The algorithm of pipelined gossiping

Dynamics of a Time-outs Management System

Vincenzo De Florio and Chris Blondia

University of Antwerp, Dept. of Mathematics and Computer Science
Performance Analysis of Telecommunication Systems group
Middelheimlaan 1, 2020 Antwerp, Belgium, and
Interdisciplinary institute for BroadBand Technology
Gaston Crommenlaan 8, 9050 Ghent-Ledeberg, Belgium

We call "time-out" a software entity that postpones a function call by a given amount of time expressed as local clock ticks. The subject of this paper is a simple software system managing lists of time-outs. We briefly describe that system and discuss its dynamics. We show some experimental results that prove the emergence of complex behaviors.

1. Introduction

A useful tool for distributed systems development is a time-outs manager. A time-outs manager is a software system that manages lists of "timeouts," i.e. objects that postpone a certain function call by a given amount of time. This feature is an interesting one, as it allows to transform time-based events into non time-based ones. A typical example is a time-out that on expiring sends an alarm message to the client process. A multiple selection statement such as the C language's switch can then be used to deal with both classes of events.

This paper focusses on a time-outs manager that had been designed to facilitate the development of some fault-tolerant applications for European projects TIRAN [1] and DepAuDE [4]. Our time-outs manager maintains a list of time-out objects ordered by ticks-to-expiration as described in [5]. When the specified amount of time elapses for the top of the list, its function is executed and the object is either thrown out of the list or renewed (in this second case, the time-out is said to be cyclic). A custom process checks periodically the expiration of their entries and executes the corresponding functions. This period constitutes a trade-off between performance overhead and maximum function call delay.

A key pre-requisite for a correct behavior of our system is that the functions associated with the time-outs do not interfere "too much" with the time-outs manager. If this is not the case, which happens e.g. when the time-out function is a long lasting and very CPU intensive one, then there is competition. This paper describes the dynamics of

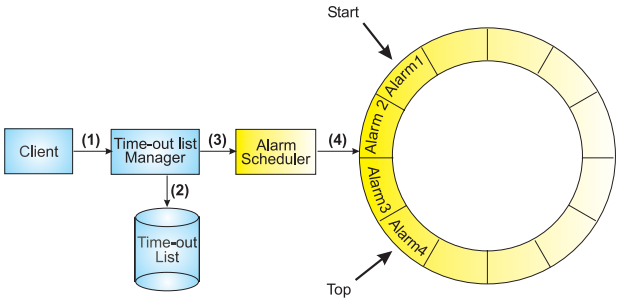

Figure 1. Architecture of the time-out management system.

our time-outs manager in the presence of such competition and the consequential emergence of complex behaviors as another example of complexity arising from simple initial conditions.

The structure of this paper is the following one: Section 2 introduces the time-outs manager. Section 3 presents the dynamics aspects of our system. Conclusive remarks are given in Sect. 4.

2. A Time-outs Management System

This section briefly describes the architecture of our time-outs manager. Such system may be regarded as a client-server application in which the client issues requests according to a well-defined protocol while a server process fulfills those requests by registering, updating, modifying, or purging entries in a time-out list, also executing the corresponding time-out functions. Let us call "alarms" such functions.

2.1 The Architecture of the Time-outs Manager

Figure 1 portrays the architecture of our time-outs manager: in

(1), the client process sends requests to the time-out list manager; in

(2), the time-out list manager accordingly updates the time-out list with the server-side protocol described in Sect. 2.2.

(3) Each time a time-out reaches its deadline, a request for execution of the corresponding alarm is sent to a task called alarm scheduler.

(4) This latter allocates an alarm request to the first available process out of those in a circular list of alarm processes, possibly waiting until one of them becomes available.

Figure 2 shows the sequence diagram corresponding to the initialization of the system and the management of the first time-out request.

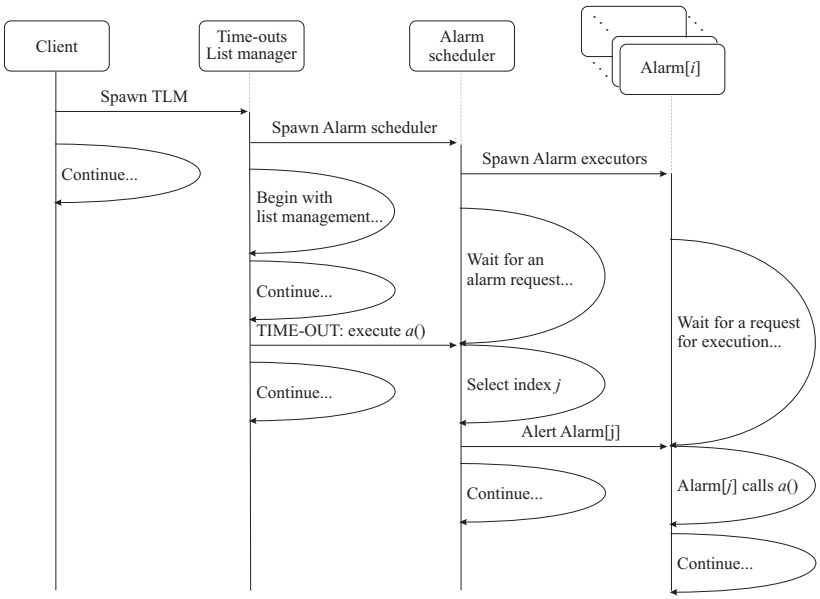

Figure 2. Sequence diagram for the tasks of the time-outs manager.

The presence of an alarm scheduler and of the circular list of alarm processes can have great consequences on performance and on the ability of our system to fulfil real-time requirements. These aspects are dealt with in Sect. 3. Our system may also operate in a simpler mode, without the above mentioned two components and with the time-out list manager taking care of the execution of the alarms.

The time-outs management class appears to the user as a couple of new types and a library of functions. Table 1 provides an idea of the client-side protocol of our tool.

The server-side protocol is run by a component called time-out list manager (TLM). TLM basically checks every TM_CYCLE for the occurrence of one of these two events:

- A request from a client has arrived. If so, TLM serves that request.
- One or more time-outs have expired. If so, TLM executes the corresponding alarms.

A full description of the server-side protocol of TLM is out of the scope of this paper—the interested reader may refer to [2]. In the following we just describe those aspects that are relevant to our following discussion in Sect. 3.

1. /* Declarations */
Declare tom **as** Timeout Manager Handler;
Declare t1, t2, t3 **as** Timeout;
Declare my_alarm, another_alarm **as** Alarm Function;
2. /* Definitions */
Initialize tom **as** Time-outs Management System (my_alarm);
Define t1 **as** Cyclic Timeout (TIMEOUT1, SUBID1) **With Deadline** (DEADLINE1);
Define t2 **as** Cyclic Timeout (TIMEOUT2, SUBID2) **With Deadline** (DEADLINE2);
Define t3 **as** Cyclic Timeout (TIMEOUT3, SUBID3) **With Deadline** (DEADLINE3);
3. /* Activation */
Register t1, t2, t3 **with** tom ;
4. /* Control */
Disable t3;
Redefine t2 **With Deadline** (NEW_DEADLINE2);
Renew t2;
Delete t1;
5. /* Deactivation */
Terminate tom;

Table 1. An example of usage of the time-outs management class. In **1.** a time-out list pointer and three time-out objects are declared, together with an alarm function. In **2.** the time-out list and the time-outs are initialized. Activation is carried out at point **3.** At **4.**, some control operations are performed on the list, namely, time-out t3 is disabled, a new deadline value is specified for time-out t2 which is then renewed to activate the changing, and time-out t1 is deleted. The whole list is finally deactivated in **5.**

2.2 Server-side Protocol

Each time-out t is characterized by its *deadline* t.deadline, a positive integer representing the number of clock units that must separate the time of insertion or renewal from the scheduled time of alarm execution. This field can only be set by functions tom_declare and tom_set_deadline. Each time-out t holds also a field, t.running, initially set to t.deadline.

Each time-out list object, say tom, hosts a variable representing the origin of the time axis. This variable, tom.starting_time, concerns in particular the time-out at the top of the time-out list—the idea is that the top of the list is the only entry whose running field needs to be compared with current time in order to verify the occurrence of the time-out-expired event [5]. For the time-outs behind the top one, that field represents relative values, viz., distances from expiration time of the closest, preceding time-out. In other words, the overall time-out list management aims at isolating a "closest to expiration" time-out, or head time-out, that is the one and only time-out to be tracked for

expiration, and at keeping track of a list of "relative time-outs."

Let us call TimeNow the system function returning the current value of the clock register. In an ordered, coherent time-out list, residual time *for the head time-out* t is given by

$$\mathsf{t.running} - (\mathsf{TimeNow} - \mathsf{tom.starting\_time}), \tag{1}$$

that is, residual time minus time already passed by. Let us call quantity (1) as r_1, or head residual. For time-out n, $n > 1$, that is for the time-out located $n-1$ entries "after" the top block, let us define

$$r_n = r_1 + \sum_{i=2}^{n} \mathsf{t}_i.\mathsf{running} \tag{2}$$

as the n-th residual, or residual time for time-out at entry n. If there are m entries in the time-out list, let us define $r_j = 0$ for any $j > m$.

It is now possible to formally define the key operations on a time-out list: insertion and deletion of an entry.

2.2.1 Insertion

Three cases are possible, namely insertion on top, in the middle, and at the end of the list.

Insertion on top. In this case we need to insert a new time-out object, say t, such that $t.\mathsf{deadline} < r_1$, or whose deadline is less than the head residual. Let us call u the current top of the list. Then the following operations need to be carried out:

$$\begin{cases} t.\mathsf{running} & \leftarrow & t.\mathsf{deadline} + \mathsf{TimeNow} - \mathsf{tom.starting\_time} \\ u.\mathsf{running} & \leftarrow & r_1 - t.\mathsf{deadline}. \end{cases}$$

Note that the first operation is needed in order to verify relation

$$t.\mathsf{running} - (\mathsf{TimeNow} - \mathsf{tom.starting\_time}) = t.\mathsf{deadline},$$

while the second operation aims at turning the absolute value kept in the running field of the "old" head of the list into a value relative to the one stored in the corresponding field of the "new" top of the list.

Insertion in the middle. In this case we need to insert a time-out t such that

$$\exists\, j : r_j \leq t.\mathsf{deadline} < r_{j+1}.$$

Let us call u time-out $j+1$. (Note that both t and u exist by hypothesis). Then the following operations need to be carried out:

$$\begin{cases} t.\mathsf{running} & \leftarrow t.\mathsf{deadline} - r_j \\ u.\mathsf{running} & \leftarrow u.\mathsf{running} - t.\mathsf{running}. \end{cases}$$

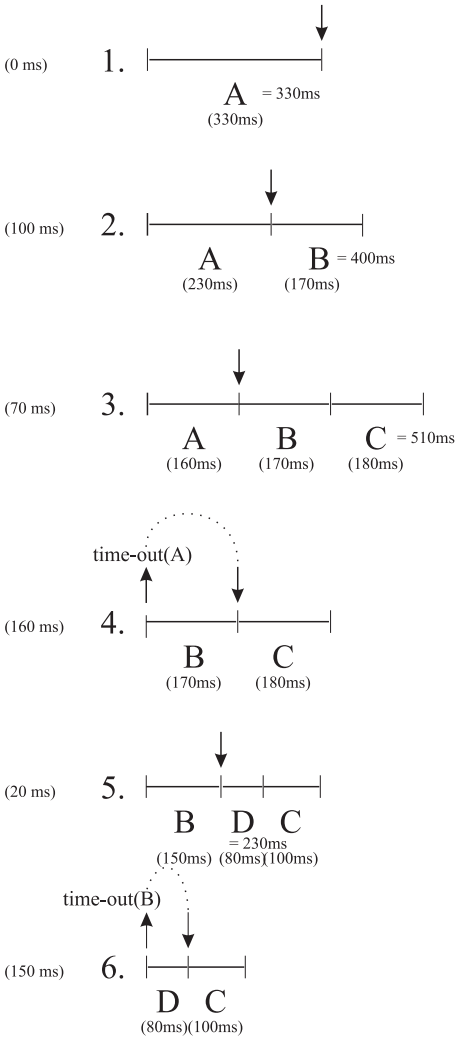

Figure 3. Operating scenario of the time-out manager. In **1.**, a 330ms time-out called **A** is inserted in the list. In **2.**, after 100ms, **A** has been reduced to 230ms and a 400ms time-out, called **B**, is inserted (its value is 170ms, i.e., 400-230ms). Another 70ms have passed in **3.**, so **A** has been reduced to 160ms. At that point, a 510ms time-out, **C** is inserted—it goes at the third position. In **4.**, after 160ms, time-out **A** occurs—**B** becomes then the top of the list; its decrement starts. In **5.** another 20ms have passed and **B** is at 150ms—at that point a 230ms time-out, called **D** is inserted. Its position is in between **B** and **C**, therefore this latter is adjusted. In **6.**, after 150ms, **B** occurs and **D** goes on top.

Observation 1. Note how, both in the case of insertion on top and in that of insertion in the middle of the list, time interval $[0, r_m]$ has not changed its length—only, it has been further subdivided, and is now to be referred to as $[0, r_{m+1}]$.

Insertion at the end. Let us suppose the time-out list consists of $m > 0$ items, and that we need to insert time-out t such that $t.\mathsf{deadline} \geq r_m$. In this case we simply tail the item and initialize it so that

$$t.\mathsf{running} \leftarrow t.\mathsf{deadline} - r_m.$$

Observation 2. Note how insertion at the end of the list is the only way to prolong the range of action from a certain $[0, r_m]$ to a larger $[0, r_{m+1}]$.

2.2.2 Deletion

The other basic management operation on the time-out list is deletion. As we had three possible insertions, likewise we distinguish here deletion from top, from the middle, and from the end of the list.

Deletion from top. If the list is a singleton we are in a trivial case. Let us suppose there are at least two items in the list. Let us call t the top of the list and u the next element, the one that will be promoted to top of the list. From its definition we know that

$$\begin{aligned} r_2 &= u.\mathsf{running} + r_1 \\ &= u.\mathsf{running} + t.\mathsf{running} - \\ &\quad (\mathsf{TimeNow} - \mathsf{tom.starting\_time}). \end{aligned} \tag{3}$$

By (1), the bracketed quantity is elapsed time. Then the amount of absolute time units that separate current time from the expiration time is given by $u.\mathsf{running} + t.\mathsf{running}$. In order to "behead" the list we therefore need to update t as follows:

$$u.\mathsf{running} \leftarrow u.\mathsf{running} + t.\mathsf{running}.$$

Deletion from the middle. Let us say we have two consecutive time-outs in our list, t followed by u, such that t is not the top of the list. With a reasoning similar to the one just followed we get to the same conclusion—before physically purging t off the list we need to perform the following step:

$$u.\mathsf{running} \leftarrow u.\mathsf{running} + t.\mathsf{running}.$$

Deletion from the end. Deletion from the end means deleting an entry which is not referenced by any further item in the list. Physical deletion can be performed with no need for any updating. Only, the interval of action is shortened.

Observation 3. Variable tom.starting_time is never touched when deleting from or inserting entries into a time-out list, except when inserting the first element: in such case, that variable is set to the current value of TimeNow.

Figure 3 shows the action of the server-side protocol.

3. Dynamic Behavior

We now discuss the behavior of our time-outs management system when it executes several long lasting alarms. The main consequence of this is that the alarms compete with each others and with the TML for the CPU or other scarce resources.

In this case there is a non negligible delay between the time in which r_1 becomes zero and the time that event is managed. Such delay can be expressed as

$$(\mathsf{TimeNow} - \mathsf{tom.starting\_time}) - t.\mathsf{running}.$$

The system deals with these events by propagating the delays to those entries that follow the top of the list. This is done by determining the integer $j \geq 1$ such that

$$r_j < 0 \quad \wedge \quad r_{j+1} \geq 0, \tag{4}$$

where this time r_1 can also be negative. The time-out management process needs therefore first to check whether r_1 is less than zero; if so, it must calculate index j such that (4) is verified; and finally command the execution of all the corresponding alarm functions.

Finally, if the list is not empty, that process must adjust the corresponding running field as follows: let t be time-out $j+1$; then

$$t.\mathsf{running} \leftarrow t.\mathsf{running} + r_j.$$

Clearly the above mechanism only works fine if there is a way to keep under control congestion that is due to alarm execution. Next subsection describes a mechanism aiming at that.

3.1 Dealing with alarm execution congestion

Alarm execution is managed via the mechanism shown in Fig. 1, items (3) and (4), i.e., through an Alarm Scheduler (AS) which gathers all alarm execution requests and forwards them to the next entry

of a circular list of processes. As we mentioned already, alarms often imply communication, hence using a list of concurrent processes might in principle result in better performance—should the underlying system offer means for managing I/O in parallel. In general, if the alarm functions do not compete "too much" for the same resources at the same times, this scheme allows a better exploitation of the available resources as well as a higher probability to control alarm congestion. Herein we evaluate the ability of this mechanism to control alarm congestion under different levels of congestion and in two opposite cases of alarm interference, i.e., no competition and full competition.

Let us call T the deadline of our time-outs, and ν the extra time due to alarm congestion, i.e. the real-time violation. Furthermore, let us call δ the *actual time to the alarm*, that is $T + \nu$. In a perfect system δ would be equal to T while in a real life it depends on the time-outs manager's period and on alarm execution congestion.

In order to estimate the average run-time violation experienced by the time-outs manager we generated 1000 non-cyclic time-outs with a deadline $T > 0$ uniformly distributed.

In order to measure the ability of the time-out manager to control alarm congestion, we run our experiment configuring the manager with no processes in the circular list of alarm processes (alarm execution managed by TLM) and then adding more and more processes in that list. Let us call τ the number of processes in the list.

Furthermore, we artificially imposed the following minimal durations to the alarm functions: minimal[1], 10ms, 100ms, and so on. These durations have been imposed either by loading the alarm functions with some purely computation-oriented tasks, so to let them compete for the same resource, being the one CPU in the system, or by executing a function (TimeWait) that puts the calling process in the wait state for the specified amount of time—this way alarm functions do not compete at all with each other. Let us use α to refer to the alarm durations.

The time-outs manager's period, i.e., TM_CYCLE was set to 50000 clock ticks, 1 clock tick being 1μs. We run the experiment on a single processing node with a PowerPC 604 at 133MHz. Let us call γ the number of time-outs that experience a run-time violation greater than TM_CYCLE, $0 \leq \gamma \leq 1000$.

Table 2 groups the parameters and variables of the experimentations and recalls their meanings.

Figure 4 summarizes the results of a first experiment in which:

- α is kept to its minimum value (approximately 50 clock ticks)

[1] Just the overhead corresponding to calling a function and copying a 20-byte message—this is the typical delay in our target applications, lasting approximately 50 ticks on our processing node.

T	A time-out's deadline, i.e., the amount of time that should elapse between the moment the time-out is inserted and the moment its alarm is invoked.
α	Minimal duration of the alarm functions.
ν	Run-time violation, i.e., actual duration of the alarm functions.
δ	$T + \nu$.
TM_CYCLE	The time-out manager's period. Time-outs are checked every TM_CYCLE clock ticks.
τ	Number of alarm processes in the system.

Table 2. Parameters and variables of the experimentations.

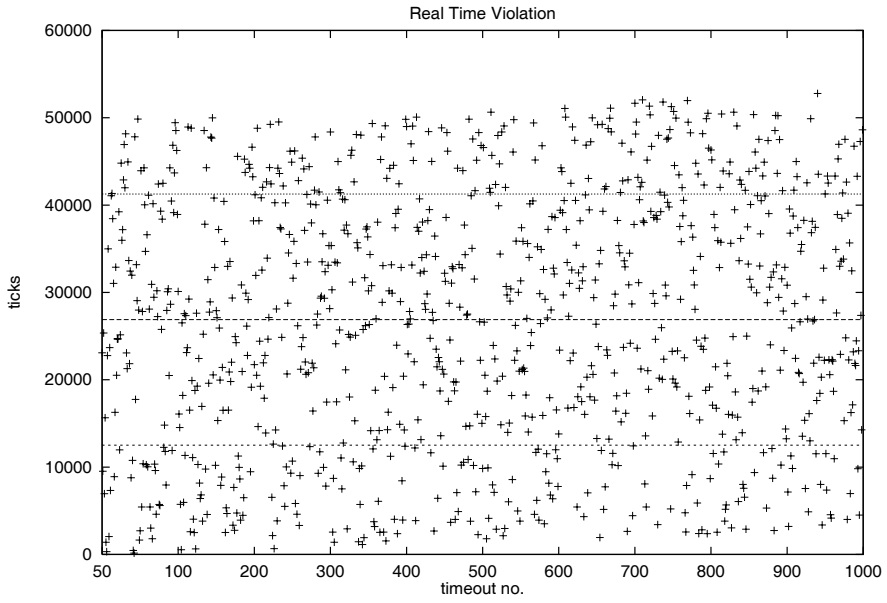

Figure 4. The graph shows the run-time violation when 1000 time-outs are uniformly generated in $(0, 100\text{s}]$, $\alpha \approx 50$ ticks, and $\tau = 0$ (that is, TLM executes alarms autonomously). Maximum observed value is 52787, minimum is 157. The average is 26892.74 with a standard deviation equal to 14376.89. γ is 20, that is, of the 1000 time-outs, 20 exceeded TM_CYCLE (in this case, 50000 ticks or 50ms). Other statistics regarding this latter sample can be found in Table 3.

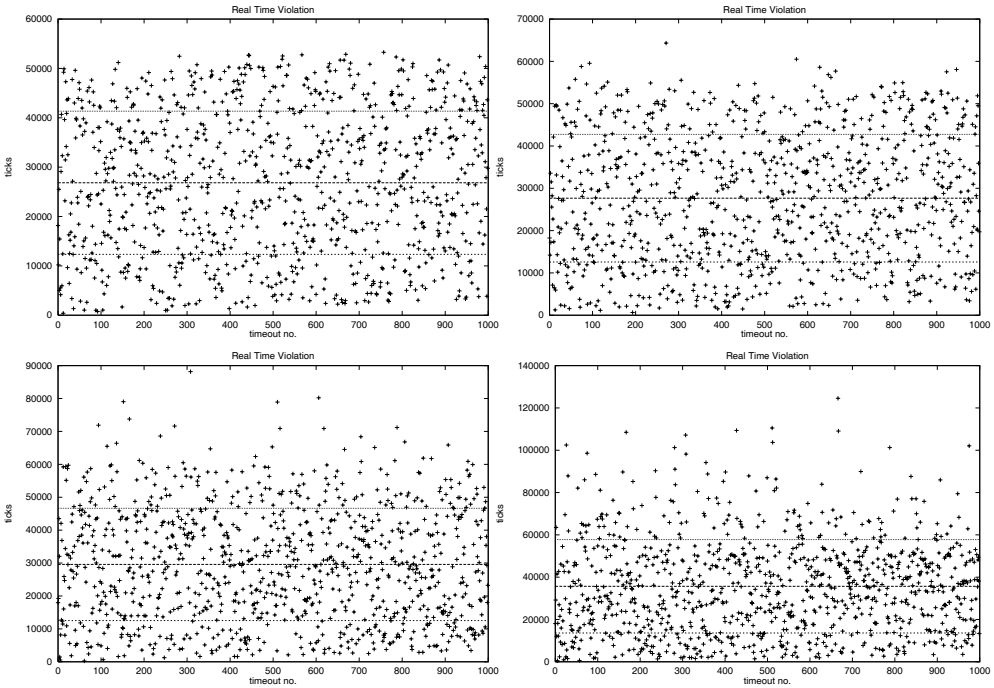

Figure 5. Real-time violation when α is equal to 1ms (upper left picture), 5ms (upper right picture), 10ms (lower, left picture), and 20ms (lower, right picture). 1000 pseudo-random time-outs with deadlines uniformly distributed in $(0, 100\text{s}]$, $\tau = 0$. In each picture, the central straight line represents the average ($y = a$,) while the others are $y = a \pm d$, d being the standard deviation.

- 1000 time-outs are executed with T distributed pseudo-randomly in $(0, 100\text{s}]$
- TLM directly executes alarm functions ($\tau = 0$).

3.1.1 Best case scenario

Figure 5 describes what happens increasing α while keeping τ equal to zero in the best case scenario—no competion among the alarm functions. The results have been also summarized in Table 3, that also reports the hugest violation (in the case, with α =20ms, the largest ν was equal to 124535 clock ticks, or 2.49 times TM_CYCLE.)

In particular, when α became larger than TM_CYCLE (the checking quantum of the time-out list manager), we measured ever increasing values for ν (real-time violation) due to alarm execution congestion (see Fig. 6).

We experimentally found that the presence of the alarm scheduler

α	population	average	stdev	max
≈ 50 ticks	20	50967.10	762.07	52782
10^6 ticks	34	51282.32	975.85	53263
5×10^6 ticks	77	53077.68	2774.48	64328
10×10^6 ticks	132	57605.80	6917.56	88183
20×10^6 ticks	238	65639.74	14571.78	124535

Table 3. This table reports about some experiments aiming at measuring alarm execution congestion. 50 ticks is approximately the average local clock time required by the alarm function used in our target applications: a context switch time plus the time to store a 20-byte message into the input mailbox of the receiver process (measurements done with the TEX operating system [3]).

τ	μ	σ	γ	$\gamma_{\max}$	μ'	σ'
0	35962.849	22683.486	264	130993	65483.295	14160.466
1	30200.690	18447.083	108	108991	64676.176	15647.9
2	27556.858	14471.615	46	63659	52228.13	2926.681
3	27286.773	14369.997	45	53934	51280.733	1014.648
4	27493.727	14058.125	47	53372	51526.17	921.993
5	27422.843	14079.798	46	54077	51357.196	1079.067

Table 4. The experiment is repeated with $\alpha = 20$ms and $0 \leq \tau \leq 5$. μ and σ are respectively the average and standard deviation of the 1000 outcomes. μ' and σ' are respectively the average and standard deviation of γ. $\gamma_{\max}$ is the largest of the γ values.

and of the circular list of processes has positive relapses on alarm congestion control. As an example, Table 4 summarizes the results of increasing τ from 0 to 5 when α is set to 20ms. In particular, when τ is equal to 2, γ drops from 264 to 46 items, while with $\tau = 3$ the worst case for γ was equal to 53934 clock units, or just 1.07868 times TM_CYCLE—a violation of less than 8%.

3.1.2 Worst case scenario

The worst case takes place when all alarm processes compete for the same set of resources at the same time. An easy way to accomplish this is to let each alarm function perform, e.g., a pure integer computation task so that each process compete with all the others for the single integer pipeline of the CPU of our system. Given a fixed $\alpha = 20$ms, we increased τ and measured real-time violations. The results have been summarized in Table 6. Adding processes did not produce any useful result in this case.

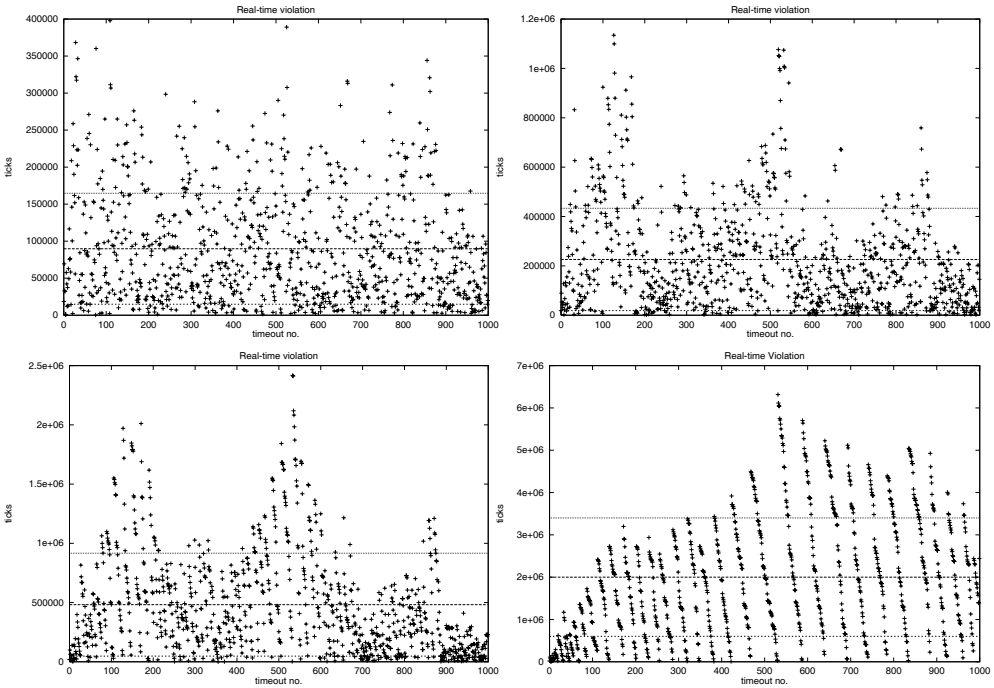

Figure 6. The experiment is repeated with α respectively equal to 60ms, 80ms, 90ms, and 100ms. Run-time violation grows without bounds. Note how the picture corresponding to the worst case exhibits some degree of self-similarity.

4. Conclusions

A simple software system for managing lists of time-outs has been introduced. Experimental results show that, in some cases, the congestion due to the concurrent execution of the alarms and of the time-out management tasks may produce severe violations of the expected beaviour. We showed how in some special cases this congestion can be controlled if not eliminated. In the general case this congestion brings the system to a chaotic state. This transition can be represented in some cases as images depicting some degree of self-similarity. Future work will include an analysis of those images' complex features and new experiments with non-uniformly distributed time-outs.

Acknowledgments

The authors wish to thank the Editor and the reviewers for their many and insightful comments and suggestions.

τ	μ	σ	γ	$\gamma_{\max}$	μ'	σ'
1	3692158.863	2966039.234	974	12277216	3790057.80	2943375.88
2	33674.461	27688.173	140	264882	83244.94	38052.06
3	28177.013	16801.971	54	146684	66737.87	20705.76
4	27621.631	14276.781	45	100260	52871.73	7358.90
5	27435.401	14087.224	44	53712	51375.93	1001.36
6	27475.845	14107.325	44	53992	51443.95	1008.77

Table 5. The experiment is repeated with $\alpha = 100$ms and $0 \leq \tau \leq 6$. $\mu, \sigma, \gamma, \mu'$, and σ' are as defined in Table 4. Note how a relatively low number of alarm workers is able to reasonably control alarm execution congestion even in the scenario of Fig. 6.

τ	μ	σ	γ	$\gamma_{\max}$	μ'	σ'
0	35251.93	21667.08	226	126931	66223.70	13713.40
1	35533.45	21584.35	248	127403	64377.88	13803.45
2	35735.19	21606.70	250	125910	64681.21	13305.83

Table 6. The experiment is repeated with $\alpha = 20$ms and $0 \leq \tau \leq 2$. This time alarm workers do compete for a unique system resource. As evident, adding processes does not produce any useful result in this case.

References

[1] O. Botti, V. De Florio, G. Deconinck, S. Donatelli, A. Bobbio, A. Klein, H. Kufner, R. Lauwereins, E. Thurner, and E. Verhulst. TIRAN: Flexible and portable fault tolerance solutions for cost effective dependable applications. In P. Amestoy, editor, *Proc. of the 5th Euro-Par Conference, Lecture Notes in Computer Science*, volume 1685, pages 1166–1170, Toulouse, France, August/September 1999. Springer-Verlag, Berlin.

[2] V. De Florio. *A Fault-Tolerance Linguistic Structure for Distributed Applications.* PhD thesis, Dept. of Electrical Engineering, Katholieke Universiteit Leuven, October 2000. ISBN 90-5682-266-7.

[3] DEC. *CS_Q66E Alpha Q-Bus CPU module: User's Manual.* Digital Equipment Corp., 1997.

[4] G. Deconinck, V. De Florio, and R. Belmans. *Architecting Dependable Systems II*, volume 3069 of *Lecture Notes in Computer Science (State-of-the-Art Survey)*, chapter Architecting Distributed Control Applications, based on (Re-)Configurable Middleware, pages 123–143. Springer-Verlag, Berlin, Germany, 2004.

[5] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall, London, 3rd edition, 1996.

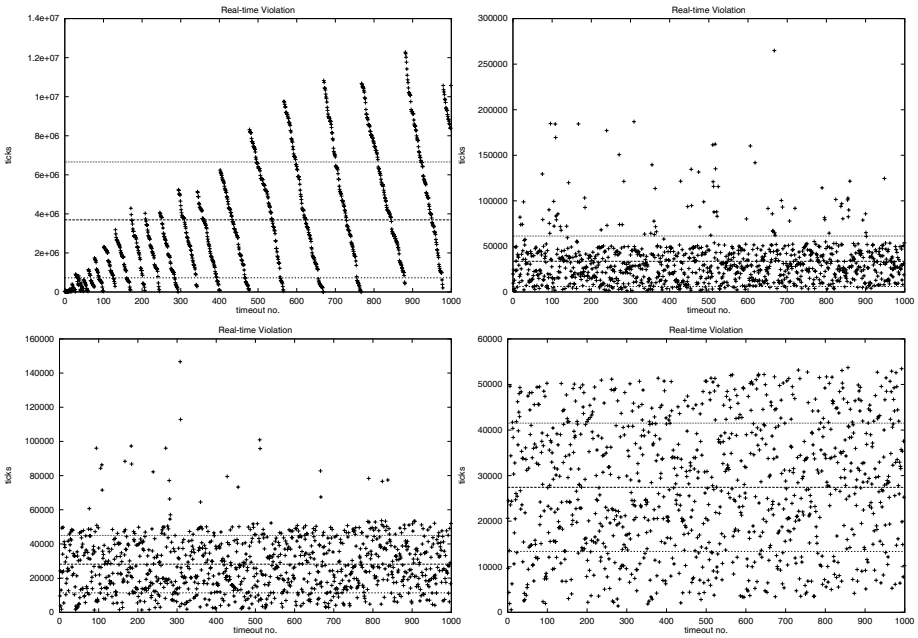

Figure 7. The pictures show how increasig τ, i.e., adding processes to the circular list of alarm processes, it is possible to control alarm execution congestion even in the scenario of the last picture of Fig. 6 ($\alpha = 100$ ticks). From left to right and up to down, picture represents the cases of τ equal to 1, 2, 3, and 5.

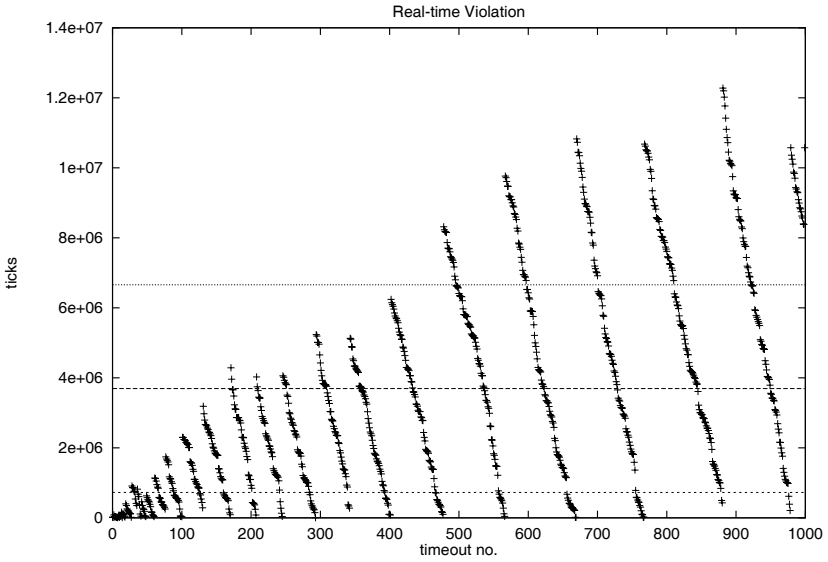

Figure 8. The top picture in Figure 7 is here reproduced at larger size to stress its self similarity.