A Generic Group Communication Approach for Hybrid Distributed Systems

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

[1] Distributed Systems Laboratory (LaSiD)
Computer Science Department
Federal University of Bahia
Campus de Ondina, Salvador, Bahia, Brazil
[2] Federal Institute of Bahia
Campus de Salvador, Salvador, Bahia, Brazil
macedo@ufba.br, allan@ifba.edu.br

Abstract. Group Communication is a powerful abstraction that is being widely used to manage consistency problems in a variety of distributed system models, ranging from synchronous, to time-free asynchronous model. Though similar in principles, distinct implementation mechanisms have been employed in the design of group communication for distinct system models. However, the hybrid nature of many modern distributed systems, with dynamic and varied QoS guarantees, has put forward the need for integrated models. Furthermore, adaptation with degraded service is a common requirement in such scenarios. This paper tackles this new challenge by introducing a generic group communication mechanism. Because of its integrated feature, our approach is capable of handling group communication for both synchronous and asynchronous distributed systems, dynamically adapting to the available QoS. For example, it can dynamically switch to the asynchronous version when the run-time system can no longer guarantee a timely operation. The properties and algorithms of the integrated approach are presented in this paper, as well as a performance evaluation through simulation, comparing this mechanism with some classical approaches.

1 Introduction

It is widely recognized that Group communication is a powerful abstraction to design fault-tolerant distributed applications, and the list of publications on the topic is vast [1,2,3,4,5]. Due to the uncertainties inherent to distributed systems, group communication protocols have to face situations where, for instance, a sender process fails when a multicast is underway or where messages arrive in an inconsistent order at different destination processes. Despite such uncertainties, group services should provide application processes with certain message delivery and consistent membership view guarantees. The provisioning of such guarantees depends, however, on the timeliness behavior observed in the underlying run-time and communication systems, ranging from synchronous, to time-free asynchronous systems.

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 [6,7,8].

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 consensus [9][3] and (primary-partition) group membership cannot be solved in this model unless some additional assumptions are considered [3]. Fortunately, in practice, most systems (specially those built from off-the-shelf components) are neither fully synchronous, nor fully asynchronous. Most of the time they behave synchronously, but can have "unstable" periods during which they behave in an anarchic way. That is why many researches have successfully identified distinct stability conditions necessary to solve fundamental fault-tolerant problems [10,11], which allows the implementation of group services in these environments [4,5].

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 [12]. Another example is the so-called real-time dependable channels (RTD) that allow an application to customize a channel according to specific QoS requirements [13]. 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 [14]. In this context, we have addressed the problems of uniform consensus and perfect failure detection for hybrid and dynamic distributed systems [15,16].

A challenge not adequately addressed so far is how to design generic group communication mechanisms suitable to hybrid distributed systems with adaptive characteristics. As it has been pointed out in [2], though similar in principles, distinct strategies have been adopted, depending on the system environment (synchronous or asynchronous). Therefore, the design of integrated group communication schemes, which can work on dynamic hybrid models, remains a challenge. The present paper tackles this challenge by introducing a mechanism called the Timed Causal Blocks (*TimedCB*), which is 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.) [17,18,19]. Because it combines physical clock time and logical time in the same infrastructure, *TimedCB* represents an integrated framework capable of handling group communication for both synchronous and

[3] consensus can be used as a building block to implement membership protocols [10].

asynchronous distributed 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 remainder of this paper is structured as follows. Section 2 presents the system model, assumptions, and message delivery and group view properties of the generic group approach. 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 Generic Group Approach

2.1 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 multicast messages[4] 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.

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 also 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.

According to the QoS related to process executions, the system observed by a process p_i, leads to the identification of sub-sets of Π that share a certain QoS (e.g., a set of processes and channels may form a synchronous component). In

[4] no particular implementation is assumed for message multicast. For example, it can either be implemented by a multi-send operation or by a network level broadcast facility.

particular, dynamic QoS modifications and process crashes lead to the observation of the sub-sets *live*, *uncertain*, and *down*, as defined in [15]. 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*. [5]. These sets are dynamically updated by the above-mentioned monitoring mechanisms and processes can fail only by prematurely halting execution (i.e. crashing). Byzantines failures are not considered. Processes that do not crash are named correct processes.

2.2 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 [4,2]. 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 [20,21]. 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 [22].

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 **agreement** (i.e. if a correct process delivers a message in a view, all correct ones must do the same), **uniform total order** and **causal order**, so the processes observe the same message delivery order, respecting potential causality [21]. 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,

[5] the way these sets are dynamically updated according to the available QoS will not be presented in this paper. Such descriptions can be found in [15].

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 Development of the Generic Approach

The proposed approach is based on the so-called Causal Blocks model that is briefly presented below. A detailed discussion can be found in [17,18,19].

Each process p_i maintains a logical clock called the Block Counter and denoted BC_i, and messages timestamped with Block Counters respect potential causality [21]. A p_i constructs Causal Blocks to represent concurrent messages it sent/received with the same block-number. Construction of Causal Blocks leads to the notion of Block Matrix that can be viewed as a convenient way of representing sent and received messages with different block-numbers.

Figure 1 shows the Block Matrix of a 6-member process group. It represents all messages sent/received by the process which owns this particular matrix. The BM matrix showed in figure 1, indicates, for example, that 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		+				

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

To enable a process to accurately determine that a given block contains all messages that can be represented in it, we use the notion of *block completion*. In the example shown in figure 1, block 2 ($BM[2]$) is complete because processes p_2, p_5, and p_6 have sent a message with block-number 2, and processes p_1, p_3, and p_4 have sent a message with block-number 3. Blocks get completed in the sequential order of their block numbers (thanks to the FIFO order channel delivery property).

A given causal block is guaranteed to complete only if processes in g remain lively by sending messages so that block counter increases with time. To accomplish that, the Causal Blocks model provides each process 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. Briefly explaining, the time-silence mechanism of p_i acts after a time period of inactivity (ts), sending a message to contribute to the completion of all incomplete blocks.

The above block completion definition has been used to implement asynchronous protocols in the Causal Blocks framework. We now introduce the notion of timely block completion, meaning the upper bound time by which a created causal block will be completed, when the system is synchronous [6].

Lemma 1. *The time bounds for a BM[m.b] to complete at a process p_i, as measured by its local clock is (a proof can be found in our technical report [22]) :*

- *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

3.1 The Generic Adaptive 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)[7]. As soon as a message becomes stable, it is then discarded from the local storage. Unstable and/or not delivered messages are kept in the local storage.

In order to assure that group members deliver the same set of messages and in the same order (*agreement* and *total order*, respectively), the following constrictions must be satisfied, where $m.b$ is the block number of message m.

- *safe1*: a received m is deliverable if *BM[m.b]* is complete;
- *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 with the same block number.

Algorithm 1 describes the steps executed every time a new message is sent or received. After creating the related causal block (if it does not exist) and setting its completion timeout, the message is stored in a local buffer. After that, the *delivery* task (Algorithm 2) will be signaled in order to check for delivery conditions in all existing incomplete causal blocks (including the new one)[8]. However, a causal block will become complete as long as processes do not fail in sending messages. Suppose now that a process p_k fails by stop functioning (crashing) and, as a consequence, a block completion timeout (*TC1* or *TC2*)

[6] 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.

[7] the interested reader should refer to [17,18] for the details on the detection of stable messages in the context of Causal Blocks.

[8] Observe that the reception or sending of a message may result in the completion of more than one causal block.

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 processes that did not contribute for the completion of *BM[m.b]*). In order 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). The *agreement property* of the reliable multicast primitive guarantees that if any operational group member delivers the message *(ChangeViewRequest,B)*, all the other operational members will do so. These requests are then processed by the changing view task (Algorithm 4), presented below.

Consider that $V_i^k(g)$ is the current view of g when *TC1* or *TC2* expires for *BM[B]*. Let $F \in \Pi$ be the set of all p_j that failed in sending a message with block-number B or larger (as required by block completion). In other to establish a new view $v_i^{k+1}(g)$ that excludes the processes in F, the adaptive *consensus* presented in [15] is used (line 9 algorithm 4). 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*). However, it is assumed that the system QoS can only degrade during the system execution[9], as required by the adaptive consensus. It is also assumed, as required by the adaptive consensus, the existence of a failure detector of class $\diamond S$[10] (named FD, where FD(p_i) = true if p_i is suspected of crash), and that the majority of processes in the *uncertain* set does not fail. The membership protocol uses repeated (possibly concurrent, but completely independent) executions of consensus where a given execution of consensus is used to decide on identical views to be installed at all group members (thanks to the *uniform agreement* property of the consensus). All the messages related to a tentative view change and a given consensus execution are tagged with the block-number B related to the timeout expiration. Hence, the *consensus* primitive for the completion of *BM[B]* is denoted as *consensus(B, v_i)*, where v_i is the consensus proposed value.

Before installing a new view, the same set of messages must be delivered at operational processes. This is because agreement must be reached not only on the new view v_i^{k+1}, but also on the set of messages delivered in view v_i^k (*agreement* property). Thus, before a view installation, all processes will collect the unstable messages from all operational and non-suspected processes. The union of such unstable messages (lines 5-7 of algorithm 4) together with the identities of the processes that provided these sets (line 8) form the new proposed value for the consensus. Afterwards, messages from *allunstable* are stored in the local buffer of processes that decide the consensus (lines 9-10). 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.

[9] this assumption can actually be relaxed by adding to the consensus proposed value the new upgraded QoS - but this alternative will not be explored in this paper.

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 timeout TC1 for BM[m.b]
   else
      set timeout TC2 for BM[m.b]
   end if
end if
store m at a local buffer and signal delivery task (Algorithm 2)
```

Algorithm 2: Delivery Task

```
1: if any causal block gets completed then
2:    deliver messages according to safe1 and safe2
3:    cancel timeouts of complete causal blocks
4: end if
```

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 a *(ChangeViewRequest,B)* message

```
1: if (unstable,B) was already been sent by p_i then
2:    exit
3: end if
4: block ordinary delivery at delivery task
5: rmcast(unstable, B)
6: wait until (∀p_j ∈ v_i^k: received (unstable,B) from p_j or p_j ∈ down_i or
   FD_i(p_j) = true) and for majority of uncertain: received (unstable,B) from p_j
7: let allunstable_i be the union of the unstable sets received from all p_j
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))
10: store messages from allunstable not yet received by p_i and apply safe1 and safe2
    only for the blocks that get completed with the messages from allunstable_i.
11: if p_i ∉ v_i^{k+1} then
12:    terminate p_i (* p_i was removed due to a false suspicion from a p_j, i ≠ j *)
13: else if v_i^k ≠ 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
```

3.2 Protocol Correctness

To be correct, the generic protocol must satisfy the properties previously described. In the following, we formalize and prove the *validity* property for mes-

sage 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 1. *(Validity): If a correct p_i sends a message m in view $v_i^r(g)$ at real time t, than, provided that it continues to function as a member of g, it will deliver m at time $t + \Delta_1$, $\Delta_1 > 0$, in some view $v_i^s(g), s \geq r$.*

Proof Assume that a correct p_i has sent a message m in view v_i^k. By the reliable channel assumption, m, timestamped with block number $m.b$, is always received at its destinations. Hence, lines 1-2 of Algorithm 1 guarantee that a causal block that contains m is eventually created at all functioning processes, including p_i. If processes do not crash, the *time-silence* mechanism guarantees that created blocks eventually get completed and its messages, including m, delivered (Algorithm 2). Now, suppose that process crashes occur so that *BM[m.b]* will not complete. But, in this latter case, a timeout for *BM[m.b]* (lines 4 and 6 of algorithm 1) will eventually expire and a message for installing a new view will eventually be reliably multicast to all processes (Algorithm 3). The reliable multicast of Algorithm 3 guarantees that all processes will execute the changing view request related to block $m.b$ (Algorithm 4). As the proposed value used in the consensus is carefully constructed to contain all unstable messages from all the members of the proposed view (lines 5 to 8 of algorithm 4), then either messages to complete *BM[m.b]* from all members in v_i^k will be received, or a new view v_i^{k+1} will be established that excludes processes those fail in sending a null (from time-silence) or application message for *BM[m.b]*. In both cases, BM[m.b] will eventually complete and m delivered (thanks to the *termination* property of consensus that guarantees all correct p_i eventually decide).

Theorem 1

4 Simulation and Evaluation

In order to simulate protocols for dynamic and hybrid distributed systems, it is required that all possible behaviors in such environments can be expressed, including 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, 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.

HDDSS is equipped with a serial execution engine of discrete-events, where tasks can be scheduled by time or by events. The processes are Java threads that invoke methods of a package of the message-passing environment - so it could be easily replaced, for instance by Unix sockets, in a real execution.

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.

We simulated our protocol with *HDDSS* and evaluated its performance in a fault-free synchronous system - this is done by comparison with the protocols proposed in [23,6], which altogether perform group communication by membership management and atomic broadcast in a synchronous distributed system. For the sake of simplicity, these protocols are referred here just by the name of the membership protocol, named Periodic Group Creator or *PGC*. Our adaptive generic protocol will be named *TimedCB*.

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, a node executes a processing step in at most 1 time unit and the system is made of deterministic communication channels with $\Delta_{MAX} =$ 14 time units and $\Delta_{MIN} = 10$ time units - so the 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 process failure in the sending path ($k = 1$). With this initialization, we obtain the minimum cost for the flooding mechanism. For comparison, the resiliency level is also adjusted to support two failures ($k = 2$). Also, the synchronization algorithm used in *PGC* is the simple algorithm presented in [24] modified to be aware of clock drifting, considering $\epsilon = 4$ time units. In *TimedCB*, no synchronization of clocks is needed.

The simulation factors considered are the number of network nodes (10, 30 or 50) and the periods ts and π (16, 24 or 32). In order to simulate the generation of application messages, each process is defined to send an application message at each advance of a time unit according to a Bernoulli probabilistic distribution function. To calculate averages, each experiment was replicated 5 times and had a time window of 500 time units. Varying the probability P of generating a message in a time unit, we run two scenarios, named A and B. The first one is characterized by the probability $p = 0.1$, resulting in an average of 517, 1526 and 2516 application messages, when 10, 30 and 50 nodes were considered, respectively. In the scenario B, the sending load is characterized by $P = 0.02$ and the corresponding averages were 105, 314 and 517. Observe that in scenario A message transmission load is roughly 5 times larger than scenario B. So, with these scenarios we tried to simulate distinct message transmission loads (high and low, respectively).

An important evaluation metric is the proportion of protocol messages against the total messages transmitted - including the application messages. That is, the overhead of the protocol. The protocol messages for *TimedCB* are the ones generated by algorithms 3 and 4 and time-silence. For *PGC*, the protocol messages are those generated by the Atomic Broadcast and *PGC*'s view monitoring mechanisms. Another important 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. Tables 1 and 2 refer to these metrics in *PGC* and *TimedCB*, respectively. The figures are the mean time (and corresponding standard deviation) for scenarios A and B.

Analyzing the results, it should be noticed that *TimedCB*, as expected, presents much less overhead than *PGC*: for instance, in scenario A (table 1), *TimedCB* overhead in the worst case (shorter checking period and greater number of nodes) is 26.82 %, against 62.81 % of *PGC* in same conditions (the best case for *PGC* is 36.72 %). One of the reasons is the fact that *PGC* requires the periodic transmission of protocol messages to monitor membership and *TimedCB* uses application messages to carry (when possible) protocol messages. 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.

Also, it should be noticed that, even in lower load scenarios (like scenario B in table 2), where there are few application messages to be used, *TimedCB* takes advantage of them, presenting always lower overhead than *PGC*. On the other hand, observe that, because *PGC* relies on synchronized clocks, it achieves a more regular delivery delay than *TimedCB*.

We observe also that *TimedCB* delivery delay can be improved for either smaller checking periods or higher message transmission loads. Another advantage of the *TimedCB* is that, in absence of failures, no additional price will be paid. For more resilient atomic broadcasts of *PGC*, the related flooding mechanism will result in much larger delivery delays (for values of $k > 1$). For *TimedCB*, if failures are considered, the price is the same (the consensus price), no matter the number of tolerated failures (from 1 to $n - 1$).

Table 1. Simulation results of scenario A

Factors		Delivery delay			Overhead (%)	
n	π or ts	PGC (k=1)	PGC (k=2)	TimedCB	PGC	TimedCB
10	16	17.696 ± 3.120	31.482 ± 4.828	21.933 ± 8.117	46.54%	21.79%
10	24	17.523 ± 3.494	31.451 ± 4.630	25.731 ± 9.238	40.37%	15.11%
10	32	17.794 ± 3.298	31.369 ± 4.462	30.683 ± 10.682	36.72%	11.62%
30	16	17.630 ± 3.996	30.616 ± 6.550	19.853 ± 6.444	56.10%	26.14%
30	24	17.660 ± 3.929	30.632 ± 6.744	24.900 ± 8.074	51.95%	18.92%
30	32	17.706 ± 3.694	30.772 ± 6.643	32.405 ± 9.150	49.57%	14.84%
50	16	17.795 ± 3.999	30.392 ± 7.478	19.799 ± 6.079	62.81%	26.82%
50	24	17.849 ± 3.787	30.504 ± 7.453	25.845 ± 7.710	59.85%	20.28%
50	32	17.870 ± 3.948	30.594 ± 7.086	34.161 ± 9.023	58.18%	15.66%

Table 2. Simulation results of scenario B

Factors		Delivery delay			Overhead (%)	
n	π or ts	PGC (k=1)	PGC (k=2)	TimedCB	PGC	TimedCB
10	16	16.659 ± 5.477	28.266 ± 10.492	20.865 ± 9.819	81.08%	71.07%
10	24	17.272 ± 4.124	28.052 ± 10.208	30.382 ± 11.396	76.92%	61.68%
10	32	17.405 ± 4.517	27.904 ± 10.671	38.960 ± 13.270	74.07%	51.83%
30	16	17.338 ± 4.536	30.328 ± 7.118	22.748 ± 7.606	86.13%	71.74%
30	24	17.112 ± 4.507	30.353 ± 7.514	30.757 ± 9.669	84.01%	61.94%
30	32	17.431 ± 4.416	30.420 ± 6.924	39.105 ± 11.043	82.69%	53.76%
50	16	17.301 ± 4.707	30.086 ± 7.943	24.307 ± 6.584	89.15%	71.79%
50	24	17.367 ± 4.789	30.024 ± 8.243	30.842 ± 8.832	87.88%	62.78%
50	32	17.313 ± 4.650	29.913 ± 8.232	40.182 ± 9.727	87.13%	55.04%

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 ($TC1$ or $TC2$) hold, and also the expiration of these bounds is an accurate indication of failures. When an asynchronous system is considered, these bounds trigger failure suspicions.

In the simulated experiments, if we consider the runs without failures and the synchronous scenario, *TimedCB* produces lower message transmission overhead when compared with classical approaches such as in [8], where messages are sent using an atomic broadcast primitive (which is equivalent to consensus as proved in [10]). In the approach presented in this paper, the cost of consensus is paid only when crashes occur. However, the execution of consensus impact the worst case time delay for message delivery. Thus, the advantages of our protocol will be more observed in scenarios less bounded to failures.

For the asynchronous case, asymmetric approaches [5] 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 number of extra transmitted messages 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 (for instance, 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 [16] - that can be used to manage the *down* set (algorithm 4).

At last, we have not assumed a specific implementation for the underlying QoS provision and monitoring systems. This can be carried out, for instance, by using hybrid real-time systems, where there are guaranteed deadlines for critical tasks and best-effort response times for aperiodic or non-critical tasks (for instance, by using aperiodic servers [25]). The same kind of solution can be used in the network level that must be a hybrid real-time network. Another possible approach is to use QoS architecture solutions [26], and we have developed a prototype implementation with this purpose [15].

References

1. Birman, K.P.: The process group approach to reliable distributed computing. Communications of the ACM **36**(12) (December 1993) 37–53
2. Cristian, F.: Synchronous and asynchronous group communication. Communications of the ACM **39**(4) (April 1996) 88–97
3. Chandra, T.D., Hadzilacos, V., Toueg, S., Charron-Bost, B.: On the impossibility of group membership. In: Proc. of the 15th annual ACM Symposium on Principles of Distributed Computing, New York, NY, USA, ACM Press (1996) 322–330
4. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Computing Surveys **33**(4) (December 2001) 427–469
5. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys **36**(4) (December 2004) 372–421
6. Cristian, F., Aghili, H., Strong, R., Volev, D.: Atomic Broadcast: from simple message diffusion to byzantine agreement. In: Proc. of the 25th International Symposium on Fault-Tolerant Computing, IEEE CS Press (June 1995)

7. Kopetz, H., Grunsteidl, G.: Ttp - a protocol for fault-tolerant real-time systems. IEEE Computer **27**(1) (January 1994) 14–23
8. Cristian, F.: Reaching agreement on processor-group membership in synchronous distributed systems. Distributed Computing (4) (April 1991) 175–187
9. Fisher, M.J., Lynch, N., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32**(2) (April 1985) 374–382
10. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (March 1996) 225–267
11. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM **34**(1) (January 1987) 77–97
12. Veríssimo, P., Casimiro, A.: The timely computing base model and architecture. IEEE Trans. on Computers **51**(8) (August 2002) 916–930
13. Hiltunen, M.A., Schlichting, R.D., Han, X., Cardozo, M.M., Das, R.: Real-time dependable channels: Customizing qos attributes for distributed systems. IEEE Trans. on Parallel and Distributed Systems **10**(6) (June 1999) 600–612
14. Aurrecoechea, C., Campbell, A.T., Hauw, L.: A survey of qos architectures. ACM Multimedia Systems Journal **6**(3) (May 1998) 138–151
15. Gorender, S., Macêdo, R.J.A., Raynal, M.: An adaptive programming model for fault-tolerant distributed computing. IEEE Trans. on Dependable and Secure Computing **4**(1) (January 2007) 18–31
16. Macêdo, R., Gorender, S.: Perfect failure detection in the partitioned synchronous distributed system model. In: Proc. of the The 4th International Conference on Availability, Reliability and Security (ARES 2009), IEEE CS Press (March 2009) 273–280
17. Macêdo, R.J.A.: Fault-tolerant group communication protocols for asynchronous systems. In: Ph.D. Thesis, Department of Computing Science, U. of Newcastle upon Tyne. (1994)
18. Macêdo, R.J.A., Ezhilchelvan, P., Shrivastava, S.K.: Flow control schemes for fault tolerant multicast protocols. In: Proc. of the IEEE Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS'95). (1995)
19. Ezhilchelvan, P., Macêdo, R.J.A., Shrivastava, S.: Newtop: a fault-tolerant group communication protocol. In: Proc. of the 15th IEEE Int. Conf. on Distributed Computing Systems (ICDCS'95). (1995) 296–306
20. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys **22**(4) (December 1990) 299–319
21. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of ACM **21**(7) (July 1978) 558–565
22. Macêdo, R.J.A.: Adaptive and dependable group communication. Technical Report 001/2008, Distributed Systems Laboratory (LaSiD) - Federal University of Bahia, Salvador, Brazil (December 2008)
23. Cristian, F., Center, I., San Jose, C.: Agreeing on who is present and who is absent in a synchronousdistributed system. In: Digest of Papers of the 18th International Symposium on Fault-Tolerant Computing (FTCS-18), IEEE CS Press (1988) 206–211
24. Lundelius, J., Lynch, N.: Upper and lower bound for clock synchronization. Information and Control **62**(2) (1984) 190–204
25. Lehoczky, J.P., Sha, L., Strosnider, J.: Enhanced aperiodic responsiveness in hard real-time environment. In: Proc. of the 8th IEEE Real-Time Systems Symposium (RTSS'87), San Jose, California, IEEE CS Press (1987) 110–123
26. Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., Weiss, W.: An architecture for differentiated services. RFC 2475 (December 1998)