Performance stabilization of a token based epidemic diffusion

Augusto Ciuffoletti [1†]

[1]*INFN-CNAF – Via B. Pichat, Bologna – ITALY – e.mail:*`augusto@di.unipi.it`

We introduce a membership management scheme based on a number of tokens that randomly propagate advertisements within the membership itself. In order to stabilize update latency, in spite of relevant variations of membership size, we introduce a distributed rule that dynamically controls the number of wandering tokens.

A formal analysis of system behavior allows to compute the relevant design parameters, and simulation results prove the validity of our claims.

By way of a use case the reader appreciates the properties of extreme scalability, security and simplicity that makes the protocol appealing in a Grid environment.

Keywords: performance stabilization, randomized diffusion, membership management

1 Introduction and related works

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 trustworthy partners, and ignore the others. This problem is just an instance of *group membership* management, which has been largely studied in recent years [4, 5].

The introduction of a 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. This infraction disrupts system reliability and scalability.

Database replication is a partial solution of the problem, when it does not rely on human intervention to adapt to a changing environment: the concept of *federated service* can be used, (as in [2]) to make the system dynamically adaptable to a variable load.

However, this approach leaves unsolved the design of the mechanisms used to access the database, since an increase in the number of replicas degrades the performance of `update` operations, which must ensure the convergence of the system to a consistent state: fulfillment of this requirement may become quite expensive when system size scales up to thousands of replicas.

An interesting concept on this side is the *epidemic broadcast*, as illustrated in [3]: update requests are diffused peer to peer, selecting the targets of successive diffusion rounds using a randomized rule. Stochastic diffusion proves to scale well with system size, while ensuring a high degree of predictability.

The application of epidemic broadcast to the management of a membership is not a new idea [6], and the main problem with an epidemic-style diffusion is bound to load distribution: this can be non-homogeneous, in time and space, thus leading to congestion. In [7], the authors observe that epidemic diffusion tends to insist on domain boundaries. In [8] a discard policy is introduced to prevent congestion.

[†]This research work is carried out under the FP6 Network of Excellence – CoreGRID project funded by the European Commission (Contract IST-2002-004265)

In a spirit similar to [9], we introduce an epidemic diffusion scheme that exhibits a predictable overhead on the network and on the processing components of the system: congestion is therefore prevented. Local overhead is independent from system size, which can therefore scale up indefinitely. Expected update latency is kept invariant with respect to system size, so that distributed applications can rely on this value. Variations of system size reflect linearly on database operation throughput: a larger system will accept less update requests per time unit, exhibiting a linear degradation trend. An explicit admission control is introduced for update requests: applications that require a given quality of service are thus enabled to take appropriate actions depending on offered throughput.

In order to implement these features we introduce a randomized circulation of a number of tokens. Also in [10] authors note (for the case of a single token) that such a paradigm is appropriate to dynamic environments, making reference to wireless networks. Here we introduce, with reference to a Grid environment, a dynamically adaptive protocol that regulates the number of tokens depending on the size of the system (without having this number stored anywhere), in order to keep update latency invariant. Since we want to keep the message length as a system costant, it happens that we need to introduce an explicit acceptance of update requests: in a larger system a (linearly) longer time will pass between *back to back* successive admissions. An alternative to this performance degradation is the introduction of variable length tokens: in that case system throughput remains constant at the expense of the length of the token, which increases linearly with the size of the system.

1.1 A use case

The scheme that we propose has been designed to solve a specific problem, from which it takes the requirements: the database supports a directory service used to manage a membership of agents that implement a Grid service. Each entry contains a certificate used to authenticate the agent, as well as its capabilities.

In such a system, the load of the database in unevenly divided into `select` queries, which do not alter the database, and `update` queries. Since the number of `select` dominates the number of `update`, we opt to replicate the database, and make it accessible through *proxies*. A generic user application will address a `select` query to a local *proxy*.

In our case `select` queries have poor or no locality, so we consider that each replica contains the whole database. This makes the creation of a new replica an expensive operation, with a cost of the order of $O(n)$ due to the download of a nearly complete database from another replica. This drawback is acceptable in our use case, since the creation of a new replica is an infrequent event.

A common feature of probabilistic algorithms (and of any real world thing, in fact), is that it may fail: appropriate recovery mechanisms must be provided to cope with these events. In our case, the content of all replicas should converge, even in the event that the probabilistic epidemic diffusion fails to reach each proxy.

The state of the art in distributed recovery binds the implementation of this statement to the fact that database content is managed according using *consistently timestamped atomic transactions*. In our case, we observe that, while `select` queries are submitted by a generic user application, `update` requests are submitted by one of the resources represented in the database; access rights are therefore well defined and quite restrictive: *only the resource itself can modify the record that defines it*.

This originates a trivial timestamping rule, which simply requires that timestamps grow linearly in time, with no other bound with real time: missing transactions in a log are identified by missing timestamps, and are recovered downloading the missing update request either from the resource itself, or from another replica. In our case, the processing of a new *update request* consists of the following steps:

1. check if the timestamp associated in the database last update to the record of the service is immediately preceding that of the current update request;

2. 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;

3. process the update request and record the attached timestamp.

Summarizing, the accessibility of the database should conform to the following requirements:

- applications are enabled to commit an update request, with high probability of success;
- recovery events must be extremely infrequent, and transparent.
- dynamic configuration of database management must be automatic in a wide range of variability of system size;
- database management must have a footprint on local system resources which is independent from the size of the system;

2 Database management

Update requests are broadcast to all proxies in the system, that process them on the local replica. Broadcast is implemented by a peer to peer epidemic protocol, that uses a number of *tokens* randomly moving from proxy to proxy. Since the communication protocol used to move a token between two proxies is at transport level, we assume that network topology is a full mesh, where each proxy is adjacent to any other. No overlay network is introduced, and routes are unknown.

The only piece of data needed to boot a new proxy is the identity of another proxy selected at random: the *boot proxy*. We assume this information to be provided by the Certification Authority, upon registration of the new proxy, together with proxy certificate. The newcomer sends one (signed) token to the boot proxy, and downloads from it the local database replica. Although the download of the whole content of the database is not needed in principle (it has been proved that epidemic membership works efficiently with small parts of the membership known to members), we opt, for sake of adherence to our use case, to assume that the whole database is downloaded.

All tokens are signed by the sender using a private key associated with its certificate; upon receiving a token, a proxy checks its signature against sender's certificate, which contains its public key: it will query the sender for a full record of its capabilities and of its certificate only in case it is not already recorded in the local replica. If the token is valid the receiver scans the update list in the token: each token contains a list of update requests, managed as a FIFO stack of bounded capacity. The proxy will perform those that have not been executed yet: timestamps attached to each update request help to identify them. In case the receiving proxy knows of transactions not recorded in the token, it pushes these transactions in the token, pushing out overflowing ones, and sends the token to another proxy, chosen at random among those in the local database.

Such a simple protocol (see figure 1) exhibits a performance that depends on many system parameters, among others the number of proxies in the system. Therefore it must be complemented with a protocol that regulates operational parameters, in order to stabilize its performance, as announced by the title of this paper.

2.1 Performance stabilization

The performance figure we want to maintain stable is the latency of a an update request. Since the protocol is inherently randomized, we need a probabilistic description of this time lapse, that we call *saturation time*, t_{sat}.

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

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

In our environment, the value of T_{sat} is a system constant, and represents the expected latency of an update request, while the value of $1 - p_{sat}$ corresponds to our concept of "high probability". We compute the distribution of t_{sat}, introducing some approximation as explained in [1]:

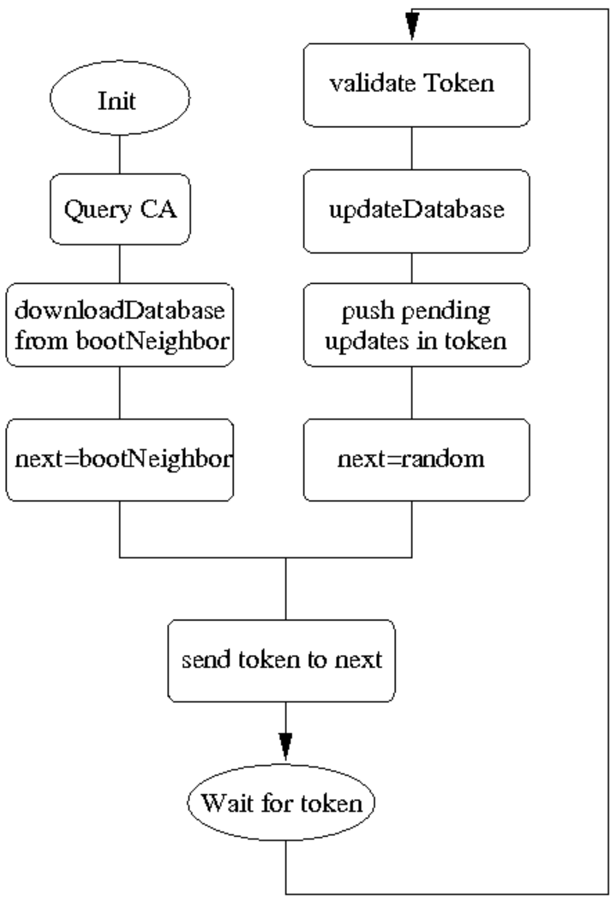

Fig. 1: Flowchart of proxy operation

T_{sat}	40	time units	expected latency of an update request
p_{sat}	0.001		probability that a proxy is not informed within T_{sat}
dt	0.03	time units	latency of a token, between receive and resend
L	10	requests	capacity of the token (in update requests)

Tab. 1: System constants used in examples and simulations

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

The appropriate number of tokens in the system, in order to meet the requirements for t_{sat}, is:

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

Therefore, by injecting or removing packets in the system, a proxy can control the target performance figure. Now we need to identify a *control parameter* to trigger compensating actions.

A node has a direct feedback about system response time: the average interarrival time between tokens, t_{wait}, is bound to the value of T_{sat} (see [1] for a formal explanation of this intuitive fact). The observed interarrival time of tokens can be therefore used to stabilize the target performance figure.

The average value of t_{wait}, if the number of wandering tokens is the expected one, should be the interarrival time of a Poisson process with a (low) probability of success $1/(n-1)$. Since N experiments (i.e., token transfers) are performed every dt time units:

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

and, using the expected value of N computed above:

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

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.

Finally, let us use the values in the table 1 for an example: if we want that t_{sat} is less than 40 seconds with a high probability (99.9%), assuming that the token has a latency of 30 msecs each time it is passed from proxy to proxy, we need that each proxy receives a token every 2.63 seconds ($exp(t_{wait})$), on the average. All this is independent from the size of the system, but in the case of a system composed of $n = 1000$ proxies we stabilize with 11 (N) tokens circulating in the system. Variations of system size should reflect in (linear) variations of the number of tokens.

2.2 A policy to accept new update requests

The size of the message should be sufficient to accommodate all updates occurring during T_{sat}; they are arranged in a stack contained in the payload of the token. A straightforward design option might be the introduction of some kind of *time to live* for each entry in the stack bound to T_{sat}, a system constant. This apparently appealing option has two practical limits: it introduces some kind of common time reference in the algorithm (which till now has none), and implies that token size grows linearly with system size. To justify the latter point, we consider that we may expect that the number of update requests per time unit grows linearly with the size of the system.

We opt for a difference strategy, which is more adequate to our use case: we keep constant the capacity of the token, and assume that update requests are submitted to an admission control that may delay them.

Once admitted, they are served within the T_{sat} deadline with known probability. In our case, this alternative is an acceptable compromise between the predictability of update latency, and the increased workload.

The goal of the admission control will be to ensure that each update request persists in the token for the time needed to reach all proxies in the system: for this each proxy will limit the frequency of admitted update requests. If update requests on a proxy are too frequent, they are delayed so that they do not prevent the diffusion of other update requests. In our use case update requests are infrequent, and local congestion is not an issue. Instead, we want to use UDP packets as token carriers, for which a fixed size is notably an advantage, and want that update requests commit are predictable, not fast.

Once the admission control is passed, the processing of an update request follows a FIFO discipline: it is pushed in the first place in token stack, while the last one is pushed out.

We consider the token as a buffer of L positions where update requests should stay for an expected time T_{sat}: in order to avoid premature overflow, the average interarrival time of update requests (considering the whole system) should be less than T_{sat}/L, in stable operation. On a single proxy:

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

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

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

Concerning the latter point, we argue that we cannot set up any sort of timer in order to regulate the occurrence of an update, since the value of this 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:

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

which is the desired rate of success for our randomized rule: note that it depends on N, the number of tokens in the system, which is unknown to the proxy.

The mechanism we propose for estimating the number of active tokens in the network is quite simple, although imprecise: tokens are identified by a uid, generated at random upon token creation, and each proxy keeps a list of *recently seen* token identifiers with an attached timestamp. Upon receiving a token the proxy updates the associated timestamp. Token id timestamps are checked, and recent ones are associated to living tokens, while those whose age exceeds by several times the value of $exp(t_{wait})$ are popped out of the stack. Note that, also in this case, timestamps are not representative of a global time reference.

2.3 Database management footprint

We require that the load on each proxy does not depend on the size of the system: this requirement holds since 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 proxy is constant, and corresponds to:

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

Since the behavior of the overall system is exposed to failures, we should take into account also the cost for recovering from them. In our case, the failure materializes as the loss of some *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:

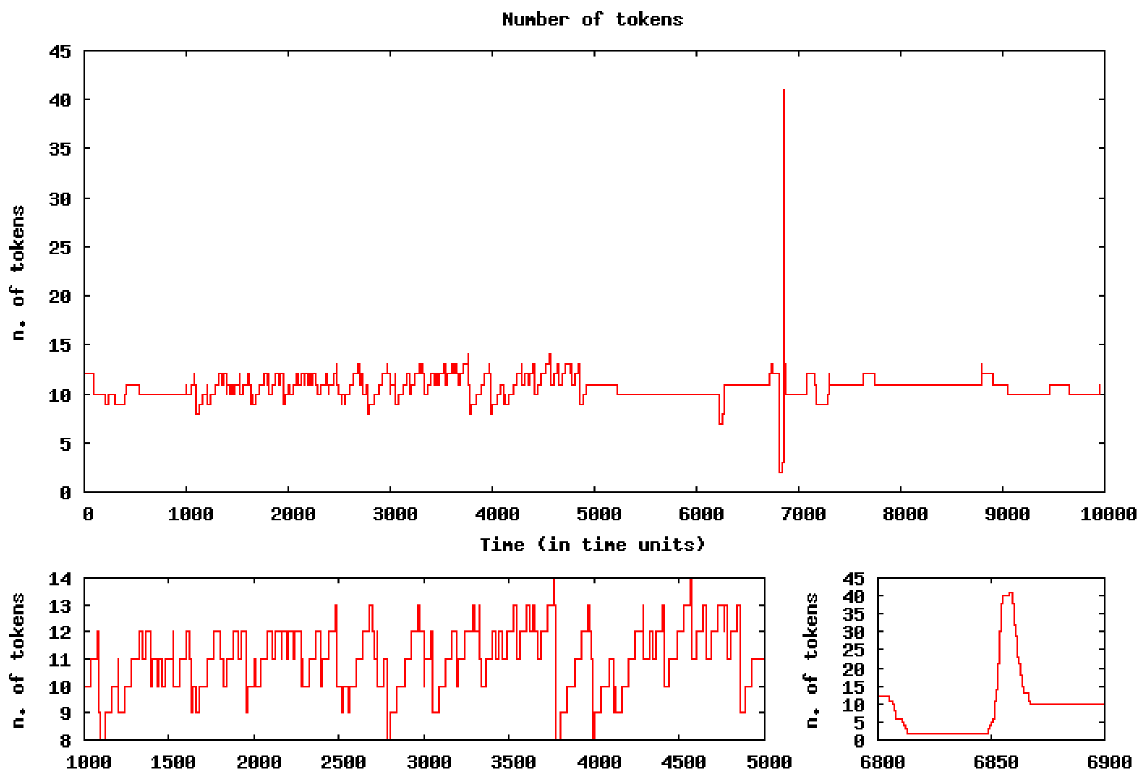

Fig. 2: Variation of the number of tokens in the system. Smaller graphs are magnified views of two transients described in the paper.

- the token containing the update request is deleted as an effect of the application of token count regulation rules;
- the request overflows from all tokens before some of the proxies is reached;

Although these events are labelled as *infrequent*, it is hard to assess analytically their frequency.

The simulation results that follow are used to give a working evidence of the behavior of the system, and to have an idea of the frequency of recovery actions. We note that the simulator is not used as a design tool: the design of a real system is based on the expressions introduced above, using simulation only to confirm the results.

3 Simulation results

We used a simulator written ad-hoc (500 lines of terse Perl code, available on demand) to simulate a system initially composed of 1000 proxies during 10000 time units: from time 1000 the system rapidly grows, and 100 new proxies join the system during the successive 4000 time units. We observe the stabilization transient during the final 5000 time units. Update request queues on proxies are assumed to be permanently busy, so that throughput is maximum. Table 1 summarizes system constants, the same used in the previous example.

Note that token capacity is quite low: 10 update requests. This is due to the limited scalability of the simulator. In a working environment a much higher capacity is more appropriate. In addition, some reader may find useful to replace “time unit” with “second”, to have an idea of the applicability of our results.

Figure 2 summarizes the stabilization scheme dynamics displaying the variations of the number of wandering tokens.

We observe that, during stable operation (intervals $[0,1000]$ and $[5000-10000]$), the number of tokens is stable as well, although slightly below the expected value (11 tokens in our case): this fact is justified by the approximations introduced with theoretical analysis, and has no practical relevance.

The number of tokens oscillates when the system is exposed to transients due to induced or random phenomena. Magnified views of these transients are shown in the same figure.

During interval $[1000,5000]$, frequent joins slightly disturb the system: however, extra tokens generated by join events are quickly removed.

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 time units (the expected interarrival time between tokens on a proxy is 2.6 time units). As a consequence, they are all deleted and only two tokens survive in the system. During the successive 40 time units the system reacts with the creation of 39 tokens, of which 31 are deleted during the successive 10 time units. The system ends up with 10 tokens, and the rebound lasts 50 time units: a long range effect is recorded on local estimates of the number of tokens in the system, and produces a number of recovery actions, as shown in figure 3.

Figure 3 aims at showing the performance of the system as perceived by a applications submitting update requests, as well as an insight of its operation. The time interval has been divided into slots of 100 time units each, and aggregated statistics are shown for each slot.

The bottom graph illustrates the number of update requests accepted during a time slot: we observe a minor performance degradation during system growth during interval $[1000,5000]$, but the throughput is overall stable. The average interval between successive services on each proxy is around 5000 time units, which agrees with the expected value of 4000 time units (ir_{update}).

The middle graph reports the number of recovery actions during each time slot: we observed 82 recovery actions. According with the recovery scheme introduced in section 1.1, 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. Since during the simulated interval approximately $4*10^6$ tokens are passed in the system, the traffic induced by recovery activity corresponds to 20 parts per million.

The top graph shows the estimated value of the number of tokens: we observe that it tends to diverge when the system is exposed to transients due to induced or random phenomena. In fact, the extra tokens generated by joins, although promptly removed, persist in proxies data structures and are considered as living for a long period of time: this distorts the estimates of the proxies concerning the number of living tokens, but has no overall performance effects.

4 Conclusions

We have introduced a broadcast scheme based on epidemic diffusion. The key feature of that scheme is the predictability of its overhead and performance, despite it is based primarily on randomized decisions, regardless the size of the system.

We discuss the applicability to such a scheme for the solution of a practical problem, taking into account issues that are relevant in practice, like security, packet sizing, and design parameter estimates. We conclude with results for a system of 1000 proxies obtained using a simulator designed ad-hoc.

References

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

[2] Francisco Matias Cuenca-Acuna and Thu D. Nguyen. Self-managing federated services. In *Proc. of the 23rd IEEE Symposium on Reliable Distributed Systems*. IEEE Press, October 2004.

[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.

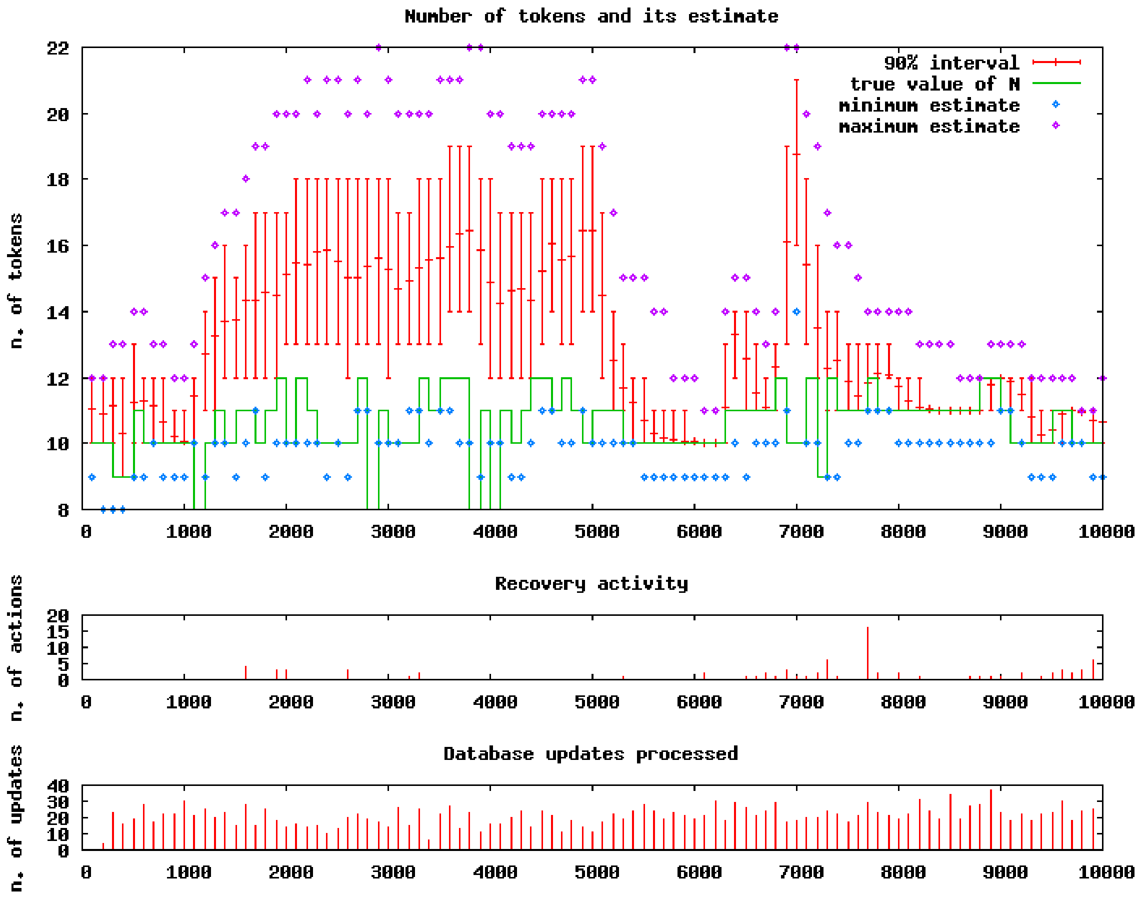

Fig. 3: Overview of a simulation lasted 10000 time units. Time is represented along x axis, and all graphs take samples every 100 time units. The top graph shows the number of tokens in the system at the beginning of each time slot, represented as a continuous green line, and the corresponding estimate of the proxies: red vertical lines represent intervals containing 90% of local estimates, dots and crosses respectively minimum and maximum local estimates. The middle graph illustrates the recovery activity during each time slot, in terms of the number of recovery actions executed during that lapse. The bottom graph outlines the activity of the system as observed by user applications, in terms of processed updates.

[4] Cristian F. Agreeing on who is present and who is absent in a synchronous distributed system. In *Proc. of 18th International Symposium on Fault-Tolerant Computing*, pages 206–210, June 1988.

[5] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massouli. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2), February 2003.

[6] Richard A. Golding and Kim Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, University of California, Santa Cruz, 1992.

[7] Indranil Gupta, Anne-marie Kermarrec, and Ganesh Ayalvadi. Efficient epidemic-style protocols for reliable and scalable multicast. In *Proc. IEEE Intl. Symposium on Reliable Distributed Systems*, 2002.

[8] J. Pereira, L. Rodrigues, J. Monteiro, M., R Oliveira, and A.-M. Kermarrec. NEEM:network-friendly epidemic multicast. In *Proc. of the 22nd IEEE Symposium on Reliable Distributed Systems*, Florence, October 2003.

[9] L. Rodrigues, S. Handurukande, J. Pereira, R. Guerraoui, and A.-M. Kermarrec. Adaptive gossip-based broadcast. In *Proceedings of the International Conference on Dependable Systems and Networks*, San Francisco (CA), June 2003.

[10] Bernard Thibault, Alain Bui, and Olivier Flauzac. Topological adaptability of the distributed token circulation paradigm in faulty environment. In *Parallel and Distributed Processing and Applications*, number 3358 in Lecture Notes in Computer Science, pages 146–155. Springer Berlin / Heidelberg, 2004.