Universidade Federal do Paraná

Departamento de Informática

Jorge Tortato Júnior
Aldri L. dos Santos
Roberto A. Hexsel

PSAD - A Protocol for Synchronization of Distributed Actions

Relatório Técnico
RT-DINF 001/2008

Curitiba, PR
2008

Resumo

Real-time systems are present in people's life and are used in several applications like industrial automation and robot motion control. These applications in distributed systems are challenging due to non-deterministic behavior of system elements and communication channels. This paper proposes a protocol called PSAD to deal with action synchronization through system elements. This protocol is based on services for clock synchronization, reliable group communication and real-time task scheduling. An experimental evaluation using local networks shows that it can be used for synchronized applications that need rates around 10 requests per second and jitter around few tens of milliseconds.

1 Introduction

Real time systems are present on people's life due to extensive usage of embedded microprocessors and microcontrollers. These devices can be found from house appliances to complex systems composed by several distributed controllers used on applications like robotics, remote control, remote surgery, industrial automation and so on. Non-distributed real time systems are well understood. However, real time distributed systems are still seem as a challenge because the performance of system elements as well as of communication network cannot be predicted in most of the applications [1]. This lack of predictability can make the implementation of these systems unfeasible due to resulting unreliability.

Real time requirements lead to discussion of synchronous systems, on which the time perception is a fundamental concept. Studies started on 80's expanded this concept with implementation of sophisticated synchronization protocols. These protocols allow clock synchronization in range of milliseconds [2], depending on distance among elements that are being synchronized and the reference clock. The development of global positioning system, GPS, on early 90's enabled new opportunities for research on synchronous systems. The synchronization restricted to elements close to each other could be expanded to more distant areas.

Beyond the facts above, due to expansion of embedded computing on the latest years, research for common development platforms has been intensified, specially for real time platforms. Efforts on this approach include the development of Real Time Java platform (RTSJ) [3] and the specification of Real Time Java Platform for Distributed Systems (DRTSJ) [4] [5]. RSTJ provides solutions for real time issues on a single machine, like task scheduling and priority control, meanwhile DRTSJ promises answers for distributed systems like remote invocation (RPC) with timing constraints and management of distributed memory.

One of the basic issues on real time distributed systems is action synchronization, it means the ability to execute tasks coordinated in time on different processes or machines. Some approaches seek for solutions independent of physical and logical communication network layers, meanwhile others change these layers to achieve timing guaranties. CASD [6] [7], for example, is a protocol that does not depend on physical or logical layers. However, it shows inconsistence and wrong behavior when used together with aggressive time constraints [8]. Other studies tried to reach timing constraints by modifying communication layers [9] [10], through statistical methods [11] [12] or by using resource reservation protocols [10] [13]. However, these studies require non-standard solutions, resulting on restrictions on their applicability, or do not provide guaranties for time deadlines on all nodes.

This paper proposes a protocol for real time distributed systems, called PSAD, that allows reliable synchronization of tasks on different places. In this way, tasks

on distant places can be synchronously executed on request of one of the elements of the system. PSAD is based on mechanisms of clock synchronization, on group communication and on real time task scheduling that allow the execution of a task on a predictable time.

The performance and efficiency of PSAD are evaluated through tests on a single machine executing several independent processes and on processes being executed on several machines connected to each other through a local network. Results show that on these scenarios it is possible to reach rates of 10 requests per second, small number of cancelled tasks and jitter of a few tens of milliseconds, which confirms the reliability of PSAD for real time applications.

The paper is structured as follows. Section 2 describes related work. Section 3 discusses basic services used to build PSAD. Section 4 details PSAD and its behavior. Section 5 describes the experiments and the results. Section 6 presents the conclusions and future directions.

2 Related work

One of the first works on task synchronization was the CASD [6] [7]. However, it presents wrong behavior when used with more agressive timing constraints [8]. Some works based on physical and logical network layers [9] [10] require non-standard solutions, thus resulting on restrictions on their applicability. Statistical methods [11] [12] or resource reservation protocols [10] [13] do not provide guaranties that time deadlines are achieved on all nodes.

Some work is found on media synchronization due to increasing demand on multimedia network services. In [14], a survey shows needs for synchronization protocols on media transmission. In [15], it is shown that synchronization mechanisms can be applied to reconstruct presentation schedule. In [16], a performance evaluation for multimedia presentation protocol highlights the importance of timing guaranties on network.

Additionally, some work is also found on robotics. In [17], it shows the need for synchronization of robots modules and describes related work on synchronization related to collaborative work.

3 Basic services

This section discusses the basic services that support the proposed protocol. These services are clock synchronization services, group communication services and real time task scheduling.

3.1 Clock synchronization

Several techniques for clock synchronization on distributed system have been researched during 80's [8]. The main issues are drift of clocks due to imprecision of each individual clock and latency of communication networks that becomes synchronization among nodes inaccurate.

Srikanth and Toueg [18] proposed a method based on the fact that the variation rate of clocks is likely to be constant. From this premise, the method tries to compensate the differences among them a-priori. Verissimo and Rodrigues [19] and Clegg and Marzullo [20] proposed another method a-posteriori for clock synchronization. This one performs periodically the delay calculation among clocks of the system via messages among the different processes and makes the required adjustments according to received values, assuming that average delays on network is constant. Today, the most used protocols are capable of reach synchronism around few tens of milliseconds or better [2] and are mainly based on Marzullo's method.

Another method used from 90's is the GPS system. Using GPS is possible to achieve higher precision than ones reached with synchronization protocols: around few milliseconds or better, except when atmospheric condition causes interference on signal reception or disturbs it significantly [21]. Thus, with GPS, it is possible to have good precision with relative low cost, although they are higher than costs of methods based on message passing.

Synchronization protocols as well as GPS allow that several machines into a system become synchronized and, so, that timestamps could be used for controlling on real time distributed systems, scheduling tasks and synchronizing them.

3.2 Group communication services

Group communication is extremely useful on distributed systems. It allows several processes to join or leave a group dynamically, to be monitored by failure detectors, to share states and group visions and to exchange unicast and multicast messages. Multicast communication protocols are classified as basic, reliable, casual, FIFO or totally ordered [8].

When applied on task synchronization protocols, the most relevant are ones classified as reliable and FIFO. Reliable protocols ensure that all processes belonging to a group receive messages even in presence of failures into communication. FIFO protocols ensure that messages sent by a process to be ordered received, i.e., if a process *p1* sends a message *m1* to the group and then a second message *m2*, all group members will receive *m1* and then *m2*, independently of communication network delays. Communication using a reliable and FIFO protocol is the most appropriated because ensures that all messages sent by a sender process to be ordered delivered in time to the group.

3.3 Real time task scheduling

Scheduling of tasks or actions at defined and predictable time is an important matter when talking about real time distributed systems. Two issues related to this are the delays on communication network and delays on scheduling by operating system. On the first case, the network characteristics can be controlled through quality of service guaranties (QoS) and/or traffic priority. However, there is not so much to do when complex networks are used, where quality guaranties cannot be easily achieved.

On the other hand, operating system scheduling, that is strongly dependent of processor load, can use process priorities and other real time features of the own operating system. The major part of this features is dependent on a specific architecture [22], x86 or PowerPC, for example, limiting applicability on heterogeneous systems. These heterogeneous systems are very common on distributed systems, where each node can have a different operating system, processor or memory capacity. So, using services that are independent of a specific platform, like Java, seems to be the most feasible way to implement scheduling of tasks on real time distributed systems context.

4 PSAD - A protocol for synchronization of distributed actions

This section describes the PSAD protocol and its behavior. PSAD is a protocol designed to solve problems on synchronizing tasks on real time distributed systems. It ensures that actions are started on all correct processes of the system or on none of them. The protocol assumes that is possible to get a minimum level of synchronization among the several processes on the distributed system. Below, the system model and the proposed protocol are described.

4.1 System model

A real time distributed system is defined as a set S composed by n processes $\{p_1, p_2, ... , p_n\}$ synchronized among them. Each process $p_i \in S$ has its own clock $c_i \in C$, where $C = \{c_1, c_2, ... , c_n\}$ is the set of all clocks in the system. The synchronization uncertainty, ϵ, is defined as the higher difference among all clocks $c_i \in C$. Additionally, processes communicate through a network with the following timing properties (PT):

PT1 .Δ_{max} : maximum delay of messages on the network.

PT2. Δ_{med} : average delay of messages on the network.

PT3. Δ_{opmax}: maximum time taken by an operation to complete through the network.

, whose message passing occurs through two communication primitives (PC):

PC1. *R_FO_UNICAST* : unicast message, reliable and FIFO ordered.

PC2. *R_FO_MULTICAST*: multicast message, reliable and FIFO ordered.

, where messages are sent through PC1 and PC2 and are also FIFO ordered between them.

Two types of failures are assumed: crash and network timing. On crash failures, a process does not respond to messages, meanwhile on network timing failures the messages have delays higher than expected.

4.2 Behavior

PSAD is based on wall clocks, it means, one or more processes own clocks that are used to synchronize the other process clocks. After all process are synchronized and organized into a closed group, the process that initiates an action (coordinator) and remote processes (clients) use communication primitives PC1 and PC2 to synchronize their actions. There are no restrictions to the roles of each process and each action can be coordinated by a different process.

Figure 1 shows typical protocol runs. On the first message exchange, the coordinator initiates a synchronized action request sending a message for the clients via PC2. This message has sending timestamp, sequence identifier of requests sent by this coordinator, action type requested and the desired time for execution (called target). Clients receive them, schedule the requested action and sent confirmation through PC1, which must be received by coordinator within a maximum time period T $< target$ - k*Δ_{med}, where k is adjusted according to the network.

Each client knows how much time is needed to execute a specific action. So, a client can reject requests if there are conflicting pending actions or whose target time is too much close current time (less than k*Δ_{med}, for example), which makes impossible to the coordinator to receive a confirmation before deciding about cancellation.

On the second message passing, coordinator initiates a request as before. In this case, however, a confirmation message is not received in time and it initiates an action cancellation sending via PC2 a cancellation message. This message contains an action identifier and the target time. Clients are responsible for canceling the action scheduled before.

The protocol is described in three parts, on the following algorithms. These algorithms describe the initialization and vision update of the groups, transmission and reception of messages and the procedure to schedule real time actions.

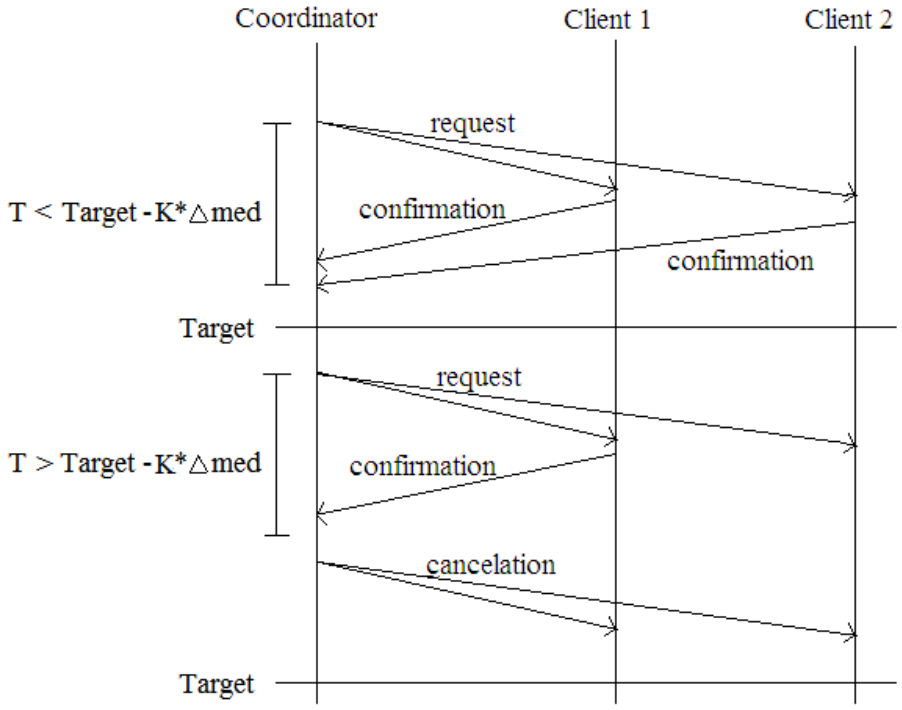

Figura 1: Typical runs of PSAD protocol.

4.2.1 Initialization and group vision update

The initialization and update algorithm is responsible for initializing control variables, for connecting a process to the communication group and keeping an updated vision of processes belonging to the group. On $Initialization()$ procedure of algorithm 1, at lines 2 to 4 (for simplicity lines will be referred as $l.$ through text), the variables *action_number*, *ack_number* and *member_num* are initialized and they refer to the sequential action counter, received number of confirmations and the number of group members, respectively. Then, at ($l.$5-6) the initial timing parameters Δ_{med} and Δ_{opmax} are configured and the join to the communication group is requested ($l.$7). On *Vision_Reception()* procedure, at each modification of group vision, the number of members is updated ($l.$10).

4.2.2 Transmission and reception

Transmission and reception algorithm handles all message exchanges with other group members. On procedure *Transmission()* of algorithm 2, at each new request the target time is calculated ($l.$2), the sequence number is incremented

Algoritmo 1 Initialization and Update of group vision

```
 1: procedure Initialization( )
 2:     action_number ← 0;                          // control variables initialization
 3:     ack_number ← 0;                             // initializes received response number
 4:     member_num ← 0;                             // initializes number of active members
 5:     Δ_med ← default Δ_med;                      // initializes average network delay
 6:     Δ_opmax ← default Δ_opmax;                  // initializes operation latency
 7:     join_group(PSAD);                           // joins to communication group
 8: end procedure

 9: procedure Vision_Reception( )
10:     member_num ← group_size(PSAD);              // upd. members num.
11: end procedure
```

(l.3) and the number of received confirmations is re-initialized (l.4). One message containing a timestamp, packet type, sequence number, target time and the higher level application message is concatenated and sent through multicast (PC2) (l.8).

After sending a message, the transmitter waits until period for confirmations arrival expires (l.11) or that all confirmation messages have been received (l.12). Period for waiting is function of target time of the request and of delays experimented on network, being factor k a configurable safety parameter. If confirmations are not received, a cancellation message specifying the action to be cancelled is sent via multicast for all processes (l.17-20). Then, parameter Δ_{opmax} is updated according to the time spent to exchange messages (l.21) and the procedure is finished with failure (l.22). On the other hand, if confirmations arrive in time, the parameter Δ_{opmax} is also updated and the procedure is finished with success (l.14-15).

On procedure *Reception()* (l.25), the average message delay through network, Δ_{med}, is updated using timestamp of received message. Then the message is processed according to its type: new action request, cancellation or confirmation. When it is a new request (l.27), feasibility of action scheduling is checked with real time scheduler (l.28). If it is possible (l.30), a positive confirmation via *unicast* is sent to the coordinator (l.31-34). Otherwise, it sends a negative confirmation, or cancellation, to the group (l.36-39). If received message is a cancellation of one action (l.42), the respective action is immediately cancelled via a request to real time scheduler. When the message is a positive confirmation (l.45), the number of received confirmations is incremented if it refers to the current action being coordinated (l.49).

4.2.3 Real time task scheduling

Real time task scheduler algorithm is responsible for receiving requests or cancellations and for verifying if schedule of a task is feasible or not. On procedure *Reception()* of algorithm 3, when a new action should be scheduled (l.2) and it

Algoritmo 2 Transmission and Reception

```
1: procedure Transmission(app_msg)
2:     target_time ← now + Δ_opmax;                          // calc. of target time
3:     action_number ← action_number + 1;                    // inc. of seq. counter
4:     ack_number ← 0;                                       // initializes rec. confirmation number
5:     pkt_type ← action_type;                               // new message is an action
6:     msg ← timestamp & ptk_type &
7:     action_number & target_time & app_msg;
8:     R_FO_MULTICAST(msg);                                  // reliable multicast of msg
9:     start_time ← now;                                     // initializes timer
10:    wait until
11:    (now < target_time − k ∗ Δ_med) or                    // waits until timeout
12:    (ack_number = member_num)                             // or until rec. all confirmations
13:    if (ack_number = member_num) then                     // if rec. confirmations
14:        Δ_opmax ← update_Δ_opmax(start_time);             // upd. latency
15:        return sucess;                                    // returns success
16:    else                                                  // if it was a timeout
17:        pkt_type ← cancel_type;                           // assembles a cancellation msg
18:        msg ← timestamp & ptk_type &
19:        action_number & target_time;
20:        R_FO_MULTICAST(msg);                              // performs reliable multicast
21:        Δ_opmax ← update_Δ_opmax(start_time);             // upd. latency
22:        return failure;                                   // returns failure
23:    end if
24: end procedure

25: procedure Reception(msg_rcv)
26:    Δ_med ← update_Δ_med(msg_rcv.timestamp);              // upd delay
27:    if (msg_rcv.pkt_type = action_type) then              // if a new action
28:        send_msg_RT(msg_rcv);                             // scheduling msg to RT algorithm
29:        wait until rcv_msg_RT(msg_RT);                    // waits RT algorithm
30:        if (msg_RT = success) then                        // if it is a success
31:            pkt_type ← ack_type;                          // assembles confirmation msg
32:            msg ← timestamp & ptk_type &
33:            action_number & target_time & msg_rcv;
34:            R_FO_UNICAST(msg);                            // sends to the coordinator
35:        else                                              // if refused
36:            pkt_type ← nack_type;                         // assembles rejection msg
37:            msg ← timestamp & ptk_type &
38:            msg_rcv.action_number & msg_rcv.target_time;
39:            R_FO_MULTICAST(msg);                          // sends to the group
40:        end if
41:    else
42:        if (msg_rcv.pkt_type = cancel_type or nack_type) then   // if it's a cancelation
43:            send_msg_RT(msg_rcv);                         // requests a cancellation
44:        else
45:            if (msg_rcv.pkt_type = ack_type) then         // if a conf.
46:                if (msg_rcv.action = action_number) and
47:                    (msg_rcv.target_time = target_time) then
48:                                                          // increments confirmation number
49:                    ack_number ← ack_number + 1;
50:                end if
51:            end if
52:        end if
53:    end if
54: end procedure
```

is feasible to be ($l.3$), the action is scheduled and confirmed to communication algorithm 2. Otherwise ($l.7$), it is discarded and not confirmed. If a cancellation is requested and the respective action was not executed yet, it is cancelled ($l.10$-12). Otherwise, a severe failure is forwarded to application level ($l.14$), which is responsible for system recovering.

Algoritmo 3 Real time task scheduling

```
 1: procedure Reception(msg_RT)
 2:     if (msg_RT.pkt_type = action_type) then                          // if new action
 3:         if (isfeasible(msg_RT.target_time)) then                     // feasible ?
 4:             sched_action(msg_RT.action, target_time);
 5:             send_com(sucess);                          // sends success to rec/trans. alg.
 6:         else
 7:             send_com(failure);                         // sends failure to rec/trans. alg.
 8:         end if
 9:     else
10:         if (msg_RT.pkt_type = cancel_type) then                      // canc. msg.?
11:             if (notfired(msg_RT.action)) then                        // if not yet fired
12:                 dis_sched_action(msg_RT.action);                     // cancels it
13:             else                                                     // if already fired
14:                 recover_action(msg_RT.action);                       // recovers
15:             end if
16:         end if
17:     end if
18: end procedure
```

5 Experiments and results

Experiments were carried out using three different computers connected through ethernet switches to a local 100Mbps ethernet network. The configurations of computers are: (A) Intel dual core 1.66GHz, 2GB DDR2 PC4200 RAM memory, FC6 (Fedora Core 6) operating system executed on a virtual machine running on Windows XP; (B) Atlhon 1.2GHz, 512MB DDR 266 RAM memory, FC6 operating system and (C) Pentium 4 3.0GHz, 472MB of RAM memory, FC6 operating system executed on a virtual machine running on Ubuntu 6.06.a.

The fact that two machines are being executed FC6 on virtual machines and that they are all of different configurations of memory and processors builds an heterogeneous environment when talking about processing capability and delays of messages exchanged by processes. So, this environment allows the protocol to be better evaluated because the behavior is closer to a real network than an isolated and controlled network.

Clock synchronization was implemented using NTP (Network Time Protocol) [23], version 3, although version 4 has been developed and are being standardized by IETF. NTP uses Marzullo algorithm to keep synchronization within a few tens of milliseconds or better [2]. NTP was chosen instead of GPS because tests were

done on a local network, where performance of NTP is enough. Group communication services were implemented from JGroups 2.4.1 SP3 [24] libraries. JGroups were used on services to create multicast groups and to implement reliable unicast and multicast communication, both FIFO ordered.

PSAD was implemented on Java version JDK1.4 and Java-RT (Java Real Time) [3] version RI 1.1 Alpha 2 from TimeSys. All algorithms dependent on JGroups libraries were coded and executed on JDK1.4 platform. Algorithms dependent on real time functions (real time task scheduling) were coded and executed on Java-RT platform. Communication between JDK1.4 and Java-RT parts were implemented with a socket connection local to the machine. This approach was required due to restrictions on Timesys Java virtual machine (tjvm). Although this implementation is specific for Fedora Core 6, other commercial versions are available for other operating systems. So, the option for Java was done due to platform independence. Figure 2 shows code architecture.

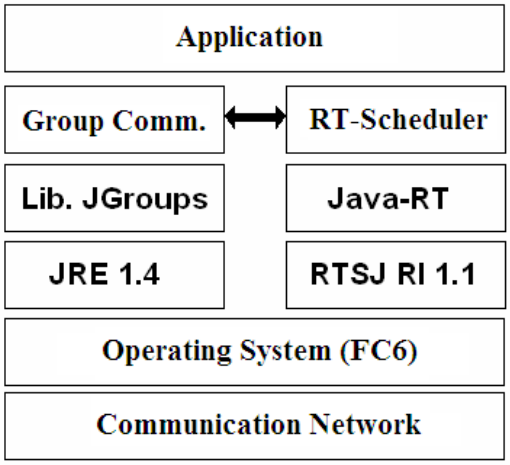

Figura 2: Implementation of PSAD.

Two types of experiments have been done. On first one, a single FC6 virtual machine on computer (A) executed 1, 2 or 3 processes, where only one of them was generating requests to the group. Once all processes were executed on a single virtual machine, a perfectly synchronized environment is simulated. On second experiment, one (B), two (B+C) or three (A+B+C) different machines were used, with one process running on each, where only processes on computer (B) was generating requests to the group.

Metrics were collected for all experiments to evaluate scalability, performance, efficiency, tolerance to delays and jitter. For scalability, were collected the rate of scheduled actions per second, for 20.000 attempts, as a function of number of processes or computers. Performance were evaluated through rate of scheduled

actions as function of time and efficiency was demonstrated via number of cancellations, both calculated over 2.000 attempts, from a total of 20.000 attempts. Tolerance to delays were evaluated changing Δ_{opmax} parameter and verifying the rate of cancelled actions, using samples with 8.000 requests for each value of Δ_{opmax}. Finally, the amount of jitter were measured, always compared to local clock, showing the difference between target time and real time of execution of a given action. Data for all metrics were collected through source code instrumentation.

5.1 Experiment 1 - perfectly synchronized system

As described before, this experiment used machine (A) to simulate a perfectly synchronized environment. Except to delay tolerance tests, the value of Δ_{opmax} were kept constant and equal to 2s. Figure 3 shows results obtained for (a) scalability and (b) performance. On (a) the average request rate decreases with number of processes. This happens because the processing capacity is limited and, thus, more concurrent processes reduces maximum request rate. (b) shows variation of request rate, which is almost constant as function of time, between 8 and 12 requests per second, depending on number of processes.

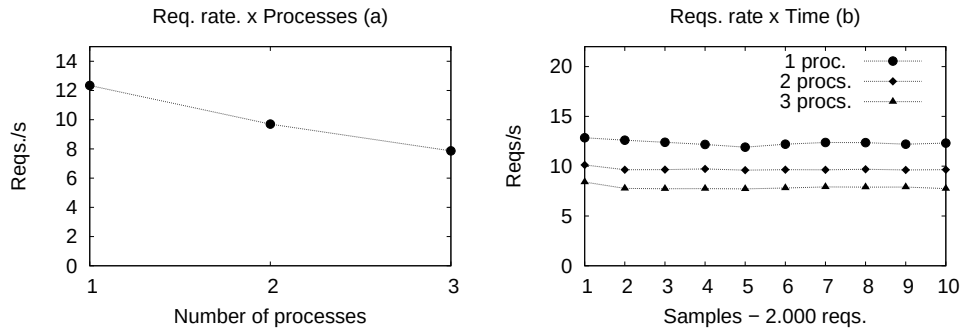

Figura 3: Scalability and performance for perfectly synchronized system.

Figure 4 shows results of (a) efficiency and (b) delay tolerance. On (a), as the number of concurrent processes increases the efficiency decreases, with a considerable increase of the number of cancellations. Considering the biggest number of cancellations (8) and total number of requests (2000), the percentage of cancellations is always less or equal to 0.4%. However, for all cases, requests have been cancelled correctly, showing atomic behavior of protocol. On (b), changing target time by modifying Δ_{opmax} parameter, the number of cancelled requests approaches to 100% when this parameter is closer to the time spent to sent a request and receive confirmations. Again, the number of processes contributes for increas-

ing delay of messages and, so, for increasing percentage of cancelled requests for a given value of Δ_{opmax}.

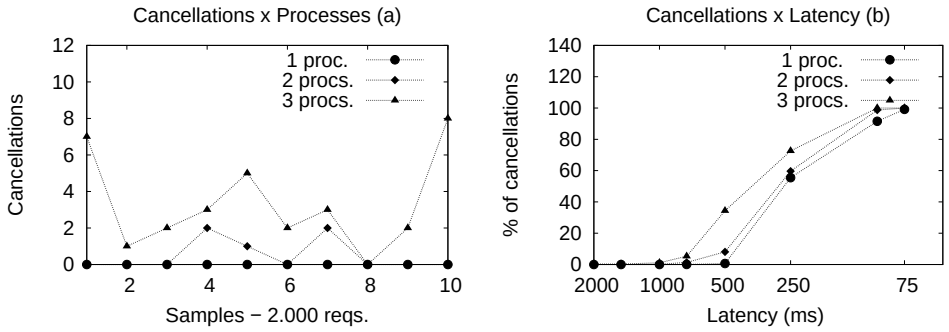

Figura 4: Efficiency and delay tolerance for perfectly synchronized system.

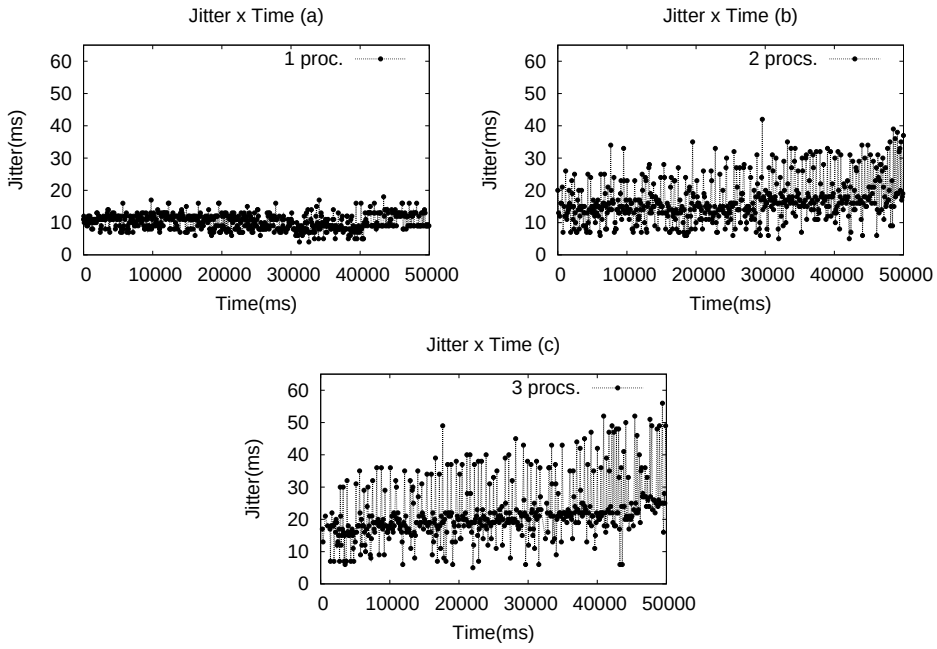

Figura 5: Jitter for perfectly synchronized system.

Figure 5 shows results of jitter. Jitter were less than 15ms for almost all samples with one process (a). However, as concurrent processing increases with more than one real time scheduler running, average value of jitter increases as well as its variation, as show on (b) and (c).

5.2 Experiment 2 - system synchronized via NTP

On second experiment, machines (A), (B) and (C) were used, with one process being executed on each one and all of them synchronized through NTP. Figure 6 shows results for (a) scalability and (b) performance. On (a) the request rate decreases slower as the number of process increase if compared to results shown in Figure 3. This happens because on experiment 1 all processes were running on same machine (A), where processing capability became the limiting factor. The smaller reduction indicates that communication network is not saturated and the request rate is only limited by processing capability, around 12 requests per second, thus ensuring system scalability. (b) shows request rate as function of time, which is almost constant.

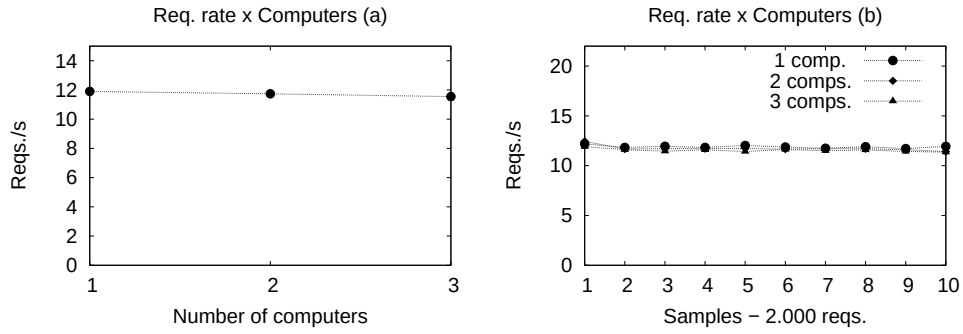

Figura 6: Scalability and performance for system synchronized via NTP.

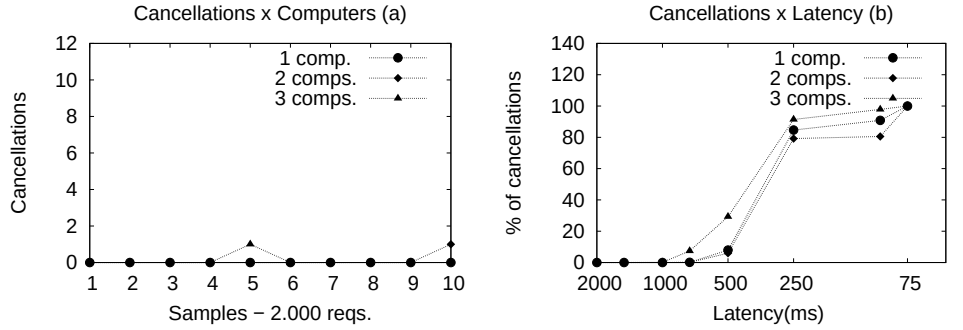

Figura 7: Efficiency and delay tolerance for system synchronized via NTP.

Figure 7 shows results for (a) efficiency and (b) delay tolerance. On (a), is observed a small number of cancellations. Differently from previous experiment, the increase of this value is smaller as number of processes increases. It is smaller because in this case processing capability is not a limiting factor. On (b), changing

Δ_{opmax} also demonstrates that successful request rate decreases as this parameter becomes closer to communication network delay. In this case, the cancellation rate is bigger than previous experiment for the same value of Δ_{opmax}. This happens because additionally to delays due to network protocol stack processing there is also delays associated to network itself, i. e., propagation, switching and interface delays. So, this result is coherent with previous result.

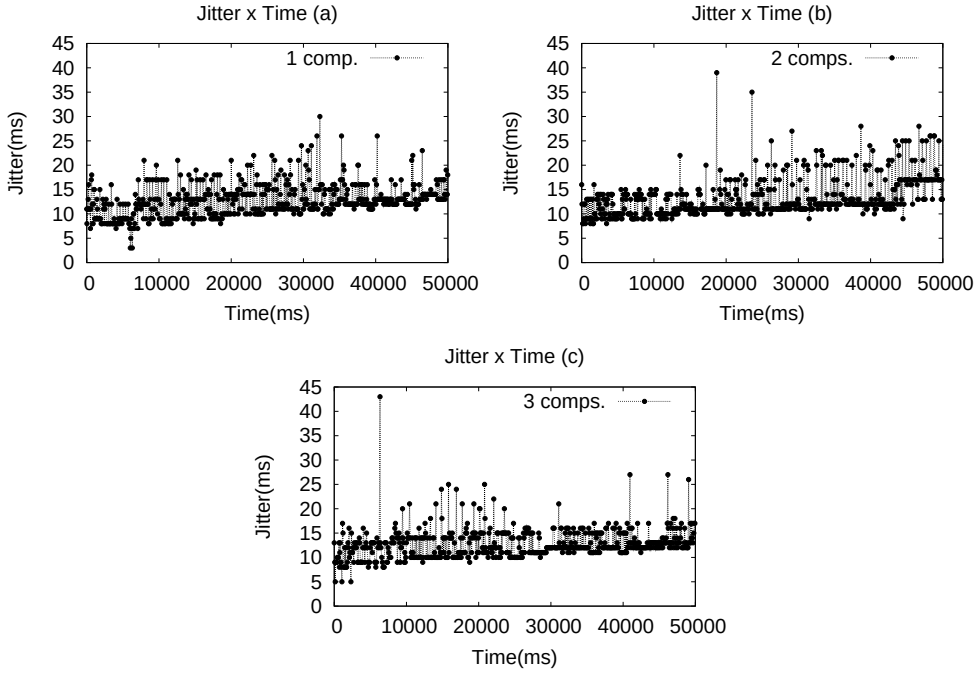

Figura 8: Jitter for system synchronized via NTP.

Analyzing the results for request rate and cancellations as function of Δ_{opmax}, notice that the protocol keeps the request rate constant, even though cancellations are present. This behavior saturates even more the system, hence input buffers keep congested. One possible enhancement would be the implementation of an adaptative control of request rate based on cancellation number, as described on [25].

Figure 8 shows results of jitter. Notice that on almost all samples the jitter value is smaller than 20ms and that an increase on number of processes does not affects this metric. In this case, however, to the jitter value should be added the difference of synchronism among machines, ϵ. These results of jitter, together with request rate, shows that this protocol can be used on applications like automation or robotics, for example, where variations of few tens of milliseconds on execution of actions and rates around 10 requests per second are acceptable.

6 Conclusions

This work presented a reliable protocol called PSAD for action synchronization on real time distributed systems. It allows synchronization of actions on distributed elements of a system in an atomic way. The protocol is based on clock synchronization service, on reliable group communication and on real time task scheduling.

PSAD uses schemes such as NTP or GPS to synchronize all processes of the distributed system. A group communication service supports a closed group and exchanges reliable and FIFO ordered unicast and multicast messages among processes. The messages exchanged allow any process to become a coordinator and request an action to be atomically and synchronously executed on each node. Actions successfully accepted by all processes are scheduled by a real-time scheduler module that ensures a minimum jitter between agreed and real execution time.

The protocol was implemented using NTP to provide synchronization service and two other independent java codes, one using JGroup for group communication services and other using Real Time Java (RTSJ) [3] for real time action scheduling. The java platform allows the use of PSAD on systems composed by several different processors and operating systems.

Evaluation was performed using some computers connected through a local ethernet network. Experiments showed that this solution was able to achieve rates of 10 requests per second, low cancellation rate and jitter around 20ms. These results indicate that PSAD can be used on applications that require distributed synchronization like industrial automation and collaborative motion control, and others.

Future works include studies on mechanisms to control request rate based on number of cancellations, on integration of group communication and real time scheduler codes for execution into a single Java virtual machine to improve performance and tests on bigger and sparser networks.

Referências

[1] Nick I. Kamenoff. One approach for generalization of real-time distributed systems benchmarking. In *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '96)*, 1996.

[2] D. L. Mills. Improved algorithms for synchronizing computer network clocks. In *IEEE/ACM Trans. Networks*. 1995.

[3] The real time specification for java. www.rtj.org, 2005.

[4] Jonathan S. Anderson and E. Douglas Jensen. The distributed real-time specification for java. In RTSJE Group, editor, *Status Report*. JTRES 2006, 2006.

[5] Distributed real-time java. www.real-time.org, 2007.

[6] F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, pages 200–206, Ann Arbor, MI, USA, 1985. IEEE Computer Society Press.

[7] Flaviu Cristian, Danny Dolev, H. Raymond Strong, and Houtan Aghili. Atomic broadcast in a real-time environment. In *Proceedings of the Asilomar Workshop on Fault-Tolerant Distributed Computing*, pages 51–71, London, UK, 1990. Springer-Verlag.

[8] Kenneth P. Birman. *Reliable Distributed Systems*. Springer, 1st edition, 2005.

[9] Paulo Pedreiras. Ftt-ethernet: A flexible real-time communication protocol that supports dynamic qos management on ethernet-based systems. In *IEEE Transactions on Industrial Informatics*. 2005.

[10] Ferdy Hanssen and Pierre G. Jansen. Real-time communication protocols: An overview. 2003.

[11] Seok-Kyu Kweon, Kang G. Shin, and Qin Zheng. Statistical real-time communication over ethernet for manufacturing automation systems. In *IEEE Real Time Technology and Applications Symposium*, pages 192–202, 1999.

[12] S. Kweon, K. Shin, and G. Workman. Ethernet-based real-time control networks for manufacturing automation systems. 2000.

[13] Mehdi Amirijoo Gunnar Mathiason. Real-time communication through a distributed resource reservation approach. In *Technical report HS-IKI-TR-04-004 University of Skovde*. 2004.

[14] Gerold Blakowski and Ralf Steinmetz. A media synchronization survey: Reference model, specification, and case studies. *IEEE Journal on Selected Areas in Communications*, 14(1):5–35, 1996.

[15] María José Pérez-Luque and Thomas D. C. Little. A temporal reference framework for multimedia synchronization. pages 721–736, 2001.

[16] Jun Sato, Koji Hashimoto, Michiaki Katsumoto, and Yoshitaka Shibata. Performance evaluation of media synchronization for multimedia presentation. In *ICPP Workshops*, pages 608–613, 1999.

[17] Feili Hou and Wei-Min Shen. Hormone-inspired adaptive distributed synchronization of reconfigurable robots. In *The 9th Intl. Conf. Intelligent and Autonomous Systems (IAS-9)*, Tokyo, Japan, March 2006.

[18] T. K. Srikanth and S. Toueg. Optimal clock synchronization. In *Journal of the ACM 34:3*. ACM, 1987.

[19] P. Verissimo and L. Rodrigues. A-posteriori agreement for fault-tolerant clock synchronization on broadcast networks. In *Proc. 22nd International Symposium on Fault-Tolerante Computing*. 1992.

[20] M. Clegg and K. Marzullo. Clock synchronization in hard real-time distributed systems. In *Technical Report CS96-478*. University of California, San Diego, Department of Computer Science and Engineering, February 1996.

[21] Júlio da Silva Dias, Ricardo Felipe Custódio, and Denise Bendo Demétrio. Sincronizacão segura de relógios para documentos eletrônicos. In *21º Simpósio Brasileiro de Redes de Computadores (SBRC'2003)*, 2003.

[22] S. Baskiyar and N. Meghanathan. A survey of contemporary real-time operating systems. In *Informatica, vol. 29, no. 2, pp 233-240*, 2005.
[23] Network time protocol. www.ntp.org, 2007.
[24] Java groups. www.jgroups.org, 2007.
[25] John A. Stankovic, Tian He, Tarek Abdelzaher, Mike Marley, Gang Tao, Sang Son, and Cenyan Lu. Feedback control scheduling in distributed real-time systems. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 59, Washington, DC, USA, 2001. IEEE Computer Society.