IN SUPREMÆ DIGNITATIS
1343

Università di Pisa

Dipartimento di Informatica

Technical Report: TR-06-02

Scalable accessibility of a recoverable database using wandering tokens

Augusto Ciuffoletti

January 17, 2006

address: via F. Buonarroti 2, 56127 Pisa, Italy. tel: +39 050 2212700 fax: +39 050 2212726

Scalable accessibility of a recoverable database using wandering tokens

Augusto Ciuffoletti

January 17, 2006

Abstract

In a large distributed system there is usually the need to keep a record of the state of the components: one example is the database of the resources available in a Grid. The design of such database should take into account that the size of the system is not constant, and may vary of orders of magnitude: manual intervention to reconfigure the system in order to meet accessibility requirements may be problematic, and a source of unreliability.

We present a scheme that provides scalable accessibility to such a database, focussing on the performance of update operations. The scalable implementation of update operations enables ubiquitous replication of the database, thus bringing at reach the scalability of read operations. Each component of the system regulates automatically the relevant operational parameters with a kind of feedback, which is based on probabilistic considerations. Many aspect of the scheme are in fact probabilistic, and we introduce a simple model that describes the performance of the system.

The overall approach as well as certain performance figures that cannot be forecast analitically, are explored with simulation.

1 Introduction

The operation of a large distributed system is usually based on some sort of shared knowledge, that allows the coordination of its parts. The *peer to peer* approach stresses this aspect of distributed computing, on which security

depends: in a word, agents need to have a way to recognize trustworth partners, and ignore the others.

However, the introduction of such common knowledge may infringe the basic architectural principles of a distributed system, by introducing a bottleneck, otherwise called a single point of failure, consisting in the *server* that supports the database representing the common knowledge. The effects of such infraction range from the reliability to scalability.

Simple replication of the database (not simple indeed) does not completely solve the problem, since the correct configuration of the system may involve complex profiling of system activity, in order to bring up (or down) database replicas. Not the kind of activity one can reliably carry out in response to the variation in size of the system in a peer to peer environment.

To make the problem statement clearer, we characterize the accessibility of such database according to the following requirements:

- database should be dynamically configured in order to guarantee accessibility to be independent from system size and location of the querier (especially for update queries);
- dynamic configuration of database management should be automatic in a wide range of variability of the system size;
- configuration activity should have a footprint on local system resources which is independent from the size of the system.

Considering the application, we can introduce one further specification that brings at reach the implementation of such demanding requirements:

- the database content is recoverable, in particular in case some transaction is lost;

Let's analyze the meaning of such specification. The state of the art in distributed recovery binds the implementation of such statement to the fact that database content is managed according with the *atomic transaction* concept, and that transactions are timestamped consistently. Under such conditions, and introducing an appropriate recovery protocol, the database content can be recovered in case some of the transactions is lost or damaged. In particular, the database can be regenerated from scratch to create a new replica. Since this may involve a significant workload, the evaluation of the impact of recovery actions is relevant in our perspective.

1.1 A use case

Another argument to justify such working hypothesis is to describe an application environment, which is in fact the application for which the schema described in this paper was initially defined. In such environment the database contains the description of the properties associated to a group of entities cooperating through a peer to peer protocol. Such properties contain the certificates used to authenticate the partner agent, and the capabilities that describe the agent (in our use case they describe the service provided as components of a Grid).

Accessibility of such data structure is granted through a set of proxies that represent the *front end* of the database. While `select` queries are addressed to a proxy by a generic user application, `update` queries are addressed by the resource itself to a *local* proxy. The creation of a new proxy should be comparable, as for complexity, to the creation of any other resource in the system, basically requiring simply its registration in the database itself, after the installation of the required software. Manual intervention should be limited to that required for security reasons, basically to interact with the certification authority.

In such environment the accessibility requirements should guarantee that the following transactions have a cost that is independent from the size of the system:

- creation/deletion of a resource (included a proxy);
- caching a certificate (the certification authority is centralized);
- modifying a capability associated with a resource;

In such environment access rights are well defined and quite restrictive:

- only the resource itself can modify the record that defines it;

This originates a trivial timestamping rule, where each resource (either proxy or non-proxy) gives a progressive stamp to each `update` query: missing transactions in a log are identified by missing timestamps, and can be recovered downloading the missing transaction either from the resource itself, or from another proxy. As long as this operation is extremely infrequent, its occurrence can be tolerated without impact on the scalability of a solution.

Having introduced the environment where our protocol operates, and the requirements for its implementation, we have given an exhaustive definition of the problem. The next step is to introduce the solution.

2 Accessibility of the database content

We consider that each proxy contains a copy of the database, maintained in a backend database support, for instance a generic SQL relational database. Such database is generated as a routine operation during the installation of a proxy itself, which in turn is one step in the configuration of a new site.

Transactions are broadcast to the proxies using a peer to peer epidemic protocol, that uses a fixed number of tokens that randomly move from proxy to proxy. Since the comunication protocol used to move the token between two proxies is at the transport level, we assume that the topology of the network is a full mesh, where each proxy is adjacent with any other.

Therefore, in order to move the token, each proxy needs to know only the information already contained in the database itself. No overlay network is introduced, since routing data is unknown.

The only piece of data needed to boot a new proxy is the identity of another proxy: the *boot proxy*. We assume this information to be provided by the Certification Agency itself, upon registration of the new proxy. The identity of the boot proxy is used by the newcomer to download the content of the database, and to send one token to the boot proxy.

Each token contains a list of transactions, managed as a FIFO stack of bounded capacity. Upon receiving the token, a proxy checks the credentials of the sender: it will access the certification authority and update the local database only in case the proxy is not in the local copy of the database. If the sender proxy is trusty, the receiver scans the update list in the token and performs those that have not been executed yet: timestamps attached to each transaction help to identify them. In case the receiving proxy knows of transactions not recorded in the token, it pushes such transactions in the token, pushing out overflowing ones, and sends the token to another proxy, chosen at random among those in the local database.

Although apparently very simple (see figure 1), the protocol exhibits a performance that depends on many system parameters, among others the number of proxies in the system. Therefore it must be complemented by a protocol that regulates operational parameters, in order to stabilize its

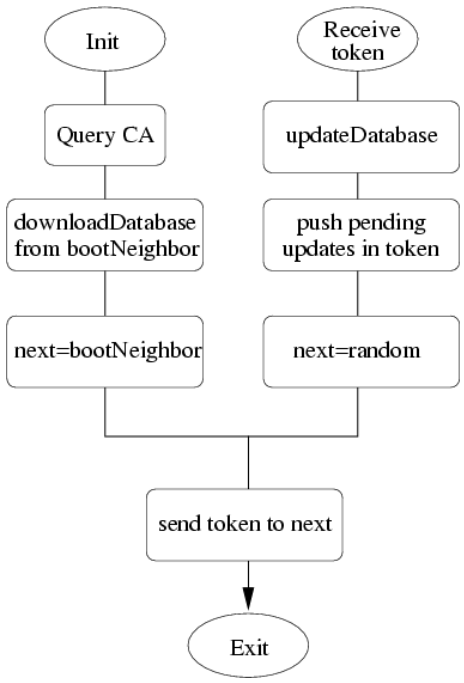

Figure 1: Flowchart of proxy operation

performance.

2.1 Performance stabilization

Performance stabilization is obtained using a feedback mechanism, based on a quantity which exhibits two basic properties:

- it reflects the performance of the system, and
- it is observable locally by a proxy

When a proxy detects a divergence of such quantity from a known expected value, which is a system constant, it initiates a compensating action, which aims at re-establishing the expected value of the control parameter. As any feedback mechanism, it is exposed to known problems, like divergence, oscillation etc. In order to avoid such problems, we need a formal understanding of the dynamics of the system.

The quantity we want to maintain stable (not the one used for performance stabilization, but the performance figure we want to keep stable independently from the size of the system), is the interval between the time when an update request is posted by a proxy, and the time when all proxies in the system have reflected the transaction in their local replicas. Since the protocol is inherently randomized, we need a probabilistic description of such time, that we call *saturation time*, T_{sat}. This corresponds to the latency of a transaction, and, for the kind of application we envisage, it should be bound in the range of the minutes.

Definition 1 *Let t_{sat} be the stochastic variable representing the interval between the time when an update transaction is posted by a proxy, and the time when the update is reflected by all proxies. Given a (low) probability value p_{sat}, we indicate with T_{sat} the time interval such that:*

$$p(t_{sat} > T_{sat}) < p_{sat}$$

The value of t_{sat} corresponds to the sum of two distinct time intervals: the interval before a new token arrives on the proxy, thus being *infected* by the transaction request (t_{wait}), and the time needed to propagate the transaction request to all proxies (t_{spread}).

The first component of the reaction time is the wait time t_{wait} before being reached by a token. Since the token wanders randomly in the system, with one wandering token in the system the probability that this time is higher than a given threshold T_{wait} is:

$$p(t_{wait} > T_{wait}) = \left(1 - \frac{1}{n-1}\right)^{T_{wait}/dt}$$

where dt is the latency of the token during its transit through a component. When, as in our case, N tokens are wandering in the system:

$$p(t_{wait} > T_{wait}) = \left(1 - \frac{1}{n-1}\right)^{T_{wait}*N/dt}$$

The spreading time after one token is infected with the update is computed for a single token wandering in the system, considering the probablity that a given proxy is not informed after 1 transmission as $(1 - 1/(n-1))$. The probability that it is not informed after k transits is $(1 - 1/(n-1))^k$, and therefore::

$$p(t_{spread} > T_{spread}) = \left(1 - \frac{1}{n-1}\right)^{T_{spread}/dt}$$

The generalization to N tokens is complicate: we simplify considering (optimistically) that all tokens contain the new update at the beginning of the diffusion. The simulation will assess the validity of this simplification:

$$p(t_{spread} > T_{spread}) = \left(1 - \frac{1}{n-1}\right)^{\frac{T_{spread}*N}{dt}}$$

Note that we have the same formula obtained for the t_{wait}.

We introduce a further simplification in order to compute the probability that the overall saturation t_{sat} be lower than a given threshold T_{sat}:

$$\begin{aligned} p(t_{sat} < T_{sat}) &> p\left(t_{wait} < \frac{T_{sat}}{2}\right) * p\left(t_{spread} < \frac{T_{sat}}{2}\right) \\ &= \left(1 - p\left(t_{wait} > \frac{T_{sat}}{2}\right)\right) * \left(1 - p\left(t_{spread} > \frac{T_{sat}}{2}\right)\right) \\ &\sim 1 - p\left(t_{wait} > \frac{T_{sat}}{2}\right) - p\left(t_{spread} > \frac{T_{sat}}{2}\right) \\ &= 1 - 2 * \left(1 - \frac{1}{n-1}\right)^{\frac{T_{sat}*N}{2*dt}} \end{aligned}$$

By using the definition of e, and acquiring that the estimate is optimistic:

$$p(t_{sat} < T_{sat}) = \left(1 - 2 * e^{\frac{-T_{sat}*N}{2*dt*(n-1)}}\right)$$

and:

$$p(t_{sat} > T_{sat}) = 2 * e^{\frac{-T_{sat}*N}{2*dt*(n-1)}}$$

by applying logarithms:

$$ln\left(\frac{p(t_{sat} > T_{sat})}{2}\right) = -\frac{T_{sat} * N}{2 * dt * (n-1)}$$

From this we obtain the appopriate number of tokens in the system, in order to meet the requirements for the reaction time:

$$N = -\frac{2 * dt * (n-1) * ln\left(\frac{p(t_{sat} > T_{sat})}{2}\right)}{T_{sat}}$$

A node has a direct feedback about the response time of the system: in fact, it knows that it should see a new token every t_{wait} time units. To maintain the system in equilibrium, each proxy should estimate the current value of t_{wait} (using ordinary exponentially weighted mean) in order to compensate token loss/duplication as well as system size changes.

The average value of t_{wait}, if the number of wandering tokens is the expected one, should be that of Poisson process of an experiment with a probability of failure of $N/n - 1$ every dt time units:

$$exp(t_{wait}) = \frac{(n-1) * dt}{N}$$

and, using the expected value of N:

$$exp(t_{wait}) = -\frac{T_{sat}}{2 * ln\left(p\left(\frac{t_{sat} \geq T_{sat}}{2}\right)\right)}$$

Let us use the values in table 1 for a use case: if we want that the reaction time is less than 40 seconds with a high probabiity (99.9%), assuming that the token has a latency of 30msecs each time it is passed from proxy to proxy, we need that each proxy receives a token every 2.63 seconds, on the average. All this is independent from the size of the system, but in the case of a system composed of 1000 unists we aim at having 11 tokens circulating in the system. Variations of system size should reflect in (linear) variations of the number of tokens.

We conclude that the compensating action of a proxy which observes an average interarrival time diverging from the expected value is to add (or remove) a token. Appropriate mechanisms are used to make the estimate of the average token interarrival time sufficiently robust.

2.2 A policy to accept new updates

The size of the message should be sufficient to accomodate all updates occurring during the expected reaction time; they are arranged in a stack of bound

capacity contained in the payload of the token. To avoid early overflow of update requests from such stack, the proxy should distribute (possibly delaying) such events in time. If we consider the token as a buffer of L positions where update requests wait for an expected time T, the global interarrival time of update requests should be less than T/L, in stable operation. For a single proxy:

$$ir_{update} = (T * n/L)$$

For instance (always using system constants in table 1), assuming a token contains 100 update requests, the average interarrival time of update request should be 400 secs, less than 7 minutes.

Since this value depends on n, the number of proxies in the system, it is exposed to two adverse facts:

- such value is in principle unknown
- it may change dynamically

As for the second fact, we argue that we cannot set up some sort of timer in order to regulate the occurrence of an update, since the value of such timer should change dynamically.

We opt for a regulation based on a probabilistic *policy*: if we compute the rate with which passing tokens can be used by a proxy to insert a new update, we are able to setup a random rule that converges to that expected value.

The target rate can be computed using the above expressions:

$$\frac{ir_{update}}{exp(t_{wait})} = \frac{T * N}{L * dt}$$

which is the desired rate of success for our randomized rule: note that it still depends on an unknown quantity, namely N, the number of tokens in the system. In our sample system (see table 1), the value of such rate is 147, which means that a generic proxy will assign a probability of 1 over 147 when deciding to insert a pending update in a passing token.

In order to compute such rate, each proxy should be able to compute the number N of tokens circulating in the system: to this end, each token is labelled with a random number upon generation. The chances of generating a duplicate label is negligible, and has limited drawbacks.

The mechanism for counting the tokens in the network is quite simple, although imprecise: a proxy keeps a list of *recently seen* token identifiers. When the proxy receives a token it checks whether its id is already in the list, and in case it inserts the new id. Otherwise it updates the timestamp. Ids in the list are checked, and older ones are popped out of the stack since corresponding tokens are considered as removed.

This concludes the description of the principles of operation of the overall scheme: we have introduced rules for the generation and removal of the tokens, as well as those that process the content of the packet. Since the capacity of the whole scheme is bound, we have introduced a simple *policy* to accept incoming updates: each proxy will process pending update requests based on a randomized rule.

2.3 Outline of an application interface

This clarifies the kind of Application Interface we can design for our distributed database: the application will submit an update request to the proxy, and wait for an acknowledgement. The request will be acknowledged promptly in case the proxy is within the capacity limits (one new request every more than 7 minutes). Otherwise the proxy will arrange the request in a queue, possibly returning the querier an estimate of its execution time. From the time when the acknowledgement is received, there is an additional latency of T time units (40 seconds in our example) before the update is visible anywhere in the system.

Note that the timing of the update is known to the application: it is in part based on a system constants, namely T, and, in case of overload, returned by the API itself. Therefore the application can be designed in order to take into account such timing.

2.4 Footprint

We conclude the analysis of our scheme with some considerations concerning the amount of traffic generated, which must be kept under control. As announced in the requirements, it should increase linearly with the number of components in the system. This is proved observing that the interarrival time of tokens on a given proxy is kept invariant (with an expected value which corresponds to $exp(t_{wait})$). Therefore the network load for each component is constant, and corresponds to:

$$bw = -\frac{2 * L * ln\left(\frac{p(t_{sat} > T_{sat})}{2}\right)}{T_{sat}}$$

In our example, let an update request description be 1KBytes long, for each proxy in the system we have an associated bandwidth of 38 KBytes/sec. For the whole system (possibly supported by a unique backbone!) this amount to 38 MBytes/sec which is negligible for a GByte class trunk.

Since the behavior of the overall is exposed to random effects, we should take into account also the cost for recovering from possible failures of the overall scheme. In our case, the inconsistency that may arise is the loss of *update requests*. There are many reasons for which this event may occur, that are bound to the probabilistic nature of the scheme. To give an account of some of them:

- the token containing the update request can be deleted by the application of token regulation rules;
- the request can overflow from all tokens before some of the proxies is reached;

Although such events are labelled as *infrequent*, it is hard to assess theoretically their cost: we apply to simulation for this aspect alone, and we anticipate that their impact is negligible (60 parts per million of the generated traffic) using an elementary recovery scheme.

We have thus concluded the explanation of the regulation mechanisms that stabilize the performance of our scheme: the interested reader may find an exhaustive *computer friendly* description of such mechanisms in the appendix.

The simulation results that follow are used to give a working evidence of the behavior of the system. We note that the simulator should not be used as a primary design tool: the design of a real system should be based solely of the expressions introduced above, using simulation only to confirm the results.

3 Simulation results

We implemented several simulators, in order to be able to test several aspects of the scheme under various conditions. Only the last simulator (`sim8.pl`)

T	40	seconds	expected latency of an update request
p	0.001		probability that a proxy is not informed within T
dt	0.03	seconds	latency of a token, between receive and resend
L	100	requests	capacity of the token (in update requests)

Table 1: System constant in a reference system

is adherent to a real implementation of the scheme: the other are used to observe certain events under controlled conditions.

In table 1 are system constants that are used for our experiments, and are realistic for a system composed of thousands of proxies in the application environment described in section 1.1.

These are the only data data are available to all proxies in the system, and are considered constants during the lifetime of the system (independently from the number and identity of proxies in the system, of course). Indeed, we consider them as hardwired in the installation package for the proxy.

3.1 Simulating the saturation time (`sim1.pl`)

This experiment simulates the diffusion of a single update request in the system, and collects statistics about the T_{sat}. The system (whose constants are those summarized in table 1) is composed of 1000 proxies and the appropriate number of tokens (11) is already circulating:

```
mean = 24.3
stddev = 5.0
T_sat = 19.1, count = 12
T_sat = 21.8, count = 26
T_sat = 24.5, count = 20
T_sat = 27.3, count = 24
T_sat = 30.0, count = 5
T_sat = 32.7, count = 7
T_sat = 35.5, count = 2
T_sat = 38.2, count = 2
T_sat = 40.9, count = 0
T_sat = 43.7, count = 2
```

We observe that the average saturation time differs from that computed analytically: the probability of $T_{sat} > 40$ should be 0.001. This is explained

observing that in our simulation we realistically assume that initially only one token is infected with the update: in the theoretical analysis we assumed instead that all tokens were informed at once. The time taken to spread the new update to all tokens, once the first one is infected, is small, but overall significant:

```
mean =  1.3
stddev =  0.2
time =  1.0, count = 4
time =  1.1, count = 9
time =  1.2, count = 13
time =  1.3, count = 29
time =  1.4, count = 21
time =  1.6, count = 8
time =  1.7, count = 7
time =  1.8, count = 3
time =  1.9, count = 3
time =  2.0, count = 2
```

As a consequence the theoretical diffusion is faster than the simulated (and expected in reality) one: however, this difference is not dramatic, and the following simulations prove that it is not noticeable in the overall operation of the system.

The only impact of this divergence is in the management of the number of tokens: the system might be inclined to inject tokens, and the number of tokens might stabilize slightly above the optimum. Although expected in theory, this effect has not been observed in practice.

3.2 Simulating a stable behavior (`sim2.pl`)

This experiment simulates the mechanism for token creation/removal. The system is initialized with the expected number of tokens (11 in our setup), and the number of proxies in the system does not change: therefore there should be few spurious creation/removal events, and the number of tokens should remain overall steady.

The creation/removal mechanism is based on the local computation of an expected value of the interarrival time: this is computed as an Exponentially Weighted Moving Average (EWMA) of observed interarrival times. Such

value is compared with the observed interarrival time, each time a token is received: if the observed value falls outside a given range around the target interarrival time (indicated as T_{wait} in the preceding formal analysis), the proxy either deletes the received token, or creates another one. The interval is set to $[T_{wait}/3, 3 * T_{wait}]$, implicitely assuming a certain simmetry in the distribution.

We observe that, during a time interval of 1000 time units, we collect 7 spurious events over approximately $3 * 10^5$ token passing events: the number of tokens in the system ranges from 11 to 8, and remains most of the time (800 time units over 1000) at 10. Higher rates can be obtained with tighter intervals: using 2.5 instead of 3 to compute the threshold values, we count 194 spurious events (number of tokens ranging from 3 to 18). Considering that each event entails some information loss, it seems that such system is too *nervous*. On the other hand, using a higher factor to compute the thresholds brings to a *lazy* behavior, which slowly adapts to variation of the system size.

```
mean:    3.01
stddev:    0.77
time = 1.8, count = 29
time = 2.2, count = 129
time = 2.7, count = 200
time = 3.1, count = 228
time = 3.5, count = 173
time = 4.0, count = 136
time = 4.4, count = 61
time = 4.9, count = 24
time = 5.3, count = 14
time = 5.7, count = 6
```

The interarrival time of tokens to the proxies, which is the figure that determines the functionality of the system, has an average value of 3.01 time units: this is above the expected 2.63. Such average is computed as the EWMA of the observed interarrival times. Local expectations are well concentrated around the average: none of the 1000 proxies is outside the range from half the average, and two times the average (standard deviation=0.77).

We explain the divergence from the expected interarrival time with a "wrong guess" concerning the shape of the distribution of the interarrival times: as explained above this is computed assuming a simmetrical shape.

This apparently makes slightly more likely to occur a *token delete* event with respect to a *token create* one, and tends to keep the system in a state with less tokens than needed (10 instead of 11, in our case). An empirical evaluation of such unbalance is hardly feasible, since involved events have an extremely low probability to occur (order of 10^{-4}): in fact we should have a significant number of values in the "tails" of the distribution, which requires a large number of values. Here we assume that such *first hit* assumption is sufficiently precise, and proceed with our experimental evaluation.

3.3 Simulating a growing system (`sim3.pl`)

This experiment simulates the instantaneous growth of the system: at time 500 the size of the system doubles, passing from 1000 to 2000 proxies. This stresses the token creation/removal mechanism beyond the capacity of the whole protocol: to avoid the overflow of token capacity, new joins should be distributed in time.

Despite the fact that the simulated event cannot occur in practice, the experiment is nonetheless significant in order to assess the efficacy of the token stabilization algorithm.

We observe that the reaction of the system takes approximately 100 seconds, from time 500 to time 590. During this lapse the number of tokens in the system increases of 12 units, reaching the number of 22 tokens in the system, which is consistent with the theoretical analysis (22.8 tokens). After that time the dynamics are overall typical of a stable behavior, and slopes down to 19 tokens.

The behavior of the system proves to be extraordinarily robust: following the stress, the number of tokens is almost immediately recovered, with no long range effects.

3.4 Simulating the diffusion of an update `sim4.pl`

This experiment simulates the diffusion of an update (a join in our case): the system is initially stable, with the consistent number of circulating tokens, and each proxy has a database containing a list of all other proxies (an anticipation of the local replica of the database). We simulate the join of a new component, and observe how other proxies learn about the corresponding update request.

The proxy joining the system is initialized with minimal knowledge, the identity of another proxy selected at random: the *boot proxy*. This is the information delivered by the Certification Authority to a joining proxy upon its registration. The first action of the joining proxy is to send a token to the *boot proxy*, which therefore learns about the presence of the newcomer. From this point on, this piece of news is spread using existing tokens.

In the simulation, the stable system is initially composed of 1000 proxies, and a new one is added at time 0, with a database containing only the identity of proxy 0.

The results are those expected in theory, and need no discussion. Instead, it may be interesting to observe the traffic induced by the join on the Certification Authority. We recall that, upon the receipt of a *join request*, a generic proxy should access the certification authority in order to certify the newcomer:

```
number of requests/time interval
    152 000000
    278 000001
    183 000002
    126 000003
     85 000004
     60 000005
     37 000006
     29 000007
     15 000008
     17 000009
      4 000010
      2 000011
      5 000012
      2 000013
      1 000014
      2 000015
      1 000017
      1 000018
```

There is a relevant accumulation of certificate requests, up to 278 during one time unit. This is a detail that may be relevant in practice.

3.5 Estimating the number of tokens in the system (`sim8.pl`)

This simulation aims at establishing the validity of the estimate made by a proxy about the number of tokens in the system: such estimate is useful to set the rate at which update requests are served, which is at the based of the acceptance *policy* outlined in the paper. This rate is bound, since the token has a fixed length, and we want that an update request persists inside the circulating tokens for at least T time units, before being pushed out by more recent request.

The number of tokens in the system is estimated by maintaining locally a list of *live* tokens. To this purpose, each time the proxy receives a token it scans that list: the estimate of the interarrival time for that token is compared with the difference between the current time and the last time the token was seen. If that difference is remarkably (10 times) too large, the token is considered as removed: the others are accounted as *live*. The interarrival time estimate is computed using the observed values for live tokens, not using theoretical results.

The simulation displays the estimates of the numner of tokens in a system of 1000 units, during an interval of 10000 seconds: every 100 seconds we check the value of the estimate on each proxy, and the plot displays the minumum, the maximum, the average value and the 90% interval of the estimates of n. From time 1000 the system is exposed to a growth at the admissible rate (one new proxy every 40 seconds): 100 new proxies are added during the interval from 1000 to 5000 time units.

In figure 2 we observe that the average value of the estimate is well above the real value. Since higher values induce a lower published capacity, this corresponds to a quite prudent attitude. A more aggressive system can be obtained by lowering the threshold for considering a token removed in the list of live tokens: as explained above in this simulation we set it to 10 (times the expected value).

The estimated value of the number of tokens tends to diverge when the system is exposed to transients due to induced or random phenomena, as shown in figure 3: during the interval from 1000 to 5000 the frequent joins evidently disturb the system. Around time 6800 an extremely unlikely event occurs: 10 of the 12 circulating tokens hit the same proxy during a time interval of less than 8 seconds. As a consequence, they are all deleted and only two tokens survive in the system. During the successive 40 seconds

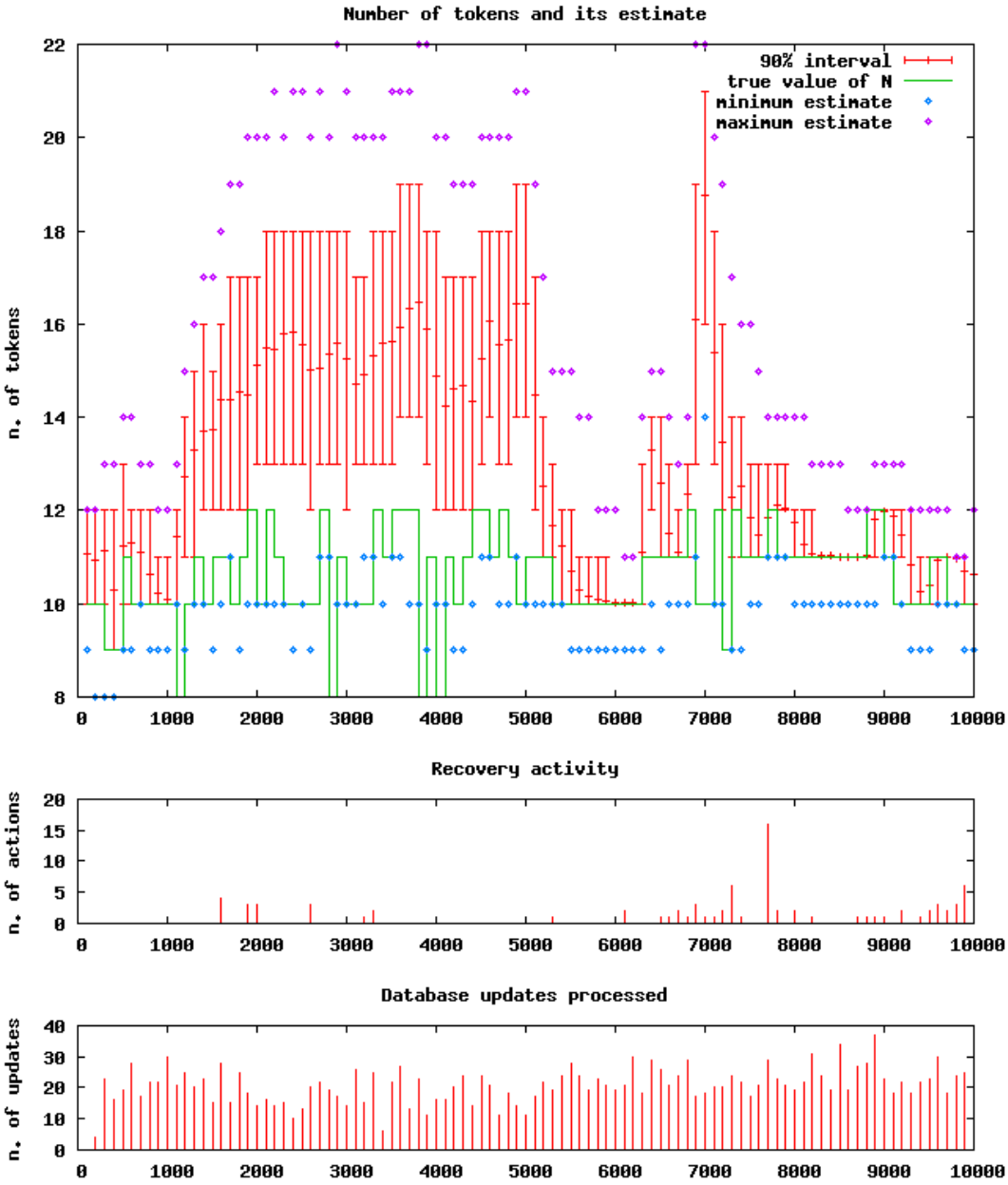

Figure 2: Overview of a simulation lasted 10000 seconds. All graphs summarize the activity of the system for time slots of 100 time units. The first graph shows the number of tokens in the system at the beginning of a time slot, and the corresponding estimate of the proxies. The second graph illustrates the recovery activity during each slot, in terms of the number of recovery actions executed during that lapse. The third graph outlines the activity of the system in terms of processed updates.

the system reacts with the creation of 39 tokens, of which 31 are deleted during the successive 10 seconds. The system ends up with 10 tokens, and the rebound lasts 50 seconds: a long range effect is recorded on the local estimates of the value of the number of tokens in the system, as well as in the number of recovery actions, as described in next section.

To observe the impact of the error in the estimate of N on the performance of the system, we have injected requests at the maximum rate, i.e. assuming that input queues on each proxy were permanently busy. For limits in computing resources, we had to reduce the lenght of the token from 100 to 10 updates slots, thus reducing the available throughput of 10 times.

We observed that the system performed 2042 updates during the simulation interval (summarized in figure 3): this corresponds to 1 event every 5000 seconds for each proxy, significantly higher than the expected interval of 4000 seconds. This is an observable drawback induced by the *liveness threshold* (which was set to 10, as noted above) used for the estimate of N.

3.6 Estimating recovery activity (`sim8.pl`)

The simulations reported in this section aims at observing the recovery activity induced by inconsistencies that may arise due to the probalistic nature of our scheme.

We use an elementary recovery rule, that consists in the following steps, performed for each update request contained in a token:

1. check if the update has been already executed on the local database replica, and in such case exit;
2. check if the timestamp associated in the database last update to the record of the proxy is immediately preceding that of the current update request;
3. if not, download and process the update requests from the proxy originating the current update request, starting from the one with timestamp following that in the local record, and up to the one with timestamp preceding the one of the received update;
4. process the update request and record the attached timestamp.

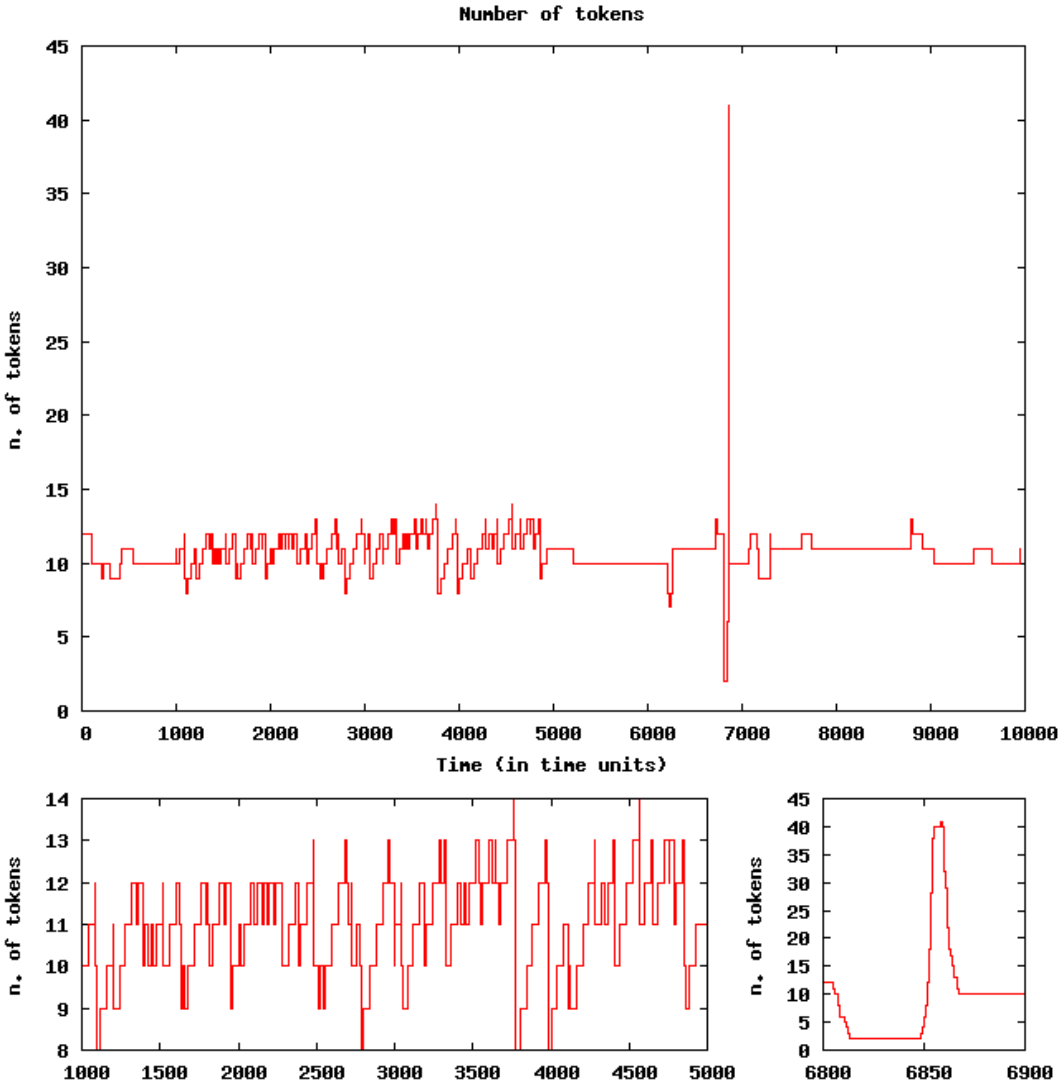

Figure 3: The variation of the number of tokens in the system. Smaller graphs are details of the larger graph during the two transients described in the paper.

Note that this simple rule is justified by our working hypotheses: only the proxy itself can request an update to his record in the database, and this allows a consistent timestamping of update requests.

According to this recovery scheme, we assimilate the load induced by the recovery of a *lost* update to that of a token passing operation: both of them consist of a point to point communication between proxies trusting each other.

The results we draw from the simulation show that the impact of recovery actions is significantly lower than expected: during the same experiment mentioned in previous section, which lasted 10000 seconds, we observed only 82 recovery actions. Estimating that, during the same interval, approximately $4 * 10^6$ tokens are passed in the system, the traffic induced by recovery activity corresponds to 20 parts per million. Instead, we expected a numer of recoveries similar to the number of performed updates, which equals to 2042. The reasoning behind the prediction is simple: a probability $p(t_{sat} > T_{sat})$ of 10^{-3} implies that, in a system of 1000 components, for each update request one component, on the average, remains uninformed. This prediction is not confirmed by simulation, and the system behaves better than expected.

It must be noted that traffic induced by recoveries appears to be clustered: in one case we had a cluster of 16 recovery actions addressed to the same record, on the same proxy:

```
25 1
13 2
 3 3
 1 6
 1 16
```

In figure 1 we summarize the occurrences of recovery actions during the experiment: note that an increase of recovery actions follows the anomalous event at time 6800 described in previous section.

4 Related works

The scheme presented in this paper takes ideas from many methodologies, each of which would require tens of bibliography items. However, the purpose of this report is not of giving a survey of the related literature, but of explaining the basics of a data sharing scheme which is meant to be original.

Therefore we will give a short listing of the basic ideas taken from research on distributed systems, giving somewhat historical references.

One basic idea is that of *random dissemination* of information, also referenced as epidemic diffusion. One key reference on this topic is [3], but also [2] is a somewhat mandatory reference.

Another key idea is that of a *probabilistic approach* to the control of a distributed system. One of the more impressive works on this subject, to which I feel indebted, is [5]. The concept is not that of a blind randomization, but that of a decision biased by an imprecise knowledge, which considers the additional task of recovering in case the decision is wrong.

Another idea is that of *self-stabilization*, introduced by Dijkstra in [4]. From that paper originated a powerful and fruitful twig of research. Here we complement the original idea with probabilistic considerations, not completely matching the flavor of [1], which introduces the term of probabilistic self-stabilization. The result is a system that undefinitely *tends* to stabiliziation, but which may deviate from this state even when the algorithm is executed correctly.

The use of multiple tokens can be traced back to [6], also in the stream of self-stabilization.

The overall approach based on feedback and filters has many unresolved relations with elementary concepts of control theory

5 Conclusions

Designing a control protocol that is able to scale up to thousands of autonomous components cannot rely on

Despite the fact that most of our activities depend on random phenomena, the introduction of probabilistic (i.e., randomized with a purpose) rules in a computing system is controversial.

We have used plenty of such rules in a scheme that solves a practical problem, which arised during the design of a network monitoring system: keep a globally accessible record of the descriptions of all resources in a large system composed of thousands of components. Such scheme must be able to regulate itself in order to maintain the expected behavior, even when the number of components in the system changes significantly.

The scheme we have defined falls in the category of randomized token circulation algorithms: thus the first randomized rule is that which determines

where to pass the token. But there are other decisions that are equally randomized:

- token creation and removal, needed to react to system size variations;
- shaping of incoming update requests from applications, needed to avoid incomplete propagations;

All these randomized rules are designed to aim at the desired result, but with a marginal probability of failure. Our goal was to prove that such failures have a minimum impact on perfomance.

Despite its complexity, the scheme can be evaluated analytically using elementary concepts of probability theory: this is a first step towards its applicability. In order to have a further insight in the work of the system, and to explore aspects that were not treatable analytically, we have implemented a simulator that reproduces the work of a real system. The use of the simulator substantially confirms the expectations, with some deviations that are discussed in the paper.

References

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

[2] Pittel B. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 1987.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithm for replicated database maintenance. In *Proc. of ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver (CANADA), August 1987.

[4] Edsger W. Dijkstra. Self-stabilization in spite of distributed control. In David Gries, editor, *Selected writings on Computing*, pages 41–46. Springer Verlag, 1982.

[5] Cristian F. *Exception handling*. Balckwell Scientific Publication, 1989.

[6] M. Flatebo, A.K. Datta, and A.A. Schoone. Self-stabilizing multi-token rings. *Distributed Computing*, 8(3):133–142, 1995.

A Computer friendly description of the algorithms

The following listings are derived from the source of the simulator (`sim8.pl`). Since the original language is Perl, the syntax used in the following is reminiscent of that language, although many simplifications have been introduced to make the result more readable.

The program makes an extended use of hashes, which a data type somewhat peculiar to Perl. The notations `->` and `=>` refer to hashes.

A.1 Proxy initialization

The following subroutine initializes a new proxy.

```
sub init ( id, proxy ) {
    id = id;
    timestamp=0;
    updatelist=[];
    active_tokens={};
# The last time a generation event occurred
    lastgen=0;
    warmup=WARMUP_COUNT;
# Initializes each node with estimated density equal to expected density
    est_ir=- T/(2*log(p));
# Download initial database from remote proxy
    download_database(proxy);
# Record the new join in the database
    update_database([newupdate("join")]);
# generate a new token
    inject_token();
}
```

A.2 Token management

The following subroutine manages a token, from its receipt to the delivery of the token to another proxy.

```
sub get_token () {
    token=udp_receive()
    target_ir=- T/(2*log(p));
    my @eventList=();
# updates the database with update requests in the token
    update_database(token->{updatelist});
# updates the active tokens list
    est_N=update_active_tokens();
# computes token interarrival time estimate
    if ( defined last_seen ) {
        est_ir = ewma(est_ir,(time-last_seen),ir_est_K);
        if ( warmup > 0 ) { warmup-- };
    }
    last_seen=time;
# Certain actions require that the local view of the system is stabilized
# after a warmup period
    if ( warmup <= 0 ) {
# open the gate for new requests from application with randomized rule
        gen_period=(T*est_N)/(L*dt);
        if ( random(gen_period) < 1 ) {
            gate_open=1;
        }
# if gate opened, check if update requests are pending
        if ( gate_open && pending_request() ) {
            update_database(accept_request());
            lastgen=time;
            gate_open=0;
        }
# injects/deletes tokens
        if ( est_ir > target_ir*LOW_THRESHOLD) {
            inject_token();
        }
if ( est_ir < target_ir/HIGH_THRESHOLD) {
            return;
        }
    }
# passes the token
    pass_token (token);
```

```
}
```

A.3 Database management

The database is represented by the `db` hash, which is indexed by a proxy id. Each entry contains an hash composed of a the following fields:

`timestamp` : a timestamp, corresponding to the timestamp of the last update performed on that entry;

any other fields relevant for the specific application.

The `update_database` function contains an abstraction of the database management: we detail only the management of the timestamp, which is relevant for our purposes.

```
sub update_database ( updatelist ) {
    foreach request in updatelist {
# define if source not included in database
        if ( ! defined db->{request->{source}} ) {
            db->{request->{source}}->{timestamp}=-1;
        }
# check if request is new
        if ( request->{timestamp} > db->{request->{source}}->{timestamp} ) {
# push new request in updatelist (to be included in tokens)
            unshift updatelist,request;
# check if database is updated
            if ( request->{timestamp} >
                 ( db->{request->{source}}->{timestamp}+1 ) ) {
# otherwise, issue recovery request(s)
                for ( i = db->{request->{source}}->{timestamp}+1;
                      i < request->{timestamp};
                      i++) {
                     recover(i, request->{source});
                }
        }
# process database transaction
        process(request);
```

```
        db->{request->{source}}->{timestamp}=request->{timestamp};
    }
# sort and cut the local updatelist
    sort(updatelist);
    while  ( #newlist >= L ) { pop updatelist; }
}
```

A.4 Tokens hash management

The set of known tokens is represented as a hash of indexed with token identifiers. Each entry contains a hash composed of the following fields:

`est_ir` : the estimated interarrival time of the token

`last_seen` : the last time the token passed

The `update_active_tokens` subroutine manages the update of the record associated to the passing token.

```
sub update_active_tokens {
    my k_est_ir=8;
# this is a threshold used to estimate the number of tokens.
# 10 is a quite conservative value, and tends to overestimate
# the number of tokens in the system, thus enabling less requests.
    my k_est_N=10;
    my t=active_tokens->{token->{id}};
# update intearrival time estimate and last seen time for the
# received token;
    active_tokens->{token->{id}}->{est_ir}=
        ewma(t->{est_ir},
             time - t->{last_seen},
             k_est_ir);
    active_tokens->{token->{id}}->{last_seen}=time;
# initialize estimate number of circulating tokens
    my est_N=0;
# count recently seen tokens
    foreach token_id ( keys active_tokens ) {
        ( defined at->{tid}->{est_ir} ) || next;
```

```
        if ( time - active_tokens->{token_id}->{last_seen} <
             k_est_N * active_tokens->{token_id}->{est_ir} ) {
            est_N++;
        }
    }
    return est_N;
}
```

A.5 Others

These subroutines are mainly introduced to improve readability.

```
sub pass_token (token) {
    udp_send ( {"id"=>token->{token_id},
                "source"=>id,
                "destination"=>random_neighbor,
                "updatelist"=>updatelist }
    };
}

sub inject_token (token) {
    udp_send ( {"id"=>rand(100000),
                "source"=>id,
                "destination"=>random_neighbor,
                "updatelist"=>updatelist }
    };
}
```