# The Wandering Token: Congestion Avoidance of a Shared Resource

Augusto Ciuffoletti
University of Pisa - Dept. of Computer Science
Largo B. Pontecorvo - I56100 - Italy

**Abstract**

In a distributed system where scalability is an issue, like in a GRID [5], the problem of enforcing mutual exclusion often arises in a *soft* form: the infrequent failure of the mutual exclusion predicate is tolerated, without compromising the consistent operation of the overall system. For instance this occurs when the operation subject to mutual exclusion requires massive use of a shared resource.

We introduce a scalable *soft mutual exclusion* algorithm, based on token passing: one distinguished feature of our algorithm is that instead of introducing an overlay topology we adopt a random walk approach.

The consistency of our proposal is evaluated by simulation, and we exemplify its use in the coordination of large data transfers in a backbone based network. **Keywords**: congestion avoidance, random walk, token circulation, self-stabilization, soft mutual exclusion.

## 1 Introduction

In an ideal distributed system there are no centralized resources, all of them being equivalently able to play any role. However, in practical applications, it is often the case that the introduction of a centralized resource may be appropriate, in order to reduce the cost, or to improve the performance. The loss of scalability and fault tolerance, which is inherent to the introduction of a centralized resource, is accepted as a tradeoff.

For instance, consider the communication infrastructure of a network: be it geographical, or located inside a building, it will probably be based on a backbone. Although this may be replicated for fault tolerance, nonetheless it will break the symmetry of the system.

Components of a large distributed system may need to regularly perform operations that require a relevant share of such centralized resources: for instance, peripheral components may want to update local caches of data from a database, or upgrade local software by downloading new revisions from remote mirrors, or simply renew their credentials. In a system that wants to scale, the performance of such operations should not be tightly bound: *occasional* congestion of the centralized resource should produce a tolerable degradation of the high level service running at the edge.

In order to keep congestion an occasional event, applications must be equipped with a congestion avoidance mechanism. Such mechanism should coordinate the access to the centralized resource so that congestion is avoided as long as the overall load is nominal. When requests overtake the capacity of the resource, it should reproduce at the edge the effect of a congestion, but without stress for the centralized resource.

One key requirement for a congestion avoidance mechanism is that it must not introduce bounds on system size: this excludes the adoption centralized algorithms, that are not scalable, as well as distributed algorithms based on deterministic consensus, that have an heavy footprint.

Summarizing, unlike traditional mutual exclusion, modeled by a concurrent write on a shared register, our problem statement includes the occasional occurrence of simultaneous instances of the *protected* operation. This is due to the nature of the controlled operation whose performance may degrade when many are executed simultaneously, but without damage for the consistency of the system. This is formally translated in the following definition:

**Requirement 1** *A* soft mutual exclusion *algorithm for the* protected operational **A** *ensures that at any time, with high probability, there is just one component enabled to perform* **A**. *The probability that more than one component is enabled falls exponentially in the number of enabled components.*

We propose a simple distributed algorithm that implements soft mutual exclusion. The algorithm falls into the *peer to peer* family, since there is no centralized agent, and all participants run the same code. The result is the circulation of a token, ensuring that, with high probability, exactly one token is present in the system. We do not assume a fixed topology or a preliminary *overlay design* phase (as in [9], aimed at multicast). The token is circulated according to a *random walk* concept, that can be tailored to run on any topology. We evaluate its performance in a full mash that represents the transport level of the Internet. Formal results (see [8]) justify the claim that our algorithm may be of interest also in networks with an average degree comparable with $\log N$, where $N$ is the number of components in the system.

The algorithm relies on the knowledge of the *membership* of components that may execute the protected operation. Here we do not introduce a solution to the membership problem, but propose a way to regulate access to a shared resource once the membership is established. However, we insist on the fact that the knowledge of the membership on each component may be limited to O(log(n)) other components, randomy selected.

The relationships with Dijkstra's (deterministic) *self-stabilization* [4] are evident: however, instead of using the knowledge of neighbor's state, we enforce mutual exclusion using time constraints computed locally. We share with some *randomized* self-stabilizing algorithms the basic idea of performing random moves in order to compensate lack of information. Our approach may be regarded as an evolution of [6]: with respect to that work, we introduce a probabilistic definition of closure, since the token management scheme may itself induce divergence from stable operation. Such event occurs with low probability, and the algorithm autonomously recovers. As for the token elimination rule, the probability of collision is augmented by widening the *collision window* so that recovery is substantially improved with respect to [6].

Although many topics discussed in this paper have been individually treated in formal papers, here we prefer a simulation approach: this is motivated by the fact that the solution we introduce uses a combination of techniques, and this makes unapproachable a formal evaluation. Whenever appropriate, we will make reference to formal works that motivate the framework of our approach.

## 2 Establishing a benchmark: random access

To have a sort of reference, we introduce a straightforward solution to the problem of granting access to a shared resource without a centrilized control: each component issues a service request periodically, without any form of coordination. The period between two successive requests is based on a system wide period $\Delta_{min}$ perturbated by a random bias, chosen in the interval $[0, \Delta_{min}/2]$, which is introduced to break synchronous behaviors.

From a simulation lasted $10^6$ time units, assuming that all instances of the protected operation take 4 time units to complete ($\Delta_{op} = 4$) and with $\Delta_{min} = 600$, we understand that such algorithm is a *low end* solution to the soft mutual exclusion problem: in figure 2, the number of time units during which more than one protected operation is running falls exponentially with the number of simultaneous operations (see *benchmark* histograms). We performed two distinct simulations to show how the performance of such solution scales up when we step from a nominal workload ($N = 150$) to an overload ($N = 300$).

As we see, the number of concurrent operations is hardly acceptable, and the system has no way to adapt itself when the size of the membership scales up. This motivates our interest for a better algorithm, with a light footprint on system load.

## 3 System model and the wandering token basics

The system is composed of a set of $N$ *components*, interconnected by a complete mesh of *links*: for each couple of components $(c_i, c_j)$ there is a link $l_{i,j}$ that connects them. This simple model is meant to represent a *transport level* view of the Internet.

We do not assume sharply synchronized clocks, but their drift should be reasonably low: hypotheses on local timing are not critical, and will be explained in section 4.2.

All components run the same *algorithm*, which basically consists in receiving a token, possibly performing the *protected* operation, and finally passing the token to another component: only exceptionally a component may perform compensation actions (described below) to enforce the presence of a unique token in the system. The token passing *protocol* is a variant of the well known three way handshake, and is not described here: interested readers may find a exhaustive description and discussion in [3].

The time between two successive protected operations can be modeled using the cover time of a random walk in the system graph: in [8] the authors prove that the distribution of such random variable is characterized by a small probability after a value that grows with $O(nlogn)$. Therefore, we may expect that its distribution is comparable with that obtained using the reference random access algorithm introduced in section 2.

The algorithm is based on some *time constants*, which characterize the system. Although the algorithm is not overly sensible to the value of these constants (as shown by simulation), their evaluation is a critical step, and depends on the specific application. Here they are listed for increasing magnitude:

$\Delta_{skip}$ the minimum latency of a token inside a component: this is the time a token will stay in a component if the protected operation is not needed. A non null value ensures that the token will not bounce too rapidly, thus dissipating network resources;

$\Delta_{op}$ the typical duration of the *protected* operation;

$\Delta_{min}$ the minimum time between two successive *protected* operations on the same component.

$\Delta_{max}$ the maximum time between two successive *protected* operations on the same component (used to regenerate a new token when we suspect its loss).

We introduce a rate that binds two of them with the size $N$ of the membership:

$$k_{load} = \frac{\Delta_{op} * N}{\Delta_{min}}$$

When such rate is greater than one, we say that the system is overloaded.

# 4 The algorithm

The algorithm falls in the category of self-stabilizing algorithms, as defined by [4] and is probabilistic, in the sense defined in [1]. Let us first examine the stable behavior, that corresponds to the case where there is exactly one token in the system.

During *stable* operation the behavior of the algorithm is simple, consisting of receiving the token, performing the protected operation only in case the component performed the protected operation more than $\Delta_{min}$ time units ago, and finally passing the token to another component.

The probabilistic model that describes the occurrence of a protected operation on a edge component cannot be described analytically, since each event can last $\Delta_{skip}$, or $\Delta_{op}$, depending on when the previous event has occurred on that component. A simulation approach seems more appropriate to study the behavior of the system, and we will address simulation results in section 5.

The loss of a token can occur as a consequence of the failure of the token passing protocol: such infrequent event breaks the *stable* behavior. The *token generation rule* recovers from such event, and consists in generating a new token when a component does not receive one within a timeout $\Delta_{max}$ from the last token.

Indeed, it is far more frequent that the token is not really lost, but simply exhibits an anomalous delay: in that case, such rule may induce the simultaneous presence of multiple distinct tokens in the system. Therefore the token generation rule, which is introduced in order to recover from a token loss event, most times has the effect of disrupting the stable property by introducing *spurious* tokens.

In order to remove spurious tokens, we apply to a *token removal rule*: the component discards a token $t$ when it has been enabled from less than $\Delta_{min}$ time units by another token preceding $t$. Precedence is based firstly on the timestamp included in the token, and next on the generator identifier.

Such rule has the property that, in case of a wrong token loss detection, with high probability the spurious token will be removed after being passed a few times. In fact, consider that, at any time, many components have not exceeded their $\Delta_{min}$ timeout: if the spurious token visits any of them it will be thrown away. Since for each token passing operation the spurious token has the same probability of being removed, the probability that the token survives falls exponentially with time.

Spurious tokens are also generated in response to token loss: in such case, many components will exceed $\Delta_{max}$ in fast sequence, generating a new token in response. Let $t_{gen}$ be the time when the first new token is generated: if the token was lost more than $t_{gen} - \Delta_{min}$ time units ago, we are sure that the first generated token will persist, and all other tokens will be removed. Otherwise, all components that were visited by the lost component during the lapse $[t_{gen} - \Delta_{min}, t_{gen}]$, may wrongly remove the new token: following that, like in a sort of domino-effect, all regenerated tokens may disappear, thus reintroducing the anomalous state.

Although the problem of token elimination is well studied in theory, and is often referred as a solution to the leader election problem (see [2]), our setting disencourages a formal approach. Indeed, tokens are continuously generated, and two token meet when they hit the same component within a certain time interval (like in a worms game). These two facts make smart theoretical results, that are based on an initial population of tokens, and on exact collision of tokens for token elimination, useless fo our purpose.

In our case, the (adverse) probability of an early token loss detection is bound to the rate between $\Delta_{op} * N$ and $\Delta_{max}$: when this rate is small (around 0.2, according with our simulation), the probability that the regenerated token is removed is sufficiently small, and the system recovers to stable operation.

The previous statement introduces a sensitivity of the algorithm to the number of components in the system, that we discuss in the next section.

## 4.1 Dynamic join and leave

The dynamic variation of the membership that runs the token circulation algorithm is a key issue: although we disregard the implementation of the membership update protocol, we must consider that the number of components participating into such protocol ($N$ in the above formulas) may vary in time.

When we analyze the effects of a variation of the size of the membership, we must take into account that the role of our protocol is that of reproducing the behavior of a congested resource, when the number of components in the system increases.

When the size of the membership $N$ is significantly near the rate $\frac{\Delta_{max}}{\Delta_{op}}$, the system shows instability when the token is lost. Until the token is not lost, the algorithm performs normally, the intervals between successive protected operation on a component increase, while the rate $k_{load}$ reaches and overtakes the unit. After a token loss event, it may take a long time before the system goes back to its normal behavior (approx 20000 time units), and this may be inappropriate for certain applications, as shown in figure 4.

In order to cope promptly with the event of a token loss, each components must dynamically adapt the timeout $\Delta_{gen}$ used to regenerate the token, setting it to a value higher than $\Delta_{max}$ when needed. This adaptivity is obtained incrementing $\Delta_{gen}$ at a constant rate (50% in our simulation) each time the component needs to regenerate a new token, and decrementing of a small fraction (5% in our simulation) of the difference from $\Delta_{max}$ each time the component receives a token.

Such rule has no effect on a component as long as it regularly receives a token and $\Delta_{gen} = \Delta_{max}$. When a token is lost, all components that generate a new token rapidly raise their value, and the new value is used only if all tokens are lost again, which indicates that the system is inclined to instability. $\Delta_{gen}$ will keep growing exponentially, until the stable condition (just one token running) is reestablished. When the system reenters that mode, the value of $\Delta_{gen}$ on all components that incremented it during the unstability will logarithmically tend to $\Delta_{max}$ again.

Note that, while $\Delta_{max}$ is a system wide constant, $\Delta_{gen}$ is a local variable that is updated in response to local events.

Simulation shows that such rule (which is inspired by TCP congestion avoidance [7], but also by many self-regulating devices) is extremely effective in coping with $k_{load}$ larger than one, which corresponds to overload.

In figure 1 we show a programmer-friendly description of the wandering token algorithm.

```
Δ_gen = Δ_max
K^1_loss = 1.5;
K^2_loss = 0.05;
lasttoken = {timestamp = 0, id = NULL}
while (true)
        do
        select(receive(token), Δ_gen);
        if (defined(token))
          then
                Δ_gen = Δ_gen − K^2_loss(Δ_gen − Δ_max)
                if ((token → timestamp) − (lasttoken → timestamp)) ≤ Δ_per
                  then if (token → id) ≠ (lasttoken → id)
                                then
                                        next
                                else
                                        sleep(Δ_skip)
                                        send(token)
                        fi
                  else
                        execute(A)
                        send(token)
                fi
          else
                Δ_gen = Δ_gen * K^1_loss
                token = {timestamp = localclock, id = newid()}
                send(token)
        fi
        lasttoken = token
od
```

Figure 1: The wandering token algorithm

### 4.2 Clock synchronization

As for relative drift among clocks, which alters the measurement of time intervals, we should ensure that the various timeouts are measured appropriately, but the rules are not critical with respect to the measurement of the duration of such timeouts. A drift of $10^{-4}$, which is typical of a low cost crystal clock, is adequate. However, badly compensated clocks may induce inconsistent operation.

The application of some of the rules explained above requires a certain degree of synchronization among the clocks of the components. Such synchronization should simply ensure that the token $t_g$, generated in response to a timeout for a token $t_x$, has a timestamp larger than $t_x$: therefore the only requirement on clock accuracy is that it should be small compared with $\Delta_{max}$, which falls in the range of thousands of time units.

## 5 Simulation results

We carried out a series of experiments using a simple (200 Perl lines) *ad hoc* discrete event simulator, which is available upon request.

The simulation takes into account token loss, token regeneration and token removal, as explained in section 4. We do not simulate variable durations of the token passing operation, which is assumed to be negligible with respect to $\Delta_{skip}$, the minimum time the token is hold by a component.

We tested the regularity of the execution of the protected operation during a long run, as well as the number of tokens in a system characterized by the time constants in table 1. All simulations last $10^6$ time units.

In figure 2 we compare the results of the benchmark algorithm, with random access, and the algorithm regulated using the wandering token: the improvement is evident. When the system is not overloaded ($N = 150$) the number of time spent with more than one protected operation running is less than 0.5%

| | | |
|---|---|---|
| $\Delta_{skip}$ | 1 | |
| $\Delta_{op}$ | 4 | |
| $\Delta_{min}$ | 600 | $1.5 * (\Delta_{op} * N)$ |
| $\Delta_{max}$ | 1800 | $3 * \Delta_{min}$ |

Table 1: Time constants used in the simulation, in arbitrary time units

of the total duration of the experiment, while using the random access algorithm this amounts to more than 15%. The comparison is less favorable when the system is overloaded: in that case the system controlled using the wandering token spends 10% of the time running two or more protected operations, while using the benchmark algorithm this amounts to 40%.

The interval between protected operations reflects the progressive saturation of the resource moving from 150 to 300 users: the distribution of such random variable is shown in figure 3.

The transient following a token loss event is a critical issue: its length is relevant, as shown in figure 4, and its distribution is dispersed. However, during the stabilization transient, the algorithm shows good properties.

During most of the time there is at most one protected operation running: in our experiments, this statement is violated during 3% of the time in case of nominal load ($N = 150$), and in 9% of the time in case of overload ($N = 300$), which are comparable with the same figure during stable operation. However the distribution of such random variable has a longer tail, when compared with stable operation. For instance, in case of overload ($N = 300$), during stable operation we observed 8 time units over $10^6$ (0.0008%) where more than 5 protected operations were running (see figure 2), while during stabilization this occurred during (0.015%) of the time.

The interval between successive execution of the protected operation on a component exhibits a similar behavior: during stabilization most of the values of this random variable are concentrated about $\Delta_{min}$, but their dispersion is significantly higher: in case of overload, during stable operation only 9 intervals over $10^9$ are longer than $3000 = 5 * \Delta_{min}$ time units, which might mean starvation for the application that wants to perform the protected operation, while during stabilization this amounts to almost 15%. The application would probably assimilate this event to the temporary failure of the shared resource, and act accordingly.

# 6 Case study: multicast of a video stream

We introduce a case study that justifies the system constants used in the simulation described in section 5.

We consider a network where a video stream at $650KBps$ is broadcast to a group of receivers: at the same time different receivers are viewing distinct parts of the stream. The shared resource in a $200MBps$ virtual circuit, that is the backbone of the infrastructure.

We assume that applications are enabled to use the backbone for $\Delta_{op} = 4$ seconds, which corresponds to a data transfer of $800MB$. From the above we conclude that each application should be enabled to use the network once every 20 minutes: we set prudentially $\Delta_{min} = 10$ minutes.

Given the above, we can support at most $N = 150$ subscribers, each having a 4 seconds share every 10 minutes. We also examine the consequences of a 100% overbooking, and observe the behavior when $N = 300$.

Based on the above parameters, we can figure out the behavior of the system: as long as one token is wandering in the system and the number of units is nominal or less, 80% of the times the applications have access to the backbone within $2 * \Delta_{min}$: a moderate level of buffering allows a smooth reproduction. Occasionally (less than one percent of the time) two or more applications are allowed to use the backbone: this can be assimilated to an allowed burst of traffic along the Virtual Channel.

When the number of receivers grows up to two times the nominal size, the chance of concurrent execution increases, but the event that more than 3 data transfers are occurring simultaneously is rare, and should be manageable by the application and by the network with a noticeable but sporadic degradation of the performance. The application will be aware of the problem, since the period between successive data transfers is 90% of the times lower than $3 * \Delta_{min}$: this may induce the application to negotiate a lower quality of service, reducing the stream bitrate of about 30%.

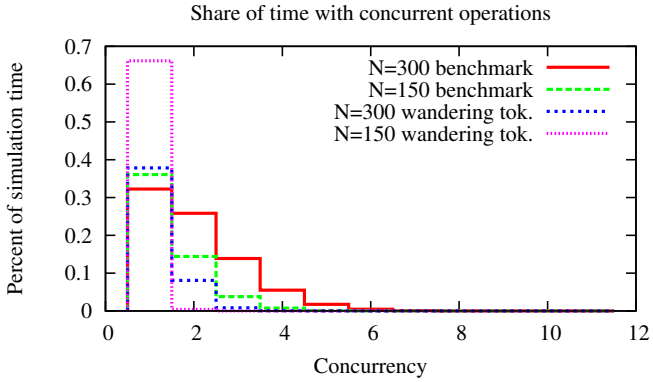


Figure 2: Benchmark algorithm vs. wandering token: distribution of number of concurrent operation on a shared resource for memberships of 150 (nominal) and 300 (overload) subscribers (simulation lasted $10^6$ time units with $\Delta_{min} = 600$ and $\Delta_{op} = 4$)

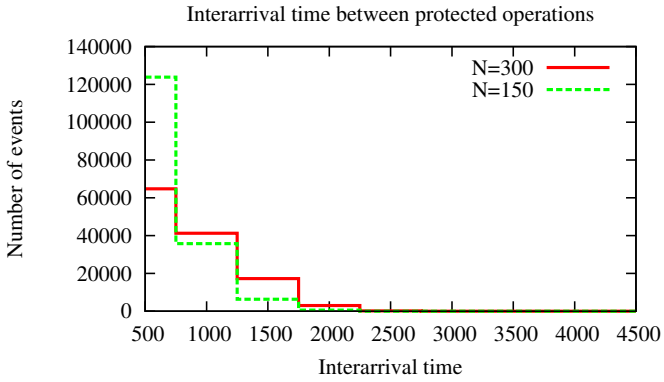


Figure 3: Wandering token: distribution of intervals between successive firing of the protected operation (simulation lasted $10^6$ time units with $\Delta_{min} = 600$ and $\Delta_{op} = 4$)

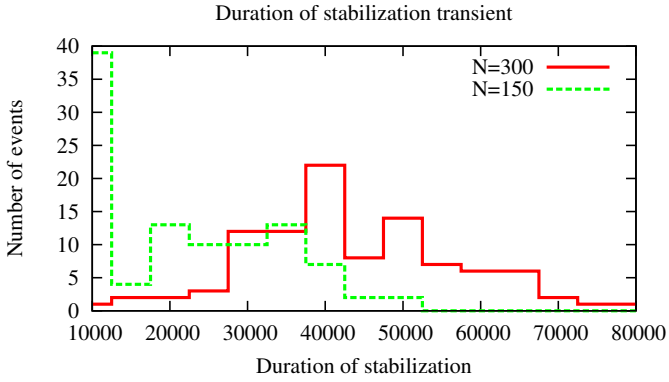


Figure 4: Wandering token: duration of stabilization after a token loss event (100 token loss transients with $\Delta_{min} = 600$ and $\Delta_{op} = 4$)

The transient after a token loss event may be quite long, taking several hours to terminate: during that time a system of nominal size performs, for the sake of simplicity, like an overloaded one. This may be admissible in our case study, since the user application will simply experience a slightly degraded quality of level of network. When, in addition, the system is overloaded, the congestion avoidance mechanisms of the underlying network may undertake traffic shaping.

# 7 Conclusions

The *wandering token* algorithm is proposed as a solution for an architecture where moderating the concurrent access to a shared resource can improve the performance. Its cost, in terms of communication and computation, is negligible.

The algorithm is fully scalable: when the scale (i.e., the number of components in the system) overtakes the capacity of the shared resource, the *wandering token* algorithm gradually reduces the resource share granted to each component, thus shielding the shared resource from the consequences of the overload.

While token duplication is avoided by a 3-way token passing protocol, the loss of the token cannot be excluded. When such event occurs, the operation becomes unstable for a quite long period: during that lapse, the features of the algorthm are degraded. We regard a better reaction to token loss as an open issue.

# References


[1] Y. Afek and G.M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.

[2] Nader H. Bshouty, Lisa Higham, and Jolanta Warpechowska-Gruca. Meeting times of random walks on graphs. *Information Processing Letters*, 69(5):259–265, 1999.

[3] Augusto Ciuffoletti. Scalable accessibility of a recoverable database using a wandering token. Technical Report TR-06-02, Universit di Pisa, Largo Pontecorvo - Pisa -ITALY, January 2006.

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

[5] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150:1–27, 2001.

[6] A. Israeli and M. Jalfon. Token management schemes and random walks yield self stabilizing mutual exclusion. In *Proceedings of the Ninth Annual ACM Symposium on Distributed Computing*, pages 119–129, Quebec City, Quebec, Canada, August 1990.

[7] Van Jacobson. Congestion avoidance and control. In *SIGCOMM ' 88*, Stanford (CA), USA, 1988. ACM.

[8] J. Jonasson. On the cover time of random walks on random graphs. *Combinatorics, Probability and Computing*, (7):265–279, 1998.

[9] G. Kwon and J. Byers. ROMA: Reliable overlay multicast with loosely coupled TCP connections. Technical Report BU-CS-TR-2003-015, Boston University, 2003.