Secure token passing at application level

Augusto Ciuffoletti
INFN/CNAF - Bologna

Abstract

We introduce an application level implementation of a token passing operation. After an introduction that explains the conceptual principles, we describe exhaustively the state machines that implement our solution. The security requirements are carefully considered, since the token is a sensitive resource, but without introducing scalability limits.

We aim at a general purpose token passing primitive: we do not enter the domain of the distributed coordination algorithms that can be implemented using the proposed operation. We discuss its practical utilization, and we indicate as primary application area the coordination of servers in a distributed infrastructure: this matches service oriented Grids as well as other emerging paradigms.

Its usage is explained with a simplified use case. A working prototype exists, and we report about experimental results that confirm our claims concerning performance.

Key words: token passing, random walk, group membership.

1. Introduction

The utilization of a token passing operation is often considered in the design of distributed algorithms: for instance, this occurs for the many members of the "self-stabilization" family, originated from an idea of E.J. Dijkstra [5] addressing mutual exclusion. We start from the observation that the design of a token passing protocol at the OSI Application Layer is not considered in the literature. Such option introduces issues that are not found in the case of an OSI Link Layer implementation, which is instead quite frequent in the literature (including FDDI (ANSI X3T9.5 and X3T12) and Token Ring (IEEE 802.5) protocols, and other original issues like in [4]).

Email address: augusto@di.unipi.it (Augusto Ciuffoletti).
[1] This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

In this paper we introduce the implementation of a token passing protocol in the Application Layer, considering its design as orthogonal to the design of the user application that makes use of the token. The result is a generic token passing protocol that may run in the user space. Our intention is to provide a building block for distributed computing infrastructures.

A token-based coordination is considered appropriate for long lived applications that have intermittent access to the shared resource. The first indication comes from the fact that the token return time is relevant, orders of magnitude higher than a typical round-trip time, while the second is bound to the fact that access to the resource is granted only in presence of the token.

In an environment considered typical for current technology, with hundreds of host applications and a resource utilization lasting tens of milliseconds, we indicate a typical return time in the range of tens of seconds. This parameter should be considered as an order of magnitude for the minimum lifetime of an application, and of the time between successive accesses to the resource.

These figures are adequate for the generality of the *server* applications in distributed infrastructures: these are long lived applications, that may need to access a shared resource for synchronization or accounting reasons. For instance, the work presented in this paper is considered for the maintenance of a distributed directory in a Grid network monitoring architecture [3].

However, the interest for a token based coordination is not restricted to distributed infrastructures only: in [2] we introduce a case study that coordinates network utilization among the clients of a stream multicast.

In the next section we describe the functional requirements of a token passing protocol; in the successive section we introduce and analyze an original protocol, using state diagrams to describe its internals. A simple use case gives the feeling of its usage. Finally we report about a prototype implementation and performance.

2. Requirements for an Application level token passing protocol

The first problem that needs to be addressed when designing an application whose coordination is based on a token is the rule used for token routing: such rule may reflect a configurable topology, like a ring [10], or a tree [14]. This mostly depends on the properties required by the application that makes use of the *privilege* attached to the token, which may be more or less sensitive to fairness issues, or to token uniqueness.

Since our discussion wants to abstract from a specific application, we do not make assumptions on a specific token routing. In principle, at a given time the token can be forwarded to any member in the network of agents that share the token.

Given that the maintenance of a global knowledge of such network would collide with basic requirements of scalability and reliability, we are in the case that a member may receive the token from an unknown peer. Since an intruder may take advantage of this to introduce a forged token, a token passing protocol at Application level must restrict token circulation within a trusted membership; the inappropriate or malicious utilization of the token by a trusted member falls outside the scope of this paper. We consider the use of public/private key-pairs for each member, delivered to a trusted member by a Certification Authority (CA).

Such schema is complicated in the case, increasingly frequent in practice, that the token may cross administrative boundaries. Federated identity management is a topic of active research and we do not propose an ad-hoc solution. Recently the Shibboleth protocol (based on the SAML OASIS protocol), has received a special attention, and VOMS has shown to be a practical solution. An interesting reading on the subject is [7].

The token passing protocol should ensure that failures do not damage token consistency: token duplication and token loss should seldom occur. This apparently indicates the TCP protocol [16] as a good candidate to support token passing. However, at a closer look it exhibits several drawbacks: functionalities like connection setup, window sizing etc. are useless for our purpose, but are resource consuming and are exposed to threats. The Internet transport protocol that, at second sight, appears as more appropriate for the task of carrying a token is UDP [15]: but then fault tolerance requirements must be implemented within the token passing protocol.

We distinguish between network failures, which are rather frequent, and application failures. We do not exclude that a token is lost when an application instance silently fails while holding it: for this reason we understand the existence of a token recovery algorithm. This algorithm falls outside the scope of our investigation, being strongly influenced by token routing rules, which we do not consider. However, since it evaluates a global property (the absence of the token) using local information, we cannot assume it is deterministically successful: hence the need for a rule to remove spurious tokens.

We consider that token latency degrades the performance of the user application: while the token is *in transit* no application is enabled to use the related privilege. In addition, token fairness properties are negatively affected by a relevant jitter of the token exchange protocol. Therefore we consider that token latency must be kept under consideration in the design of the token passing protocol: token latency should approximate delay times, and exhibit a small dispersion.

The relevant conclusions of this section, which are used in next sections, are the *requirements* for a general purpose token passing operation at the application level:

(i) token loss occurs with a probability that is negligible with respect to packet loss;
(ii) token duplication is excluded;
(iii) token passing latency distribution approximates packet delay with a small jitter.
(iv) each token passing operation is validated;

3. Design of a token passing operation

Our implementation of the token passing operation consists of a sequence of datagram exchanges (see figure 1) between the *sender* and the *receiver* of the token, similar to what happens when a TCP connection is established. A token passing operation is identified by a *session identifier*, and each token is characterized by a unique *token identifier*.

Each datagram in the token exchange protocol is a data object containing:

– a *token identifier*;
– a *session identifier*: this is generated by the token sender, by incrementing the *id* of the session with which it received the token;
– a *type* tag for the datagram;
– a *timestamp* field, which may contain random data;
– a *signature*, optionally replaced by a *certificate*, as detailed below;

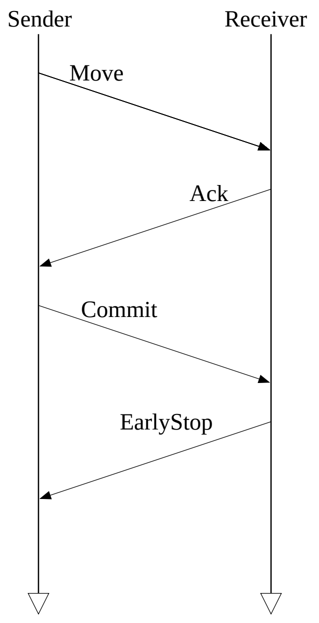

Fig. 1. Datagrams during token passing: *Move*, *Ack*, *Commit* and *EarlyStop* labels are used in protocol description

The first datagram is sent between the source and a well known port at the destination. It announces the intention to move the token and is indicated as a *move* datagram. The datagram contains a DSA signature [12], obtained using the private key of the sender on the content of the datagram and on the IP address of the destination. A certain *move* datagram is therefore bound to a given (source,destination) pair. It is relevant for security issues that the same datagram, received by a different destination or with a different source is immediately detected as illegal.

After sending a *move* datagram, the sender starts waiting for a *ack* datagram before a timeout expires. In case the timeout expires, the source repeats the send operation, until either an *ack* is received, or the maximum number of retries is exceeded. In the latter case the token passing operation fails, and the source holds the token. After the first timeout, the source piggybacks a certificate to the datagram. The certificate consists of the public key of the *source*, encrypted and signed by the Certification Authority.

Upon receipt of the *move* message, if the destination already knows the public key of the source, it checks the signature of the datagram. Otherwise, if the public key of the source is unknown, it discards the message. It is in charge of the source, whenever the *ack* timeout expires, to send the certificate of the sender. In this case, upon receipt of the certificate, the destination decrypts and stores the public key of the source in a local cache.

After successfully checking the *move* datagram, the destination records the *session id* associated with the token, as part of the *soft state* of the token. Next it prepares and sends the *ack* datagram to the source, using the port indicated in the *move* datagram. The first two fields, namely the *token id* and the *session id*, are the same as of the *move* datagram, but the destination updates the timestamp using a local timestamp, indicates a local port for protocol continuation, and replaces the signature with one obtained using the local private key. The *ack* datagram is sent back to the source, and the destination will timeout the receipt of a *commit* datagram. As in the case of the *move* datagram, after the timeout expires the first time the *ack* datagram is resent a limited number of times, piggybacking the public key signed by the Certification Authority. In case the number of retries exceeds the limit, the token passing fails and the *destination* considers the source still holds the token.

As soon as the source receives an *ack* datagram on the expected port, it checks the validity of the content using the public key of the source, in case it is known. Otherwise, it silently drops the datagram and waits for the successive, which contains the certificate. If the *ack* is valid it considers the token as successfully passed, and sends a *commit* datagram to the destination. Any activity connected with the presence of the token is (forcibly) terminated at that time.

The *commit* datagram contains, besides the identifiers of the token and of the token passing operation, a local timestamp and the signature. It is resent a limited number of times, until an *early stop* datagram is received. We do not consider to complement the *commit* datagram with a certificate. Also in case the number of resend operations exceeds the limit, the token passing protocol terminates successfully, since the receipt of the *early stop* datagram has effect only on the timing of the protocol.

The number of retries allowed for the *commit* packet should be significantly higher than the number of *move* retries: for instance, 10 versus 2 may be a reasonable *rule of thumb*. The reason for this is that the successful delivery of the *move* datagram is used as an indication that the route between the sender and the receiver is working, and congestion free. In the adverse case the *move* fails to reach the receiver, and another receiver is selected. Once excluded the main reasons of auto-correlation of network delays, the delivery of the each of the *commit* retries has comparable probability, and the failure of n successive retries exponentially decreases with n.

Upon receipt of a valid *commit* datagram, the destination considers the token passing operation to be successfully concluded, and the activity related to the presence of the token is triggered. After this it sends a single *early stop* datagram.

The rationale behind the introduction of the *early stop* datagram is in the first requirements stated in section 2: the receipt of this type of datagram interrupts the sequence of repeated *commit* datagrams, which would otherwise significantly increase the token passing latency.

An exhaustive description of the protocol informally described so far is in figures 2 and 3. Two separate state diagrams are needed to describe the behavior of the source and of the destination. Blue arrows indicate transitions in absence of token losses and authentication failures. Green labels indicate transition that entail a send operation. Rounded boxes indicate exit (or entry) points: red ones indicate a failure (`return undef` in Perl idiom), otherwise the returned value.

Whenever a new member requests to join the membership, the Certification Authority (CA) delivers it, using a secure channel, a certificate consisting of a public/private key pair, and of the signature of the public key by the (CA). It is intended that the public key of the CA is available to every possible destination of the token: this is the only piece of *global* data.

The operation of joining the membership falls outside the scope of the token exchange protocol: we consider that the credentials provided by the CA are used to enter the token

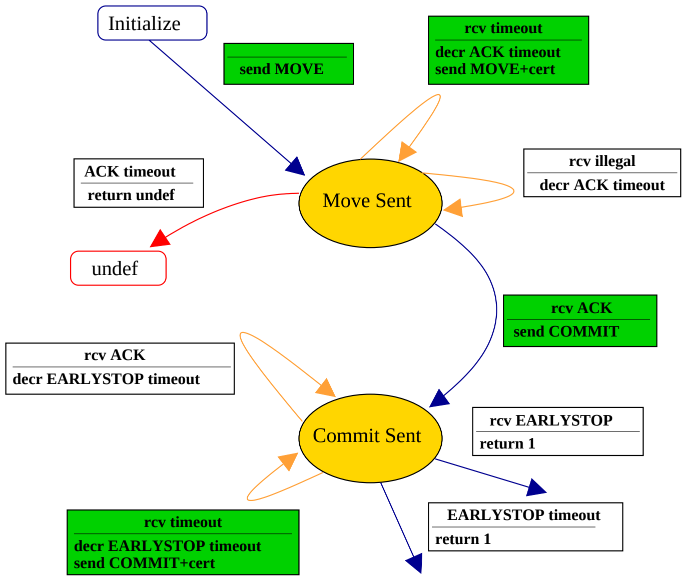

Fig. 2. Finite state machine of the source process

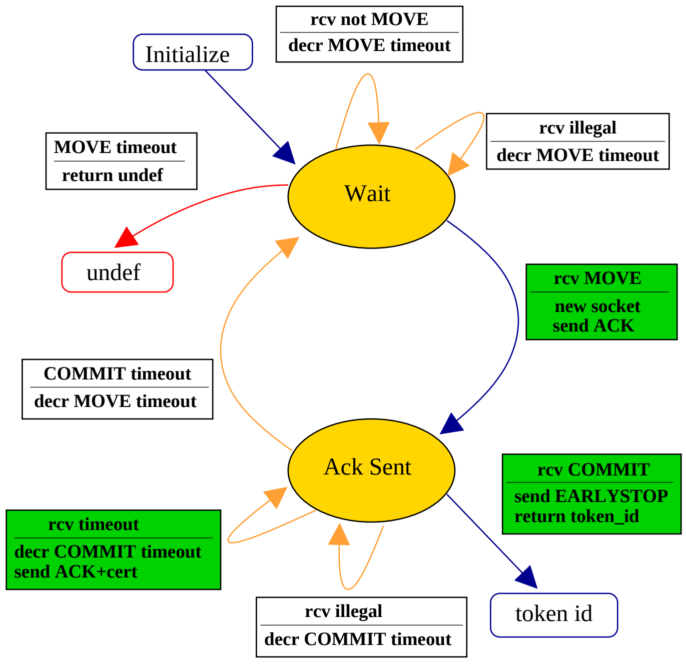

Fig. 3. Finite State machine of the destination process

routing algorithm.

3.1. *Discussion*

This section is devoted to a number of points that emerge during the detailed description of the protocol: we prefer to separately justify them to make the protocol description more fluent.

We start by briefly proving that our solution conforms the requirements described in section 2, next we give a precise statement of security issues.

Requirement ii) in section 2 states that the protocol must exclude deterministically the duplication of the token. To prove that our protocol satisfies the requirement, we note that the event of duplication occurs only if i) the source detects a protocol failure, and therefore holds the token, and ii) the destination detects a success, and holds the same token. Since the second condition occurs only when a *commit* datagram is received, which is sent by the *source* only when it considers the token passing operation successful, we conclude that the occurrence of ii) negates the occurrence of i). Therefore token duplication is deterministically excluded.

Requirement i) from section 2 indicates that the protocol should seldom incur token loss. In order to evaluate the occurrence of token loss, we need to analyze the other case of inconsistent termination of the protocol, when i) the source detects a protocol success and ii) the destination detects a protocol failure. For this case to occur, we need that the destination fails to receive the *commit* datagram, while the source correctly receives the *ack* datagram. The occurrence of both events cannot be excluded, and we briefly explore the possible scenarios that bring to this.

One event is the failure of the destination after sending the *ack* message. The source considers the token successfully passed when it receives the *ack*, and therefore in this case the token is lost.

Another similar event is the failure of the source after sending the *move* datagram, but before sending the *commit*. In that case the destination considers the token passing failed, and the token still held by the destination. But the destination failed meanwhile, and the token is lost.

Both cases are covered by the assertion *if the token holder fails, the token is lost*: we observe that the mean time between the occurrence of this event is not altered by the token exchange protocol, except for a negligible increment due to the fact that during the time between the receipt of the *move* datagram and the receipt of the *ack* datagram the switch off of *either* partners causes a token loss, doubling the probability of this event during that short lapse, the duration of which roughly corresponds to the latency of the *commit* operation.

Another event that causes the loss of the token is the failure in delivering the *commit* datagram due to a connectivity problem. The design of a protocol that takes into account the demonstrated auto-correlation of network delays [13] is a difficult task. In our case, the successful delivery of the *move* datagram excludes the permanent failure of the route, and, given the limited number of retries of the *move* datagram (though greater than two), also a heavy congestion. That given, the residual event is the excessive delay of a *commit* datagram for independent events bound to the application of network policies (like [6]).

To harden our protocol against this event, we repeatedly send the *commit* datagram: since datagram loss events are poorly auto-correlated, we conclude that the probability to lose all datagrams in a sequence of n drops exponentially with n.

As for latency jitter (in requirement iii) of section 2), we consider that this corresponds to the interval between the send of the *move* datagram, and the latest time between the receipt of the *earlyStop*, and the send of the last *commit* datagram of the sequence (in case the *earlyStop* is lost). This time is bound to the communication delay between the source and the destination, and to the time between the successive resend of the same datagram. Since each datagram is successfully delivered after the first try with high probability, latency distribution is quite similar in shape to that of a ping, meeting requirement iii.

Notice that the presence of the *earlyStop* datagram is a key feature on this respect: without this message, the token latency should take into account the repeated delivery of n *commit* datagrams.

Another infrequent event that brings to a resend event is the failure in authenticating the datagram. As a general rule, the receiving agent will successfully check the datagram using the public key stored in the local cache: only in case the sending agent is not yet cached the check will fail (as a consequence of a recent join), and the sender will resend the datagram together with its certificate.

As for token validation (requirement iv) in section 2), most of the work is done by the signatures carried by datagrams. The role of timestamps included in each of them is not of carrying any timing information, but of changing the signature of each and every datagram. Such a "timestamp" can be dependent on the application.

Flushing an agent of invalid datagrams is not specially harmful, since these datagrams are only checked and silently discarded, without response. This fact, together with the use of the UDP protocol, makes identification of agents difficult using port scanning [11].

In case an intruder obtains and replays a valid datagram (for instance, sniffing the network and identifying the datagram), it does not damage the consistency of the protocol. In order to make use of the sniffed packet, the intruder also needs to spoof [11] the sender of the sniffed packet, and send it to the same destination. As a consequence:

– if the datagram refers to the current session, the sender either ignores it, or correctly makes use of it if the good one went accidentally lost: the protocol is prepared to receive and discard multiple copies of any type of datagram in the current session;
– otherwise, the *session id* necessarily corresponds to or precedes the one recorded as completed in the local *soft state* of the token: we recall that datagram signature binds its destination. The datagram is detected as illegal and discarded.

Therefore a malicious agent has no way to intrude the protocol at any stage, unless it manages to obtain either the private key of a member, or a certified key-pair.

A misbehaving Internet router may cause token loss events by corrupting or discarding valid datagrams: such behavior is regarded as marginal, since routers are kept under strict control by system administrators. The kind of inconsistency introduced by such behavior is quickly recovered after removal of the misbehaving agent.

A real threat comes from a misbehaving agent in possession of a valid private key or of a certified key-pair. Such an agent may damage the consistency of the protocol: the worst scenario is the corruption of local caches, as well as token duplication. Such damages will persist for a long time after the removal of the misbehaving agent. We note that such scenario is an instance of a *Byzantine generals* problem [8], an efficient solution of which is still an open issue.

We do not consider datagram encryption, since its potential benefits do not compensate the footprint of the decryption/encryption operations. In fact, the data that a potential intruder might *stole* from the content of the datagram are the *token id* and the *session id*. These are two serial numbers that are of little use, unless the misbehaving agent holds a correct private key and certificate. In addition, the intruder may obtain otherwise, and with little effort, the public key usable for decryption, sniffing the first retransmission of *move* or *ack* messages, that are moderately frequent events.

3.2. *A case study*

We describe a case study based on a randomized routing rule: at each step, the token is routed to a peer chosen at random among those recorded in a local cache. We only aim at demonstrating that the token passing protocol fits a practical use case, and the description is not meant to be exhaustive.

To enforce the presence of a token, which is the first step to ensure its uniqueness, we need to introduce a way to detect and recover from its loss, an event that cannot be excluded: despite the required resilience of the token passing protocol to network failures, we cannot avoid that a token holder silently fails.

The detection of such an event is based on an estimate of the *return time*, the time between two successive hits of the token on a certain agent (see [9] for details about the distribution of the *return time* of a random walk). If a token is not observed for a time exceeding by several times the expected *return time*, a new token is generated by the agent that observes the delay. The token passing protocol introduced in the paper ensures (see requirement i)) that this event is not bound to the quite frequent network failures.

The above discussion about the expected *return time* relies on the hypothesis that each token passing operation takes a time characterized by a low dispersion, near to the minimum: this corresponds to requirement iii). Failure of this requirement introduces frequent generation of *spurious* tokens (distinguished from *duplicated* tokens).

Given the probabilistic nature of the rule used to cope with token loss the introduction of a spurious token in the system cannot be avoided deterministically. This justifies the existence of a token removal rule, which is the second step to ensure token uniqueness. Such rule is based on the system-wide assumption that each token has a unique identifier (a consequence of requirement ii)), and that token identifiers are ordered: *spurious tokens* are defined as all those whose identifier is not minimal in the set of the identifiers of the tokens circulating in the system. At any time, exactly one token in a system is non-spurious, if at least one exists.

That given, if an agent is hit, sequentially, by token A, token B, and again by token A, then it can conclude that the one with larger id is spurious. If A is spurious, the agent has a chance to remove it.

The simple correctness proof is as follows. We assume that $A > B$, and consider an agent observes the pattern $A - B - A$. When token B was last observed at time t by the agent, both token A and B were circulating in the system; in fact A was observed before and after time t, and by id uniqueness the same token was present in the system at time t. By definition A is spurious. In conclusion, the first agent in the system that observes the pattern $A - B - A$ removes A. Those observing $B - A - B$ do not take any action.

Random events may determine the infrequent removal of both tokens: if token B is lost after time t the system ends up without tokens. The token recovery rule is in charge of introducing a new token in this infrequent event.

We observe that a new agent willing to enter the membership must be given the opportunity to get in touch with one of the members. A simple way to implement a *join* event is to give the joining agent a new token, and the address of one member: its first action is to authenticate (along requirement iv)) and send the token to the indicated member, thus gaining the opportunity to enter the membership. The spurious token thus generated will be eventually removed by the above removal rule. Leave events are treated as silent failures: the agent stops responding to *move* messages.

The *token passing* operation takes place within a trusted membership: the solution we indicate for membership maintenance consists in associating with each token passing operation a synchronization of the cached directories. To secure this operation the data passed during directories synchronization must be authenticated: requirement iv) gives the basis for this. Concerning directory synchronization, the interested reader finds an applicable result in [1]. We do not discuss further the issue, that falls outside the scope of this paper.

3.3. *Experiments*

An implementation of the token exchange operation explained in this paper has been written in Perl, as well as part of the randomized routing described as a use case. The prototype was used to verify the feasibility of a protocol conforming to the requirements stated in section 2, using the formal statement in section 3. To obtain a realistic feedback about protocol reliability we run our tests in the open Internet, not in a restricted or simulated environment.

We refrain that our purpose is to study and design the token passing protocol, not the randomized routing use case; therefore the preparation of a large testbed was not only expensive, but pointless. We configured a network of 4 hosts, located in Italy and Greece, thanks to the cooperation of the FORTH Institute in Crete. Three of them (located at CNAF, in Bologna, and FORTH) were used to carry out an endurance test, of which we report here, while the fourth one (located inside the Dept. of Computer Science of the University of Pisa) was used to test and debug join and leave operations.

We were able to obtain experimental evidence that the protocol tolerates Internet packet losses and delays while preserving the required properties of the token passing protocol, as introduced in the statement of the problem in section 2, namely: i) token loss seldom occurs, ii) the token is never duplicated, and iii) token passing latency dispersion is low.

During the experiment in the Internet, the token was delayed 10 seconds before being resent, to reproduce the execution of an operation controlled by the presence of the token.

We observed the first *token loss* event after $1,792,498$ seconds of activity (approx. 20 days): during this time one of the members (A) was hit by the token $61,206$ times, with an average return time of 29.29 seconds (expected 30 seconds), 99% of the times below 77.64 seconds. The *token latency* from agent A to agent B, located in the same network, and from A to C, located in a different country, were significantly different. Token latency between A and C was on the average 0.121 seconds, 99% of times below 0.615 seconds in a sample of $30,563$ operations. Latency between A and B was 0.009 seconds, 99% of times below 0.010. These observations prove that token latency has low dispersion (requirement iii) in sec. 2).

The token passing protocol tolerated 60 network failures without producing a token loss: 37 network failures were detected by a timeout of the *acknowledge* datagram, 23 by a timeout of the *commit* datagram. The token was finally lost after a crash of the agent holding the token, which proves that token loss is an infrequent event (requirement i) in sec. 2).

No token duplication events were observed, matching requirement ii) in sec. 2.

Another series of experiments has been carried out in a virtual testbed with 10 hosts, implemented on a PC hosting User Mode Linux, used for development and testing. While these experiments indicate a successful development methodology based on infrastructure virtualization, their quantitative results are not relevant for the topics discussed in this paper.

4. Conclusions

The design of the token passing operation should not be overlooked, since it must exhibit specific features in order to efficiently and securely support distributed coordination tasks. In this paper we introduce the implementation of a general purpose token passing operation.

We propose a solution that specifically targets security aspects: all datagrams used to exchange the tokens are signed by the sender, and authenticated by the receiver. The public key distribution functionality is embedded in the token passing protocol itself, and has a negligible cost.

The protocol is not bound to a specific token routing rule or network overlay: such generality does not impact on performance, which is shown to adhere to generic requirements of consistency, reliability, and speed. To give an working example of application, we describe the implementation of a randomized token circulation.

The performance of the protocol has been assessed with a prototype implementation running in a testbed wired using public Internet links. After a 20 days long run, when the token was finally lost, we observed that the results are compatible with the requirements.

References

[1] Ziv Bar-Yossef, Roy Friedman, and Gabriel Kliot. RaWMS - random walk based lightweight membership service for wireless *ad-hoc* networks. *ACM Transactions on Computer Systems*, 26(2):66, June 2008.

[2] Augusto Ciuffoletti. The wandering token: Congestion avoidance of a shared resource. *Future Generation Computing Systems*, page to appear, 2009.

[3] Augusto Ciuffoletti and Michalis Polychronakis. Architecture of a network monitoring element. In *CoreGRID workshop at EURO-Par 2006*, page 10, Dresden (Germany), August 2006.

[4] Adam M. Costello and George Varghese. The FDDI MAC meets self-stabilization. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems – Workshop on Self-Stabilizing Systems*, pages 1–9, Austin – Texas, May 1999.

[5] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[6] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[7] R. Groeper, C. Grimm, S. Piger, and J. Wiebelitz. An architecture for authorization in grids using shibboleth and voms. In *33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 367–374, August 2007.

[8] L. Lamport, R. Shostak, and R. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

[9] L. Lovasz. Random walks on graphs: a survey. In D. Miklos, V. T. Sos, and T. Szonyi, editors, *Combinatorics, Paul Erdos is Eigthy*, volume II. J. Bolyai Math. Society, 1993.

[10] Navneet Malpani, Yu Chen, Nitin H. Vaidya, and Jennifer L. Welch. Distributed token circulation in mobile ad hoc networks. *IEEE Transactions on Mobile Computing*, 4(2):154–165, Mar/Apr 2005.

[11] Stuart McClure, Joel Scambray, and George Kurtz. *Hacking exposed: network security secrets & solutions*. McGraw-Hill Professional, 2005.

[12] National Institute of Standards and Technology (NIST). *Digital Signature Standard (DSS)*, January 2000.

[13] Vern Paxson and Sally Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[14] Frank Petit and Vincent Villain. Optimal snap-stabilizing depth-first token circulation in tree networks. *Journal of Parallel and Distributed Computing*, 67(1), January 2007.

[15] J. Postel. User datagram protocol. Request for Comment 768, Network Working Group, August 1980.

[16] Jon Postel. Transmission control protocol. Technical Report RFC 793, DARPA, 1981.