Adaptive and Dependable Group Communication

Raimundo José de Araújo Macêdo
Distributed Systems Laboratory (LaSiD)
Computer Science Department
Federal University of Bahia
Campus de Ondina, Salvador, Bahia, Brazil
macedo@ufba.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 (real-time) 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 called the Timed Causal Blocks. Because of its integrated feature, the Timed Causal Blocks mechanism 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. Formal properties of the integrated model and related mechanisms, with proof sketches are presented.

Keywords
group communication, fault tolerance, hybrid systems

1. INTRODUCTION
Group communication is a powerful abstraction that can be used whenever groups of distributed processes cooperate for the execution of a given task [3, 13, 1, 7]. In group communication, processes communicate in a group basis where a message is sent to a group of processes. With a group is usually associated a name to which application processes will refer, making transparent the location of the distributed processes forming the group. Due to the uncertainties inherent to distributed systems (emerging from communication or process failures), 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. Moreover, a consistent view of the set of functioning processes is fundamental for some applications. For instance, when server replication is employed to attain high dependability of services, surviving server replicas must take over the responsibilities of failed ones. Therefore, a central problem that has to be solved in such a context is to maintain consistent the server replica states and the views that distinct replicas have about the set of functioning ones. Group Communication is being widely used to manage such problems in a variety distributed system models, ranging from synchronous, to time-free asynchronous model.

In synchronous systems, message transmission and process execution delays are bounded, and, in most cases, these bounds are assumed to be known in advance. Synchronous model is natural for hard real-time distributed systems, since it guarantees bounded reaction time for events. This model assumption simplifies the treatment of failures since a process failing to send a message (or processing it) within the delay bound can be considered to have failed. As a consequence, several problems related to fault-tolerant computing, such as membership, consensus, and atomic broadcast have been solved in such a model. The price that has to be paid is the real-time scheduling and hardware redundancy techniques that need to be used to guarantee that such a bounded time assumption is achieved with high probability [17, 6, 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, it is well known that some fundamental fault-tolerant problems have no deterministic solution in such a model (e.g., the consensus problem [14]). In practice, however, 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 [4, 12]. Other researches have considered hybrid systems composed by a synchronous and an asynchronous part. So, we can regard such systems as being hybrid in the space dimension. This is the case of the TCB model, which relies on a synchronous wormhole to implement fault
tolerant services [20]. The concept of real-time dependable channels (RTD) that has been introduced in [16] to handle QoS adaptation at channel creation time is another example. In their paper, the authors show how an application can customize a RTD channel according to specific QoS requirements, such as bounded delivery time, reliable delivery, message ordering, and jitter (variance in message transmission time). Such QoS requirements are stated in terms of the probabilities and are enforced by a composite protocol based on the real-time version of the Cactus system. 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 [2].

Contributions of this paper. 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 [7], though similar in principles, distinct strategies have been adopted to group communication, depending on the system environment (synchronous or asynchronous) and application requirements (timely message delivery, total and causal message ordering, etc). Therefore, the design of integrated group communication schemes, which can work on dynamic hybrid models, remains a challenge in the area. 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. Causal Blocks is a framework for developing group communication protocols and related services with a number of ordering and reliability properties, such as ordered message delivery in overlapping groups, flow control, etc. [9, 10, 13]. Because they combine physical clock time and logical time in the same infrastructure, TimedCB represents an integrated framework capable of handling group communication for both synchronous hard real-time and asynchronous soft real-time 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 remaining of this paper is structured as follows. Section 2 presents the model and assumptions from which the generic approach is built. Section 3 presents the message delivery and group view properties that must be satisfied by a generic group communication system. The development of TimedCB is then 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, Π = \{p₁, p₂,...,pₙ\}. Processes communicate and synchronize by sending and receiving multicast messages through channels and every pair of processes (pᵢ, pⱼ) is connected by a reliable bidirectional channel: they do not create, alter, or lose messages. In particular, if pᵢ sends a message to pⱼ, then if pᵢ is correct (i.e., it does not crash), eventually pⱼ 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 ρ.

It is assumed that the underlying system is capable of providing timely and untimely QoS guarantees for both message diffusion and process scheduling times. For a given timely channel it is known the maximum and minimum bounds for message transmission times, denoted Δₘₐₓ and Δₘᵢₙ, respectively. Similarly, for timely processes there is a known time bound φ for the execution of a step. For untimely processes and channels, there is no such a known time bound. We assume that Δ > φ, so φ can be neglected when calculating end-to-end message latencies. That is, the time to transmit messages is much larger than the time to process them, so we can ignore φ for the sake of message delivery time calculations. It is assumed that the underlying system behavior can change over time, such that processes and channels may alternate their QoS. This may happen due to QoS renegotiations and/or failures. It is also assumed that the underlying system is equipped with a monitoring mechanism that provides processes with the information of the current QoS guarantee ensured for a given channel or process (the change of state can be informed within some time lag and two distinct processes are not required to have the same view of the QoS related to a third process or channel at a given time). According to the QoS related to process executions, the system observed by a process pᵢ leads to the identification of sub-sets of Π that share a certain quality-of-service (e.g., a set of processes and channels may form a synchronous component). In particular, dynamic QoS modifications of QoS and occurrences of process crashes lead to the observation of the sub-sets of P, live, uncertain, and down, as defined in [15]. That is, if pᵢ ∈ live, pᵢ is timely and pᵢ is connected to (at least) another timely process pⱼ (not necessarily k = i), by a bidirectional timely channel (pᵢ, pⱼ). Otherwise, pᵢ ∈ uncertain. If processes that crash are in live, they are moved from live to down. 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 tolerated. Processes that do not crash are named correct processes. Otherwise, they are named faulty processes.

3. GENERIC GROUP PROPERTIES

Processes form a unique group g, whose initial configuration is Π. That is, g = \{p₁, p₂,...,pₙ\}. Due to space limitations, multiple groups are not considered in this paper. In a group g four operations are defined: join(g): issued by a process when it wants to join group g; leave(g): issued by a process when it wants to leave group g; send(g, m): issued by a process when it wants to multicast a message to the mem-

\[^2\]the way these sets are dynamically updated according to the available QoS will not be presented in this paper due to lack of space. The interested reader can find such descriptions in [15].
bers of group \( g \); \( \text{deliver}(g, m) \): issued by a process to receive a message multicast from a group \( g \). A process \( p_i \) of a group \( g \) also installs views, named \( v_i(g) \subseteq \Pi \). A view represents the set of group members that are mutually considered operational in a given instant of the group existence. This set can change dynamically on the occurrence of process crashes (suspicions) or when processes deliberately leave or join \( g \). Every time a change occurs in the group view, a new view is installed by a group membership protocol. Each view installed by a process is associated with a number that increases monotonically with group view installations. \( v_i^k(g) \) denotes the view number \( k \) installed by \( p_i \). Where suitable (i.e., there is no ambiguity), the process identity of a view will be omitted (e.g., \( v^k(g) \)), or the group identity (e.g., \( v^k \)).

When a group \( g \) is created, every group member \( p_i \) installs the same initial view \( v_i^1 = \Pi \). Any subsequent modifications on the configuration of the group will result in new views being installed, forming the sequence \( v_i^1, v_i^2, \ldots, v_i^k \) where \( k \) represents a given moment on the view evolvement history. 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 and applications [11, 5, 7]. 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 the stringent properties of total order message delivery (respecting causality) and agreement on a linear group view history [19, 18]. Such message delivery properties are formalized below.

**Message Liveness Property**

- **Validity**: If a correct \( p_i \) sends a message \( m \) in view \( v_i^s(g) \) at real time \( t \), then, 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 \( v_i^r(g), s \geq r \).

A message sent in a view will be delivered (possibly) in a future view that reflects (possibly) new view configurations due to crashes, leaves or joins. \( \Delta_1 \) is known \textit{a priori} when all processes in view \( v_i^s(g) \) are timely and mutually connected by timely channels (i.e., they form a synchronous distributed system), and its value can be calculated from the system model (this is an important requirement for real-time systems). Otherwise, the system is asynchronous and \( \Delta_1 \) is an arbitrary positive value, but its value cannot be known \textit{a priori}.

**Message Safety Properties**

- **Agreement**: If a correct process delivers a message \( m \) in view \( v_i^r \), then all correct processes \( p_j \) deliver \( m \) in view \( v_j^s \).

- **Uniform Integrity**: For any message \( m \), every process delivers \( m \) at most once, and only if \( m \) was previously sent.

- **Causal Order**: If a process sends message \( m' \) after it has sent or delivered message \( m \), then all processes delivers \( m' \) only after they have delivered \( m \).

- **Uniform Total Order**: If processes \( p \) and \( q \) both deliver messages \( m \) and \( m' \), then \( p \) delivers \( m \) before \( m' \) if and only if \( q \) delivers \( m \) before \( m' \).

For static process groups (where changes on the group membership are not considered), an Atomic Broadcast primitive [8] satisfies all but the Causal Order property. However, Causal Order is necessary to implement active replication of servers [18, 19]. Moreover, dynamic membership is required in order maintain the specified number of functioning replicas.

Our specification of message delivery properties implies that whenever membership changes occur, messages will be delivered in the context of the new views being installed. In some applications (for instance, load-balancing between distributed processes), the knowledge of the current membership is an important decision-making factor when messages are being delivered. Therefore, membership properties must be specified that allow applications processes to make the best decisions, respecting a known time bound in the case of synchronous distributed systems. Due to space limitations, it will be shown only the view delivery properties related to process crashes (membership changes due to leaves and joins are similar). Such view delivery properties are specified in the following.

**View Liveness Property**

- **Failure Detection**: if a process \( p_j \in v_i^s(g) \) crashes at time \( t \), then all correct processes will install a new view \( v^s(g), s > r \), where \( p_j \notin v^s(g) \), at time \( t + \Delta_2, \Delta_2 > 0 \).

The failure detection property states that a new view will be installed by correct processes whenever a process crash occurs. As for \( \Delta_1, \Delta_2 \) can only be calculated when the system is synchronous. When the system is asynchronous, \( \Delta_2 \) is a positive, but arbitrary value.

**View Safety Properties**

- **Self-Inclusion**: if a process \( p_i \) installs a view \( v_i^k(g) \), then \( p_i \in v_i^k(g) \).

- **Exclusion Justification**: if a process \( p_i \in v_i^{k-1}(g) \) \( v_i^k(g) \), \( k > 1 \), then either \( p_i \) has crashed or it has been suspected of crashing by some group member.

\(^{\text{a}}\)note that in case of asynchronous systems, a failure suspicion may also lead to the installation of a new view that excludes the suspected process.
The self-inclusion property states that only the members of a group view install the corresponding view, and the exclusion justification property states that modifications on the group view are justified only by crashes or crash suspicions.

- **Unique Sequence of Views:** Let $v^k_i(g)$ and $v^k_p(g)$ be the view of number $k$ installed by $p_i$ and $p_j$, respectively. Then, $v^k_i(g)$ is necessarily equal to $v^k_p(g)$. In other words, $\forall k, i, j$: if $p_i$ and $p_j$ install views $v^k_i(g)$ and $v^k_p(g)$, then $v^k_i(g) = v^k_p(g)$.

This property states that distinct group members always install a mutually consistent view history.

## 4. DEVELOPMENT OF THE GENERIC APPROACH

As mentioned earlier, the implementation of the generic approach is based on the so-called Causal Blocks for time-free distributed systems. Thus, before showing how the generic approach is implemented for the hybrid environment, the Causal Blocks approach is briefly presented in the next section.

### 4.1 Overview of the Causal Blocks Model

Each process $p_i$ maintains a logical clock called the Block Counter and denoted as $BC_i$. $BC_i$ is an integer variable and its value increases monotonically. When a group $g$ is created, the $BC_i$ of every $p_i$ is initialized to any non-negative integer, and without loss of generality, it is supposed that they are all initialized to zero. Transmitted messages are timestamped with Block Counters, and, as it is the case in Lamport’s Logical Clock [18], timestamping using Block Counters will respect potential causality (i.e., the happen-before relation denoted by $\rightarrow$). That is, if $m \rightarrow m'$, then the timestamp associated with $m'$ will be larger than the timestamp associated with $m$. However, the reverse will not always hold.

If two messages have different timestamp values, they may be concurrent. Unlike Lamport’s Logical Clock that is advanced on send/receive events, Block Counters advances on send/delivery events, since a transmitted message can only be causally dependent on previously delivered ones. Just before a process $p_i$ sends a message to a group $m$, it advances $BC_i$ by one. The contents of the incremented $BC_i$ is assigned to $m$ as its block-number in the message field $m.b$. $BC_i$ may also be advanced by $p_i$ on the delivery of $m$ (denoted $delivery_i(m)$) if the current value of $BC_i$ is less than $m.b$. Thus, the two events under which $BC_i$ may be advanced are:

- **CA1** (Counter advances during $send_i(m)$): Before $p_i$ multicasts $m$, it increments $BC_i$ by one, and assigns the incremented value to $m.b$; and,

- **CA2** (Counter advances during $delivery_i(m)$): Before $p_i$ delivers $m$, it sets $BC_i = \max\{BC_i, m.b\}$.

Based on **CA1** and **CA2** it can be stated the three following properties of group messages, where $send_i$ denotes a send event executed by a process $p_i$.

- **pr1**: $send_i(m) \rightarrow send_i(m') \Rightarrow m.b < m'.b$

- **pr2**: $\forall m, p_j : deliver_j(m) \rightarrow send_j(m') \Rightarrow m.b < m'.b$; and,

- **pr3**: $\forall m, m' : m'.b = m''b \Rightarrow m'$ and $m''$ are concurrent.

The properties **pr1** and **pr2** follow directly from **CA1** and **CA2** respectively. Together they imply that for any distinct $m, m'$: $send(m) \rightarrow send(m') \Rightarrow m.b < m'.b$. The property **pr3** states that distinct messages multicast with the same block-number are necessarily concurrent and these messages must have been multicast by distinct processes, as **CA1** forbids two send events to occur in a given process with the same value of $BC$.

#### 4.1.1 Causal Blocks and the Block Matrix

Using **pr3**, a process $p_i$ constructs Causal Blocks to represent concurrent messages it sent/received with the same block-number for a given group $g$. 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. A Causal Block is a vector of size equal to the number $n$ of $g$ members. Whenever a process $p_i$ receives or sends a multicast message with a new block-number, say $B$, it creates an empty vector of length $n$; for any message multicast with block-number $B$, it sets the $i_{th}$ entry of the vector to ‘+’; and, for any multicast message received with block-number $B$ from another process $p_j, j \neq i$, it sets the $j_{th}$ entry of the vector to ‘+’. Causal Blocks, maintained by a process in this way, will have the following property:

- **PR1**: in a given Causal Block, only concurrent messages are represented.

When group communication is active, i.e. member processes continually send multicast messages, the number of Causal Blocks constructed will grow. Causal Blocks maintained by a process are arranged in the increasing order of the message block-number they represent, giving rise to a matrix that is called the Block Matrix and denoted as $BM$. Thus, $BM[B]$ will represent the Causal Block for message block-number $B$. Referring to $BM$ of any member process in $g$, we can state another property of Causal Blocks, stated below.

- **PR2**: if $m$ and $m'; m'$ represented in $BM[B]$ and $BM[B']$, respectively, are causally related such that $m \rightarrow m'$, then $B < B'$. 

![Figure 1: The Block Matrix of a 6-member Group Process](image-url)
same in a given moment of physical time. 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.

### 4.2 Block Completion and Message Delivery

By **PR1**, all messages that belong to a given Causal Block, say $BM[B]$, are concurrent. From **PR2**, any causal order message delivery based solely on block numbers will require that a received message represented in $BM[B]$, $B > 1$, be delivered only after all multicast messages which can be represented in $BM[B']$, for all $B' < B$, have been delivered. To enable a member process to accurately determine that a given block completely represents all messages which can be represented in it, we use the notion of block completion.

In the example shown in figure 1, block 2 is complete because processes $p_2$, $p_3$, and $p_4$ have sent a message with block-number 2, and processes $p_1$, $p_4$, and $p_6$ have sent a message with block-number 3. In the following it is defined the precise condition for block completion.

**Lemma 1.** A Causal Block $BM[B]$, $B \geq 1$, maintained by a process $p_i$, is complete, if and only if for all $1 \leq j \leq n$, the following condition hold:

- $LC1$: The $j^{th}$ entry of $BM[B]$ either (i) has ‘+’ or (ii) is blank and there exists $B_i$, $B' > B$, such that the $j^{th}$ entry of $BM[B']$ has ‘+’.

**Proof** Suppose that $LC1$ is satisfied for causal block $BM[B]$, $\forall j$, $1 \leq j \leq n$. By (i), it is ensured that $p_i$ has sent a message for $BM[B]$. Because the conditions $CA1$ and $CA2$ guarantee that a process will always multicast messages with monotonically increasing block-numbers, and by assumption, transmitted messages are received in the sequential order of their block-numbers (FIFO assumption), then, by (ii), it is ensured that $p_j$ has not (and will not) multicast any message for $BM[B]$. Therefore, causal block $BM[B]$ is complete. □

From **PR2** it is obvious that the existence of $BM[B]$, $B > 1$, also implies the existence of some Causal Block $BM[B']$, for some $B' < B$ ($B' = B - 1$ if Block Counters are incremented by 1). Since messages are assumed to be transmitted in FIFO order, Causal Blocks in the $BM$ maintained by a process will complete in the sequentially increasing order of block-numbers they represent.

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: a necessary condition to deliver ordered messages (as will be seen below). 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 the process is required to be lively but is not generating computational messages.

**Time-silence Mechanism.** The time-silence mechanism of $p_i$, $timesilence_i$, works as follows. A process $p_i$ maintains in the integer variables $SENT_i$ and $RECD_i$, the largest block number of the messages $p_i$ has sent and received at any given time, respectively. Whenever $p_i$ receives a multicast message with block number $B$, it checks whether $SENT_i \geq B$. If $SENT_i < B$, then a local-time-silence timeout, denoted as $ts(B)$, for some predetermined time period is set. This timeout period indicates the duration within which $p_i$ is expected to multicast a message with a block-number $B$ or larger - thus, making its contribution towards the completion of $BM[B]$, by all $p_j \in v_i(g)$ (including itself). When $ts(B)$ expires, $p_i$ multicasts a null message $m$ with the block-number $m.b = RECD_i$, if $SENT_i < B$ is still true; if $SENT_i \geq B$, the expiry of $ts(B)$ is ignored. Note that by multicasting a null message with block-number $m.b = RECD_i$, $p_i$ contributes to the completion of all $BM[B']$, $B' \leq RECD_i$. Stated below is the third possibility $CA3$ by which the $BC_i$ is advanced due $timesilence_i$.

**CA3** (Counter Advance due to $timesilence_i$): Before $p_i$ multicasts a null message, it sets $m.b = RECD_i$ and $BC_i = m.b$.

Finally, we observe that the time-silence mechanism of a given $p_i$ ensures that $SENT_i = RECD_i$ becomes true in finite time if $RECD_i$ ever becomes larger than $SENT_i$. 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.

**Time Bounds for Block Completion.** Precise upper bounds for block completion can only be determined if all processes and channels are synchronous 4. Hence, let us assume that the underlying system is totally synchronous (i.e., $\forall p_i, p_j \in \Pi$: $p_i, p_j$, and channel ($p_i, p_j$) are all timely).

**Lemma 2.** The time bounds for a $BM[m,b]$ to complete at a process $p_i$, as measured by its local clock is:

- $TC1$: $(t_i + ts(m,b) + 2\Delta_{max})(1 + \rho)$, 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 $p_i$

**Proof** A causal block $BM[m,b]$ is created for a process $p_i$ either when $p_i$ sends or when it receives a message $m$ and block $BM[m,b]$ has not been created yet. First, let us consider that $BM[m,b]$ is created for $p_i$ at local time $t_i$ when it sends $m$. In this scenario, $p_i$ will have to wait until time $t_i + ts(m,b) + 2\Delta_{max}$ to receive a message (non-null or null) from every $p_j$ with block-number $\geq m.b$, $i \neq j$. The first $\Delta_{max}$ is due to the time necessary for $m$ to reach $p_j$, and $ts(m,b)$ is the time bound for $p_j$ to send a message for

4actually, 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
$BM[m.b]$, say $m'$, with $m'.b \geq m.b^5$. Finally, the other $\Delta_{max}$ is the time bound for $p_i$ to receive $m'$. Now consider that $BM[m.b]$ is created for $p_i$ at local time $t_i$ when $p_i$ receives $m$ from a process $p_j$. Soon after $BM[m.b]$ is created at $p_i$, $ts(m.b)$ is set up to expire at $t_i + ts(m.b)$ - as a time bound for $p_i$ to send a message for $BM[m.b]$. As every other $p_j, j \neq i$, also receives $m$ (reliable channel assumption) and sets $ts(m.b)$, $p_i$ will have to wait until $t_i + (\Delta_{max} - \Delta_{min}) + ts(m.b) + \Delta_{max}$. The difference $(\Delta_{max} - \Delta_{min})$ is to account for the extreme case where a process $p_j$ may receive $m$ in $\Delta_{max}$ while $p_i$ receives $m$ in $\Delta_{min}$. Finally, $ts(m.b)$ is the time necessary for $p_j$ to send a message in $BM[m.b]$ and $\Delta_{max}$ the time necessary for that message to travel from $p_j$ to $p_i$.

4.3 The Generic Adaptive Protocol

In this section the generic protocol is briefly described. The generic dimension of the protocol lies in the fact that it can either work in a synchronous or asynchronous manner, depending only on the current synchrony of the underlying run-time system. Moreover, it can also work on a system that is hybrid. So, the generic protocol is adaptive in the sense that it will adapt to the available environment at run-time.

As assumed in section 4.1assumptions, a message is sent to a group of processes and such a message reaches all destinations in FIFO order, as long as the sender process does not crash during transmission; in case of crash, some destination processes may not receive a message. 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)$^6$. As soon as a message becomes stable, it is then discarded from the local storage. Unstable and/or not delivered messages are stored in a local buffer.

Condition for Safe Message Delivery. The properties specified in section 3 require that all correct group members deliver the same set of messages and in the same order (agreement and total order, respectively). To meet such properties, a given process member $p_i$ must verify the following delivery conditions:

- $safe_1$: a received $m_i$ is deliverable if $BM[m.b]$ is complete;
- $safe_2$: 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.

$^5$note that $m'$ will only be artificially generated by the time-silence mechanism if $p_j$ has not sent any application message by the expiration of $ts(m.b)$

$^6$the interested reader should refer to [9, 10] for the details on the detection of stable messages in the context of Causal Blocks

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 a timeout for its completion, 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 created one), and the two safety conditions, $safe_1$ and $safe_2$, ensure that received messages are delivered in total order (respecting causality)$^7$. 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 ($TC_1$ or $TC_2$) expires at $p_j$ 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 initialization 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 $(ChangeViewRequest,B), B$ all the other operational members will do so. These requests are then processed by the changing view task (Algorithm 4).

### Algorithm 1

**Executed by a process $p_i$ on a send/receive event of a message $m$**

1. if $BM[m.b]$ does not exist then
2. create $BM[m.b]$
3. if $p_i = m.sender$ then
4. set timeout $TC_1$ for $BM[m.b]$
5. else
6. set timeout $TC_2$ for $BM[m.b]$
7. end if
8. end if
9. store $m$ at a local buffer
10. signal delivery task (Algorithm 2)

### Algorithm 2

**Delivery Task**

1. if any causal block gets completed then
2. deliver messages according to $safe_1$ and $safe_2$
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. $rmcast(ChangeViewRequest,B)$

$^7$Observe that the reception or sending of a message may result in the completion of more than one causal block
Algorithm 4 Executed by $p_i$ on the reception of a (Change-View-Request, $B$) 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 ($\forall p_j \in v_i$: received (unstable, B) from $p_j$ or $p_j \in down$, or $FD_i(p_j) = true$) and for majority of uncertain: received (unstable, B) from $p_i$
7: let allunstable 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.
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: end if
14: if $v_i^{k} \neq v_i^{k+1}$ then
15: install the decided view $v_i^{k+1}$ at $p_i$
16: end if
17: signal delivery task (Algorithm 2) for resuming ordinary message delivery

Fault Tolerant Message Delivery. Consider that $V_v^k$ is the current view of $g$ when $TC1$ or $TC2$ expires for $BM[B]$. Let $F \in \Pi$ be the set of all $p_i$ that failed in sending a message with block-number $B$ or larger (as required by condition $LC1$ of Lemma 1). In other to establish a new view $v^{k+1}(g)$ 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, as required by the adaptive consensus. It is also assumed, as required by the adaptive consensus, the existence of a failure detector of class $\alpha S[4]$ (named FD, where $FD_i(p_j) = true$ if $p_j$ is suspected of crash), and that the majority of processes in the uncertain set do 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 execution of consensus related to 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) together with the identities of the processes that provided the unstable sets (line 8, the set of unstable messages from each process) 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 of algorithm 4). As the decided view may not include a given $p_i$ (that fails in sending its unstable set), it might be terminated (line 12 of algorithm 4). Finally, a new view is only installed if some process has been removed from the current view (lines 14-15 of algorithm 4). Otherwise, the missing messages to complete $BM[B]$ have been recovered from allunstable and no view change is necessary.

4.4 Protocol Correctness

To be correct, the generic protocol must satisfy the properties described in section 3. Below, it is presented the proof sketch for validity. The proofs of 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 $v_i^k(g)$ at real time $t$, then, 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, $m$ (which, by $CA1$, is timestamped with a 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 (see section 4.2) 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 view $v_i^{k+1}$ will be established that excludes those processes that failed 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).
5. FINAL REMARKS

TimedCB has been introduced to handle group communication in hybrid systems, with synchronous and asynchronous characteristics. The underlying system model provides upper-layer applications with process state information according to the current system synchrony (or QoS). Moreover, the underlying system is hybrid, composed by a synchronous part (where there are time bounds on processing speed and message transmission delay) and an asynchronous part (where there is no time bound). However, such a composition can vary over time in such a way that the system may become totally synchronous or totally asynchronous [15]. 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 can be used to trigger failure suspicions.

Analyzing the protocol complexity in a deterministic way (number of communication steps and transmitted messages) cannot be realized for the general case where failure patterns cannot be predicted. Nevertheless, if we consider the runs without failures and the synchronous scenario, TimedCB will produce lower message transmission overhead when compared with classical approaches such as in [6], where messages are sent using an atomic broadcast primitive (which is equivalent to consensus as proved in [4]). In the approach presented in this paper, the cost of consensus is paid only when crashes occur. However, the calculation of the proposed value through the formation of the 

For the asynchronous case, asymmetric approaches [11] may be more efficient in terms of the number of messages transmitted. However, asymmetric approaches will also need extra heartbeat messages to detect failures, whereas in the timeCB, failure detection and ordered message delivery are integrated and no extra heartbeat messages need to be transmitted.

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.

6. REFERENCES