Avoiding Priority Inversion on the Processing of Requests by Active Replicated Servers *

Y. Wang
Southeast University
Nanjing, PR China
yunwang@seu.edu.cn

F. Brasileiro
Federal University of Paraíba
Campina Grande, Brazil
fubica@dsc.ufpb.br

E. Anceaume, F. Greve, M. Hurfin
IRISA
Rennes, France
{anceaume,fgreve,hurfin}@irisa.fr

Abstract

We consider the priority inversion problem in an actively replicated system. Priority inversion was originally defined in the context of non-replicated systems. Therefore we first introduce the concept of group priority inversion, which extends the concept of (local) priority inversion to the context of a group of processors that perform an actively replicated processing. We then present the properties of a request scheduling protocol to enforce a total ordering for the processing of requests while avoiding group priority inversions. These properties have been implemented in a protocol that relies on a timed asynchronous system model equipped with a failure detector of the class $\diamond S$. The proposed solution allows to replicate a critical server while ensuring that the processing of all the incoming requests is consistent (mechanisms for solving the atomic broadcast problem) and predictable (mechanisms for solving the group priority inversion problem). Thus, the described request scheduling protocol is a key component which can be used to develop fault tolerant real time applications in a timed asynchronous system.

1 Introduction

Many execution platforms for real-time applications use a priority-based scheduling policy. In such systems, detecting and limiting the effects of priority inversions is a key requirement to attain predictability. Informally, a priority inversion occurs when a higher priority *activity* is delayed by a lower priority one. An activity is a generic term that may correspond to a process, a thread, a transaction or even the processing of a request. Furthermore, when the execution time of lower priority activities is potentially much larger than that of a higher priority activity, then priority inversions should ideally be avoided altogether. This in turn requires detecting the occurrence of the priority inversion and preempting the lower priority activity.

*This work is partially supported by a France Telecom R&D grant 98 1B 123 and by CNPq/Brazil grant 200 323/97

Priority inversions can occur at different levels in a system [13]; in this paper we deal with this problem in a specific system context. We consider the design of an infrastructure that allows the development of client/server applications with the following requirements: *i)* requests issued by clients must be processed by a server in accordance with their associated priorities - thus priority inversions must be detected and eliminated; and *ii)* servers are actively replicated to tolerate processor faults. An example of such an infrastructure is one that implements the specifications of both Real-Time CORBA [7] and the active replication style of Fault Tolerant CORBA [8], where part of an application can be implemented by a *group* of replicated server objects using a *client propagated* priority model policy.

Active replication is a well established strategy to implement fault-tolerant systems. Compared to the *primary-backup* replication scheme, active replication avoids long recovery procedures. Even when no failure occurs, active replication contributes to enhance performances of distributed applications by improving the response time provided by servers. Moreover, as indicated in [9], "the scope for the implementation of standby redundancy is limited". For these two reasons, active replication is the preferred approach to implement fault-tolerant real-time systems [10, 11]. In these systems, failures are masked by processing *replicas* of an application in a sufficiently large group of processors whose components fail independently. Replicas execute a total ordering protocol, which guarantees that input requests issued by clients will be processed by all functioning replicas in the same order; assuming that the behavior of a replica is deterministic, all functioning replicas undergo the same state transitions and issue identical replies to any given request. If necessary, a validation function (*e.g.*, majority voting) can be applied to the outputs produced, masking those that could be generated by faulty replicas.

The total ordering protocol which allows replicas to process the same sequence of input requests is the core of a fault-tolerant system that employs active replication. Further, when the extra requirement of dealing appropriately with priority inversions is introduced, the ordering on which input requests are processed must not only be consistent among the replicas, but also "sensible". It is interesting to point out that when the computation is replicated, the very notion of priority inversion is not clearly defined [12, 14]. Since there is no synchronization on the instants of time when input requests are received, made available for processing, or processed at different replicas, it is possible that one replica experiences a "local" priority inversion, while another does not. Thus, the first contribution of this paper is the introduction of the concept of *Group Priority Inversion* (GPI), which extends that of local priority inversion to the context of a group of processors that perform an actively replicated processing. Informally, a group priority inversion occurs when *too many* local priority inversions have occurred, causing the processing of some higher priority activity to be delayed by that of lower priority ones. Thus, in our target system, "sensibly" ordering requests means doing that in such a way that group priority inversions are avoided.

In a synchronous system, if group priority inversions are avoided and the arrival law of input requests is known, the predictability of the replicated system will not be affected. Predictability is the capability of the system to perform a set of activities in a well-defined fashion, such that each activity's timing requirements are satisfied: when an activity is predictable, its result and its worst case execution time are known before its execution. Therefore, predictability is essential to support real-time applications. The fault-tolerant real-time systems which have previously been mentioned [10, 11] are based on synchronous systems in which predictable runtime behaviors can be ensured. However the definition of a synchronous system relies on the existence and the knowledge of temporal bounds within the system, *i.e.*, the delay of message transmission and the speed of processing. Therefore relying on strict timing assumptions is unsafe especially in systems where performance failures (*e.g.*, network congestions) are possible. Such failures are generally infrequent but may prevent most of the synchronous protocols from being used. In this paper rather than considering a "pure" synchronous system, we suppose that the system alternates between "good" and "bad" periods, as in the Timed Asynchronous Distributed System Model of [1]. During a good period, the system behaves as a synchronous system, while in bad periods, the system is asynchronous (*i.e.*, performance failures may occur). Of course, the durations of the good periods must be longer than those of the bad ones. It would be futile to develop real-time application in an completely inadequate environment. Clearly, the predictability analysis is valid only during good periods. The second contribution of this paper is a solution to avoid GPI within a set of replicated servers even when the system enters bad transient periods. Moreover, this solution guarantees the equivalence between the avoidance of GPI and predictability during good periods.

A number of total ordering protocols have been presented in the literature. As far as we know, only two of them consider the priority associated with requests to establish total ordering [12, 14]. These protocols were not designed to avoid group priority inversions which is not surprising, since group priority inversion is a new concept introduced in this paper. They differ from the protocol we propose in several aspects. We discuss these differences in detail in Section 7. In this paper, the term *request scheduling* protocol refers to a total ordering protocol that takes priority of input requests into account and avoids the occurrence of group priority inversions. We focus on the design of request scheduling protocols. On the other hand, we do not discuss details associated with any protocol that could be required to filter the replies produced by a replicated server, either to validate them, or to discard duplicates.

The paper is organized in the following way. Section 2 presents our target system model. Section 3 recalls some key definitions about the active replication model. Section 4 defines the notion of group priority inversion in a general setting. In Section 5, we formally define the problem that has to be solved and we discuss the main difficulties related to the design of a solution. Then, in Section 6, a request scheduling protocol is presented. In Section 7 we discuss related works. Finally, Section 8 brings our concluding remarks.

2 Model

2.1 System Model

We assume that an application functionality is implemented by a number of cooperating entities (*e.g.*, objects, processes) that interact only via the exchange of messages. *Client* entities send *input request* messages to *server* entities that process them and send back a *reply* message as response.

We consider that the system follows the timed asynchronous distributed system model of [1]. All services are timed and entities have access to hardware clocks that proceed within a linear envelope of real-time. We consider a progress assumption extension of this core model. This progress assumption states that the system alternates between "good" and "bad" periods. During a good period, entities behave like in a synchronous system and communicate with each other in a timely manner.

Entities have crash/performance failure semantics. The execution of an entity might stop prematurely (crash failure), or an entity can fail to process some code in a timely manner. A crashed entity does not recover. An entity behaves correctly, that is according to its specification, until it (possibly) crashes. By definition, a *correct* entity is an entity that does not crash. A *faulty* entity is an entity that is not correct. Both correct and faulty entities can suffer from performance failures: this occurs either when an activity is activated later than expected or when an activity misses its deadline.

The datagram service is assumed to have a performance failure semantic: it can fail to deliver messages in a timely manner, but one can neglect the probability that a message delivered by the system is corrupted, loss or is delivered multiple times.

Finally, we assume that each non-crashed entity has access to a correct hardware clock H_p, *i.e.*, has access to a hardware clock with a known and finite drift rate ρ. This assumption simplifies applications since they have to deal with crash failures anyhow but they do not have to deal with faulty clocks like "fast" or "slow" clocks [1].

2.2 Replication of a Critical Service

The growing dependency on computer systems has created a need for fault tolerant systems to ensure uninterrupted business activities. Replication is a key to provide a higher degree of fault-tolerance in distributed systems. A classical way to make a critical server reliable is to replicate it on different sites of the distributed system. If sites fail independently, an invocation to a fault tolerant server will succeed even if some replicas may crash before they have completed their tasks. Without loss of generality and in order to simplify both the presentation and the notations, we consider a single particular critical server denoted S. To tolerate crash failures, this server is replicated on n processors: each processor p executes one replica of S, denoted S_p. Let us denote by Π the group of processors associated to the server S. The cardinality of this set $|\Pi| = n$ is called the replication degree of the replicated server S. Among the n processors, we assume that f are faulty while $n - f$ are correct, *i.e.*, will never crash: $0 \leq f < n$.

2.3 Failure Detection

Failures have to be tolerated by the implemented mechanisms: their impact on the safety has to be null while their impact on the liveness has to be limited. However, in contrast to performance failures, crash failures have to be equally detected. To detect crash failures, several approaches have been investigated. One of them is based on the concept of an *unreliable failure detector* [2]. In this approach, each entity has access to a failure detector module that provides it with a list of processors that it suspects of having crashed. According to the properties a failure detector is assumed to satisfy (namely the completeness and accuracy properties), several classes of failure detector have been defined [2]. Among them, the class denoted $\diamond\mathcal{S}$ is of particular interest (since it is the weakest one that solves consensus [3]). This class is defined by the following two completeness and accuracy properties: any process that crashes is eventually suspected (completeness), and there is a time after which there is a correct process that is no longer suspected (eventual weak accuracy). In order to present the most general solution to the problem of group inversion priority, we consider herein asynchronous distributed systems equipped with $\diamond\mathcal{S}$ failure detectors. In this context, agreement problems can be solved assuming that a majority of the group replicas never fails [2]. Of course, stronger assumptions, like for example the existence of perfect failure detectors, are also compliant with the proposed solution and offer some advantages: perfect failure detectors allow to use more efficient agreement protocols and to tolerate up to $n - 1$ crashes.

3 Active Replication

3.1 Principles

In the active replication style, the copies of the critical server run concurrently. All of them receive the requests from clients, execute the corresponding operations, and send back the responses. When a failure occurs on one server, the failure is masked and the computation continues as long as there is some running server. This replication style assumes a deterministic behavior on these replicas and requires a reliable atomic broadcast mechanism to maintain consistency.

Why a reliable broadcast is needed: any broadcast request has to be received and processed concurrently by all the operational replicas belonging to Π. As indicated in Section 2.1, communication channels are reliable. Therefore, the broadcast primitive supported by the network is also reliable.

Why an atomic broadcast is needed: different requests submitted by different clients have to be processed by all the correct replicas in the same order. If the operations on the replicated object are deterministic, the local states of the replicas remain consistent and all responses sent back to the client are identical. Note that regarding faulty replicated entities, these properties guarantee that a prefix of this sequence of requests is processed. To satisfy the above requirement, an actively replicated system relies on the properties provided by an atomic broadcast primitive. Within this paper, we will outline a particular way to implement such a primitive (see Section 6.2). An atomic broadcast primitive guarantees three fundamental properties: *i)* **Uni-**

form Agreement, which ensures that for any request r, if r has been processed by a replicated entity then all correct replicated entities eventually process it, *ii)* **Termination**, which guarantees that for any request r, r is eventually processed by some correct replicated entity; and *iii)* **Total Order** which says that for any request r and r', if both requests have been processed by any two replicated entities, then they have been processed in the same order.

3.2 Validation Mechanism.

A value which is neither incorrect nor altered is called a *good* value. By definition, the data integrity of the good values are preserved even when faults occur. Note that a good value is not necessarily a timely value. To guarantee that despite the failures of processors in Π only good responses are sent back to the clients, some validation mechanism is needed. This mechanism is not intended to ensure a timely submission of the response to the client. We denote by $\vartheta(a)$, $\vartheta(a) \subseteq \Pi$, the set of all the processors that generate a good value for activity a. This set contains all the processes that deliver their reply. The validation mechanism is applied to the output generated by the processors and ensures that only one good response is delivered to a client. It enables to mask processor failures by applying a given function on the output values. The function applied by the validation mechanism requires to get as input at least v values to compute the good value that is sent back to a client. The value v typically depends on the processor failure mode. In our case, this function is trivial since all generated outputs are good. Thus $v = 1$: the validation mechanism returns the first received response. The introduction of the parameter v is motivated by the fact that we want to define the Global Priority Inversion independently from the failure model. To illustrate the interest of this parameter, let us consider a Byzantine failure mode. If some non-crashed processes (at most f_e) can generate erroneous (or bad) values, then the values returned by the replicas have to be compared. If at least $|\Pi| - f_e$, with $f_e \leq \lfloor (|\Pi| - 1)/3 \rfloor$, output values are good, then a voting procedure can be implemented by the validation mechanism. Thus, in this case $v = (2|\Pi| + 1)/3$.

4 Group Priority Inversion

4.1 Terminology

As previously said, clients send input requests to the replicated server S. A fixed priority is affected to each input request. Let r be the (unique) identifier of an input request, and *Priority(r)* its level of priority. By definition,
Priority(r)<Priority(r') iff input request r has a priority level smaller than the one of r'.

Each processor $p \in \Pi$ executes a local *Request Scheduler* module denoted RS_p. The role of this module is *i)* to receive the input requests submitted by the clients, *ii)* to schedule them according to their priorities, *iii)* to avoid the occurrence of group priority inversions, and *iv)* to send the reply of each input request to the corresponding clients. The interactions between replica S_p and RS_p (see Figure 1) are very simple: whenever S_p has completed the processing of an input request and thus is ready to process the next one, S_p queries RS_p to obtain a new input request. If no input request is ready at RS_p, then replica S_p remains blocked until RS_p is able to provide the next one.

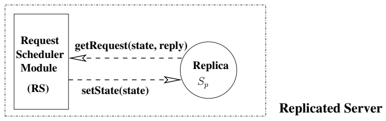

Figure 1. Interactions between S_p and RS_p

At any time t, each module RS_p maintains a local history denoted *History*$_{p,t}$. The sequence *History*$_{p,t}$ is the concatenation of two lists *Completed*$_{p,t}$ and *Ready*$_{p,t}$. The list *Completed*$_{p,t}$ contains the sequence of identifiers of all the input requests that have already been processed by S_p by time t (if any), while *Ready*$_{p,t}$ contains the sequence of identifiers of all the input requests that are ready to be processed by S_p by time t (if any). The time at which a request r is inserted in *History*$_{p,t}$ is denoted by $t_p^{history}(r)$. Time $t_p^{submit}(r)$ is the time at which request r is submitted by a client to module RS_p. Similarly, $t_p^{compl}(r)$ is the time at which replica S_p completes the processing of r. Obviously, $t_p^{submit}(r) < t_p^{history}(r) < t_p^{compl}(r)$. The i^{th} element of *History*$_{p,t}$ is denoted by *History*$^i_{p,t}$. Note that in the proposed solution, the rank of a request is not fixed. It is first computed at $t_p^{history}(r)$ and can increase later. Let *History*$^i_{p,t} = r$ and *History*$^j_{p,t} = r'$, with r and r' the identifiers of two input requests. In the following a distinction is made between the scheduling of an input request r—which is done by the request scheduler RS_p, and the processing of this input request r—which is done by S_p. In the latter case, we simply say that S_p processes the activity r. When S_p *completes* the processing of r at $t_p^{compl}(r)$, then RS_p adds r at the tail of *Completed*$_{p,t}$. Thus the number of elements of *Completed*$_{p,t}$ represents the *local execution progress* of replica S_p at time t, and is denoted by $LEP_{p,t}$. RS_p updates the value of $LEP_{p,t}$ right after the completion of an activity by S_p. Therefore, $LEP_{p,t}$ represents the number of activities S_p has already processed by time t, or equivalently, $LEP_{p,t}$ indicates the border line between *Completed*$_{p,t}$ and *Ready*$_{p,t}$ in *History*$_{p,t}$. Finally, we use the notation $Running_{p,t}$ when referring to the activity replica S_p is processing at time t. When replica S_p is idle at time t, then $Running_{p,t} = \perp$, otherwise, $Running_{p,t}$ is the head of *Ready*$_{p,t}$, denoted either *Ready*$^1_{p,t}$ or *History*$^{LEP_{p,t}+1}_{p,t}$.

4.2 Definitions

In this context, we formally define the *Local Priority Inversion* (LPI) in the following way:

Definition 1 (Local Priority Inversion) *for any processor p and activity r, r has suffered a local priority inversion on p (i.e., $LPI_p(r)$ is true), if and only if, the following expression holds:*
$\exists t$: *Running*$_{p,t} \neq \perp \wedge$
$r \in$ *Ready*$_{p,t} \wedge$ *Priority(Running*$_{p,t}$*)* <*Priority(r)*

Recall that the effect of a local priority inversion is that a higher priority activity that is available for processing is delayed by a lower priority activity that is being processed. By extension, a group priority inversion will occur whenever the processing of a higher priority replicated activity r is delayed by lower priority activities. As discussed before, the processing of a replicated activity r is good if it is processed by at least v processors. Thus, r may exhibit a group priority inversion if its processing is delayed on *too many* processors. In other words, it occurs if the number of processors in $\vartheta(r)$ that did not suffer a local priority inversion with respect to r is less than v. Therefore, we formally define the *Group Priority Inversion* (GPI) in the following way:

Definition 2 (Group Priority Inversion) *for any activity r, r has suffered a group priority inversion (i.e., $GPI(r)$ is true), if and only if, the following expression holds:*
$\exists \Psi \mid \Psi \subseteq \vartheta(r) \wedge (\forall p \in \Psi : LPI_p(r)) \wedge |\vartheta(r)| - |\Psi| < v$

Note that in our case $v = 1$. We can now relate the definition of the predictability of an activity with the GPI definition. Let $[t, t']$ be a time interval during which the system is in a good period. Then any activity r (submitted by a client after time t and whose reply has been received by the client by time t') is predictable if $GPI(r)$ is false during this time interval.

4.3 Avoiding Group Priority Inversion

Priority inversions can only be eliminated if the system allows preemptions; this in turn can be accomplished by either suspending the processing of a lower priority activity r_l that is causing the priority inversion, or aborting activity r_l. In replicated systems, the latter approach is preferable, because the synchronization mechanism required to guarantee that all replicas r_l will be suspended at the same execution point is typically very costly. We therefore assume that when querying the RS_p module for a new activity (invocation of the `getRequest(state,reply)` function on Figure 1), replica S_p gives to this module *i)* its new local state (which is defined by the essential local variables managed by S_p), and *ii)* the result of the activity it has just processed . This local state is used by RS_p to save a local checkpoint before allowing the processing of the next activity. Thus, if required, the RS_p module can roll-back replica S_p to the previous consistent state (Due to space limitations, the checkpointing algorithm and the description of the roll-back mechanism used are not discussed in this paper.). More generally, the RS_p module can roll-back replica S_p to a previous consistent state different from the last one if several consecutive activities need to be undone. Inconsistencies may occur if a client receives the reply for a request r whose processing has been rolled-back. To prevent such a problem, the RS_p module logs the reply of r and only sends it to the requesting client when RS_p is guaranteed that the state of S_p will never be rolled-back to a state that precedes the processing of r. When a roll-back is required, the RS_p module asks S_p to stop the processing of the current request (if any), uses a checkpoint to re-initialize S_p in a previous state (invocation of the `setState(state)` function on Figure 1), and provides to S_p the activity which has entailed a modification of the processing order in *History*$_p$. Thus to summarize, an activity r can be processed several times by a replica (or equivalently a replica may have completed the processing of r several times) because of multiple roll-backs but the client that sent r receives at most one reply for this request. Note that the notation $t_p^{compl}(r)$ always refers to the last execution of activity r.

To limit the potential cascade of roll-backs, the notions of irreversibility and conflict are introduced. By definition, the processing of a request r is *irreversible* at S_p if: *i)* S_p has completed the processing of r, and *ii)* r will never be rolled-back by S_p in the future.

Definition 3 (irreversible) *r is irreversible if*
$\exists S_p \in \Pi$, $\exists t$: $r \in$ *Completed*$_{p,t}$ *and* $r \notin$ *Ready*$_{p,t'}$ $\forall t' > t$

By definition, a request r is *conflict-free* if for all r' with a higher priority level than r, either the submission of r' occurs before the submission of r on all S_p or the submission of r' occurs after the completion of r on all S_p.

Definition 4 (conflict-free) *r is conflict-free if*
$\forall r'$ *with Priority(r') > Priority(r):*
either $t_p^{submit}(r') < t_p^{submit}(r)$ $\forall S_p \in \Pi$
or $t_p^{submit}(r') > t_p^{complete}(r)$ $\forall S_p \in \Pi$

5 The Problem

5.1 Formal Specification of the Problem

We address the combination of the atomic broadcast problem and the group priority inversion problem in a distributed system subject to crash and performance failures. Informally, avoiding priority inversion in an actively replicated system turns out to guarantee that all the correct replicated entities process the same sequence of requests while preventing priority inversion to occur. Thus, a specification of the problem we address combines the properties of both the atomic broadcast and the group priority inversion problems. To guarantee that requests are not indefinitely undone (*i.e.*, their processing is irreversible), we need to add a non triviality property. A specification of the GPI problem in an actively replicated system is as follows:

Total Order: for any two activities r and r' irreversibly processed by any two replicas $S_p \in \Pi$ and $S_q \in \Pi$, if r' has been irreversibly processed by S_p before r, then r' has also been irreversibly processed by S_q before r.

Priority Order: for any activity r, $GPI(r)$ is false.

Uniform Agreement: any activity r that is irreversibly processed by replica $S_p \in \Pi$, is eventually irreversibly processed by all correct replicas $S_q \in \Pi$.

Termination: for any activity r, if r is received by a correct replica $S_p \in \Pi$, then r is eventually completed by S_p.

Non Triviality: for any activity r, if r is conflict-free, then the processing of r is irreversible, and r is processed once.

Note that the second property, namely the Priority Order property, is equivalent to the Predictability property when the system evolves in good periods.

5.2 Difficulties in Solving the Problem

To implement the active replication scheme within the set Π, each replica S_p needs to receive all the input requests in the same order. By construction, replicas receive the input requests from the RS modules. Thus, because of crash and performance failures, an agreement on the request ordering needs to be found among all the RS modules in Π. This agreement is obtained by executing a total ordering protocol on the flow of input requests clients are issuing. Typically, a total order protocol runs as follows: a `waiting for` loop waits for the input requests submitted by clients. Each time a RS module receives such requests, this module asks all the other RS modules to agree on a (unique) order for these requests. This unique order, called the decision value of the protocol, must be known by all RS modules. When a new decision is learnt, each RS module updates its *Ready* list. In addition, to cope with the priority inversion requirement, the ordering in which input requests will be processed must also be updated to prevent priority inversions.

A local priority inversion on a new input request r can be trivially detected by RS_p by simply testing if *Priority(r)>Priority(Running*$_{p,t}$*)* when RS_p schedules and adds r to *Ready*$_{p,t}$ at time t. This test directly derives from the definition of LPI. On the other hand, since each new input request is received and scheduled by each RS module gradually and asynchronously, the detection of a group priority inversion is rendered difficult. Indeed, a local priority inversion detected by a module RS_p may never occur from the point of view of another module RS_q because their *History*, or more precisely their *Ready* list, is updated (addition or removal of an element) in a desynchronized way. To illustrate this problem, let us consider two simple scenarios. These scenarios aim at underlying the inherent RS "desynchronization" whatever the processing context. In both scenarios, r_h and r_l are two requests such that *Priority*(r_h)>*Priority*(r_l).

Scenario a) observations of new requests are not synchronized

Let S_p and S_q be two replicas such that *i)* both start the processing of activity r_l at time t_1, and *ii)* both complete this processing at time t_3, with $t_1 < t_3$ (*i.e.*, $Completed_{p,t3}^{LEP_{p,t3}} = Completed_{q,t3}^{LEP_{q,t3}} = r_l$). During the processing of r_l, RS_p receives r_h and adds it to *Ready*$_{p,t2}$ at time t_2, while r_h is received by RS_q and added to *Ready*$_{q,t4}$ at time $t4$. If we consider $t_1 < t_2 < t_3 < t_4$, RS_p detects a local priority inversion at time t_2 because *Priority(Running*$_{p,t2}$*)<Priority*(r_h). On the other hand, RS_q does not detect a local priority inversion at time t_4.

This scenario shows that even if the processing of activities is perfectly "synchronized" at all replicas (*e.g.*, the processing of activity r_l), the desynchronized observation of new input requests (insertion of r_h in *Ready* at different times at each replica) may lead the local *Histories* to diverge.

Scenario b) updates of the local execution progress are not synchronized

Let S_p and S_q be two replicas such that *i)* input request r_l is simultaneously received by RS_p and RS_q at time t_1, and *ii)* request r_h is simultaneously received by RS_p and RS_q at time t_3, with $t_1 < t_3$. At time t_2, with $t_1 < t_2 < t_3$, replica S_q completes the processing of r_l (*i.e.*, r_l is removed from *Ready*$_{q,t_2}$ and added to *Completed*$_{q,t2}$). At time t_3 replica S_p has not yet completed the processing of r_l. Thus RS_p observes a local priority inversion at time t_3, while RS_q does not.

This scenario shows that even if all RS modules observe the same sequence of input requests at the same time (insertion of r_l and r_h in *Ready* by each RS module at the same time), there is no guarantee that they will detect the same local priority inversions, and thus, the processing of replicas might be different.

These scenarios demonstrate that to detect and avoid group priority inversions, all the RS modules must clearly have a consistent group view of their history but also of their execution progress. Otherwise, either non-existing group priority inversions will be detected or real ones will be missed. Therefore, the RS modules must counterbalance the inherent lack of synchrony of the system by achieving an agreement on the execution progress within the group.

5.3 Overview of the Proposed Solution

A trivial solution to avoid group priority inversions is to guarantee that at any time t the whole $History_{p,t}$ maintained by RS_p is ordered according to the increasing level of priorities of input requests. An implementation of this solution may be as follows: each time RS_p observes a new input request r (*i.e.*, upon receipt of a new decision value from the total ordering protocol), it examines its whole $History_{p,t}$. Then whenever it finds some input request r' whose priority level is less than the one of r, it puts r in r' place and shift by one the preceding requests. In the worst case, the whole $History_{p,t}$ may be reordered (request r has the higher level of priority among all the requests in $History_{p,t}$), and thus all the requests in $Completed_{p,t}$ may have to be rolled-back.

Clearly, this solution guarantees that group priority inversions are avoided whatever the desynchronization of the system could be. It is simple and its cost in number of messages is very low since no agreement is necessary. Unfortunately, this solution does not solve the problem we address. Indeed, this solution does not meet the Non-Triviality property because nothing guarantees that the processing of a request will eventually be irreversible.

It is necessary to agree on some threshold below which input requests will never be undone. This threshold can be seen as a *Group Execution Progress* (GEP) within Π. Clearly, this value, computed and agreed among all the correct RS modules, must be related to the local execution progress of each replica in Π. By testing GEP, each *RS* module is then able to detect whether a given input request may possibly be reordered or not. The value of GEP delimits a second border line within the *History* of each *RS* module by separating *stable* input requests from *unstable* ones. A stable request is a request whose ordering in *History* will never be changed in the future, *i.e.*, a stable request will never cause another input request r to satisfy *GPI(r)=true*. By definition, for all replicas S_p, all the input requests r such that $i \leq GEP$ with $History_p^i = r$ are stable. Consequently, once stable requests r have been processed their processing is irreversible. On the other hand, an unstable request r' may possibly be reordered and even undone to avoid a group priority inversion. Thus processing of r' is not irreversible. Note that the checkpoints taken by each RS module upon receipt of the `getRequest(state,reply)` function (see Figure 1), need to be taken only for unstable requests, since by definition such requests can possibly be undone.

Finally, the evolution of *GEP* must be so that group priority inversions are avoided while detections of non-existing group priority inversions are prevented or limited. The next section describes the protocol run by the request scheduling module to solve the GPI problem in an actively replicated system, and shows how the value of GEP is computed.

6 A Request Scheduling Protocol

6.1 Use of GAF, a Generic Agreement Service

Any traditional consensus protocol could be used as the agreement service to satisfy the requirements of our solution (*i.e.*, agreement on the input requests ordering and agreement of the GEP value). The General Agreement Service (GAF) described in [5] extends the properties of the Consensus problem with regard to the decision value. Indeed, the decision value may be obtained by combining all (or a subset of) the proposed initial values. These features led us to use this framework for our solution, especially for its capacity to decide on the sequence combining all initial values. Note that GAF is based on $\diamond S$ failure detectors.

In the following, we describe the solution we propose to solve the problem addressed in Section 5 in a progressive way. First, in Section 6.2 we present a description of the Request Scheduler module algorithm by assuming that all the input requests submitted by clients have the same priority level. In Section 6.3, this algorithm is extended to take into account the input request priorities and thus to solve the addressed problem.

6.2 Ordering Input Requests

Request Scheduler modules are composed of two submodules called respectively the *Total Order* module (TO for short) and the *Basic Scheduler* module (BS for short), see Figure 2(a). The goal of the TO module is twofold: first to take into account the flow of input requests submitted by the clients, and second to ensure that all the non-crashed BS modules observe the same sequence of input requests. The BS module prevents group priority inversion by possibly changing the total order initially defined by the TO module. In this section, we assume that BS modules only buffer input values (*i.e.*, those issued by the TO module), and progressively deliver these requests to the replicas.

Without loss of generality, we can consider that the flow of input requests is continuous. A TO module progresses step by step. At each step, one or several new requests are ordered: this new subsequence extends the global sequence of requests previously observed. This subsequence is obtained by executing an agreement protocol (by using GAF), and is similar to the consensus-based solution described in [2]. It allows the TO modules to agree on a sequence of input requests. This order is then discovered by all the replicas in Π gradually and asynchronously. More precisely, a TO module is formed by two basic services, the *buffer unit*, that is responsible for managing the requests and the *agreement unit*, that is responsible for ordering the requests. Figure 2(b) illustrates these basic modules.

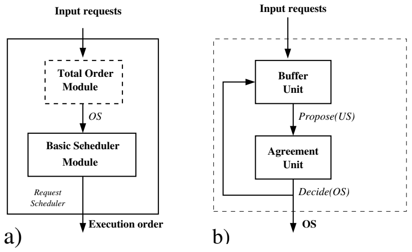

Figure 2. The TO Module

The Buffer Unit The buffer unit maintains a set of unordered requests, called the *Unordered Set*, and denoted US. This set is locally updated each time a input request is received (addition of a request) and each time a new decision is reached by the agreement unit (removal of requests). Thus, US contains all the input requests that have been received but not ordered yet. The buffer unit repeatedly invokes the agreement unit. To disambiguate between these invocations, a sequence number is associated with each invocation. The $k+1^{th}$ invocation is done when *(i)* the buffer unit receives the decision value of the k^{th} invocation (and it has removed from US all the input request identifiers that equally appear in the k^{th} decision value), and *ii)* new requests have to be ordered (*i.e.*, the local US is not empty). The agreement protocol guarantees that the decision value, called an *Ordered Set* and denoted OS, is eventually received by all non-crashed buffer units and BS modules (see Figure 3.b).

The Agreement Unit The agreement protocol is encapsulated within the agreement unit. This *agreement protocol* is at the core of the proposed solution. To solve the total order problem, the decision value provided by GAF combines all the gathered sets of input requests proposed by each buffer unit. This feature enables better performances with regard to the traditional single decision value.

6.3 Avoiding Group Priority Inversion

Agreeing on a Group Execution Progress As presented in Section 5.3, the RS modules need to achieve an agreement on the value of the group execution progress GEP. The value of GEP must satisfy two properties: *i)* it must result from an agreement among all the RS modules in Π, and *ii)* it must be updated whenever a new batch of ordered requests is observed by any RS module. A natural solution to meet both properties consists in combining the two agreement problems (*i.e.*, the agreement on GEP and the agreement on OS) into a single one so that any execution of the agreement protocol leads to decide on both values simultaneously. Therefore, the agreement unit manages two pairs of values, namely *i)* a set of input requests US and an execution progress value *EP* in input (`Propose(US,EP)` on Figure 3.b), and *ii)* a set of ordered input requests OS and the group execution progress value in output (`Decide(OS,GEP)` on Figure 3.b).

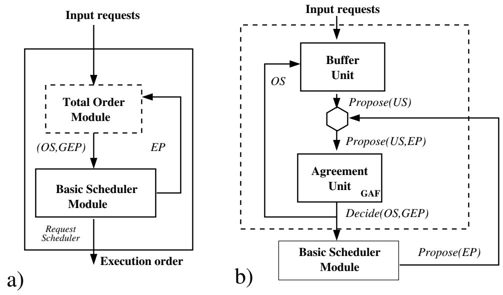

Figure 3. The Extended TO Module

Invoking a New Agreement As previously said, a new agreement protocol can only start if the previous agreement is completed. Once this condition is met, the invocation of this protocol can be done by both the buffer unit and the BS module. In the former case, new input requests have to be ordered, while in the latter case each BS_p module proposes a value which is equal to the maximum value between its current LEP_p value and the last decided value of GEP. Proposing the maximum value between the current LEP value and the GEP value ensures a monotonic progress of GEP. If one of these modules (*i.e.*, the buffer unit or the basic scheduler module) has nothing to propose, then a non-significant value is proposed: an empty set of input request for the buffer unit or the current value of GEP for the BS module (in particular for the first agreement invocation, $EP = max(LEP, GEP) = 0$). Practically, when the agreement unit is invoked by only one of these two modules (for example by the buffer unit), then it asks the silent module to propose a value. This strategy prevents any blocking behavior from the agreement protocol.

Computing GEP The computation of the GEP value is done by the GAF protocol. The GAF protocol gathers initial values from a majority of processors in Π. Then it applies a given function $\mathcal{F}$ on the set of gathered values to obtain the new value of GEP. Two steps are required to compute this value. First, the subset containing the $f+1$ greater initial values is determined. Note that enough values have been collected because, in our model, $f < |\Pi|/2$. The selection of the greatest values (*i.e.*, the values issued by the fastest BS modules) allows a quicker progress of GEP. By considering $f+1$ values, we ensure that at least one of the values has been provided by a correct BS (and thus by its associated replica) that is, by a BS that will never crash. In the worst case, f BS modules can propose their current value and crash later. In a second step, the function $\mathcal{F}$ returns a value of GEP which is the minimal value contained in this set of $f+1$ values. This strategy ensures that at least one of the correct BS modules has already processed at least GEP requests. In other words, the parameter function $\mathcal{F}$ gives a safe estimation of the execution progress of one of the processors that never crashes. Moreover, as this estimation is based on the values provided by a majority of BS modules, any computation done during any future agreements will guarantee that the computed value increases monotonically: any set of gathered values contains at least one value which is greater or equal to the value computed during the previous execution of the agreement protocol.

Reaction to a Decision Reacting to a new decision value is the main activity of the BS module. Let us first introduce some notations. A variable *Max* is used to store the current number of elements contained in *History* before the reaction to a decision. The notation *History*$[x..y]$ is used to identify a sub-sequence of the original sequence which begins at position x and ends at y. When a new decision value *(OS,GEP)* is received at BS, the following actions are performed.

1. Updating GEP and the Log The value of *GEP* is updated with the new value contained in the received decision value. By definition of *GEP*, with *GEP*>0, the prefix *History*[1..*GEP*] is stable. All the checkpoints related to stable requests are useless and thus are destroyed. For each stabilized request, the log data is used to transmit any postponed replies to the clients. If *LEP*<*GEP* further processing can be done without taking a checkpoint and delaying the transfer of information to clients. Otherwise, both checkpointing and reply data logging are necessary.

2. Re-ordering The cardinality m of the new ordered set OS is determined. Then, *History*[*Max+1..Max+m*] is initialized with the identifiers of the new input requests of *OS*. The subsequence *History*[*GEP+1..Max+m*] is sorted according to the priorities associated to the requests. In the following, SR denotes the first position in *History* at which a new ordered request is inserted.

3. Rolling-back According to the both *GEP* and *LEP*, three cases have to be analyzed.

3.1. *LEP<GEP*: No roll-back is necessary since by definition the processing mode is stable.

3.2. *LEP=GEP*: The last processed request, *i.e.*, *History*$_p^{LEP}$, is stable by definition of *GEP*. On the other hand, the request r that is currently being processed (if any) satisfies *GPI(r)=true* if *SR=LEP*+1. In that case, a roll-back to the last checkpoint needs to be performed.

3.3. *LEP>GEP*: Some of the processed requests are unstable. These requests may have been re-ordered during the merge procedure. The SR value is used to detect from which point a roll-back is necessary (when $SR \leq LEP+1$).

Note that a roll-back implies to stop the request that is currently being processed (if any) and to remove from the log all information regarding the requests that will have to be re-processed. Also, the value of *LEP* is reset accordingly (*LEP*=*SR*-1). Finally, the *RS* module launches the processing of the first waiting request (*History*[*LEP*$+1$]).

7 A Comparison with Related Work

As indicated in the Introduction, the problem that we analyze has connections with the problems studied in [12] and [14]. Let us denote these two related solutions by the initial of their authors, namely TS and RVC.

In the TS solution, the concept of group execution progress overlaps the concept of local execution progress. Any request which has been processed is stable. In particular, for any RS_p module $LEP_q \leq GEP$. The RS modules have different *Ready* lists whose updates are not synchronized. To ensure that every replica will process the requests in the same order, an agreement protocol is executed by the RS modules each time a new request has to be processed by the replicas. Thus all RS modules obtain an unanimous view of the current set of available requests before selecting within this set the request with the highest priority. Note that this solution always blocks the delivery of the next request to the server until an agreement is reached. This is the case even if no local priority inversion has been observed since the time when the last request has been delivered.

RVC adopts almost the same strategy as the one described in the TS solution. However, a consensus between the different RS module is only required when a local priority inversion has been detected. Each time a new ordered request is observed, the RS module verifies its priority with regard to the requests already available. If the new request has a lower priority than the existing ones, it is inserted at the tail of $History$. Otherwise, an agreement is required to ensure that everybody will insert the request at the same place within the history list. During the agreement phase, the consumption of lower priority requests is no more allowed: a part of the history list is locked. On the contrary, higher priority requests can be delivered to the server for processing without delay. The computation of the insertion point takes into account the progress of the fastest replica. All modules will respect the prefix order already chosen by the fastest RS module. At any time, GEP is equal to the LEP of the fastest RS module. Therefore no roll-back is necessary. For all the other RS_q modules $LEP_q \leq GEP$. As an agreement is no more always required before processing a request, the RVC solution improved the TS solution. To guarantee consistent decisions, the RVC solution makes the strong assumption that there is a service available that is able to provide a "view synchronous multicast" and a dynamic group membership.

Both solutions ensure that, for any module RS_p, $LEP_p \leq GEP$. This conservative approach ensures that requests processed by a server replica are always stable and roll-backs are not necessary. Even the request which is currently being processed cannot be preempted: a request is stable as soon as it has been selected for processing. In counterpart, this strategy requires a strong degree of synchrony among replicas, leading to a temporal blocking on the processing of requests while priority inversions are treated. The solution proposed in this work follows an optimistic approach by admitting a free relation between LEP and GEP. Thus, replicas progress freely at their own execution speed. Of course, to avoid a group priority inversion, roll-backs may be required from time to time. Note, however, that the roll-back requirement is a necessary mechanism when one wants to preempt lower priority activities to ensure the predictability of higher priority activities.

The TS and RVC protocols were not designed to the generic client-server model. In fact, the RVC solution is not adapted to this model. When a single crash occurs, the replicas can exhibit a non-uniform behavior which leads clients to observe inconsistent replies. The following situation illustrates this fact. Let us assume that the fastest replica S_p processes a request r, returns the reply to the client and crashes thereafter. Suppose the other replicas S_q are slow ($LEP_q < i$) with $History_q^i = r$, and that a new request r' with a higher priority level is observed by all RS_q modules. To avoid a priority inversion, the RS_q modules will agree to process r' before r. Of course, as the fastest processor p is crashed, it will not participate on the agreement and thus is unable to impose to the others its view of the history. Thus, a client may observe different replies to its request r that will reflect (or not) the processing of a previous request r'.

8 Conclusion

We have proposed the concept of group priority inversions. This new concept extends that of local priority inversions to the context of a system where computation is replicated to tolerate faults. We then presented a request scheduling protocol that is able to provide replicated processes with an ordered stream of input requests in such a way that global priority inversions are avoided. Due to space limitations, proof of correctness of the request scheduling protocol are given in [15].

References

[1] Fetzer C. and Cristian F., The timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 642–657, june 1999.

[2] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *JACM*, 43(1):225–267, March 1996.

[3] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *JACM*, 43(4):685–722, July 1996.

[4] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *JACM*, 32(2):374–382, April 1985.

[5] Hurfin M., Macêdo R., Raynal M., and Tronel F., A General Framework to Solve Agreement Problems *Proc. 18th IEEE Int Symposium on Reliable Distributed Systems (SRDS'99)*, pp. 56-65, Lausanne, 1999.

[6] Hadzilacos V., and Toueg S., Fault tolerant broadcasts and related problems, In Sape J. Mullender editor, *Distributed Systems*, chapter 5, Addison-Wesley, pp 97-145, 1993.

[7] Realtime CORBA 1.0 Revised Submission, OMG TC Document orbos/98-12-05, Dec., 1998.

[8] Fault Tolerant CORBA Specification V1.0, OMG ptc/2000-04-04, April, 2000.

[9] S. Mullender, editor, Distributed systems, 2nd edition, ACM Press NYC, 1993.

[10] Powell D., and Arlat J., and Beus-Dukic L., and Bondavalli A., and Coppola P. and Fantechi A., and Jenn E., and Rabjac C., and Wellings A., Guards: A Generic Upgradable Architecture for Real-Time Dependable Systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(26):204–226, june 1999.

[11] Reisinger J., and Steininger A., and Leber G., Predictably Dependable Computing Systems. In Chapter: The PDCS Implementation of Mars Hardware and Software, 209–224, *Springer-Verlag*, 1995.

[12] Rodrigues L., Verissímo P., Casimiro A., Priority-Based Totally Ordered Multicast. *3rd IFIP/IFAC workshop on Algorithms and Architectures for Real-Time Control (AARTC'95)*, 1995.

[13] Schmidt D., Mungee S., Flores-Gaitan S., Gokhale A., Software Architectures for Reducing Priority and Non-determinism in Real-time Object Request Brokers, *Kluwer Journal of Real-Time Systems*, 1999.

[14] Tully A., Shrivastava S., Preventing State Divergence in Replicated Distributed Programs. *9th Symposium on Reliable Distributed Systems (SRDS'90)*, pp.104-113, 1990.

[15] Wang Y.,Brasileiro F., Anceaume E., Greve F., Hurfin M., Avoiding Priority Inversion on the Processing of Requests by Active Replicated Servers. *Research Report IRISA PI 1388, March 2001*