIGHT} = \frac{\left|X_{|L|/2} - X_{|L|/2-1}\right|}{2}$$
$$I = [X_{-|L|/2} - d_{LEFT}, X_{|L|/2} + d_{RIGHT}]$$

3. X sends a message containing interval I to every node in set N. Upon reception of the message, each node replies with the set of all the log identifiers with key $K \in I$ it stores, along with the last recursive hash corresponding to each log. Thus the format of the response is a variable-size set of pairs $\{K, hash^K\}$.

4. X builds the set of key-hash pairs Y from the union of all the received log identifiers and recursive hashes. X then discards all pairs $\{K_i, hash^{K_i}\}$ that appear less than $\frac{|L|}{2}+1$ times in Y, since these may have been generated by colluding nodes. Once this is done, X synchronizes all the logs whose identifier matches one of the remaining identifiers in set Y.

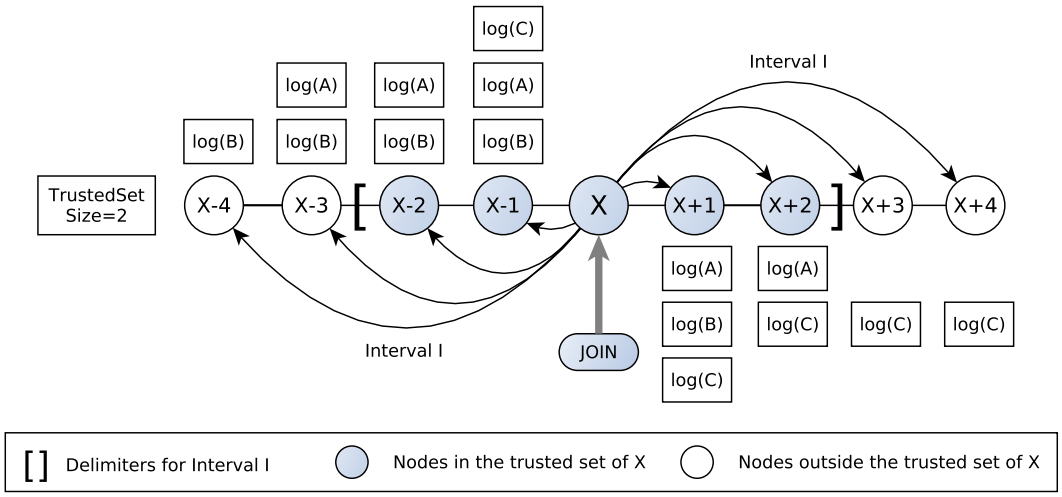

Figure 12: Certificates logs maintenance

For every log that X must synchronize, X selects a node Z in N that holds this log. If X does not have any entry for this log, then X must ask Z for the entire log. Otherwise, X sends the last q recursive hashes it has for the log. The recursive hashes act as checkpoints for the log. The purpose of the checkpoints is to prevent nodes from downloading entire logs when they already store parts of logs. This may happen when a node re-enters a trustset it had previously left. Our checkpointing scheme allows reentrant nodes to identify the missing parts of the log. Upon reception of the last q recursive hashes from X, Z looks for matches in its log entries. If the search is successfull, Z sends back all its entries that are subsequent to the most recent match. If Z cannot find any match, then it requests the next q hashes from X. If Z states that it cannot find any match, then X can consider that Z is lying, and X must contact another node in N.

Upon reception of the log entries from Z, X adds the entries to its log and computes the recursive hash. If the recursive hash differs from those provided by at least $\frac{|L|}{2}+1$ nodes in set Y then X must consider that Z is lying, and can proceed to retrieve the log entries from another node in N.

It is essential for X to handle log synchronization as an atomic operation, as delivering a transaction notification may modify a log entry and invalidate log consistency. Therefore X postpones all transaction notification deliveries until log synchronization is over.

The last recursive hash will suffice for almost every log entry retrieval. In some rare cases, X may receive a transaction notification from service S immediately after entering the trustset and before the beginning of the log synchronization process. This is possible if A makes a very short transaction with S when X is part of the trustset of the corresponding quasi certification authority, but just before the synchronization of the logs on X. In this case, this will produce a shift in the log of X, and its last recursive hash will not correspond to the ones of the other nodes that manage the same certification log. Since this recursive hash is useless, X can discard it and proceed to a new log synchronization.

3.4.3 Node Failures and Node Departures

The *CORPS* [36] trustset maintenance algorithm defines three situations that call for the removal of a node.

1. The node leaves the DHT, and hence also leaves the trustset it belongs to. In this situation, called a *normal departure*, an honest node notifies its departure to the nodes of its trustset.

2. The reputation value of the node decreases below the threshold set by the *CORPS* to define reputable nodes. In this *shameful departure* situation, the node gets pushed out by the peers that reference it in

their trustset. Nodes lead periodical reputation assessments for every node in their trustset to identify those that have become disreputable, and trigger shameful departures when necessary.

3. In a *crash departure* situation, the node fails independently of both the DHT and the reputation system. Periodical heartbeats are exchanged among nodes in the same trustset in order to detect such failures.

Upon removal of a peer in its trustset, a node X will look for a replacement. X starts by identifying which side of its trustset it must fix: the left side or the right side, depending on the nodeId of the node that was removed. Let $N = \{N_\alpha, N_\beta, \ldots, N_\omega\}$ be the set of $\frac{|L|}{2} - 1$ nodes corresponding to the side of the trustset in need of fixing. In order to find the next node to add to its trustset, X requests the trustset of the farthest node N_ω in set N. Upon reception of the answer, X then inserts in its own trustset the reference to the node closest to N_ω that does not yet belong to N. If N_ω remains silent, X must repeat the operation with the second to farthest node in N, and so on until reaching node N_1 if no timely answer ever comes back (see algorithm 7). If X fails to receive any answer from the nodes in its trustset, then it cannot repair its trustset. Please see the evaluation of the probability of failure of the trustset repair in section 5 (table 2).

Once it has successfully fixed its trustset, X must synchronize its logs. X uses the $GetTrustset()$ function to acquire the trustset TS of the node it has just added. X then synchronizes its logs with the nodes in TS by means of the scheme described in section 3.4.2.

3.5 Verifying a certificate

In order to verify whether a transaction has occurred between client A and service S, a node Z must first acquire the trustset associated with key $K = SHA(NodeID_A || ServiceName)$ by calling $GetTrustset(K)$. Depending on the application, Z may be interested in the last transaction that occurred between A and S, or the last n^{th} transaction, or even in checking whether a transaction occured between two dates. In order to simplify the explanation of the protocol, let us assume that Z wants to check if the last transaction between A and S has occurred before time t_0.

Z sends a request to every node in C for the last certificate delivered to A. The last certificate is the last entry in the log file associated with A and stored on C. When Z receives at least $\frac{|L|}{2} + 1$ answers from the nodes in C with the same recursive hash value, then Z considers this answer as correct. After a time $\triangle t$, if there is no answer from C, or if there are less than $\frac{|L|}{2} + 1$ answers from C, then Z considers that there is no such certificate.

Let W be the set of certificates received by Z. Z builds the time interval $T = [t_{min}, t_{max}]$ with t_{min} the oldest certificate timestamp in W and t_{max} the most recent. Since we assume that every node of the DHT uses an NTP server to synchronize its clock, timestamping certificates with the UTC time ensures that T will be relatively small. Z then computes:

$$t_{avg} = \frac{t_{max} - t_{min}}{2}$$

Algorithm 7 Find next node to repair the trustset

```
N = Halftrustset() //from left or right
for i = |L|/2 − 1 downto 1 do
    if TS = AskFortrustset(N[i]) ≠ ∅ then
        break
    end if
end for
if i == 0 then
    return FAILED
end if
Node = MIN(distance(N[N/2 − 1], TS)
return Node
```

Z considers that the transaction between A and S occurred before the deadline t_0 if

$$\begin{cases} t_{avg} < t_0 \\ OR \\ |t_0 - t_{avg}| \leq \alpha \end{cases}$$

where α is a tolerance threshold which depends on the application. Otherwise, Z can still attest that a transaction did occur between A and S, but that it was not carried out within the required time.

4 Formal Verification

This section deals with the formal verification of the protocol when considering that processes may behave maliciously. This model does not consider corruptions of the trustset; these are handled in our probabilistic analysis in section 5. The objective is to prove that, from the initial configuration of the protocol (*i.e.* before a transaction occurs), only safe situations are reachable (*i.e.* a certificate is produced *iff* the service is completed).

We use Petri nets [20, 23], a mathematical notation suitable for concurrent systems, for the modeling of our protocol .

4.1 A Small Introduction to Symmetric Petri Nets

Among the numerous variants of Petri nets, we choose Symmetric Petri Nets[1] [12] for their capacity to capture symmetries in a system. This is particularly useful for scalable applications, as it allows to store all the configurations of a system in an exponentially more compact way than without considering the symmetries (as shown later in figure 15).

This section gives an informal presentation of Petri nets as a help for the reader for navigating through our formal model presented in section 4.3.

A Petri net is a bipartite graph composed of places (circles) and transitions (rectangles) connected by arcs. Places usually represent resources and may hold tokens (defining the number of occurrences of the corresponding resource). Transitions let the system evolve. They require a given number of input resources (*i.e.* they consume tokens from input places) and produce other ones in output (*i.e.* they produce tokens in output places). Tokens can contain a value.

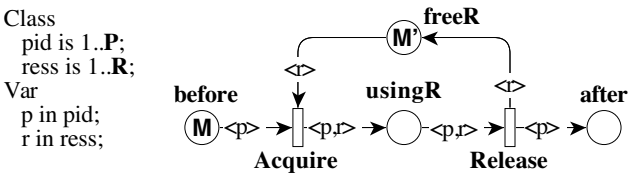

Figure 13: A small Symmetric net example.

Figure 13 shows a small example of a system where processes (known through their id) deal with resources (also referenced by an id). Let us imagine that the initial configuration is

$$M = \langle pid.all \rangle, M' = \langle ress.all \rangle$$

where $M = \langle pid.all \rangle$ denotes one value per possible identity of a process – here, {1, 2, ..., **P**} – and $M' = \langle ress.all \rangle$ one value per possible identity of a resource – here, {1, 2, ..., **R**}.

From this initial marking, only transition **Acquire** can fire (we say it is enabled) and then consume any couple $\langle p, r \rangle$ (from places **before** and **freeR**). When it fires, it also generates a composed token in **usingR**,

[1] In [12], stochastic analysis is also considered but the tool we used only deals with colored aspects.

for instance $\langle 1,1\rangle$. As soon as **Acquire** fired once, there is a concurrency between **Acquire** and **Release**, that are now both enabled (except if $\mathbf{P}=1$ or $\mathbf{R}=1$ because the firing of **Acquire** empty one of its input places).

The Petri net model allows us to generate valid sequences in the system. For the system represented in figure 13, a portion of a possible sequence (odd numbers denote a state in the system and even numbers the firing of a transition) is:

1. **freeR**$=\{\langle 1\rangle,\langle 2\rangle\}$+**before**$=\{\langle 1\rangle,\langle 2\rangle\}$
2. [**Acquire**$(p=1,r=2)>$
3. **freeR**$=\{\langle 2\rangle\}$+**before**$=\{\langle 1\rangle\}$+**usingR**$=\{\langle 1,2\rangle\}$
4. [**Release**$(p=1,r=2)>$
5. **freeR**$=\{\langle 1\rangle,\langle 2\rangle\}$+**before**$=\{\langle 1\rangle\}$+**after**$=\{\langle 1\rangle\}$

The Petri net can then be seen as an automata generator: the reachability graph, where nodes represent a configuration of the system. Numerous types of analysis can be performed on Petri nets. The most popular ones are structural analysis [14] (dealing with the Petri net as a graph) and model checking [13, 18]. We use the second technique in section 4.4 (dealing with the reachability graph) to verify our protocol.

4.2 Abstraction on the System

Of course, the full system is far too complex to be formally modeled as is. In particular, several abstraction had to be made. First, it is useless to model a full DHT with numerous transactions in since we are interested in the behavior of one transaction. So, only one actor (**A** in figure 14), together with its associated leafset offering the service (**S** in figure 14), and the leafset dealing with certification (**C** in figure 14)are needed to be modeled for our study. We exploit here the fact that any group of an actor interacting with the two leafsets is symmetric to any other comparable group in the system.

Then, we assume that the DHT algorithmic provides us with a constant number of nodes in the leafsets, and thus, we do not model the fact that, over the execution of a service and its certification, these nodes may change over time (due to a failure for example).

On top of these abstractions, and to reduce the complexity of the system and its related state space, we finally state several hypotheses that do not alter the behavior of the system with regards to its expected properties (presented in section 4.4):

H_1 the service is reduced to one bidirectional communication (transitions **AstartCS**, **Sperform** and **AendCS**),

H_2 we assume that, since there is no communication problem, an actor waits for n answers (instead of any $\frac{|L|}{2}+1$). This does not change the global behavior since we analyze qualitative properties (*i.e.* we do not count all the possible configurations where the certification is successful),

H_3 We simplify the computation of the trustset of S by A. It is reduced to the emission of n messages to S (and n messages to C from each S in the trustset). For example, A acquiring the trustset of S is modeled by transitions **AreqTS**, **SackTS** and **AgetTS**.

4.3 The Formal Model

This section presents the Symmetric Petri net specifying the behavior of our quasi certification protocol. This model is shown in figure 14 and roughly follows the structure presented in figure 10.

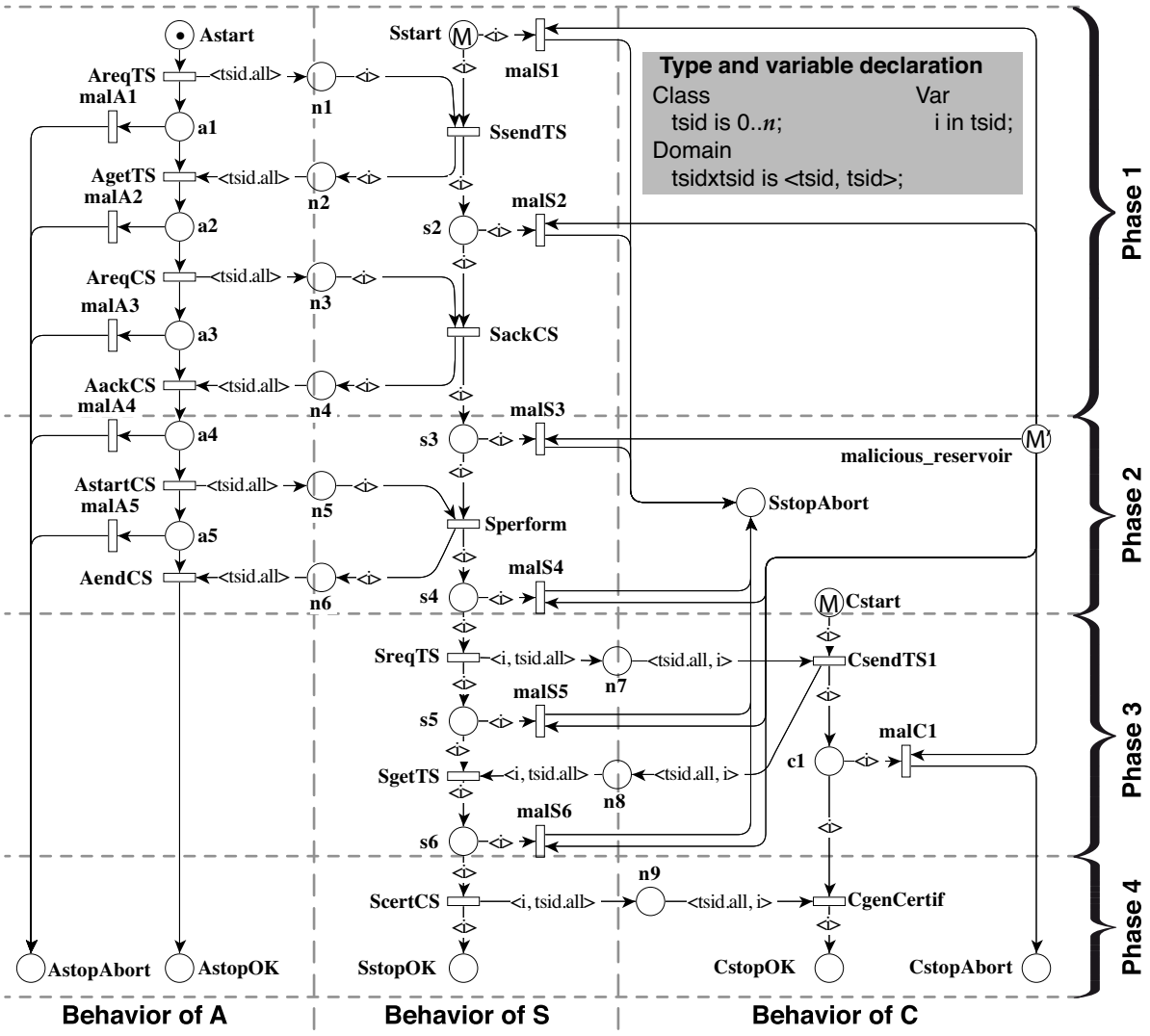

Figure 14: behavioral model of the quasi certification protocol. Columns denote the various "actors" of the system (the actor initiating a transaction and the two leafsets dealing with the service and its certification). Lines correspond to he various phases of the protocol as it was described in figure 4.

In this model, the respective behaviors of client A, server S and certification authority C are displayed in separate columns. Places **Astart**, **Sstart** and **Cstart** are the respective starting points of each class of process.

In the initial configuration:

- place **Astart** holds one uncolored token (•, there is only one A);
- the initial marking of places **Sstart** and **Cstart** is M= $\{\langle tsid.all \rangle\}$ (*i.e.* one distinct token per process in the trustset, every token contains the identity of the process it is associated with);
- place **malicious_reservoir** initially holds M'= $\{m \times \bullet\}$ uncolored tokens, m being the maximum malicious nodes in the system. We compute m from L, the size of the trustset (with 30% of malicious nodes).

Places $\mathbf{a}\langle x \rangle$, $\mathbf{s}\langle x \rangle$ and $\mathbf{c}\langle x \rangle$ repectively represent intermediary states in the protocol of A, S and C. Places $\mathbf{n}\langle x \rangle$ represent message exchanges over the network. Places **AstopAbort**, **SstopAbort** and **CstopAbort** represent "error states" where either the involved process ends when he behaves maliciously, or the fair process eventually ends if no other decision can be taken (*e.g.* the number of expected answers is not reached despite several retries). A fair and reliable client A should end with a token in **AstopOK**. When the service is completed, S-node S should end with a s token in place **SstopOK**. When the certificate is emitted, a C-node C should end with a c token in place **CstopOK**.

Transitions $\mathbf{malA}\langle x \rangle$, $\mathbf{malS}\langle x \rangle$ and $\mathbf{malC}\langle x \rangle$ all lead to the "error state" of the corresponding process. They are fired on one of two conditions:

- either the corresponding actor is fair and it cannot behave differently (*e.g.* firing **malA1** instead of **AgetsTS** because the minimum number of answer cannot be reached),
- or the corresponding actor is malicious and intentionally leaves the protocol there in order to disturb the service.

Other transitions are steps in the protocol.

4.4 Analysis of the Formal Model

We used several tools embedded in both the CPN-AMI environment [22] and the CosyVerif environment [4] to analyze the above specification.

Properties. The properties stated below assert that the protocol behaves appropriately:

$$\begin{aligned} F_{ok}: &\quad |SstopOK| = |L| \wedge |CstopOK| = |L| \\ F_{abort}: &\quad |SstopAbort| > 0 \vee |CstopAbort| > 0 \\ P: &\quad AF(F_{ok} \vee F_{abort})) \end{aligned} \tag{2}$$

$|L|$ corresponds to the size of the trustset. F_{ok} corresponds to a state where a certificate gets issued. F_{abort} corresponds to the situations where something went wrong: no certificate gets issued.

Let us note that, for both F_{ok} and F_{abort}, we are not interested in the status of A. In particular, A may refrain from acknowledging the completion of the service (*.i.e* firing transition **AendCS**); S issues the certificate anyway.

P is the final property to be verified. It is a Computational Temporal Logic (CTL) formula stating that, from the initial configuration of the system, all executions lead to F_{ok} or to F_{abort}.

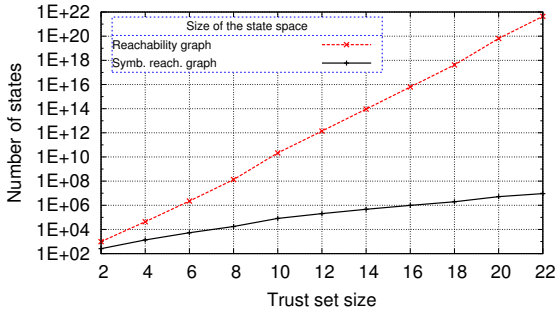

Figure 15: Compared sizes of the reachability graph and the symbolic reachability graph according to the size of the trustset ($|L|$) in the model of figure 14.

Evaluation of these properties. As an indication, the number of configurations computed to verify the Petri net of figure 14 via model checking is presented in figure 15. The complexity of the reachability graph is high (there are 4.4095×10^{21} possible configurations for a trustset of size $|L| = 22$).

We first used the GreatSPN [3] that is a recognized implementation of the Symbolic Reachability Graph. Computation of the reachability graph failed for $|L| = 24$ after 11 hours and 45 minutes[2]. Here, the symmetry-based techniques used in the tool allow to compute the reachability graph and evaluate the properties thanks to this exponential gain obtained by abstracting all equivalent configurations to a permutation of the involved process identifiers. The symbolic reachability graph [12] is exponentially smaller: 9.2838×10^6 symbolic nodes were computed in 7 hours for $|L| = 22$ instead of an estimated number of 4.4095×10^{22} explicit nodes (that could not have been computed in a similar machine).

We conducted a first evaluation of property P successfully on this model. Although we can construct reachability graphs for trustset sizes of up to $|L| = 22$, we restricted our checks to values of P up to $|L| = 6$. Since we deal with a qualitative analysis, larger values would not demonstrate the correctness of our protocol any further. Property P cannot change as $|L|$ increases: this adds new configurations due to the increased number of actors, but in a pattern that remains the same – that of the protocol. This would not have been the case for a quantitative analysis such as a probabilistic one since the computed values then rely on the number of actors.

To confirm our first analysis and to strengthen our validation, we used another technique based on hierarchical decision diagrams [41] to encode the corresponding Petri net, thus considering more symmetries than the ones automatically computed with the previous technique. The combination of symmetry-based analysis with the compact representation of the state space in memory provided by the decision diagrams allowed us to verify (unsurprisingly) property P for $|L| = 32$. Analysis was performed using ITS-Tools [42], that implements this technique.

The techniques exploited by the tools we used for verification (exploitation of symmetries for greatSPN, exploitation of hierarchy and locality for ITS-Tools) is typical of the techniques that scale well in distributed systems where a large part of the complexity comes from interleaving of actions. Moreover, when numerous processes share the same behavior (*e.i.* the same code where the behavior is conducted by local data), the situation is even better. We recommend such techniques for the analysis of distributed asynchronous systems.

5 Evaluation

In this section, we present a probabilistic assessment of ASCENT, and show that it is highly improbable for our quasi certification authority to fail. We also evaluate the message complexity of our solution, and

[2]in fact, at this stage, we reach an implementation limit of the tool (size of a hash table). The required memory when $|L| = 22$ is about 4 Gbytes, far from the 128 Gbytes we had on the machine.

	Probability to fail	
Size of Trusted Set (L)	**DHT**	**CORPS**
8	0.0081	6.25×10^{-6}
16	6.56×10^{-5}	3.9×10^{-11}
32	4.3×10^{-9}	1.52×10^{-21}

Table 2: Probability for GetTrustset to fail

deduce that it is both scalable and efficient.

Let p be the probability for a single node to be malicious, and N the total number of nodes in the Trusted Ring.

5.1 The Gettrustset Algorithm

This section first evaluates the probability of failure of the Gettrustset Algorithm and then assesses its message complexity.

5.1.1 Probability of Failure

The algorithm fails to get the trustset of a given node K_{ROOT} if $\frac{|L|}{2}$ consecutive nodes are malicious in a trustset. This probability is given by

$$P_{=L/2} = p^{\frac{|L|}{2}} \tag{3}$$

As mentioned before, the main reason for using the *CORPS* is to prevent potentially malicious nodes from participating to ASCENT. If we consider a DHT with 30% of malicious nodes, the theoretical limit above which a reputation system collapses, then the *CORPS* can reduce this probability in the trustset to 5%. Table 2 compares the probability of failure for the Gettrustset algorithm when built above a raw DHT to the same probability when built above the *CORPS*. In a system that handles billions of transactions, and even for a large trustset ($|L| = 32$), the algorithm will fail several times if it is implemented directly above a DHT. If it uses the *CORPS*, the probability become so low that the algorithm will likely never fail.

5.1.2 Message complexity

According to [10] the average number of tries required to make a successful routing to any node K_{ROOT} in the Trusted Ring is around 1.5 with a ring of 150 millions of nodes. Equation (3) evaluates around 6.25×10^{-6} the probability of having more than 4 consecutive malicious nodes when using the Diversity Trusted Routing. We can then consider it highly improbable to have to make more than 4 routing attempts to get the trustset of a node near K_{ROOT}. An upper bound for the cost of getting the trustset of K_{ROOT} is given by

$$n = \underbrace{O(log_{2b}(N)) + 4 \times O(log_{2b}(N))}_{First\ and\ Diversity\ Routing} + \underbrace{Q+4}_{Direct\ IP}$$

$$n = 5 \times O(log_{2b}(N)) + Q + 4 = O(log_{2b}(N)) \tag{4}$$

where Q is the maximum number of tries required to get a trustset from a node that belongs to the trustset of K_{ROOT}. The probability of not being able to retrieve a trustset from a node that belongs to the trustset of K_{ROOT} after $Q = 16$ tries with a trustset size $|L| = 16$ is 1.52×10^{-21}. This is highly improbable, so we can consider that $\frac{|L|}{2} = 8$ is a reasonable upper bound for Q.

In the best case, the cost is reduced to $n = O(log(N))$ when the root node K_{ROOT} is honest. Thus, the Gettrustset algorithm easily scales when the size of the Trusted Ring increases.

5.2 The Transaction between A and S

This section first evaluates the probability of failure of a Transaction between A and S and then assesses its message complexity.

5.2.1 Probability of failure

A combination of network failures and/or malicious nodes that choose to remain silent may prevent A from receiving at least $\frac{|L|}{2}+1$ answers from the nodes in S. More precisely, the transaction between A and S can fail for the following reasons:

- S does not respond to A (RequestInit or during the transaction progress)
- S does not send the ACKs to A to terminate the transaction.

In both cases, this corresponds to the probability that more than $L/2$ nodes of S are malicious.

Since S retains the same properties as Pastry's LeafSet, its nodes are numerically close in the trusted ring. The random uniform distribution of nodeIDs operated by Pastry makes it extremely likely that nodes on S will be geographically far on the Internet. For this reason, we consider that each node is malicious independently and uniformly at random with probability p.

Therefore, the probability of $\frac{|L|}{2}+1$ simultaneous IP routing failures or node failures, that is the probability for more than $\frac{|L|}{2}$ answers to not reach A, is very close to zero. The computation of this probability is as follows.

The probability to have exactly k malicious nodes among a set of $L+1$ random nodes is given by the binomial distribution

$$P_{Kmalicious} = \binom{|L|+1}{k} p^k (1-p)^{|L|+1-k} \tag{5}$$

where p is the probability for a single node to be malicious. It follows that the probability of having at most k malicious nodes is

$$P_{\leq k} = \sum_{i=1}^{k} \binom{|L|+1}{k} p^i (1-p)^{|L|+1-i} \tag{6}$$

Therefore, the probability of having more than k malicious nodes in n is given by

$$P_{>k} = P_{\leq n} - P_{\leq k} \tag{7}$$

and the probability that S will not respond is

$$\begin{aligned} P_{>|L|/2} &= P_{\leq |L|+1} - P_{\leq |L|/2} \\ &= \sum_{i=1}^{|L|+1} \binom{L+1}{i} p^i (1-p)^{|L|+1-i} \\ &- \sum_{i=1}^{\frac{|L|}{2}} \binom{|L|+1}{i} p^i (1-p)^{|L|+1-i} \end{aligned} \tag{8}$$

Figure 16 represents the likeliness of dealing with an increasing number k of malicious nodes among a trustset comprising 32 nodes.

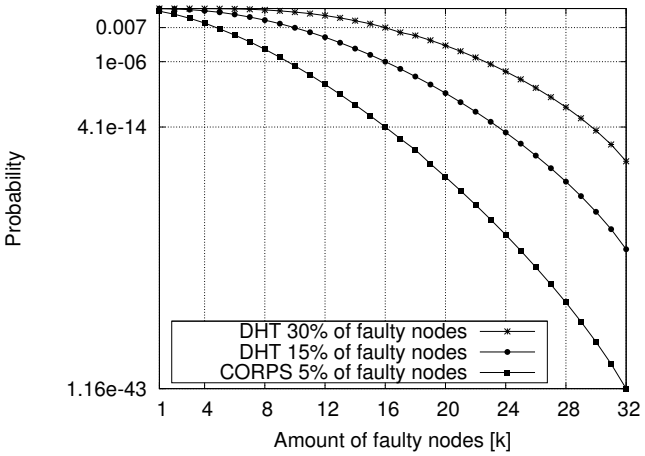

Figure 16: Probability to have more than k malicious nodes

L	**DHT -** $p = 0.3$	**CORPS -** $p = 0.05$
8	0.188	6.64×10^{-5}
16	0.079	6.57×10^{-8}
32	0.016	8.24×10^{-14}

Table 3: Probability of failure of a transaction between A and S. To account for this very rare and highly improbable case, A aborts the transaction if it has not received a sufficient amount of identical answers after a timeout $\triangle t$.

From equation 8 we can deduce the probability that S won't respond to the RequestInit or won't send the final ACKs as

$$\begin{aligned} P_{AS} &= (1 - P_{>L/2})P_{>|L|/2} + P_{>|L|/2} \\ P_{AS} &= 2P_{>|L|/2} - P^2_{>|L|/2} \end{aligned} \tag{9}$$

Table 3 shows the probability of failure of a transaction between A and S as the trustset size $|L|$ increases, and compares a quasi CA built directly above the DHT with one built above the CORPS. We consider in the worst case that malicious nodes represent 30% of all DHT nodes. In this case, the *CORPS* reduces the probability of failure for each node of the trustset to $p = 0.05$.

5.2.2 Message complexity

First, A must get the trustset of S. The associated cost is $n = 5 \times O(log_{2b}(N)) + Q + 4$ (see section 5.1.2).

The number of messages inherent to the transaction itself is given by

$$n = \underbrace{2(|L|+1)}_{Init} + \underbrace{r(|L|+1)}_{Data} + \underbrace{|L|+1}_{ACKs}$$

$$n = (r+3)(|L|+1)$$

r corresponds to the number of data messages sent by A to S, and fully depends on the transaction. The total cost is then

$$n_{total} = 5 \times O(log_{2b}(N)) + Q + 4 + (r+3)(|L|+1)$$

		P_1		P_2		P_{total}	
	$\|L\|$	**DHT**	**CORPS**	**DHT**	**CORPS**	**DHT**	**CORPS**
Case 1	8	0.00243	3.125×10^{-7}	0.19410	3.72×10^{-4}	0.1960	3.72×10^{-4}
	16	1.96×10^{-5}	1.95×10^{12}	0.07435	3.5×10^{-7}	0.07437	3.5×10^{-7}
	32	1.29×10^{-9}	7.63×10^{-23}	0.01384	4.24×10^{-13}	0.0138	4.24×10^{-13}
Case 2	8	0.027	1.25×10^{-4}	0.194	3.71×10^{-4}	0.216	4.97×10^{-4}
	16	2.19×10^{-4}	7.81×10^{-10}	0.0743	3.49×10^{-7}	0.0745	3.5×10^{-7}
	32	1.43×10^{-8}	3.05×10^{-20}	0.0138	4.24×10^{-13}	0.0138	4.24×10^{-13}

Table 4: Probability of failure for the certificates logs maintenance

The total cost only depends on the size L of the trustset, which is a constant, and $O(log(N))$. In the best case,

$$n_{total} = O(log(N)) + (r+3)(|L|+1)$$

Therefore, the cost of the transaction between A and service S is scalable when the size N of the TrustedRing increases.

5.3 Lazy Log Coherency Maintenance

Case 1 *- A new node X enters the trustset.* The maintenance of the certificate logs fails if X encounters $\frac{L}{2}$ erroneous results consecutively when building the node interval to retrieve the logs, or if more than $\frac{L}{2}$ nodes are malicious and make it impossible to obtain at least $\frac{L}{2}+1$ identical answers.

The probability $P_1$of having $\frac{L}{2}$ consecutive malicious nodes is given by equation (3)

$$P_1 = p^{|L|/2}$$

The probability of not being able to retrieve at least $\frac{|L|}{2}+1$ identical answers is given by equation (8)

$$P_2 = P_{>|L|/2}$$

Hence the total probability for a maintenance operation to fail is

$$P_{total} = P_1 + (1-P_1)P_2$$

Case 2 *- A node is leaving the trustset.* The certificate logs cannot be repaired if the node cannot repair its trustset (ie. add a node on the left or right side), or if it is impossible to find at least $\frac{|L|}{2}+1$ identical answers upon downloading the logs.

A node cannot repair its trustset if $\frac{|L|}{2}-1$ consecutive nodes are malicious with a probability of

$$P_1 = p^{|L|/2-1}$$

and the download of the logs fails with a probability of

$$P_2 = P_{>|L|/2}$$

Hence the total probability of failure is

$$P_{total} = P_1 + (1-P_1)P_2$$

Table 4 gives the results for each probability for a varying trustset size of $|L| = \{8, 16, 32\}$ nodes.

5.4 Certificate Generation

This section first evaluates the probability of failure of a Certificate Generation and then assesses its message complexity.

5.4.1 Probability of failure

The generation of a certificate fails if S fails to send a sufficient amount of ACKs to C, or if a sufficient amount of nodes from C do not attest that the certificate exists. S or C will fail if at least more than $L/2$ of their nodes are malicious. We have

$$P_{>|L|/2} = P_{\leq |L|+1} - P_{\leq |L|/2}$$

Hence, the probability P_{CF} of failure of the certification is given by

$$P_{CF} = 1 - (1 - P_{>|L|/2})^2$$

$$\begin{aligned} P_{CF} &= 1 - \left(1 - \sum_{i=1}^{|L|+1} \binom{|L|+1}{i} p^i (1-p)^{|L|+1-i} \right. \\ &\quad \left. + \sum_{i=1}^{|L|/2} \binom{|L|+1}{i} p^i (1-p)^{|L|+1-i} \right)^2 \end{aligned} \tag{10}$$

Table 3 also shows the evaluation of the probability of failure of the certification in ASCENT according to the size L of the trustset, and compares the raw DHT construct with the *CORPS* construct. Indeed, the certificate generation corresponds to the transaction itself.

5.4.2 Message complexity

To generate a certificate, every node of S must first acquire the trustset that corresponds to C, then the public keys of each node of C, and then send a notification (ACK) to each node of C. Let $n_{TRUSTED}$ be the cost of getting the trustset of C. The total cost is then given by

$$n_{total} = \underbrace{(|L|+1) n_{TRUSTED}}_{Get trust set} + \underbrace{2(|L|+1)^2 + (|L|+1)^2}_{Public\,Keys + ACK}$$

$$n_{total} = (|L|+1)\,(3(|L|+1) + 5 \times O(log_{2b}(N)) + Q + 4)$$

and in the best case

$$n_{total} = (|L|+1)\,(3(|L|+1) + O(log(N)))$$

This complexity confirms that the certification protocol remains scalable when the size N of the Trusted Ring increases.

6 Discussion and Related Work

Similarly to all scientific contributions, our approach may raise several debatable issues such as its overall usefulness. In this section we try to address some of these issues, namely:

- how reliable can our solution be in practical terms, and how well it compares to centralized certification authorities,

- what kind of applications may benefit from our approach,
- how interchangeable are the building blocks we use to design our solution.

6.1 Scope of Applicability

Many applications may benefit from a scalable one-side certification service. We propose three examples of applications, explain in every case why a centralized solution is not fully satisfactory, and give an overview of how they might be implemented over ASCENT.

Digital filing for tax purposes. Many governments expect their citizens to file a revenue declaration form every year; for instance in the USA this process is called tax returns. Usually there is an annual deadline beyond which filings will incur substantial extra fees. Digital filing is becoming ever more popular as it saves significant amounts of money and time both for the taxpayers and for the collector. However the servers that process filings must handle huge peak workloads as the deadline approaches.

Since digital declaration has been validated in France, not a year has gone by without a total server crash on the day of the deadline; some filers have faced gruesome consequences because of this. ASCENT offers a simple load balancing solution for this application. Since the data size of a declaration is very small, it is perfectly affordable to store the encrypted declaration directly with the timestamped certificate. The scalable service will remain available despite intense activity from last minute filers, and the taxation authority may then process declarations in its own time well after the deadline.

Certified e-mail. This application represents an extension of the digital declaration of revenues. In many countries, certified mail receipts are accepted as legal evidence for sending or the receipt of a particular piece of mail on a particular date. As a consequence many postal delivery companies, be they private or public, offer certified mail services. Certified electronic mail (CEM) is by no means a new research topic [19, 30, 7]. But to the best of our knowledge our solution is the first that prevents repudiation from both the sender and the receiver simultaneously without requiring a centralized third party infrastructure.

If both parties acquire an ASCENT certificate for the same data exchange, then the encrypted data can be stored either on the set S of server nodes, or on the set C of certification nodes, or even on another set of nodes entirely. The important contribution is that at any moment anyone (the sender, the receiver, or any other independent party) can check whether an exchange did truly occur, and around what time if it did.

Online game refereeing. Online games regroup large numbers of players that interact in the same virtual universe. It is not uncommon for players to tamper with the game commands in order to gain an unfair advantage against other players. The current solution adopted by the software game industry is to deploy potent server architectures to centralize the refereeing process.

Yet in reality, Massively Multiplayer Online Games (MMOGs) do not live up to their name. For instance in the case of *World of Warcraft*, every server imposes a limit on the number of simultaneous players that is evaluated by unofficial sources around $15,000$. Besides their huge maintenance cost, central servers are not failproof and hacks have occurred in the past.

Distributing the refereeing process over the player nodes would allow to cut the costs and to remove the limit on the size of the virtual universe. For instance, [44] proposes such a distributed refereeing architecture. It is also based on a reputation system but it cannot detect complex cheating behaviors such as *gold farming*, where some players spend their time online gathering virtual resources in order to sell them for profit in the real world. Every game action could lead to a certificate in ASCENT, and the logs would allow to analyze whether a specific player behaves honestly over long periods of time.

6.2 Reliability in Practical Terms

The reliability of a centralized certification authority depends mostly on the supporting host: its availability, its securedness and its capacity to handle significant amounts of concurrent requests. In practice this entails that a centralized CA is not foolproof: it is as trustworthy as the (group of) server(s) it runs on. Well-established centralized CAs have been corrupted before. For instance, in April 2011 the Sony Playstation Network servers [35] got hacked and the attackers retrieved vital information. The very same happened repeatedly to VeriSign [28] in 2010.

In a decentralized approach, the main additional issue is that the supporting hosts are not accountable for their decisions and therefore cannot be trusted. That is precisely the point of our solution: a reputation system makes the CA nodes accountable. Furthermore, we build sets of accountable nodes selected for their trustworthiness and enforce an agreement among these sets.

The secondary second issue linked to decentralization is that the distributed algorithm may be faulty. We think the formal verification we conducted in Section 4 is enough to set this issue aside.

Overall the resulting probability of a false positive, where a decision made by malicious nodes is adopted, is of the order of 10^{-14}. This value is way lower than any we have encountered in the literature about decentralized CAs. It is obtained with a trustset containing 32 nodes, and we believe it guarantees that in practice no false positive will occur during a system run. We also showed in Section 5 that a trustset size of 32 remains very affordable in terms of communications overhead.

Another possibility of failure is the loss of a certificate, which is evaluated to be of the order of 10^{-13} (see table 4). For the reasons mentioned above, chances are that no certificate loss will ever happen in practice.

For instance, let us consider Digital filing for tax purposes applicable to France (about 36.5 million revenue declarations every year). Such values would lead to issue a wrong tax certificate once every 5132 years or to lose a tax certificate once every 997 years. This makes the system theoretically quasi reliable.

6.3 Portability

We wish to emphasize that ASCENT is a modular, and therefore portable solution. Although our prototype relies on Pastry as its DHT and WTR as its reputation system, our approach can rely on other building block implementations.

For example it is very simple to switch the underlying DHT from Pastry to Chord [39], as it maintains a structure called the successors list that is equivalent to the Pastry leafset and to the CORPS trustset. Each node in Chord also maintains a routing table equivalent called the finger table.

Similarly, any reputation system such as PowerTrust [46] that matches the properties described in Subsection 2.4 will do. The same principles apply to the CORPS membership system, as discussed in [36].

6.4 Security Issues

As the protocol of ASCENT uses a public key exchange during the initialization of each transaction, an attacker may attempt a *Man In The Middle* (MITM) attack. A MITM attack requires to intercept publics keys from both sides of the transaction and replace them by the attacker's own key in order to be able to decrypt and modify further traffic during the transaction. In ASCENT, such an attack only makes sense between service S and the certification layer C, or between client A and service S.

6.4.1 MITM attack between S and C

To carry out a MITM attack between service S and the certification layer C it is necessary to intercept at least $2 \times \frac{|L|}{2} + 1$ publics keys between unique nodes S_i and C_i in order to succeed. Indeed, you must be able to send fake messages to at least $\frac{|L|}{2} + 1$ node of C in order to make them take a wrong decision using the local consensus. Intercepting the keys requires to be on the routing paths (at IP level) between S and C.

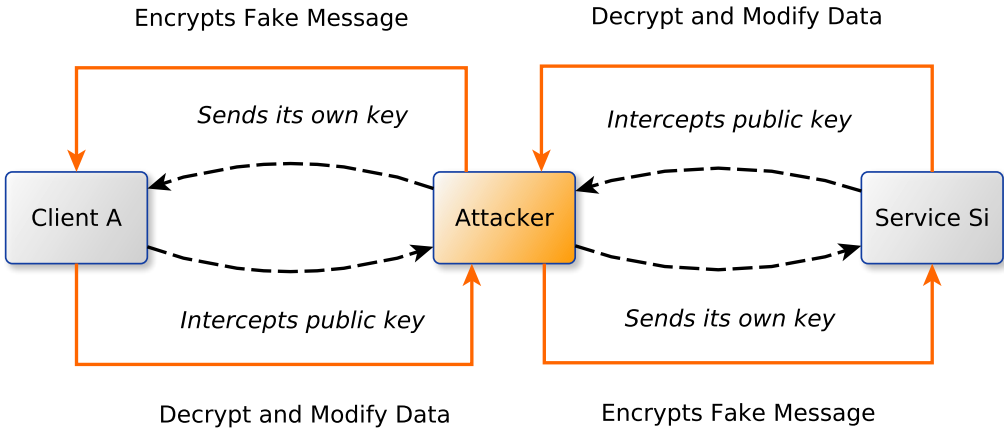

Figure 17: Man In The Middle attacke between A and S

- **In the best case**: the attacker has to control at least $\frac{|L|}{2}+1$ nodes of S and $\frac{|L|}{2}+1$ nodes of C, that is $2\times\frac{|L|}{2}+1$ nodes. However, by construction, and due to the properties of the DHT (and the Trusted Ring), these nodes have very different IP addresses and in practice they belong to $2\times\frac{|L|}{2}+1$ different IP subnetworks.
- **In the worst case**: the attacker will have to control $2\times\left(\frac{|L|}{2}+1\right)^2$ paths on the Internet to control the traffic between S and C. This is because the routing path between a node S_i and a node C_i at IP level is not necessarily the same as the return routing path.

Provided that the size $|L|$ of a leafset/trustset is reasonably high, a MITM attack between service S and the certification layer C is theoretically possible, but would be very difficult to carry out in practice due to the number of nodes or routing paths that an attacker would have to control. This is especially true because of the physical remoteness of DHT nodes in the same leafset; indeed, by construction they are very likely to be in different IP subnetworks.

6.4.2 MITM attack between A and S

The MITM attack between A and S, illustrated in figure 17, consists in intercepting public keys between the client A and at least $\frac{|L|}{2}+1$ nodes of S in order to generate fake messages. Controlling $\frac{|L|}{2}+1$ nodes of S is a difficult task, as explained above, because of the property of physical remoteness for nodes in a leafset. The most effective way to carry out a MITM attack is then to control node A or the local area network of A. Avoiding an attacker to control node A completely depends on the security level of A (software updates, anti-virus and anti-malware), as well as the security level of its local network. This security fully depends on the administration of node A and the associated local area network, and we have no way to control this. The possibility of a MITM attack in a client server model would be exactly the same.

The attacker can also try to control the routing paths between A and service S in order to intercept the public keys. This consists in potentially controlling $2\times\frac{|L|}{2}+1$ different routing paths, which is theoretically possible but practically infeasible considering that the $\frac{|L|}{2}+1$ nodes of S very likely belong to different IP subnetworks.

Moreover, using the Diffie-Hellman [34] public key exchange algorithm to compute a secret shared symmetric key won't solve the problem as the Diffie-Hellman algorithm is also vulnerable to MITM attacks [1].

A physical distribution of a private shared symmetric key would be the only secure way to prevent MITM attacks, but this is an impossible task in a large scale distributed system. Nevertheless, the number of nodes and their different locations, or the number of routings paths that an attacker would have to control

in ASCENT makes this attack very improbable to carry out successfully. For this reason we consider that, even if it is theoretically possible, a MITM attack against ASCENT is practically infeasible.

6.5 Related Work

A considerable amount of former research addresses the issue of creating a distributed certification system.

Some solutions, such as [5] and [6], are hierarchic: when a node suspects a certificate of coming from a faulty principal, it can refer to one or several root CAs that are hosted on fully trusted servers. The main advantage of root CAs is that the certificates they deliver are both reliable and verifiable. However, the trusted server nodes remain single points of failure and limit the scalability of these approaches.

[2] and [40] attempt to extend the scalability of the hierarchic approaches by adding DHT support. Both solutions use Chord [39] to store copies of the certificates issued by a fully trusted CA. While this partially mitigates the formation of a bottleneck around the CA, the reconstruction and validation of a certificate from its copies requires a byzantine consensus which is extremely costly over an asynchronous network.

Since our solution aims for scalability, fully decentralized solutions are better suited for comparison. In particular, distributed public-key infrastructures (PKIs) constitute a potent way of delivering certificates. [16] identifies three main sub-classes of decentralized PKIs : *web of trust*, *statistical*, and *hybrid* approaches.

Web of trust approaches rely on transitive trust models to coalesce nodes into chains of certification. For instance, KeyChains [29] combines an unstructured P2P lookup protocol with a PGP-like *web of trust* model to build certificate chains. However, the strength of a chain is determined by the weakest link. This makes the simple transitive trust assumed by KeyChains highly vulnerable, and thus unreliable.

Statistical approaches form quorums made of multiple random and thus presumably independent peers. As in ASCENT, signing a certificate thus requires the cooperation of a minimal number of honest nodes. Many statistical solutions that scale, such as [24], [45] and [16], split private keys among the nodes and maintain virtual distributed search trees on top of structured P2P overlays to index the partial keys. The main problem with quorum-based approaches that disregard node reputation is that they exhibit a high probability of certification failure.

Ω [33], another statistical solution that scales, relies on state-machine replication and on byzantine agreement to provide a powerful and extensive PKI service. ASCENT also relies on deterministic server replication, but there are important differences. Conversely to our approach where the client synchronizes the exchanges with the servers, Ω requires *atomic multicast* which is impossible to implement with unreliable failure detectors over an asynchronous network [21]. Also, Ω manages the groups of server replicas as part of its agreement protocol; this makes it vulnerable to denial-of-service attacks, particularly those lead by either mobile or Sybil adversaries. ASCENT delegates the group management to the *CORPS* [36], which is designed to handle highly volatile groups of reputable nodes over a DHT.

Hybrid approaches [32, 31] form weighted quorums dependent on the relative trust that is acquired among peers by exchanging public key information. Despite the multiplication of alternate trust paths, guaranteeing the emergence of a fully trustworthy path remains impossible since a single malicious node can invalidate one or more paths. Besides, the authentication metrics used to quantify the reliability of a path induce a costly consumption of network and computation resources on every node.

Initially aimed at decentralizing PKIs, ***secure logging systems*** [26, 27] also allow to store and access persistent proofs of transactions. They generally involve a trusted service that supports the issuance of certificates by domain owners and another service that logs these certificates. An auditing service checks logs periodically and/or upon client request to reinforce the security of the overall architecture. The main difference with our approach is the definition of both the adversary and the Trusted Computing Base in the system model; for instance ARPKI [8] can tolerate much stronger adversaries capable of compromising up to n-1 server nodes. However our model assumes a network that is both dynamic and scalable: servers may join or leave at any point and are never expected to constitute a static set, and no operation ever requires global synchronization. Our solution trades off the strength of a fail-proof protocol for these network properties.

7 Conclusion

In this paper, we present ASCENT (A Scalable Certification through Endorsement by Nodes that are Trustworthy), a distributed *quasi-certification* entity that scales. This entity (called C) ensures that a given client A has performed a given service S and may provide a certificate proof of this action. We demonstrate that our solution scales out and remains reliable even in the presence of arbitrary failures, and can withstand a significant degree of maliciousness. Our certification authority also guarantees a timestamp for recorded actions, thus allowing checks for precedence.

A major strength of this work is to combine formal methods (here, model checking on colored Petri nets) together with a probabilistic analysis. We thus (i) demonstrate that our protocols are sound and do not generate false positives in the absence of failures, and (ii) show that the probability of failure or interference due to a malicious node is low to the extreme (10^{-13}). If applied to Digital filing for tax in France, the most frequent problem would arise once every 997 years (see section 6.2).

Further validation of our work could be achieved by measuring its efficiency on a full-fledged experimental deployment of our architecture.

Acknowledgements

The work presented in this paper was partially funded by a CNRS/CONICYT grant number 25289.

We also thank Maricel Nuñez for her participation in the elaboration of a preliminary version of some of the algorithms presented in this paper.

References

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 5–17, New York, NY, USA, 2015. ACM, New York.

[2] Sameer Ajmani, Dwaine E. Clarke, Chuang-Hue Moh, and Steven Richman. Conchord: Cooperative sdsi certificate storage and name resolution. In *proceedings of the International workshop on Peer-To-Peer Systems*, volume 2429, pages 141–154, Cambridge, MA, USA, 7-8 March 2002. Springer, Berlin.

[3] Elvio Gilberto Amparore, Marco Beccuti, and Susanna Donatelli. (stochastic) model checking in greatspn. In *Application and Theory of Petri Nets and Concurrency - 35th International Conference – PETRI NETS. Proceedings*, volume 8489 of *Lecture Notes in Computer Science*, pages 354–363, Tunis, Tunisia, 2014. Springer, Berlin.

[4] E. André, Y. Lembachar, L. Petrucci, F. Hulin-Hubard, A. Linard, L-M. Hillah, and F. Kordon. *CosyVerif*: an Open Source Extensible Verification Environment. In *18th IEEE International Conference on Engineering of Complex Computer Systems – ICECCS*, pages 33–36, Singapore, July 2013. IEEE Computer Society.

[5] Mohammad A. Asmaran and Sulieman Bani-Ahmed. A new mechanism for evaluating subordinate and root certification authorities. *International Journal of Research and Reviews in Computer Science (IJRRCS)*, 2(4):977–982, 2011.

[6] Giuseppe Ateniese, Breno de Medeiros, and Michael T. Goodrich. Tricert: A distributed certified e-mail scheme. In *Network and Distributed System Security Symposium (NDSS)*, pages 47–56, San Diego, California, USA, 8-9 February 2001. Internet Society, Virginia.

[7] Alireza Bahreman and Doug Tygar. Certified electronic mail. In *Symposium on Network and Distributed Systems Security*, pages 3–19. Internet Society, Virginia, February 1994.

[8] David A. Basin, Cas J. F. Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: attack resilient public-key infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale*, pages 382–393, Scottsdale, AZ, USA, 3-7 November 2014. ACM, New York.

[9] Xavier Bonnaire and Erika Rosas. A critical analysis of latest advances in building trusted p2p networks using reputation systems. In *WISE'07: Proceedings of the 8th International Conference on Web Information Systems Engineering*, pages 130–141, Nancy, France, 3-6 December 2007. Springer, Berlin.

[10] Xavier Bonnaire and Erika Rosas. WTR: a reputation metric for distributed hash tables based on a risk and credibility factor. *J. Comput. Sci. Technol.*, 24(5):844–854, September 2009.

[11] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 299–314, New York, NY, USA, 2002. ACM, New York.

[12] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. Stochastic well-formed coloured nets for symmetric modelling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, 1993.

[13] E.M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, Cambridge, MA, 1999.

[14] J.-M. Colom, E. Teruel ad M. Silva, and S. Haddad. *Structural Methods*, chapter 15. Springer, Berlin, 2003.

[15] George Danezis, Chris Lesniewski-Laas, M. Frans Kaashoek, and Ross Anderson. Sybil-resistant dht routing. In *ESORICS 2005: 10th European Symposium on Research in Computer Security*, pages 305–318, Milan, Italy, 12-14 September 2005. Springer, Berlin.

[16] A. Datta, M. Hauswirth, and K. Aberer. Beyond "web of trust": enabling p2p e-commerce. In *IEEE International Conference on E-Commerce*, pages 303–312, Newport Beach, California, USA, 24-27 June 2003. IEEE, New York.

[17] John R. Douceur. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, 7-8 March 2002. Springer, Berlin.

[18] C. Dutheillet, I. Vernier-Mounier, J-M. Ilié, and D. Poitrenaud. *State-Space-Based Methods and Model Checking*, chapter 14. Springer, Berlin, 2003.

[19] Josep Lluis Ferrer-Gomilla, Jose A. Onieva, Magdalena Payeras, and Javier Lopez. Certified electronic mail: Properties revisited. *Computers & Security*, 29(2):167 – 179, 2010.

[20] C. Girault and R. Valk. *Petri Nets for Systems Engineering*. Springer, Berlin, 2003.

[21] Rachid Guerraoui. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254:297–316, 2001.

[22] A. Hamez, L. Hillah, F. Kordon, A. Linard, E. Paviot-Adet, X. Renault, and Y. Thierry-Mieg. New features in CPN-AMI 3 : focusing on the analysis of complex distributed systems. In *6th International Conference on Application of Concurrency to System Design (ACSD)*, pages 273–275, Turku, Finland, 2006. IEEE, New York.

[23] K. Jensen and L. Kristensen. *Coloured Petri Nets : Modelling and Validation of Concurrent Systems*. Springer, Berlin, 2009.

[24] François Lesueur, Ludovic Mé, and Valérie Viet Triem Tong. An efficient distributed PKI for structured P2P networks. In *9th International Conference on Peer-to-Peer Computing (P2P)*, pages 1–10, Seattle, Washington, USA, 9-11 September 2009. IEEE, New York.

[25] Chu-Hsing Lin, Chen-Yu Lee, Yi-Shiung Yeh, Hung-Sheng Chien, and Shih-Pei Chien. Generalized secure hash algorithm: SHA-X. In *2011 IEEE EUROCON - International Conference on Computer as a Tool (EUROCON)*, pages 1–4, Lisbon, Portugal, 27-28 April 2011. IEEE, New York.

[26] Di Ma and Gene Tsudik. A new approach to secure logging. *ACM Transactions On Storage*, 5(1):2:1–2:21, 2009.

[27] Stephanos Matsumoto, Pawel Szalachowski, and Adrian Perrig. Deployment challenges in log-based PKI enhancements. In *Proceedings of the Eighth European Workshop on System Security, EuroSec*, pages 1:1–1:7, Bordeaux, France, April 2015. ACM, New York.

[28] Joseph Menn. Key internet operator verisign hit by hackers. http://www.reuters.com/article/2012/02/02/us-hacking-verisign-idUSTRE8110Z820120202, 2012. Reuters.

[29] Ruggero Morselli, Bobby Bhattacharjee, Jonathan Katz, and Michael A. Marsh. Exploiting approximate transitivity of trust. In *International Conference on Broadband Communications, Networks, and Systems (BROADNETS)*, pages 515–524, Raleigh, North Carolina, USA, 10-14 September 2007. IEEE, New York.

[30] Rolf Oppliger. Providing certified mail services on the internet. *Security Privacy, IEEE*, 5(1):16 –22, jan.-feb. 2007.

[31] Vivek Pathak and Liviu Iftode. Byzantine fault tolerant public key authentication in peer-to-peer systems. *Computer Networks*, 50(4):579–596, 2006.

[32] M. Reiter and S. Stubblebine. Authentication metric analysis and design. *ACM Transactions on Information and System Security*, 2(2), 1999.

[33] Michael K. Reiter, Matthew K. Franklin, John B. Lacy, and Rebecca N. Wright. The omega key management service. *Journal of Computer Security*, 4(4):267–288, 1996.

[34] Eric Rescorla. Diffie-Hellman Key Agreement Method. RFC 2631, June 1999.

[35] Shane Richmond and Christopher Williams. Millions of internet users hit by massive sony playstation data theft. http://www.telegraph.co.uk/technology/news/8475728/ Millions-of-internet-users-hit-by-massive-Sony-PlayStation-data-theft.html, 2011. The Telegraph.

[36] Erika Rosas, Olivier Marin, and Xavier Bonnaire. CORPS: building a community of reputable PeerS in distributed hash tables. *The Computer Journal*, 54(10):1721–1735, 2011.

[37] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, 12-16 November 2001. Springer, Berlin.

[38] Gustavus J. Simmons. Symmetric and asymmetric encryption. *ACM Comput. Surv.*, 11(4):305–330, December 1979.

[39] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, New York, USA, 27-31 August 2001. ACM, New York.

[40] Anuchart Tassanaviboon and Guang Gong. A framework toward a self-organizing and self-healing certificate authority group in a content addressable network. In *6th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 614–621, Niagara Falls, Canada, 11-13 October 2010. IEEE, New York.

[41] Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. Hierarchical Set Decision Diagrams and Regular Models. In *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*, pages 1–15, York, UK, March 2009. Springer, Berlin.

[42] Yann Thierry-Mieg. Symbolic model-checking using its-tools. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 231–237. Springer, Berlin, 2015.

[43] United Nations Department of Economic and Social Affairs. *United Nations E-Government Survey 2012, E-Government for the People*. United Nations, New York, USA, 2012.

[44] Maxime Véron, Olivier Marin, Sébastien Monnet, and Zahia Guessoum. Towards a scalable refereeing system for online gaming. *Multimedia Systems*, 20(5):579–593, 2014.

[45] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. Coca: A secure distributed online certification authority. *ACM Trans. Comput. Syst.*, 20(4):329–368, November 2002.

[46] Runfang Zhou and Fellow-Kai Hwang. Powertrust: A robust and scalable reputation system for trusted peer-to-peer computing. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):460–473, April 2007.