# Preventing the collision of requests from slave clocks in the Precision Time Protocol (PTP)

Augusto Ciuffoletti (augusto@di.unipi.it)
Department of Computer Science — University of Pisa
Largo B. Pontecorvo, I-56126 Pisa (Italy)

***Abstract*—The Precision Time Protocol (PTP) distributes a time reference across a network: it specifically addresses demanding environments, where it can reach sub microsecond precision using appropriate technologies. Its scalability is primarily limited by message packet delay variations induced by packet collisions.**

**While it is possible to avoid collisions with non-PTP packets using appropriate traffic management technologies, collision between PTP packets is an open problem in large systems with critical clock precision requirements.**

**We propose a coordination algorithm that avoids the occurrence of such collisions. It assumes the availability of a multicast facility from the timing reference source, the master clock, to all the slaves: this is not a restrictive hypotesis since PTP itself takes advantage of this kind of connectivity, and it is also compatible with typical wireless environments.**

**The algorithm operates without introducing additional traffic, it ensures an upper bound to the time between two successive synchronizations of any given slave, it does not alter the structure of the standard PTP messages, it envisions a dynamic number of slaves, it tolerates the replacement of the master with a hot spare in case of failure, and does not rely on specialized hardware.**

**The algorithm has a footprint that does not insist on activities that are already time sensitive, and its operation is mostly concentrated on the master.**

**The algorithm inherits the security and fault tolerance limits of PTP: in particular this refers to malicious nodes, and to broken devices that may jam the network.**

**Keywords: Precision Time Protocol, Packet Collisions, Coordination protocol, token routing, wandering token.**

## I. Introduction

The IEEE1588 Precision Time Protocol (PTP) is a clock synchronization protocol designed for a very wide applicability range. It computes data needed for the syntonization, intended as the tuning of clock frequency, and the synchronization, intended as the consensus about the clock value at a certain time. It is based on a regular flow of packets carrying timestamps between a unit known as the *master* clock, and the units known as *slave* clocks; it is integrated by a master election algorithm, also based on packet exchange [1].

The traffic carrying the timestamps used for clock synchronization has stringent timing requirements: the delayed delivery or the loss of a single packet may deteriorate the precision of a slave clock, with effects that depend on the application environment [3]. Excluding hardware failures, these events are always related to packet jamming: packets related to distinct control flows collide for the utilization of a resource, and the resource responds with a penalty for one or both flows. The CSMA/CD policy is just one example of this. One solution is that communication related to clock synchronization uses dedicated resources: this is implemented in several ways considering time synchronization packets as high-priority traffic [4]. In this way we obtain that collisions are limited within time synchronization packets, but they are not entirely prevented.

To solve this problem, we introduce an algorithm that globally serializes the production PTP packets that are prone to collision. Collisions are thus completely eliminated.

In order to be of practical interest, a solution must satisfy a number of requirements:

- to be compatible with the existing standard;
- to have no impact on clock synchronization traffic;
- to ensure regular access to the master by the slaves;
- not to introduce single points of failure;
- to be scalable.

We introduce a solution that complies with the above requirements, but that requires that communication between the master and the slaves is in multicast: in fact this requirement is not far from IEEE1588 philosophy, which envisions certain messages to be delivered from the master to all slaves. Communication in the reverse direction, from the slaves to the master, is not required to show the same property for the application of the algorithm. As a side effect, the reciprocal visibility of slaves is not required.

In a nutshell, our solution consists in managing the circulation of a token carrying the privilege of requesting a clock synchronization to the master. The token circulates in a network that, for our purposes, can be assimilated to a star, with the master clock at the center. The token is passed from one slave to the other, bouncing on the master: token duplication and loss are therefore avoided by the presence of a centralized control. However this control does not extend to the selection of the next token destination.

The decision about the next destination of the token is usually demanded to the slave holding the token, that operates using data available locally: with infrequent exceptions, the role of the master is limited to forwarding the packet to the destination indicated by the token source. The advantage is that the replacement of the master, as well as the cold start of the system, do not require the preliminary reconstruction of a directory of the slaves in the master clock.

The decision about the next destination of the token is randomized: at each step any of the slaves can be selected as the next destination for the token [8]. This rule is simple to implement when each slave holds a limited list of other slaves

(see [9] for theoretical aspects related with this approach), not necessarily adjacent at link level, but leaves open the possibility that a slave waits for a token for an indefinite time.

The master is able to avoid the occurrence of such a case, since it participates to all synchronization sessions: when it detects that one of the slaves is about to starve, it de-routes a token in order to allow it to run a synchronization session.

We take into account that the additional complexity introduced by a new protocol should be in balance with the expected benefits. So when the target application tolerates sporadic clock inaccuracies, or when collisions are so rare to approximate hardware failures, it may be questionable to complicate the system to avoid such events. In the paper we discuss and evaluate:

- the relevance for the specific application of a collision event;
- the frequency of such an event in the specific network.

It is not possible to give an exhaustive answer to both questions, since they widely depend on the specific use case, and PTP itself is designed to be as much generic as possible. So we give generic guidelines, by analyzing the consequences of a collision event.

A collision event is usually managed buffering one of the colliding messages, and transmitting it at a later time: this introduces jitter, that is the primary source of imprecision in PTP. The relevance of sporadic clock imprecision depends on the specific application, but we consider that typical applications of the PTP are quite demanding about this. However, even in case of clock sensitive applications, events that are so rare to be assimilated to a master clock failure may become negligible. So the frequency of collision events is also relevant, and must be considered.

The evaluation of the frequency of collision events depends on system networking and cannot be solved with a general purpose statement: however, a simple model is sufficient to give an idea of when the algorithm we propose is of practical interest.

When clock synchronization events are homogeneously distributed in time — and the PTP ensures this property, as discussed below — we may consider them as a Poisson process. Its density, the number of events per time unit, corresponds to the rate between the number of slaves, $n$, and the average interval between two successive synchronization requests from the same slave: for PTP we have that this average corresponds to $\frac{\delta}{2}$ where $\delta$ is the maximum interval between two successive synchronizations of the same slave.

From the properties of Poisson distributions, the probability of a collision during a lapse of given amplitude (for instance, the gap between two successive time slots reserved to clock synchronization) grows exponentially with the number of slaves, and with the frequency of clock synchronization sessions. This value is tightly bound to the target clock precision.

To give an idea, in a system composed of 10 slaves, a synchronization period $\delta$ of 2 seconds, and one synchronization slot every 1 msec, the probability of a collision during a slot is in the $10^{-4}$ range. If the number of nodes rises to 100, or if the period $\delta$ is decremented to 200 msecs, the probability of collision rises to $0.01$.

So the introduction of a collision avoidance protocol may be questionable in a system where collision events are rare. However their probability grows rapidly with system size and synchronization frequency; if we envision the evolution of PTP towards larger systems with tighter timing requirement, then collision avoidance should be seriously taken into consideration. This paper is the first step in this direction.

The rest of the paper is organized as follows: the next section describes PTP synchronization messages, and in section III we introduce the algorithm that serializes synchronization requests. In section IV and V we focus on two relevant subproblems that are opened by the algorithm: the implementation of a unique identifier for the slaves, where we explore the possibility of using the MAC address for the purpose, and the maintenance of the random neighborhoods of the slaves, comparing the protocol adopted for our solution with another recently published to solve a similar problem.

## II. Problem description

The PTP features a clock synchronization protocol which is built on three messages: a SYNC message, sent in multicast from the master clock to the slaves, a DELAY_REQ message sent from a slave to the master, and a DELAY_RESP sent in the reverse direction. In our system, SYNC and DELAY_RESP messages are sent in multicast, with the indication of a specific destination in the case of DELAY_RESP messages.

The SYNC message transfers from the master to the slave the value of the clock of the master: a sequence of such messages, when network delay from the master to a certain slave is constant, carries enough information to enable the syntonization of the clock of the slave.

But network delays and clock offsets must be frequently measured since they are prone to a slow drift due to a number of reasons, firstly temperature; to this purpose PTP introduces the DELAY_REQ and DELAY_RESP messages. A DELAY_REQ message is sent to the master, which replies with a DELAY_RESP message that contains the timestamp recorded when the request was received; the slave uses it to rescale the clock value using also the timestamps collected during the SYNC message exchange.

Considering the traffic associated to the operation of the PTP, while the SYNC message is sent in multicast, each slave requires a distinguished pair of DELAY_REQ/DELAY_RESP messages. This fact is the reason of limited scalability of the PTP: traffic grows linearly with the size of the system, and eventually saturates carrier capacity. We call *two-way session* this latter request/response exchange (not to be confused with the peer-to-peer variant of the PTP, which falls outside the scope of the paper).

A slave initiates two-way sessions at regular intervals, whose maximum extent is determined by the quality of the slave clock, and by the target precision, that we assume to be constant in time. For each session the slave uses the most recent SYNC message as a reference, also recorded in the `sequenceId` field of DELAY_REQ message header.

The standard requires that the interval between successive DELAY_REQ messages from the same slave are taken from

a uniform random distribution between zero and a maximum value, that here we indicate with $\delta$ (see clause 9.5.11.2 *Timing Requirements* in [1]); this corresponds to an average interval of $\frac{\delta}{2}$. Assuming uniform timing requirements throughout the $n$ slaves in the system, two-way sessions can be modeled with a system-wide Poisson process, as anticipated in the introduction, whose density is $\lambda = \frac{2n}{\delta}$.

We obtain an analytic prediction of the probability of a collision if we are able to determine a $\tau$ such that two DELAY_REQ issued at times separated by less than $\tau$ are in collision. However, a reliable evaluation of this quantity is extremely hard to obtain, depending on many aspects eventually hidden in the implementation of networking devices, and being not stable in time (see [10] for an analysis of the early Ethernet from a similar point of view). That is the reason why the adoption of an approach like the one presented in this paper is of interest for critical applications. Here below we assume that a value of $\tau$ exists, and we use it for drawing qualitative conclusions that are valid under this assumption.

Using a Poisson model for the DELAY_REQ message production process, the probability of a collision is[1]:

$$P_{coll} = 1 - (1 + 2r)e^{-2r} \quad \text{with} \quad r = \frac{n\tau}{\delta} \tag{1}$$

As anticipated in section I, the probability of collision increases exponentially with the number of slave clocks in the system, represented by $n$, and with the required precision, which grows with the inverse of $\delta$. The value of $\tau$ can be used to moderate the growth, but while this value is mostly bound to the networking technology used, the other values may vary within the same application.

This is an evidence of the problem of collisions of two-way sessions: the probability of such an event is difficult to evaluate a priori, and rapidly grows in demanding environments. We conclude that an algorithm that eliminates the problem, by serializing two-way sessions, is of interest for a successful evolution of the PTP protocol.

## III. A COORDINATION PROTOCOL BASED ON TOKEN CIRCULATION

The algorithm is based on a *token circulation* mechanism [7]: each slave clock that executes a two-way session grants the privilege of executing the next two-way session to another slave. The basic idea is extremely simple, but must be complemented with a number of additional features, in order to be applicable. We divide the explanation of our instance of token circulation in three steps:

- the algorithm that is run while there is exactly one slave holding the privilege (*legal conditions*) and
- the implementation of the *token passing* operation, and
- the actions undertaken when certain events break legal conditions.

The explanation of how token passing is implemented is delayed, since it is not needed to explain algorithm operation in legal conditions.

[1] an exhaustive justification of the formula is found in [5]

### A. Operation in legal conditions

In order to ensure a fair share of synchronization opportunities, we need to overlay a sub-network used for token circulation: a number of deterministic solutions that enforce an overlay ring are found in literature. However, they are very slow in recovering from failures. Here we propose a token circulation algorithm that is targeted on the peculiar features of our environment.

We observe that slave clocks do not need to run two-way sessions at regular intervals: the time between two successive executions may be variable, as long as a given maximum is not exceeded. That opens the way to non-deterministic algorithms: the token may not visit all slave clocks in a deterministic sequence, but on the contrary it may follow a route obtained applying repeatedly a non-deterministic token routing rule. One simple rule consists of passing the token, at each round, to another slave chosen at random: the rule is effective, since the time between two consecutive visits of the token to a given slave clock, the *return time* of the token, has favorable properties, included resilience to node failures. An example of application of this technique to a congestion avoidance problem is in [6], but the same technique is also used for distributed coordination in ad-hoc networks [8].

A plain random walk approach introduces an assumption that contrasts with scalability: in order to select another slave clock *at random*, the sender needs to access a registry of all clock identifiers. The presence of a centralized registry introduces a single point of failure and re-instantiates a mutual exclusion problem. On the other hand, the maintenance of a local cache on each slave is impractical since it might contain an unpredictable number of slave identifiers.

A viable compromise consists in maintaining a neighborhood of limited size: it is a known result that the *random walk* properties (included the distribution of the *return time*) are preserved when the graph is not complete.

A solution based on a random walk exhibits another issue, since for any $\Delta > 0$ there is a non-null probability of a return time larger than $\Delta$. Instead we need to ensure that a two-way session occurs at least every $\delta$ time units.

We consider that an effective way to cope with this deterministic requirement consists in exploiting a global view of the system, which is maintained on the master clock: whenever one of the slave clocks appears to have been waiting $\delta$ time units, the master *re-routes* the token to feed the starving slave.

The algorithm used for the management of slaves neighborhoods is a key aspect of our solution, since it determines the fairness of token visits. We introduce an algorithm that we call *swap on timeout* that works as follows: whenever a slave that proposes to pass the token to peer $X$ detects that the master reroutes the token to another slave $Y$, it replaces $X$ with $Y$ in its neighborhood. In section V we compare our algorithm with another, recently published, and with a full mesh.

One important property of our algorithm is that, since all nodes are characterized by the same $\delta$, the possibility that more than one timeout is triggered simultaneously is excluded in legal conditions. In fact, at time $t$ only one slave may starve, which is the one that executed the two way session at time $t - \delta$, necessarily unique.

```
 1 Neighbors: array [1..degree];
 2 forever
 3   A = TwoWaySession.observe();
 4   I = A.Dest;
 5   if ( LocalId == I )
 6     i = randomInteger(1..degree);
 7     B = new(TwoWaySession);
 8     B.ProposedDest = Neighbors[i];
 9     B.run();
10     Neighbors[i] = B.Dest;
```

TABLE I
SLAVE CLOCK ALGORITHM

```
 1 LastVisit[Slave]: array [1..n];
 2 forever
 3   A = TwoWaySession.receive();
 4   S = select Slave
         with LastVisit <= (now-delta);
 5   if (defined S)
 6      A.Dest = S;
 7   else
 8      A.Dest = A.ProposedDest;
 9   A.complete();
10   LastVisit[A.Dest] = now;
```

TABLE II
MASTER CLOCK ALGORITHM

We have therefore deterministically excluded the occurrence of collisions, and ensured that all nodes execute one synchronization every $\delta$ time units. This result comes at the price of the design complexity introduced by the token passing protocol.

We conclude with a summary of the algorithm.

In table I we show the algorithm run by the slave, which consists of a `forever` loop that is executed each time a new two-way session A is observed on the wire. Such operation entails a token passing operation, and each slave in the system records the destination of the token, represented as `A.Dest` in the program. If the destination of the token matches with the local slave identifier, the operation continues selecting one of the `Neighbors` at random as the proposed next destination of the token; a new two-way session B is run and the token is passed. At the end of session B, in the `Neighbors` array the proposed destination is replaced with the real destination of the token, that differs from the proposed one in case of rerouting.

The master clock executes a distinguished algorithm, as shown in table II. As a general rule it is transparent (i.e. stateless): the destination of the token is the same proposed by the slave. The master forces a different destination only to feed a slave that is detected as starving.

### B. Reacting to infrequent events: leave, join and restart

In this section we discuss how the protocol reacts to events that perturb legal conditions: *leave* and *join* are considered firstly, next we consider how to deal with the *startup* transient.

A *leave* event is detected when the slave indicated as the destination of the token does not perform as expected: the master clock observes the event and records it in the registry. Following that, tokens directed to this slave will be regularly re-routed, with the usual feedback on slaves that have the leaving node in their neighborhood.

A consequence of the leave event is the loss of the token: there is no candidate slave for the next two-way session. The master recovers from this event entering the *warm start* procedure described below.

The *join* event begins when the new slave clock advertises its presence to the master. In response, the master delivers one or more slave identifiers to the joining slave, and it enters its identifier in the registry. This preliminary conversation is carried out using *Signalling* messages (see [1], section 13.12), and it does not interfere with time sensitive traffic. Finally the master reroutes a message to the new slave: this induces a feedback on the slave that proposed a different destination for the token, and the joining slave enters its membership.

Concerning system restart, we distinguish a *warm start* and a *cold start*: the difference between the two is in the master registry state, that is empty in the latter case.

At the beginning of a *cold start* the master waits for signalling messages from the slaves. When the first is received, it is not yet possible to initiate the token circulation algorithm: this will only start when at least two slaves have successfully performed the join. At this point the master enters the *warm start* procedure.

A *warm start* consists in creating and delivering a new token to one of the known slaves.

### C. Token passing implementation

*Token passing* is implemented as a part of the two-way session: the slave clock that sends the DELAY_REQ message includes in it the unique identifier of a proposed destination of the token using some space left available in the packet for extensions[2] (octets at offset 5 and from 16 to 19). The master clock has a chance to re-route the token, following the algorithm II, by indicating a different destination in the DELAY_RESP packet: otherwise it transparently copies the proposed destination into the DELAY_RESP packet. Since we assume that DELAY_RESP packets are sent in multicast, all slave clocks in the network observe the final destination of the token.

Note that the token is not represented by any sort of data structure, either inside a slave clock or in a piece of communication. The property of *holding the token* is simply embedded in the control flow of the slave clock. The token is (virtually) passed from the slave clock that sends the DELAY_REQ message to the slave clock indicated in the DELAY_RESP packet.

The *token creation* is implemented including the unique identifier of a slave in a SYNC message.

### D. Fault tolerance and security

We consider separately three kinds of failure: slave failures, master failures and network failures. In case of slave failures, we also consider the event of malicious behaviors.

[2]Note that the attachment of ad hoc TLV field is deprecated for this kind of messages (see [1], clause 13.4)

*Silent failure* and *recovery* of a slave correspond respectively to leave and join events: we have already explained how the algorithm deals with such events. Regarding the malicious behavior of a slave, the algorithm itself does not introduce significant security holes: one slave might break token passing fairness by indicating always the same slave as the token destination, or drop the token each time it receives it, but the algorithm tolerates such adverse behaviors. Spoofing the unique id used in the protocol allows to damage one specific slave clock. The case is even worse if the slave is able to embody several slaves at the same time: it may damage the overall serialization algorithm if it produces a flood of joins with distinguished identities. A secure implementation of PTP, currently a matter of discussion within PTP working group, would therefore improve the robustness of our algorithm, wherever it is an issue.

A broken or malicious slave might also damage the overall algorithm by jamming the link layer with packets, thus making communication impossible: this event equally impacts also PTP, and should be addressed using fault confinement tools that fall outside the scope of this paper.

The *failure of the master* is an event that is taken into serious consideration by PTP: regarding our algorithm, as long as the PTP protocol is able to replace the failed master with a new and warm spare, our algorithm is not affected by the event, that is ignored. In the case the master is replaced with a cold spare, there is the possibility that some of the slaves starve: to understand why, we distinguish two cases that refer to the state of the registry of the slaves maintained in the master, with respect to the state of the slave. For a certain slave such state may be either *joined* or *not joined*, and we explore the two inconsistent conditions.

When the state is marked as *joined* in the registry, but the slave is either failed or considers itself as *not joined*, its state is fixed in the registry with the first event regarding the slave. If it does not respond when it is passed the token, it will be marked as *not joined* in the registry; if it produces a *join* request, its state becomes consistent with the registry. We conclude that the case is covered by the regular operation of the algorithm without local or global drawbacks.

When the state is marked as *not joined* in the registry, but the slave considers itself as *joined*, the master is unable to detect when it is the case to reroute a packet to the slave to avoid starvation. So, if the token is not passed to the slave by another slave before the timeout expires, the slave is going to starve. This event is anomalous, since the slave is considered operational and joined to the membership of synchronized slaves: however its starvation cannot be entirely excluded in such a case, and it is managed by leaving the membership, thus fixing the inconsistency in master's registry. Later on the slave may re-join the membership.

The frequency of occurrence of this anomaly depends on the master reroute event frequency, which is related to the fairness of the token routing, which depends on the effectiveness of membership management. The net effect is that the membership maintenance algorithm contributes to avoid anomalous starvations. The last section in this paper is devoted to the discussion of such issue, comparing our membership maintenance rule with another recently published.

*Network failures* that cause the partition of the network are considered in PTP, and dealt with by activating a spare master. We apply the same considerations made for the master replacement procedure: network partition will induce inconsistencies in the registries of the masters, and compensating actions will follow.

## IV. Implementation of the unique Id using the MAC address

There are many ways to implement unique identifiers: here we focus on a solution based on MAC addresses, which are ready to use pieces of data that satisfy uniqueness. Unfortunately, there is a tradeoff between simplicity and effectiveness: the simplest variant of this solution incurs into the possibility of network collisions when the master registry is inconsistent. It happens since a MAC address is 6 octets long, while only 5 are available for extensions in the packets of interest.

We recall that a MAC address may be encoded in two different ways. In case the MAC is left with its factory settings (*universally administered*), the three most significant octets encode the manufacturer of the network card, while the three least significant octets contain an identifier which is unique for a given manufacturer. Otherwise the MAC address can be rewritten with a locally unique identifier (*locally administered*).

The case of *locally administered* addresses is not of particular interest, since it is straightforward to assign distinct MAC addresses with less than 5 significant octets. However, open environments may preclude the configuration of MAC addresses in network devices, so we explore also a solution that relies on universally administered MAC addresses. In [5] we present another solution under the same assumptions, but with a different footprint: it is computationally lighter, but requires more intervention from the master.

Our solution uses 4 octets and one flag S in DELAY_REQ and DELAY_RESP messages. A 4 octets digest is obtained from a generic MAC address using the following rule:

- the three least significant octets of the MAC are unchanged in the hash;
- the most significant octet of the hash is obtained using a hash function of the three most significant octets of the MAC address and of the SequenceId of the message.

The hashing function used to obtain the most significant octet may be very simple: for instance, it may consist of the UDP-style 1 octet long checksum of the 3 octets of the MAC and of the least significant octet in the SequenceId. In the unlucky case that two devices in the network have identical values for the 3 least significant octets, the probability of a hash collision is one over $2^8 - 1$, nearly $1\%$, each time one of the addresses is used. This frequency is considered acceptable only because hash collisions are effectively managed by the master clock, as explained below.

The slave includes the 4-octets long digest of the address of a *proposed destination* of the token in the DELAY_REQ.

The master collects MAC addresses from the frame headers of slaves that send DELAY_REQ, as well as from those

that dynamically join the network. It also records the last DELAY_RESP sent for each specific slave, and it is so that the master is able to evaluate timeouts.

In the absence of starving slaves, the master copies the *proposed destination* received with the DELAY_REQ into the DELAY_RESP, and the flag S is reset. Otherwise the master records the digest obtained from the MAC address of the starving slave in the DELAY_RESP, and sets the S flag.

Each of the slaves, observing a DELAY_RESP message, is able to decide if it is the destination of the token, by matching the digest reported in the message against that obtained from the local MAC address and the SequenceId contained in the message.

As noted above, a *hash collision* may occur when a DELAY_REQ message contains a hash matching the MAC addresses of more than one slave. The master is able to detect the problem, and to reroute the token to a non-ambiguous destination, thus avoiding packet collision. When one of the ambiguous destinations is starving the S flag disambiguates the destination, that corresponds to the starving one.

A serious problem emerges when the registry of the slaves maintained by the master is inconsistent: in this case a hash collision is not detected when colliding hashes correspond to slaves that have already joined, but are not present in the registry. A frame collision will occur in this case. However, the event will not happen again, since the slaves hit by the problem will starve, leave and re-join the system.

## V. EVALUATION OF THE NEIGHBORHOOD MAINTENANCE ALGORITHM

Evaluating the effectiveness of the neighborhood maintenance algorithm is a relevant issue, since a fair circulation of the token minimizes the master intervention needed to avoid slave starvation. This is relevant since the intervention of the master breaks the distributed nature of our protocol, and therefore introduces a vulnerability: the master needs an updated global view of the system to perform correctly, that contrasts with the claimed absence of a single point of failure. Our design option is not to totally avoid the presence of single points of failure, but to reduce their impact. In our case only a flaw in the master registry may have as a consequence, with a probability bound to the master intervention rate, the timeout of a slave: in fact we do not introduce a new single point of failure (the master is already essential for clock synchronization), and further we reduce its impact minimizing the frequency of its active intervention.

Since the evaluation of the effectiveness of the neighborhood maintenance algorithm is relevant, we propose analytical and simulation results that prove the effectiveness of the algorithm used in this paper. We use an analytical model of an optimal — yet inapplicable — algorithm as a benchmark, and simulation results for the others. We carried out the simulations using an *ad hoc* simulator written in Perl, whose code (about 100 lines) is available at `http://code.google.com/p/ispcs2009/`.

Our investigation aims at estimating the ultimate relevant parameter: the frequency with which the master has to reroute a packet to avoid the starvation of a slave. The results are shown in figures 1 and 2, and commented here below. They show that our algorithm significantly approaches the ideal case of a full mesh, and that it is substantially better than other alternatives.

We start giving an analytical model of the random walk of the token when connectivity corresponds to a full mesh: although inapplicable to our case, it provides a reference since it returns a sort of lower bound. An evaluation of the stochastic process of rerouting events is obtained analyzing the *return time* of the token during the operation in *legal conditions*. At each move of the token each slave clock has an identical probability $\frac{1}{n-1}$ to be selected as the destination of the token. In such a case, the return time has an exponential distribution with a mean of $n-1$ moves (here we approximate the geometric distribution of Bernoulli trials with an exponential since $n$ is assumed to be large). If two-way sessions occur at regular intervals, every $\tau$ time units, the time between two following visits of the token on a certain component has exponential distribution with mean $(n-1)\tau$.

The delay between successive visits has exponential distribution with a mean $(n-1)\tau$. Using an approximation that is valid for large systems, the probability of a wait $w$ longer than $\delta$, corresponding to a timeout event, is:

$$P(w > \delta) = e^{-\frac{\delta}{(n-1)\tau}} \approx e^{-\frac{1}{r}} \tag{2}$$

This function is labelled as *full mesh* in figure 2.

Since we envision neighborhoods significantly smaller than $n$, we expect, and the simulations confirm, an increment of master interventions compared to the above analytical model.

The *neighborood* relation is represented by the edges of a graph whose nodes are the slave clocks. For our purposes, the number of outgoing edges (the *outdegree*) from each node is a constant $d$ for all the nodes of the graph. Upon forwarding a token, the slave sends it along one of the outgoing edges, selected at random.

Excluding a deterministic computation of an overlay network of degree $d$, one appealing approach to the problem is to enforce a randomized composition of neighborhoods: the feeling is that, at least in the long run, the *return time* is going to be distributed evenly.

In order to conform to the assumption that each slave has an identical probability to be selected as the next destination for the token, each of the $n$ slaves should have the same number of inward edges, or *indegree*: in other words, in a network with outdegree $d$, each slave should be present in the neighborhood of other $d$ slaves (that derives from classical results that can be found in the tutorial paper [9]). This condition is difficult to enforce in our dynamic network without a centralized management of neighborhoods. To introduce a viable reference case, we evaluate by simulation an algorithm that randomly associates $d$ neighbors to each slave, without guarantees concerning the indegree: the resulting frequency of token rerouting events is shown in figure 1 as *randomized neighborhood*. But this algorithm relies on the existence of a globally accessible directory of slaves.

Distributed algorithms that maintain a list of $k \ll n$ neighbors selected at random has been recently investigated with reference to *ad hoc* networks in [2]. The authors introduce an algorithm (that we indicate as *reverse sampling*) that, transposed in our environment, consists in replacing one of the neighbors with the one that executed a two-way session $m$ token passing events ago, the $m$ parameter being around 10 for very large networks. The slave clocks keep updated a FIFO containing the identifiers of the $m$ most recently visited slave units by observing token passing operations in the network. The same algorithm applied to the PTP is explained in [5].

This algorithm has the beneficial effect of breaking potential *clusters*, but negatively affects the balance of incoming and outgoing neighborhood links. In fact, the event that the slave visited $m$ rounds ago is one of those with a lower indegree is low, since those with a lower indegree have fewer chances to be visited. As a consequence, slaves that are not sufficiently represented in the neighborhoods of other slaves for stochastic reasons tend to be visited even less frequently: in the absence of other compensating actions, the system eventually collapses to a clique of 2 slaves, thus making token circulation totally ineffective. The simulation results are indicated with *reverse sampling* in Figure 1.

The algorithm described in section III envisions a centralized management limited to the case of a timeout. When a timeout occurs, the starving slave — which is probably poorly represented in neighborhoods — is added to one of the neighborhoods, at expenses of a recently visited slave. This tends to stabilize the indegree of all the slaves around the outdegree. The result is shown in the line labeled with *swap on timeout* in figure 1.

The simulations referenced above have been run with a simulation time unit corresponding to $\tau$, in a system of 100 slave clocks with an outdegree of only 3 slaves. The rate $r = 0.1$ makes the system reasonably prone to collision events (with a probability around $1\%$, using equation 1), so to justify the introduction of a collision avoidance algorithm, but not so close to PTP scalability limits ($r = 1$) to be considered simply badly designed.

At startup, we assume that all slaves have a unique neighbor: all outbound edges point to the same peer. This is certainly an unfavorable condition from which the system has to recover.

In the case of *reverse sampling* the master intervention rate marginally improves with time, and stabilizes around $6\%$: this means that every 100 token passing operations the master reroutes 6 times the token to a starving slave. In case of a cold restart of the master, one slave starves and leaves the system every $18\tau$ on average. At the end of the experiment only $4$ slaves have an indegree of 3, and 33 are not present in any neighborhood.

This result is worse than that obtained starting with a static randomized neighborhood. In that case the master intervention rate is around $1\%$. As explained, the applicability of this solution to a dynamic distributed system is controversial.

The *swap on timeout* technique exhibits a dramatically better performance: the intervention rate rapidly drops to values around $0.03\%$, two orders of magnitude better than that obtained with reverse sampling. To understand the meaning of this result, consider the case of a system running steady: in case of a cold restart of the master with an empty registry, there is a lapse of $3000\tau$ on average to reconstruct the master registry before one slave starves. At the end of the experiment $67$ slaves out of 100 have an indegree of 3, and all the slaves are included in at least one membership.

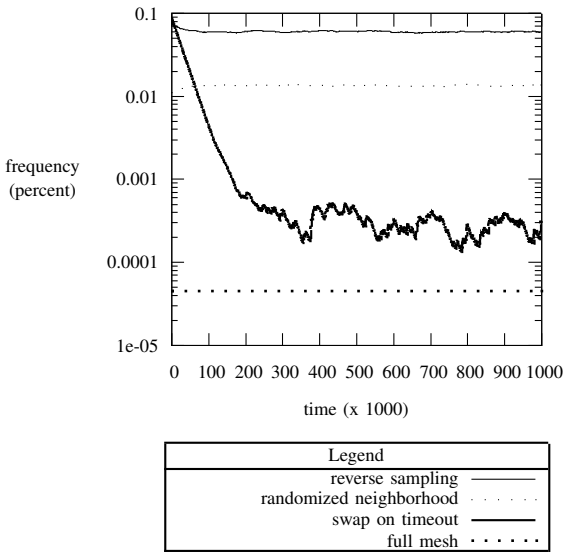


Fig. 1. Comparison of neighborhood maintenance algorithms ($r = 0.1$, $n = 100$): moving average ($k = 32$) of the frequency of token rerouting events. Logarithmic scale on y axis.

| a | $f(i \geq a)$ | $p(i \geq a) * N$ |
|---|---|---|
| 0 | 235 | 245 |
| 1000 | 169 | 180 |
| 2000 | 114 | 133 |
| 3000 | 86 | 98 |
| 4000 | 67 | 72 |
| 5000 | 51 | 53 |
| 6000 | 40 | 39 |
| 7000 | 30 | 29 |
| 8000 | 23 | 21 |
| 9000 | 16 | 16 |
| 10000 | 12 | 11 |
| 11000 | 11 | 8 |
| 12000 | 7 | 6 |

TABLE III
FREQUENCY OF INTER-ARRIVAL TIMES IN SIMULATION AND ACCORDING TO A POISSON PROCESS OF MASTER INTERVENTION EVENTS

The distribution of inter-arrival times between successive master interventions during the stationary behavior after time $200000\tau$ (see figure 1) is shown in table III. It is characterized by an average of $3266\tau$, with a standard deviation of $3658\tau$: this is in good agreement with an hypothesis of a Poisson process for master intervention events, with a confidence level of $87\%$ using a $\chi^2$ test.

A useful reference for evaluating the efficacy of the *swap on timeout* technique is given by the expected intervention rate in case of a full mesh network. In that case the analysis given at the beginning of this section returns a predicted intervention rate $e^{-\frac{1}{r}} \approx 0.005\%$. It is only one order of magnitude better than that obtained using the *swap on timeout* scheme with an outward degree of 3 in a system of 100 slaves.

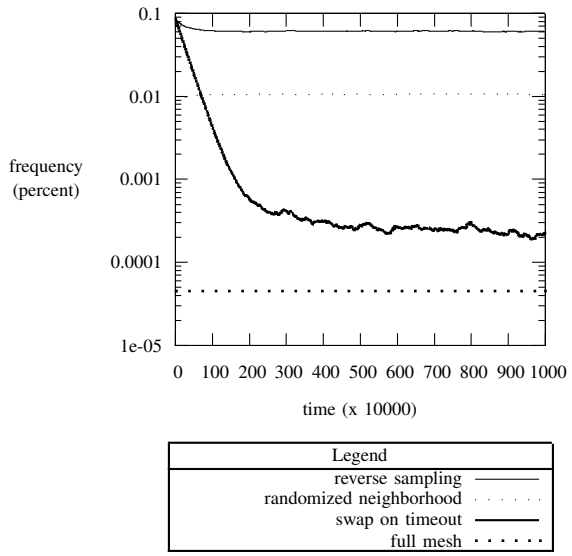


Fig. 2. Comparison between neighborhood maintenance algorithms ($r = 0.1$, $n = 1000$): moving average ($k = 32$) of the frequency of token rerouting events. Logarithmic scale on y axis.

We verify the scalability of our solution in a network of 1000 slaves, probably oversized with respect to current applications, keeping the outdegree to 3 neighbors. In figure 2 we observe that all three solutions appear to scale homogeneously, and that the *swap on timeout* technique keeps offering better performance. There is no apparent degradation in the master intervention frequency.

## VI. Summary and conclusions

The Precision Time Protocol is extremely flexible, and fits a number of operational environments: however collisions may degrade its performance and reliability.

We first characterized the application environments where collisions make a problem, and we quantified their occurrence using a simple analytic model.

The solution we propose falls within the boundaries of the standard: it uses messages defined in the standard in a way that conforms to the standard. A new message content, recorded in fields that the standard leaves available for extensions, adds semantics that are used to coordinate the slaves, thus avoiding collisions.

This results in a mutual exclusion algorithm, based on a token that performs a random walk in the network of slave clocks. The idea is not original *per se*, but here it is targeted to a specific case, exploiting the exceptional flexibility of PTP.

In order to make the basic idea of practical interest, we introduce techniques to enforce a maximum time between successive visits of the token to a given slave, and to dynamically optimize the interconnection graph.

The algorithm exhibits the fault tolerance features required for critical applications, that are typical for PTP: it is resilient to slave failures, and master failures have consequences only in case of an inconsistent replacement, and even in this case they affect only a limited number of slaves and for a limited time. Such consequences cannot be eliminated completely, but their likelihood is minimized by an appropriate design.

We follow an approach that is agnostic with respect to the link layer technology and to specific environments, so we are confident that our solution will be applicable also to future link layer technologies and in environments yet to define.

## References

[1] Standard for a precision clock synchronization protocol for networked measurement and control systems. Technical Report IEEE Std 1588-2008, IEEE Instrumentation and Measurement Society, 3 Park Avenue, New York, NY 10016-5997, USA, July 2008.

[2] 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.

[3] Jeff Burch, Kenneth Green, John Nakulski, and Dieter Vook. Verifying the performance of Transparent Clocks in PTP systems. In *International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pages 148–153, Brescia, October 2009.

[4] Callado, J. A.C., Kelner, A. C. Frery, and D.F.H. Sadok. Providing quality of service for clock synchronization. In *Telecommunications and Networking - ICT 2004*, volume 3124/2004 of *Lecture Notes in Computer Science*, chapter 3, pages 39–56. Springer Berlin / Heidelberg, 2004.

[5] Augusto Ciuffoletti. Collision avoidance for Delay_ Req messages in broadcast media. In *International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2009.

[6] Augusto Ciuffoletti. The Wandering Token: Congestion avoidance of a shared resource. *Future Generation Computing Systems*, 26(3):473–478, March 2010.

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

[8] Schlomi Dolev, Elad Schiller, and Jennifer Welch. Random walk for self-stabilizing group communication in ad-hoc networks. In *Proceedings of the 21st Simposium on Reliable Distributed Systems (SRDS)*, Osaka, Japan, October 2002.

[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] James Meditch and Chin-tau Lea. Stability and optimization of the CSMA and CSMA/CD channels. *IEEE Transactions on Communications*, COM-31(6):763–774, June 1983.

**Augusto Ciuffoletti** is a senior researcher at the Department of Computer Science of the University of Pisa. He participated to several EU funded research projects, recently to the *CoreGRID Network of Eccellence* in collaboration with the *Italian Institute of Nuclear Physics*, and he currently participates to the *CoreGRID Working Group*, where he represents topics related to Network Monitoring. He teaches a course on networking technologies for IP convergence.