Group Communication for Self-Aware Distributed Systems

Raimundo José de Araújo Macêdo[1] **and Allan Edgard Silva Freitas**[1,2]

[1]Laboratório de Sistemas Distribuídos (LaSiD) - Departamento de Ciência da Computação
Universidade Federal da Bahia (UFBA) - Campus de Ondina - Salvador - BA - Brazil

[2]Instituto Federal da Bahia (IFBA) - Campus de Salvador - BA - Brazil

macedo@ufba.br, allan@ifba.edu.br

Abstract. *Since the eighties that Group Communication is being proposed as a powerful abstraction to design fault-tolerant distributed applications in a variety of distributed system models, ranging from synchronous, to time-free asynchronous model. Though similar in principles, to date, distinct specifications and implementations have been employed for distinct system models. However, the nature of many modern distributed systems, with dynamic and varied QoS guarantees, imposes new challenges where integration, self-awareness and adaptation to available QoS are common requirements. This paper tackles this challenge by proposing a group communication mechanism capable of handling group communication for self-aware distributed systems whose model properties can vary with time. For example, it can dynamically switch to the asynchronous version when the run-time system can no longer guarantee a timely operation. The protocol is proved correct in this paper and a performance evaluation is presented for distinct QoS scenarios.*

1. Introduction

Since the eighties that Group Communication is being proposed as a powerful abstraction to design fault-tolerant distributed applications in a variety of distributed system models, ranging from synchronous, to time-free asynchronous model. Though similar in principles, to date, distinct specifications and implementations have been employed for distinct system models [Birman 1993, Cristian 1996, Chandra et al. 1996, Chockler et al. 2001, Bessani et al. 2003, Défago et al. 2004]. The choice of a given system model determines the quality-of-service (QoS) to be observed for the related group communication service, such as message delivery and consistent membership view guarantees.

In synchronous systems, message transmission and process execution delays are bounded. This model simplifies the treatment of failures because a process failing to send a message (or processing it) within the delay bound can be considered to have crashed. As a consequence, several problems related to fault-tolerant computing, such as membership, consensus, and atomic broadcast have been solved in such a model [Cristian 1991, Kopetz and Grunsteidl 1994, Cristian et al. 1995].

In an asynchronous system, on the other hand, there is no known bound for message transmission or processing times. This makes the system more portable and less sensitive to operational conditions (for example, long unpredictable transmission times will not affect safety properties of the system). However, problems such as distributed consen-

sus [Fisher et al. 1985][1] and (primary-partition) group membership [Chandra et al. 1996] cannot be solved in this model unless some additional assumptions are considered. Based on the observation that, in practice, most systems (specially those built from off-the-shelf components) behave synchronously, but can have 'unstable' periods during which they behave asynchronously, researches have successfully identified stability conditions necessary to solve fundamental fault-tolerant problems such as *consensus* and *atomic broadcast*[Dolev et al. 1987, Chandra and Toueg 1996]. Such theoretical results allowed the implementation of group services in these so-called partially synchronous environments [Chockler et al. 2001, Défago et al. 2004].

Other researches have considered hybrid systems composed by synchronous and asynchronous parts. This is the case of the TCB, which relies on a synchronous wormhole to implement fault tolerant services [Veríssimo and Casimiro 2002]. Another example is the so-called real-time dependable channels (RTD) that allow an application to customize a channel according to specific QoS requirements [Hiltunen et al. 1999]. Resource reservation and admission control have been used in *QoS architectures* in order to allow processes to dynamically negotiate the quality of service of their communication channels, leading to settings with hybrid characteristics that can change over time [Aurrecoechea et al. 1998]. In this context, we have addressed the problems of consensus over a spanning tree of timely channels [Gorender and Macêdo 2002], perfect failure detection [Macêdo and Gorender 2009] and uniform consensus for hybrid and dynamic distributed systems [Gorender et al. 2007]. To cope with hybrid and dynamic distributed systems we defined non-homogeneous distributed systems where each process or channel can have a distinct QoS (timely or untimely) that can change over time [Macêdo and Gorender 2009]. Moreover, we introduced the notion of self-awareness where the set of system processes is partitioned into three sets (*life, uncertain and down*) that are automatically updated by monitoring mechanisms to reflect the current state of the underlying computing and communication systems.

A challenge not adequately addressed in the existing literature concerns the development of group communication services that can be configured to work with either synchronous or asynchronous environments [Cristian 1996]. Besides simplifying system design, an integrated approach would allow the implementation of group communication for hybrid distributed systems whose characteristics can change over time. This paper tackles this challenge by presenting a generic solution for group communication over self-aware distributed systems. This solution that can be used in any configuration, ranging from synchronous, to asynchronous, to dynamic and hybrid distributed systems - no need to change the group communication algorithms.

Our solution is based on an extension of the so-called Causal Blocks model used for asynchronous systems - a framework for developing group communication protocols and related services with a number of ordering and reliability properties (e.g. ordered message delivery in overlapping groups, flow control etc.) [Macêdo et al. 1995]. Because it combines physical clock time and logical time in the same infrastructure, the extended model (denoted Timed Causal Blocks - *TimedCB*) represents an integrated framework capable of handling group communication for both synchronous and asynchronous dis-

[1]Consensus can be used as a building block to implement membership protocols [Chandra and Toueg 1996].

tributed systems. This is especially relevant to achieve dynamic adaptation (one could switch to the asynchronous version when timely conditions can no longer be met) and fast message delivery (for instance, there is no need to wait for a timing condition when some logical time property is already satisfied within a time window - for example, for timely total ordered message delivery).

The present paper builds on our previous publications [Freitas and Macêdo 2009, Macêdo and Freitas 2009a], by extending the previous results in the following way: the group communication protocol now implements the *uniform agreement on message delivery* property, instead of the *non-uniform* version of [Macêdo and Freitas 2009a]. The new property is very important since it guarantees that the set of delivered messages is the same for any process (crashed or not) of a given group view. This is an important requirement for applications such as distributed commit protocols used in database systems [Pedone et al. 1998]. Another enhancement of the present publication is the evaluation of the protocol in the presence of faults, which forced us to implement the adaptive consensus protocol [Gorender et al. 2007]. Finally, besides simulating a conventional group communication protocol for synchronous systems, this time we have also implemented a conventional group communication protocol for asynchronous systems [Kaashoek and Tanenbaum 1991]. Hence, we are able to compare the generic approach working in the extreme case scenarios (synchronous and asynchronous behavior). A final observation is that the three simulated protocols (the generic, the conventional synchronous and the conventional asynchronous) implement the same *uniform agreement on message delivery* property, which makes the present comparison fairer than the one of the previous publication. The evaluation carried out shows the advantages of our approach and provides insights on how the generic protocol can be adjusted to produce the best trade-off between message delivery delay and overhead.

The remainder of this paper is structured as follows. Section 2 presents the model, system assumptions and group communication properties from which the generic approach is built. The development of *TimedCB* is then presented in section 3. A simulation environment and a performance evaluation of the proposed protocol in a synchronous environment is presented in section 4, and conclusions are drawn in section 5.

2. The System Model and Assumptions

A system consists of a finite set Π of $n > 1$ processes, namely, $\Pi = \{p_1, p_2,...,p_n\}$. Processes communicate and synchronize by sending and receiving messages through channels and every pair of processes (p_i, p_j) is connected by a reliable bidirectional channel: they do not create, alter, or lose messages. In particular, if p_i sends a message to p_j, then if p_i is correct (i.e., it does not crash), eventually p_j receives that message unless it fails. Transmitted messages are received in FIFO order.

A process executes steps (a step is the reception of a message, the sending of a message, each with the corresponding local state change, or a simple local state change), and have access to local hardware clocks with drift rate bounded by ρ. Processes are assumed to fail by prematurely halting execution (i.e. crashing). Byzantines failures are not considered. Processes that do not crash are named correct processes.

It is assumed that the underlying system is capable of providing timely and untimely QoS guarantees for both message transmission and process scheduling times. For

a given timely channel it is known the maximum and minimum bounds for message transmission times, denoted Δ_{max} and Δ_{min}, respectively. For timely processes, there is a known upper bound ϕ for the execution time of a step. For untimely processes and channels, there is no such a known time bound. We assume that $\Delta >> \phi$, so ϕ can be neglected when calculating end-to-end message latencies. It is assumed that the underlying system behavior can change over time, such that processes and channels may alternate their QoS - due to failures and/or QoS renegotiations. It is assumed that the underlying system is equipped with a monitoring mechanism that provides processes with the information about the current QoS ensured for a given channel or process. Besides, we assume the mechanisms described in [Macêdo et al. 2005, Gorender et al. 2007], where dynamic QoS modifications and process crashes lead to the observation of the sub-sets *live*, *uncertain*, and *down*. That is, if $p_j \in live_i$, p_j is timely and p_j is connected to (at least) another timely process p_k (not necessarily k = i), by a bidirectional timely channel (p_j, p_k). Otherwise, $p_j \in uncertain$. If processes that crash were in *live*, they are moved from *live* to *down*. Application processes become aware of dynamic modifications of the distributed systems properties by reading the content of these sets. That is way we named these systems as self-ware distributed systems.

Generic Group Properties Processes form a unique group g, whose initial configuration is $g = \Pi$. Due to space limitations, multiple groups are not considered in this paper. A process p_i of a group g installs views, named $v_i(g) \subseteq \Pi$. A view represents the set of group members that are mutually considered operational. This set can change dynamically on the occurrence of process crashes (suspicions) (or when processes leave or join g - but these events are not considered in this paper). Every time a view change occurs, a new view is installed, and each one is associated with a number that increases monotonically. $v_i^k(g)$ denotes the view number k installed by p_i. Where suitable, the process identity of a view will be omitted (e.g., $v^k(g)$), or the group identity (e.g., v_i^k). A process p_i multicast messages only to the processes of its current view. In general, a group communication protocol must satisfy a number of safety and liveness properties, related to both the views installed by distinct processes and the set of messages delivered. Such properties vary from one implementation to another, following a given target computing environment [Chockler et al. 2001, Cristian 1996]. The group communication suite presented in this paper aims at, among other applications, the implementation of the so-called active replication of servers. Therefore, the properties specified for the presented protocol must satisfy total order message delivery (respecting causality) and agreement on a linear group view history [Schneider 1990, Lamport 1978]. In the following the properties of our generic group communication protocol will be informally presented. A formal and complete description of the protocol properties can be found in [Macêdo 2008].

To achieve message delivery liveness, the **validity** property assures that a correct process will deliver at $t + \Delta_1$ a message sent by it at time t. To achieve message delivery safety, the properties that must be satisfied are **uniform agreement** (i.e. if a process delivers a message in a view, all correct processes must do the same), **uniform total order** and **causal order**, so the processes observe the same message delivery order, respecting potential causality [Lamport 1978]. To assure view delivery liveness, a **failure detection** property guarantees that if a process fails at a time t in a view, all correct processes will detect it at time $t + \Delta_2$ and install a new view that excludes the failed process. For view delivery safety, correct processes must agree on a view, according to the **unique sequence**

of views property. Also, exclusions from a group must be justified by process crashes or suspicions. That is, if a process does not belong to a new view, then either it failed or it was suspected (**exclusion justification**). Finally, a process only installs a new view if it belongs to it (**self-inclusion**). The bounds Δ_1 and Δ_2 are known if the system is synchronous, and unknown, otherwise.

3. The Proposed Generic Approach

The Causal Blocks model is briefly presented in the following (a complete description can be found elsewhere [Macêdo et al. 1995]. Each process maintains a logical clock [Lamport 1978], named Block Counter, denoted BC_i, and messages are sent timestamped with the current block counter value. A process p_i uses a Causal Block to represent concurrent messages, which are sent or received with the same block number. The set of Causal Blocks ordered by their block numbers allow us to construct a Block Matrix BM, as showed in figure 1 to a 6-member process group. It represents all messages sent/received by the process which owns this particular matrix. In the figure, for example, the block-numbers of the last messages received from processes p_1 and p_2, are 4 and 5, respectively.

	P_1	P_2	P_3	P_4	P_5	P_6
1	+			+		
2		+			+	+
3	+		+	+		
4	+				+	
5		+				

Figure 1. The Block Matrix of a 6-member Group Process

Thanks to the FIFO and reliable channel assumptions, once the block matrix in p_i indicates the receipt of a message m with block number $m.b$ from p_j, no other messages from p_j with a smaller block number $b' < m.b$ will be ever received by p_i. Hence, the notion of *block completion* can be built to determine if a given block contains all related messages (i.e. no more messages with the same block number are expected) - said then a complete block. In the example above, blocks 1 and 2 ($BM[1]$ and $BM[2]$) are complete.

The block completion can be used to provide causal and total order delivery. To provide total order, after the completion of the block $BM[B]$ is possible to deliver its messages in a pre-defined order (e.g. according to the sender's unique identifier). In both cases, the delivery should occur in the increasing order of block numbers. To guarantee liveness in block completion, and therefore, in message delivery, each process is provided with a simple mechanism, called the time-silence, which enables a process to remain lively during those periods when it is not generating computational messages. The time-silence mechanism of p_i acts as follows. When a block $BM[B]$ is created at p_i, it sets a timeout ts, after that if the process does not sent a message m to contribute to that completion ($m.b \geq B$), a null message timestamped with the largest locally known block number is sent.

We extend this framework to consider the notion of timely block completion by giving a time upper bound by which a created causal block will be completed, when the system is synchronous [2]. The lemma below specifies these upper bounds.

[2] Actually, it is sufficient that all processes are timely and there exist a spanning tree of timely channels covering all processes. However, for simplifying our presentation this particular case is not considered.

Lemma 3.1. *The time bounds for completion of BM[m.b] at p_i, as measured by its local clock are:*

- *TC1:* $(t_i + ts(m.b) + 2\Delta_{max})(1 + \rho)$*, if m was sent by* p_i
- *TC2:* $(t_i + ts(m.b) + 2\Delta_{max} - \Delta_{min})(1 + \rho)$*, if m was received by* p_i

The Protocol A message m sent to a group reaches all destinations if the sender process does not crash during transmission; in case of crash, some destination processes may not receive m. Hence, when a message is received by a destination process, it can not be immediately discarded as its retransmission may be required to satisfy the *agreement* property; instead, the received message must be stored until it is known that all processes have received it. Messages that have not been acknowledged by all member processes are called unstable messages (stable messages, otherwise)[3]. In the previous version of the protocol [Macêdo and Freitas 2009a], as soon as a message becomes stable, it is then discarded from the local storage. For the present version, we require that a message be super-stable so that the *uniform agreement on message delivery* property can be guaranteed. Therefore, messages are only discarded after being super-stable (being known stable by all processes).

Uniform Agreement on Message Delivery. To determine when a block is stable, group members inform in every transmitted message the last complete block (LCB). By collecting all LCB information available, we can derive the last stable block $LSB = min\{LCB_i\}$, $\forall p_i \in g$. Hence, all blocks of a process p_i with number equal or less to LSB are stable to p_i. In order to assure that group members deliver the same set of messages and in the same order with *uniform agreement on message delivery*, the following conditions must be satisfied, where $m.b$ is the block number of message m:

- *stable-safe1*: a received *m*, is deliverable if *BM[m.b]*is stable;
- *stable-safe2*: deliverable messages are delivered in the non-decreasing order of their block numbers; a fixed pre-determined delivery order is imposed on deliverable messages of equal block number.

To achieve liveness in message delivery, the time-silence mechanism must act even in absence of incomplete blocks. That is, when there is an idle period ts after block completion a null message is sent in order to transmit the LCB information to all processes. In these occasions, time-silence messages should not create new blocks.

Timeouts to Block Stability The upper-bounds for block stability of *BM[m.b]* at p_i, as measured by its local clock are:

Lemma 3.2. *The time bounds for stability of BM[m.b] at p_i, as measured by it:*

- *ST1:* $(t_i + 2ts(m.b) + 3\Delta_{max})(1 + \rho)$*, if m was sent by* p_i
- *ST2:* $(t_i + 2ts(m.b) + 3\Delta_{max} - \Delta_{min})(1 + \rho)$*, if m was received by* p_i

We have also exploited the notion of stable blocks to implement joins and leaves operations. That is, when a process wants to join or leave a group, this information is first propagated to the group. Once this information (joins or leaves) gets stable the new view can be installed.

[3]The interested reader should refer to [Macêdo et al. 1995] for the details on the detection of stable messages in the context of Causal Blocks.

Algorithms Here, we present the algorithms for the uniform agreement version. Algorithm 1 is triggered by a send/receive event of a message m, setting the proper timeouts in case of block creation, storing the message m in a local buffer and triggering the delivery task (Algorithm 2). This task will deliver stable messages according to the delivery conditions. Suppose that a process p_k fails by stop functioning (crashing) and, as a consequence, a timeout expires at p_i for a *BM[m.b]*. In order to proceed with message delivery, a new membership for *g* must be established that excludes p_k (or any other faulty processes). So, to guarantee that all group members engage in the same view installation procedure, a reliable multicast primitive, denoted *rmcast(ChangeViewRequest,B)*, *B = m.b*, is employed to launch the change view procedure (Algorithm 3). This reliable multicast will be received in all functioning processes, which in turn execute the changing view task (Algorithm 4). Finally, this changing view task uses reliable multicast to disseminate the LSB value and the set of unstable messages between the functioning processes. So, it be expected that all unstable messages not yet delivered by the correct processes will be known. Also, the maximum value gathered of LSB (LSB_{max}), allows us to infer if which blocks are stable in a process but are not yet in another. A guard condition is used to wait until a proper quorum of functioning processes be reached. This quorum considers the QoS of processes and channels, based on updated information about the sets *live, down and uncertain.* Once each process collects its own view of the set of unstable messages and of the functioning processes and LSB_{max}, this information will be used with the adaptive consensus presented in [Gorender et al. 2007] to agree on identical views to be installed at all group members (thanks to the *uniform agreement* of consensus). Such a consensus algorithm makes progress despite distinct views of the QoS of the underlying system, adapting to the current QoS available (via the sets *live*, *uncertain*, and *down*). At the end of consensus, the set of unstable messages are stored in local buffer, the LSB is updated and messages are delivered according to message delivery conditions. As the decided view may not include a given p_i (that fails in sending its unstable set), it might be terminated (line 12). Finally, a new view is only installed if some process has been removed from the current view (lines 13-14). Otherwise, the missing messages to complete BM[B] have been recovered and no view change is necessary.

Algorithm 1: Executed by p_i on a *send/receive* event of a message *m*

```
if BM[m.b] does not exist then
   create BM[m.b]
   if p_i = m.sender then
      set completion timeout TC1 for BM[m.b]
      set stability timeout ST1 for BM[m.b]
   else
      set completion timeout TC2 for BM[m.b]
      set stability timeout ST2 for BM[m.b]
   end if
end if
store m at a local buffer and signal delivery task (Algorithm 2)
```

Protocol Correctness To be correct, the protocol must satisfy the properties previously described. In the following, we formalize and prove the *uniform agreement* property

Algorithm 2: Delivery Task

1: **if** any causal block gets stable **then**
2: deliver stable messages according to *stable-safe1* and *stable-safe2*
3: **end if**
4: updates LCB and LSB and cancel related timeouts for complete or stable causal blocks

Algorithm 3: Executed by p_i on the expiration of a timeout for BM[B]

1: *rmcast(ChangeViewRequest,B)*

Algorithm 4: Executed by p_i on the reception of *(ChangeViewRequest,B)*

1: **if** *(unstable,B,LSB)* was already been sent by p_i **then**
2: exit
3: **end if**
4: block ordinary delivery at *delivery task*
5: rmcast(unstable, B,LSB)
6: *wait until* ($\forall p_j \in v_i^k$: received (unstable,B,LSB) from p_j or $p_j \in down_i$ or $FD_i(p_j) = true$) and for majority of $uncertain$: received (unstable,B,LSB) from p_j
7: *let* $allunstable_i$ be the union of the *unstable* sets received from all p_j and LSB_{max} the maximum value of LSB collected.
8: *let* v_i^{k+1} be set of all p_j from which (unstable,B) was received.
9: *consensus(B,*$(v_i^{k+1}, allunstable_i, LSB_{max})$*)*
10: store messages from $allunstable$ not yet received by p_i, sets $LSB = LSB_{max}$ and apply *stable-safe1* and *stable-safe2* to blocks that get stable.
11: **if** $p_i \notin v_i^{k+1}$ **then**
12: *terminate* p_i (* p_i was removed due to a false suspicion from a $p_j, i \neq j$ *)
13: **else if** $v_i^k \neq v_i^{k+1}$ **then**
14: install the decided view v_i^{k+1} at p_i
15: **end if**
16: *signal delivery task* (Algorithm 2) for resuming ordinary message delivery

for message delivery. The proofs for the remaining properties are omitted due to space restrictions, but can be easily derived from the system assumptions, the properties of the causal block framework, and the adaptive consensus.

Theorem 3.1. *(Uniform Agreement): If a process delivers a message m in view v_i^r, every correct process p_j delivers m in view v_j^r.*

Proof

Assume that a message m is sent by p_i in view v_i^r. The message m will be delivered in p_j as soon as $BM[m.b]$ gets stable in p_j. Let us first consider the fault free case. If application messages are not generated to complete $BM[m.b]$, the time-silence mechanism will sent null messages to $BM[m.b]$ and eventually $BM[m.b]$ complete. Similarly, after a timeout of block completion, a null message will be sent to transmit the LCB to all processes. As messages are not lost, all processes will receive the LCB information, so $BM[m.b]$ get stable and its messages delivered.

Suppose now that p_i delivers m and crashes, and as a consequence, m is not delivered in a correct process p_j. If p_i delivered m, it is because $BM[m.b]$ is stable in p_i and therefore complete in all processes (including p_j). Thanks to the timeouts for block stability, this crash will trigger the reliable multicast to execute the change view task. Because of the delivery property of the reliable multicast, this message is received by all processes and the adaptive consensus is eventually executed. Because $BM[m.b]$ is complete in all processes, m will be present in the proposed view of all processes and will, therefore be delivered at all functioning processes before installing the new view, including p_j (thanks to the uniform agreement property of consensus).

Theorem 3.1

Finally, the complete version of this paper, including remaining proofs, is available through our technical report [Macêdo and Freitas 2009b].

4. Simulation and Evaluation

In order to simulate protocols for self-aware distributed systems, it is required that all possible behaviors in such environments can be expressed, including to provide distinct QoS for channels and processes, changes in topology, processes and channels. Because we have not found in the literature a simulation environment with the required characteristics, we had to develop a new one, which was done in Java. By using our simulator[Freitas and Macêdo 2009], named *HDDSS* (after 'hybrid and dynamic distributed system simulator'), one can define a system that can be composed by a mix of different kinds of processes and channels, each of them implementing a distinct fault model, and allowing the change of component behavior dynamically. For instance, one could define a set of processes that communicate to each other by asynchronous channels, but forming a spanning tree of synchronous channels; still, this system could degrade its QoS, so the spanning tree eventually split, changing dynamically the system properties. In *HDDSS*, a fault model is defined according to a chosen probabilistic density function. Moreover, fault models and timeliness properties can be combined in the definition of the behavior of channels and processes. For instance, a given system can be made of a sub-set of correct processes, another sub-set of processes that fail by crashing with certain probability, and, yet, another sub-set of processes that fail by omission with another probability. The same can be applied for channels. For instance, a channel can be reliable and characterized by a Poison density function for message delivery and another one can fail by omission but with deterministic message delivery delay. Furthermore, during the simulation, one can replace a channel between two processes by an instance of another channel class - switching dynamically its behavior. An instance of the main class *Simulator* defines the sets of processes and channels. Processes and channels inherit from the classes *Agent* and *Channel*, respectively. Arbitrary topologies are defined at the beginning of the simulation, and can be changed during its evolution. Due to space restrictions, a more detailed description of the simulation environment is omitted in this paper.

Also, the simulator allows to provide each process with a set of local variables representing sensored information - and these local values can vary over time, according to the dinamicity of the system. This simulation feature allowed us to simulate the behavior of the self-aware distributed system we assumed. running protocols.

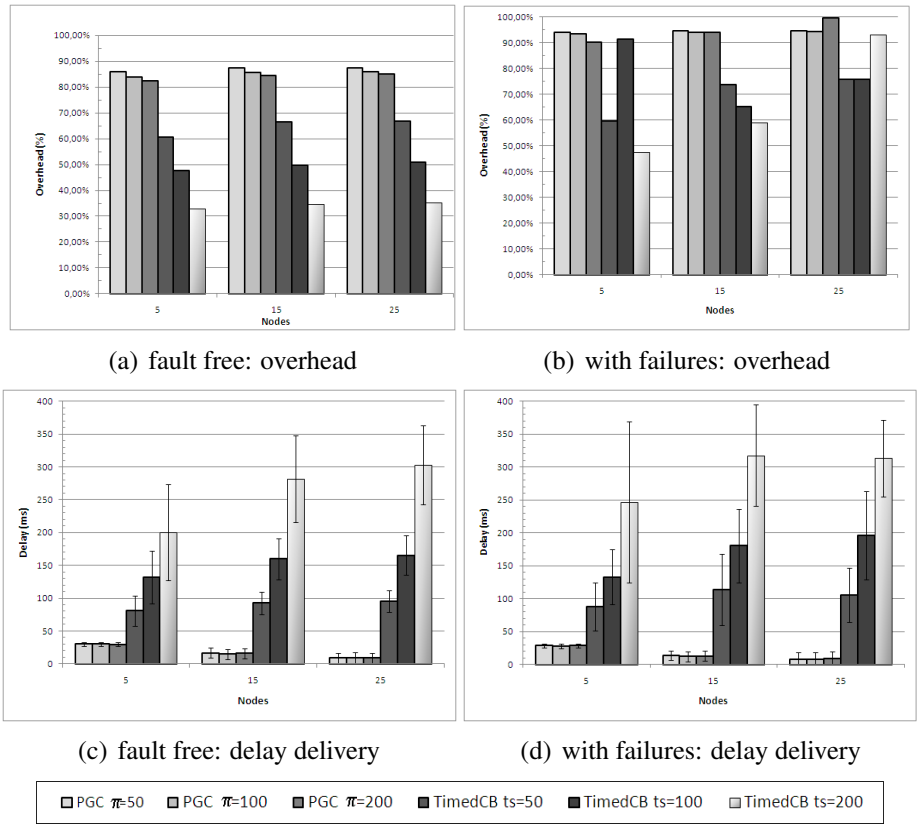
We evaluated the performance of our protocol with *HDDSS* in synchronous and asynchronous cases. This was done by comparison with two classical approaches: Periodic Group Creator [Cristian 1988, Cristian et al. 1995], to a synchronous distributed system, and Amoeba's protocol[Kaashoek and Tanenbaum 1991], to an asynchronous distributed system. For the sake of notational simplicity, they will be named *PGC* and *Amoeba*, respectively, and our protocol, *TimedCB*.

Synchronous case In *PGC*, each process periodically sends a membership checking message (period π). The related atomic broadcast algorithm is based on flooding and delivers messages using synchronized clocks, considering the network maximum delay, the number k of retransmissions and a maximum difference ϵ between the synchronized clocks. In absence of application messages, the *TimedCB* uses the time-silence mechanism to guarantee block completion at each period ts. This allow us to monitor the membership in a similar way to *PGC*. So, in our experiments, we will consider scenarios where $ts = \pi$. For this experiment, we create a synchronous scenario, that could be established in a QoS enhanced switched Ethernet network, where QoS channels are negotiated in a process-to-process basis to provide $\Delta_{MAX} = 15ms$. We assume that $\Delta_{min} = 1ms$ and communication delay is much larger than the processing delay. The network topology is full-connected. We consider that *PGC* is initialized so that its flooding mechanism tolerates just one or two process failures in the sending path ($k = 2$) and is equipped with a clock synchronization algorithm. In *TimedCB*, no synchronization of clocks is needed. The simulation factors considered are the number of network nodes (5, 15 or 25) and the periods ts and π (50, 100 or 200ms). In order to simulate the generation of application messages, each process uses a Bernoulli probabilistic distribution function to decide to send a message. This was adjusted to provide a transmission rate around 10 messages per second. Each protocol is simulated with and without crash failures.

An important evaluation metric is the overhead of the protocol (control messages against the total messages transmitted). Carrying protocol information on application messages, *TimedCB* presents much lower overhead. Even when failures occur and *TimedCB* runs consensus, we can see that it is more effective than *PGC*. For *PGC*, the protocol messages are those generated by the Atomic Broadcast, according to the desired resilience level, and view monitoring *PGC*'s mechanisms. *TimedCB* only generates protocol messages in absence of application messages (time-silence mechanism) or when a failure occurs (consensus). Analyzing the results (figures 2.a and 2.b), it should be noticed that *TimedCB*, as expected, presents, in absence of failures, much less overhead than *PGC*; it can also be observed that increasing the checking period of both protocols (ts to *TimedCB* and π to *PGC*), decreases, significantly, the respective overhead. When failures occurs, the *TimedCB* overhead increases, despite still keeps lower than *PGC*.

An important evaluation metric is the delay for message delivery, here measured from the reception of the message at a local buffer up to the corresponding delivery to the application. Analyzing the results (figures 2.c and 2.d - mean time and the corresponding standard deviation represented as a thin line), PGC for this resilience ($k = 2$) presents a better efficiency than *TimedCB*, with similar behavior in absence or presence of failures. *TimedCB* presents slightly increased delays in presence of failures, due the cost of running consensus to deliver non-stable messages.

However, we observe that *TimedCB* delivery delay can be improved for either



(a) fault free: overhead (b) with failures: overhead

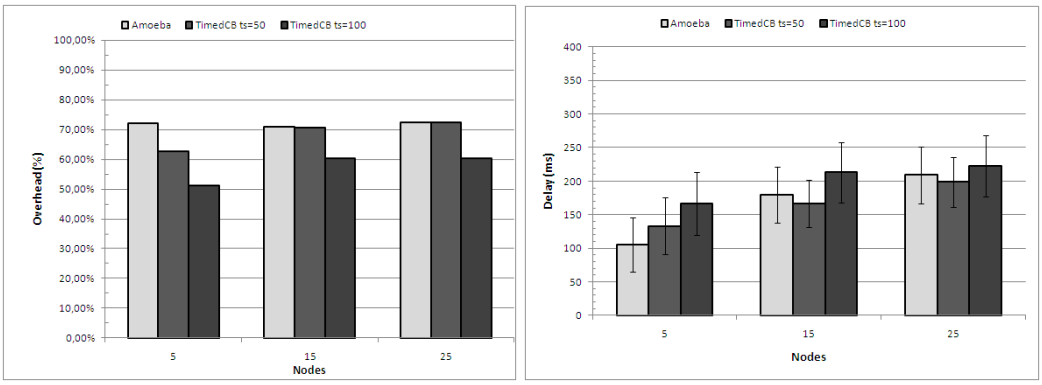
(c) fault free: delay delivery (d) with failures: delay delivery

Figure 2. simulation results in synchronous scenario

smaller checking periods or higher message transmission loads. Moreover, for more resilient atomic broadcasts of *PGC*, the related flooding mechanism will result in much larger delivery delays (for values of $k > 2$). For *TimedCB*, the price is the same (the consensus price), no matter the number of tolerated failures (from 1 to $n - 1$). Hence, the more failures to be tolerated the more advantageous in *TimedCB*.

Another advantage of the *TimedCB* is that, in absence of failures, no additional price will be paid and that is why the protocol presents a low message overhead.

Asynchronous case We simulated an asynchronous scenario without failures and compared our protocol against the uniform-delivery version of the Amoeba's protocol that uses a fix sequencer to produce message ordering. In this protocol, a message is first passed to the sequencer that multicast it to the group. Processes acknowledge to the sequencer that in turn waits for a quorum of acks before sending a control message allowing delivery. We configured asynchronous channels with exponential behavior ($mean = 20ms$). In this simulation, we varied the number of network nodes (5, 15 or 25) and the *TimedCB* period ts (50 or 100ms). The first column of the figure 3 represents Amoeba's data. The second and the third columns represent *TimedCB* for $ts = 50$ and $ts = 100$, respectively. As can be seen in figure 3, *Amoeba* produces a constant overhead and usually larger than *TimedCB*. The delivery delays for bot protocols are similar. are comparable.



(a) protocol overhead (b) delay delivery

Figure 3. simulation results in asynchronous scenario

5. Final Remarks

TimedCB has been introduced to handle group communication in hybrid systems. With *TimedCB*, the same algorithms and information structure can be instantiated in distinct system models (synchronous, asynchronous, or a hybrid system), which simplifies system design. When a pure synchronous system is considered, *TimedCB* can provide early delivery, since logical block completion can be achieved before the pessimistic bounds (obeying known timeouts) hold, and also the expiration of these bounds is an accurate indication of failures. In an asynchronous system, these bounds trigger failure suspicions.

In our experiments, we can see that the cost of consensus for our solution is paid only when crashes occur. Protocols based on atomic broadcast (which is equivalent to consensus [Chandra and Toueg 1996]) will pay an equivalent price for every multicat. For the asynchronous case, asymmetric approaches [Défago et al. 2004] may be more efficient in terms of the number of messages transmitted. However, these will also need extra heartbeat messages to detect failures, whereas in *TimedCB*, failure detection and ordered message delivery are integrated. The presented approach can be particularly relevant for applications that require run-time adaptiveness characteristics, such as those running on networks where previously negotiated QoS cannot always be delivered between processes and in which the overhead should be minimized during failure-free executions.

There are also hybrid, not necessarily dynamic, system settings and applications that can benefit from this new approach. Consider for instance, grid clusters interconnected via the Internet, and that tasks are distributed among distinct clusters (i.e. for parallel computations). Maintaining a mutually consistent view of the functioning processes distributed in distinct clusters is an important requirement, for instance, for re-executing failed tasks when the task coordinator is replicated for improving availability. Such a functionality can be achieved with the presented approach, where each grid cluster forms a synchronous partition. However, as connections among the clusters are realized via TCP/IP, the whole grid system is non-synchronous. We call such hybrid configuration *partitioned synchronous*, and elsewhere we presented an algorithm for the related perfect failure detector [Macêdo and Gorender 2009] - that can be used to manage the *down* set.

At last, we observe that the simulated experiments show that the generality of our solution does not mean a worse performance - since the performance of our protocol can

be compared to well-known solutions for synchronous or asynchronous systems. Hence, a system designer can also choose the generic approach for the sake of a better software engineering practice.

References

Aurrecoechea, C., Campbell, A. T., and Hauw, L. (1998). A survey of qos architectures. *ACM Multimedia Systems Journal*, 6(3):138–151.

Bessani, A. N., Lung, L. C., Alchieri, E., and Fraga, J. (2003). Mjaco: Middleware corba para comunicação de grupo. In *Anais do XXI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2003)*, pages 957–964.

Birman, K. P. (1993). The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53.

Chandra, T. D., Hadzilacos, V., Toueg, S., and Charron-Bost, B. (1996). On the impossibility of group membership. In *Proc. of the 15th annual ACM Symp. on Principles of Distributed Computing*, pages 322–330.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

Chockler, G. V., Keidar, I., and Vitenberg, R. (2001). Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469.

Cristian, F. (1988). Agreeing on who is present and who is absent in a synchronous distributed system. In *Digest of Papers of the 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, pages 206–211.

Cristian, F. (1991). Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–187.

Cristian, F. (1996). Synchronous and asynchronous group communication. *Communications of the ACM*, 39(4):88–97.

Cristian, F., Aghili, H., Strong, R., and Volev, D. (1995). Atomic Broadcast: from simple message diffusion to byzantine agreement. In *Proc. of the 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25)*, pages 200–206.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421.

Dolev, D., Dwork, C., and Stockmeyer, L. (1987). On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97.

Fisher, M. J., Lynch, N., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Freitas, A. E. S. and Macêdo, R. J. A. (2009). Um ambiente para testes e simulações de protocolos confiáveis em sistemas distribuídos híbridos e dinâmicos. In *Anais do XXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2009)*, pages 826–839.

Gorender, S., Macêdo, R. J. A., and Raynal, M. (2007). An adaptive programming model for fault-tolerant distributed computing. *IEEE Trans. on Dependable and Secure Computing*, 4(1):18–31.

Gorender, S. and Macêdo, R. J. d. A. (2002). Um modelo para tolerância a falhas em sistemas distribuídos com qos. In *Anais do XX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2002)*, pages 277–292.

Hiltunen, M. A., Schlichting, R. D., Han, X., Cardozo, M. M., and Das, R. (1999). Real-time dependable channels: Customizing qos attributes for distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 10(6):600–612.

Kaashoek, M. and Tanenbaum, A. (1991). Group communication in the Amoeba distributed operating system. In *Proc. of the 10th Int. Conf. on Distributed Computing Systems (ICDCS 1991)*, pages 222–230.

Kopetz, H. and Grunsteidl, G. (1994). Ttp - a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565.

Macêdo, R. J. A. (2008). Adaptive and dependable group communication. Technical Report 001/2008, Distributed Systems Laboratory (LaSiD) - Federal University of Bahia.

Macêdo, R. J. A., Ezhilchelvan, P., and Shrivastava, S. K. (1995). Flow control schemes for fault tolerant multicast protocols. In *Proc. of the IEEE Pacific Rim Int. Symp. on Fault-Tolerant Systems (PRFTS'95)*, pages 80–88.

Macêdo, R. J. A. and Freitas, A. E. S. (2009a). A generic group communication approach for hybrid distributed systems. *Lecture Notes on Computer Science: Proc. of the 9th IFIP Int. Conf. on Distributed Applications and Interoperable Systems (DAIS 2009)*, 5523(4):102–115.

Macêdo, R. J. A. and Freitas, A. E. S. (2009b). Group communication by using timed causal blocks. Technical Report 001/2009, Distributed Systems Laboratory (LaSiD) - Federal University of Bahia.

Macêdo, R. J. A. and Gorender, S. (2009). Perfect failure detection in the partitioned synchronous distributed system model. In *Proc. of the The 4th Int. Conf. on Availability, Reliability and Security (ARES 2009)*, pages 273–280, to appear in an extended version in Int. Journal of Critical Computer–Based Systems (IJCCBS).

Macêdo, R. J. A., Gorender, S., and Cunha, P. (2005). The implementation of a distributed system model for fault tolerance with qos. In *Proc. of 23rd Brazilian Symp. on Computer Networks and Distributed Systems (SBRC 2005)*, pages 827–840.

Pedone, F., Guerraoui, R., and Schiper, A. (1998). Exploiting atomic broadcast in replicated databases. *Lecture Notes in Computer Science: Proc. of the 4th Int. Euro-Par Conf. on Parallel Processing (Euro-Par'98)*, 1470:513–520.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319.

Veríssimo, P. and Casimiro, A. (2002). The timely computing base model and architecture. *IEEE Trans. on Computers*, 51(8):916–930.