Enhancing group communication with self-manageable behavior

Raimundo José de Araújo Macêdo a,∗, Allan Edgard Silva Freitas a,b, Alírio Santos de Sá a

a Distributed Systems Laboratory – LaSiD, Computer Science Department, Federal University of Bahia, Campus de Ondina, 40170-110, Salvador, Brazil
b Federal Institute of Bahia, Campus de Salvador, 40301-015, Salvador, Bahia, Brazil

ABSTRACT

Group communication protocols (GCPs) play an important role in the design of modern distributed systems. A typical GCP exchanges control messages to provide message delivery guarantees, and a key point in the configuration of such a protocol is to establish the right trade-off between message overhead and delivery latency. This trade-off becomes even a greater challenge in systems where computing resources and application requirements may change at runtime. In such scenarios, the configuration of a GCP must be continuously re-adjusted to attain certain performance goals, or to adapt to current resource availability. This paper addresses this challenge by proposing self-managing mechanisms based on feedback control theory to a GCP especially designed to be self-manageable; in the proposed protocol, message overhead and delivery latency can be adjusted at runtime to follow some new operating set-point. The evaluation performed under varied scenarios shows the effectiveness of our approach.

1. Introduction

Modern computing systems delivered over the Internet, such as cloud computing, allow higher flexibility for end users in terms of resource allocation and service compositions. The fact that users do not have to plan ahead for resource provisioning, and computing resources (such as network bandwidth, CPU, etc.) are allocated on demand to meet user defined SLA (Service Level Agreement), makes these new paradigms very appealing for users and new business opportunities for IT providers. These benefits come, however, with a price for infrastructure and service providers: how to ensure continuous availability and prescribed SLA in such a highly dynamic environment? Adapting to resource variability and changing user requirements at runtime while still maintaining desired characteristics like scalability and continuity of service is a great challenge. As a consequence, these systems need the ability to be self-manageable, continually reconfiguring and tuning themselves to attain certain goals while keeping their complexity hidden from the users.

The study of self-manageable systems has received a great deal of attention from academia and industry, in areas such as robotics, software engineering, network management, automation and control systems, biological computing etc.

Among the efforts to study and understand self-manageable systems, the autonomic computing paradigm, inspired by the human autonomic nervous system, is being adopted by many for the design of systems and applications that must work in accordance with high-level guidance, and are characterized by the so-called self-* properties like self-configuration, self-healing, self-optimization, self-protection, etc. [17,18]. The need for autonomic or self-manageable systems has posed new challenges to the dependability community for revisiting conventional dependable mechanisms design to cope with self-* properties of autonomic systems.

In this paper we address this challenge by introducing self-managing mechanisms to equip group communication, a building block commonly used to design dependable distributed applications [4,9,5,8,10,2]. Group communication protocols (GCPs) designed to date present great variability on the related properties, depending on the target environment and applications. In general, the idea behind group communication is to shield upper-layer applications from the complexity and uncertainty of the underlying communication system, such as failures and unordered message delivery. Besides, group communication provides application processes with membership view guarantees that can be used to, for instance, consistently manage a replica group. In order to provide upper-layer applications with continuous message delivery and updated information on membership changes, due to process failures, leaves and joins, group communication protocols must exchange messages and continuously monitor all group members, which in certain load conditions may incur unacceptable message overhead to the underlying communication system. Therefore, the design of such protocols must deal with the trade-off between performance requirements such as speed (e.g., delivery latency) and cost (e.g., message overhead).
Conventional adaptive strategies address this issue by trying to adapt the group communication protocol to current system and application load patterns, on the basis of a priori knowledge of variation of such patterns. However, when the behavior of the computing environment is unknown and can change over time, or when the application requirements can dynamically change, self-configuring is required—an issue usually ignored in existing implementations. One of the difficulties to implement such self-configuring strategies is the proper modeling of the application and underlying system dynamics. Moreover, modifying optimization goals or set-points at runtime (e.g., desired trade-off between message overhead and delivery latency), makes this challenge even harder.

In this paper we present group communication self-manageable mechanisms inspired on feedback control theory, which have been integrated and implemented in a GCP especially designed to be self-manageable. In our approach we implement a feedback controller that periodically senses the current protocol behavior and underlying system, reconfiguring the protocol when is needed to dynamically meet a set-point, which is a target value, in terms of a trade-off between protocol overhead and delivery latency, that the protocol controller will follow, automatically adjusting the protocol behavior when needed. Such a set-point can also be dynamically modified, at runtime, by the application layer without producing any interruption in the message delivery process. As far as we know, this is the first group communication protocol to address this challenge.

The evaluation performed under various scenarios, which includes the dynamically changing of set-point values and distinct group configurations, shows that the designed group communication controller can indeed self-configure the protocol parameters to follow the aimed trade-offs.

The group communication self-manageable mechanisms presented in this paper can be easily adapted and applied to other existing group communication protocols: to illustrate this point we developed and evaluated a self-manageable version of the AMOEBA group communication protocol [19].

The work presented here is an enhanced and improved version of a previously published conference paper [26].

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 presents the basic non-autonomic group communication protocol and related system model and assumptions. Section 4 presents the group communication self-manageable mechanisms and their integration with the proposed group communication protocol. Section 5 shows how our protocols have been simulated and evaluated. Finally, Section 6 presents our conclusions.

2. Related work

Some authors have considered mechanisms to tune group communication protocols to a trade-off between message delivery latency and message overhead. In NEWTOP [13], the message diffusion style, either sequencer-based or symmetric, can be configured offline by the user to trade message overhead against message latency. Other existing approaches deal with the latency and overhead compromise dynamically switching to distinct group protocol implementations to follow current computing condition loads (for instance, [33,23,31,28,20]). These approaches, however, do not consider the dynamic defined operational set-points in terms of required message latency and overhead trade-off. In contrast, our self-manageable approach does allow for dynamic defined operational set-points in terms of required message latency and overhead trade-off. Moreover, our approach does not realize protocol switching; avoiding, therefore, the usual agreement on protocol switching that leads to even more message overhead.

In another line of research, partition-aware group communication has been proposed [27], where the related protocols monitor and discover partitions of the system, adjusting the group communication accordingly.

Adaptive approaches for group communication based on dynamically defined priorities – used to establish message latencies for distinct process groups –, have been proposed [22,7]. As such, priorities are used according to distinct applications requirements [22]; for instance, interactive applications such as chat would have higher priority than non-interactive applications. However, these works do not consider that application requirements can change at runtime. Besides, a centralized server is used to order and handle membership changes. An adaptive consensus-based group communication protocol has been proposed [7]. In such protocol, named ATOP, messages are given priorities according to the application message transmission rates, and these priorities are then used in the consensus to define the delivery order.

Similarly to other approaches [7], our work also utilizes adaptive timeouts that reflects application message transmission rates. However, in our approach the adaptation is carried out dynamically, at runtime, by an autonomic manager, to adhere to a trade-off between message latency and message overhead.

As mentioned earlier, in our work we consider environments such as cloud computing where applications consume resources as needed and affordable, like networking bandwidth, CPU, and storage facilities. In such environments the trade-off between resource consumption and performance should be established in a very dynamic way, as resources can be dynamically added and released to the cloud. As far as we know the work presented here is the first to address self-configuring mechanisms to dynamically regulate the desired trade-off between speed (message latency) and cost (message overhead) for group communication.

3. The basic group communication structure

In the following we present the group communication structure which subsequently will be extended with self-manageable behavior. In the description of the group communication protocol we focus on the relevant aspects that will be exploited to the design of the self-manageable approach. We start by presenting the system model and assumptions. Then, we state basic group communication properties and briefly discuss their implementation.

3.1. The system model and assumptions

A system consists of a finite set $\mathcal{P}$ of $n > 1$ processes, namely, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, which may fail by prematurely halting execution (crash failure). 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 FIFO bidirectional channel: they do not create, alter, or lose messages. In particular, if $p_i$ sends a message to $p_j$, then if $p_j$ is correct (i.e., it does not crash), eventually $p_j$ receives that message unless it fails. Though precise lower and upper bounds on message transmission ($\delta_{\text{max}}$ and $\delta_{\text{min}}$, respectively) cannot be defined, in the autonomic mechanisms developed in this paper, such bounds are continuously estimated from observed latencies, and are used to suspect process failures.

A process executes steps (a step is the reception or sending of a message, each one with its corresponding local state change, or a simple local state change), and have access to local hardware clocks with drift rate bounded by $\rho$. Processes that do not crash are named correct processes.

We target a distributed system where there is no known upper bound for message transmission or processing times. This makes the related protocols more portable and less sensitive to operational conditions (for example, long unpredictable transmission
times will not affect safety properties of the protocols). On the other hand, we want to design a group communication protocol that can be used in applications such as active replication [32], which require a unique sequence of views being installed: thus consensus is required so that group members can eventually agree in such views. However, as the consensus problem cannot be solved in such a time-free model [14], we assume a partially synchronous model that allows consensus to be solved only eventually [6]—although no upper bound on group operations can still be guaranteed. Thus, we assume the existence of such a consensus protocol and the related failure detector $\mathcal{OS}$ named $\text{FD}$ [15]. $(\text{FD}(p)) = \text{true}$ means that the failure detector suspects the failure of process $p$. These conditions are necessary to solve many fundamental fault-tolerant problems in distributed systems [12].

### 3.2. The basic group communication properties

Processes form a unique group $g$, whose initial configuration is $\mathcal{I}$. That is, $g = \{p_1, p_2, \ldots, p_n\}$. For the sake of simplicity, we will present a single group approach that can be easily extended to support multiple groups [13].

A process $p_i$ of a group $g$ also installs views, named $v_i(g) \subseteq \mathcal{I}$. A view represents the set of group members that are mutually considered operational at a given instant of the group existence. A view $v_i(g)$ can dynamically change on the occurrence of process crashes (or suspicions), or when processes deliberately leave or join $g$. When a change occurs in the group view, a new view is installed by a group membership protocol. It is assumed a majority of correct processes in a view $v_i(g)$. Each view installed by a process is associated with a number that increases monotonically with group view installations. $v_i(g)_{k}$ denotes the view number $k$ installed by $p_i$.

When a group $g$ is created, every group member $p_i$ installs the same initial view $v_i^0 = \mathcal{I}$. Any subsequent modifications on the configuration of the group will result in new views being installed, forming the sequence $v_i^0, v_i^1, \ldots, v_i^k$. A process $p_i$ multicasts 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 and the set of messages delivered by distinct processes. Such properties vary from one implementation to another, following a given target computing environment and applications [10,8,9,3]. 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 satisfy total order message delivery (respecting causality) and agreement on a linear group view history [32,21]. The other properties satisfied by our protocol are similar to those of related literature and will not be further discussed in this paper.

### 3.3. The basic group communication approach

The basic group communication protocol introduced in this work is derived from the causal blocks model, which is enhanced here to allow uniform message delivery. In the following the Causal Blocks model is briefly presented (a complete description can be found elsewhere [24]). Each process maintains a logical clock [21], 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 allows us to construct a Block Matrix $BM$, as shown in Fig. 1 to a 6-member group process. 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.

![Fig. 1. The Block Matrix of a 6-member group process.](image)

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 $m \cdot b$ from $p_j$, no other messages from $p_i$ with an equal or smaller block number $b' \leq m \cdot b$ 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, only 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 $BM[8]$ 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 $BM[8]$ is created by $p_i$, it sets a timeout $t_s$; after the expiration of $t_s$, if $p_i$ has not yet sent a message $m$ to contribute to that block completion ($m \cdot b \geq B$), a null message timestamped with the largest known block number is sent by $p_i$.

Though precise upper-bounds on message transmission cannot be defined, we can estimate them on the basis of message delay history (the way we do such estimation is explained in details in Section 4). From the estimated bounds for message transmission, we can then estimate an upper bound for block completion, as follows. Assume that $BM[m \cdot b]$ was created at time $t_i$. The estimated time bounds for completion of $BM[m \cdot b]$ at $p_i$, as measured by its local clock are:

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

**Proof Sketch for TC1 and TC2.** A causal block $BM[m \cdot b]$ is created for a process $p_i$, either when $p_i$ sends or when it receives a message $m$ and block $BM[m \cdot b]$ has not been created yet. First, let us consider that $BM[m \cdot b]$ 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 $t_i + t_s(m \cdot b) + 2\delta_{\text{max}}$ to receive a message (non-null or null) from every $p_j$ with block-number $\geq m \cdot b$, $i \neq j$. The first $\delta_{\text{max}}$ is due to the time necessary for $m$ to reach $p_i$, and $t_s(m \cdot b)$ is the time bound for $p_i$ to send a message for $BM[m \cdot b]$, say $m'$, with $m' \cdot b \geq m \cdot b$. Finally, the

---

1 Note that $m'$ will only be artificially generated by the time-silence mechanism if $p_i$ has not sent any application message by the expiration of $t_s(m \cdot b)$. 
other $\delta_{\text{max}}$ is the estimated time bound for $p_i$ to receive $m'$. Now consider that $BM[m \cdot b]$ is created for $p_i$ at local time $t_i$, when $p_i$ receives $m$ from a process $p_j$. Soon after $BM[m \cdot b]$ is created at $p_i$, $ts(m \cdot b)$ is set up to expire at $t_i + ts(m \cdot b)$—as a time bound for $p_i$ to send a message for $BM[m \cdot b]$. As every other $p_i$, $i \neq j$, also receives $m$ (reliable channel assumption) and sets $ts(m \cdot b)$, $p_i$ will have to wait until $t_i + (\delta_{\text{max}} - \delta_{\text{min}}) + ts(m \cdot b) + \delta_{\text{max}}$. The difference $(\delta_{\text{max}} - \delta_{\text{min}})$ is to account for the extreme case where a process $p_i$ may receive $m$ in $\delta_{\text{max}}$ while $p_i$ receives $m$ in $\delta_{\text{min}}$. Finally, $ts(m \cdot b)$ is the time necessary for $p_i$ to send a message in $BM[m \cdot b]$ and $\delta_{\text{max}}$ the estimated time necessary for that message to travel from $p_i$ to $p_j$.

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 cannot be immediately discarded as its retransmission may be required. Messages that have not been acknowledged by all member processes are called unstable messages (stable messages, otherwise). In a protocol presented elsewhere [25], as soon as a message becomes stable, it is then discarded from the local storage—which is enough to assure agreement on message delivery. In the protocol proposed in this paper, we want to assure uniform agreement on message delivery, where even crashed processes that delivered messages must agree on the set of messages delivered. Hence it is required that a message be super-stable before discarding it: a block or message is said to be super-stable when it is known to be stable at all processes [24].

Uniform agreement on message delivery. To determine when a block is stable, group members inform in every transmitted message its last complete block number (LCB). By collecting all LCB information available, we can derive the last stable block $LSB = \min\{LCB\}$, $\forall p_i \in g$. Hence, all blocks of a process $p_i$ with number equal or less to $LSB$ are stable to $p_i$ (i.e., the related messages will be delivered 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 $m \cdot b$ is the block number of message $m$:

- **stable-safe1**: a received $m$ is deliverable if $BM[m \cdot b]$ 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 the absence of incomplete blocks. That is, after block completion, when there is an idle period $ts$ without incomplete blocks, a null message is sent in order to transmit the LCB information to all processes. As before, time-silence messages will not create new blocks.

Timeouts for block stability. Assume that $BM[m \cdot b]$ is created at local time $t_i$ by $p_i$ (as measured by its local clock). The timeouts for block stability can be estimated as follows.

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

**Proof Sketch for ST1 and ST2.** After a block gets completed, a message to notify such a completion will be sent by $ts(m \cdot b)$ and received by an estimated bound of $\delta_{\text{max}}$. Therefore, the estimated bounds for block stability will be that of block completion plus $ts(m \cdot b) + \delta_{\text{max}}$. □

### 3.3.1. Causal blocks and timeout management procedures

Algorithm 1 is triggered by a send/receive event of a message $m$. Such an algorithm sets proper timeouts in case of causal block creation, storing message $m$ in a local buffer and triggering the delivery task (Algorithm 2). This latter task will deliver stable messages according to the delivery conditions.

**Algorithm 1**: Causal Blocks Management Task

1. **on event** (send or receive of $m$) at $p_i$ **do**
   2. if $BM[m \cdot b]$ does not exist **then**
   3. create $BM[m \cdot b]$;
   4. if $p_i = m$.sender **then**
   5. set completion timeout $TC_1$ for $BM[m \cdot b]$;
   6. set stability timeout $ST_1$ for $BM[m \cdot b]$;
   7. else
   8. set completion timeout $TC_2$ for $BM[m \cdot b]$;
   9. set stability timeout $ST_2$ for $BM[m \cdot b]$;
   10. store $m$ in a local buffer;
   11. **signal** Delivery Task (Algorithm 2);

**Algorithm 2**: Delivery Task

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

Group membership procedures. Suppose that a process $p_i$ fails by stop functioning (crashing) and, as a consequence, a timeout expires at $p_i$ for $BM[m \cdot b]$. In order to proceed with message delivery, a new membership for $g$ must be established that excludes $p_i$ (or any other faulty processes), which is conducted by a consensus procedure [26].

### 4. Self-managing mechanisms for group communication

The self-manageable approach extends the basic group communication protocol to deal with the issue of adjusting time-silence in dynamic computing environments. Choosing an appropriate time-silence in such scenarios is a challenge because: (a) longer time-silence intervals decrease message overhead and resource consumption, but, on the other hand, it can imply longer blocking times for message delivery when processes are not sending application messages to complete blocks; (b) in the opposite direction, shorter time-silences can reduce blocking time, but it increases message overhead and resource consumption which can be a problem when the computing environment is subjected to high workload conditions, leading to longer end-to-end delays and message delivery blocking times. To address such trade-off, we dynamically adjust time-silence using feedback control theory [16], which is traditionally applied for controlling dynamic systems in many areas (e.g., in the automation industry). The basic idea of the feedback control is the implementation of a closed loop in which a controller uses a sensor to observe the actual outputs of a controlled object (or plant), and executes a control algorithm (or control law) to define new input values in order to guarantee that the related outputs follow the set-points (desired behavior) [29].

The feedback control loop implemented here monitors and controls the causal block structure embedded in each distributed process. As such, the control loop is made up of two components (see Fig. 2): sensor and controller.

The sensor collects, treats and shares data about the computing environment and performance of the group communication protocol (e.g., end-to-end communication delays, protocol overhead,
application workload etc.). The controller uses the sensed or estimated information to dynamically adjust time-silence according to the desired values (set-points). Set-points are defined in terms of resource consumption.

Observe that as we cannot know the real amount of the resources available in the environment (it is a distributed system), resource consumption is actually a metaphor to represent a relative measure which depends on the availability of resources at a certain time. The idea behind such metaphor is that more resources (like network bandwidth, nodes, memory etc.) will likely produce shorter delays, and vice-versa. For instance, if a node is short of memory or a network node fails, delays will be likely longer than previously observed which in turn is taken as an indication of a smaller percentage of available resources. More precisely, resource consumption is taken as a function of end-to-end communication delays, as follows: (a) if the mean end-to-end delay is close to the minimum delay observed then the actual resource consumption is close to zero; (b) if the mean end-to-end delay is close to the maximum delay observed then the actual resource consumption is close to one; otherwise, the resource consumption is a value between zero and one. Then, using this assumption, we can estimate current resource consumption for current observed end-to-end delays.

It should be noted that the definition of the set-point depends on the current number of group members and system workload. Because these conditions vary at runtime, set-points must be dynamically recalculated according to desired computing resource consumption. Thus, the dynamic set-point is defined from: (i) the user requirement in terms of resource consumption; (ii) the actual resource consumption in the computing environment; and (iii) the number of active group members.

In the following, we present more details about the design of the sensor and controller components. We begin by presenting, in next section, the main algorithms and components used in the feedback control loop. The mathematical details related to the control analysis are then presented in Section 4.2.

4.1. Control loop design

The two main components of the feedback control loop, the sensor and the controller, are described below.

4.1.1. The sensor component

The sensor component is composed by two basic tasks (see Fig. 3): sensing and transducing.

Sensing task. The sensing task collects raw data about the protocol execution and about the computing environment—see Algorithm 3. As such, on each received message $m$, the sensing task accounts $m$, incrementing $n_{rv}$, the number of received messages (Line 2). If $m$ is a control message, then $n_{ct}$ also is incremented (Lines 1–4); otherwise, the arrival time of $m$ is stored in vector $A$ at the $j$-th position (related to the sender process identifier $p_i$), see Line 6. Lastly, the task collects the current round-trip-time delay (Line 7).

Transducing task. The transducing task uses the sensed data to estimate the cost of the protocol, to characterize the behavior of the computing environment, to observe the application workload, and to estimate the computing resource usage.

Algorithm 4 shows the transducing task which executes on each delivered message.

Algorithm 3: Sensing Task

```
1 on event receive of $m$ at $p_i$ from $p_j$ do
2   $n_{rv} = n_{rv} + 1$;
3   if $m$ is a control message then
4     $n_{ct} = n_{ct} + 1$;
5   else
6     /* clock($p_i$) returns the current time according to the
7     local clock of $p_i$ */
8     $A[j] = $clock($p_i$);
9     obtain rtt from FIFO channels;
```

Algorithm 4: Transducing Task

```
1 on event delivery of $m$ from $p_j$ at $p_i$ do
2   $ovh = \frac{n_{rv}}{n_{ct}}$;
3   $ovh_{max} = \frac{(n - 1)}{n}$;
4   $\delta = \frac{rtt}{2}$;
5   if $\delta_{mean}$ is unset then
6     $\delta_{mean} = \delta$;
7     $\delta_{max} = \delta$;
8     $\delta_{min} = \delta$;
9     $\delta_{mean} = \alpha \ast \delta_{mean} + (1 - \alpha) \ast \delta$;
10    if $\delta_{max} < \delta$ then
11       $\delta_{max} = (1 + \beta) \ast \delta$;
12    if $\delta_{min} > \delta$ then
13       $\delta_{min} = \delta$;
14    $\delta_{min} = \phi \ast \delta_{min} + (1 - \phi) \ast \delta$;
15    $\delta_{max} = \phi \ast \delta_{max} + (1 - \phi) \ast \delta$;
16    $rc = \delta_{max} - \delta_{min}$;
18    if $\tau_{max} < \tau[j]$ then
19       $\tau_{max} = \tau[j]$;
20    $\tau_{max} = \phi \ast \tau_{max} + (1 - \phi) \ast \tau[j]$;
21    $\tau_{min} = (1 + \beta) \ast \tau_{max}$;
```

For estimating the cost of the group communication protocol, the transducing task computes the mean and maximum overheads, named $ovh$ and $ovh_{max}$, respectively. The mean overhead is the ratio of the total number of control messages ($n_{ct}$) to the total number of messages ($n_{rv}$) – see Line 2 of Algorithm 4. The maximum overhead is computed assuming the worst-case scenario, where one group member sends a message and block completion occurs only after the $(n-1)$ other group members have sent null messages for that block – i.e., for each set of $n$ messages, $n - 1$ are control messages—see Line 3 of Algorithm 4.

The transducing task characterizes the behavior of the computing environment using the mean, maximum and minimum end-to-end delays, named $\delta_{mean}$, $\delta_{max}$ and $\delta_{min}$, respectively. As such,
the task first estimates the end-to-end delay (δ) as half of the round-trip-time delay (rtt)—see Line 4 of Algorithm 4. Then, δmean is calculated from the moving average of the last delays observed considering a smoothing factor $\alpha = \frac{w-1}{w}$, where $w$ is the size of the history of delays considered (Line 9 of Algorithm 4). The maximum and minimum end-to-end delays are, respectively, the longest and shortest delays observed at runtime (Lines 10–13 of Algorithm 4).

The variables $\delta_{\text{min}}$ and $\delta_{\text{max}}$ are also used by the group communication protocol to estimate the timeouts for completion and stability of causal blocks (see Section 3.3). In order to prevent unnecessary executions of the membership procedure, we consider a safety margin ($\beta > 0$) in the estimation of the maximum end-to-end delay. As the bounds $\delta_{\text{min}}$ and $\delta_{\text{max}}$ depend on characteristics of the computing environment, then when the environment characteristics change, such variables may change too. Thus, we consider a forgetting factor ($\phi$) in the estimation of $\delta_{\text{min}}$ and $\delta_{\text{max}}$. In such a way that recent samples are the most meaningful, providing adaptive timeouts for causal blocks (Lines 14–15 of Algorithm 4).

In order to estimate the percentage of resource consumption ($rc$) in the computing environment, as discussed earlier, we use a metaphor that relates the variation of resource consumption as a function of the variation of the observed delays, in the following way: the longer the end-to-end delays, the higher the percentage of resource consumption (see Fig. 4). Though a nonlinear relation could be used to relate resource consumption to end-to-end latencies – and we intend to explore such a nonlinear approach in future work –, the linear relation modeling we assumed in this paper was effective enough for the construction of the controller as we later observe in the evaluation section.

To find resource consumption as a function of the end-to-end delays, we first normalize $rc$ and $\delta_{\text{mean}}$, establishing a linear relation between them. Thus, assuming maximum and minimum percentages of the resource consumption as $rc_{\text{max}} = 1$ and $rc_{\text{min}} = 0$, respectively, we find a value for $rc$ as a function of end-to-end delays as in Line 16 of Algorithm 4—see Fig. 4 for an illustration.

Time-silence bounds should be carefully chosen by the autonomic manager so as to avoid degrading the group communication protocol performance when changes in the distributed computing environment occur.

Note that the shortest possible value for time-silence bound is zero (i.e., $t_{\text{min}} = 0$), which forces group members to send a null message immediately after receiving an application message. For the maximum time-silence ($t_{\text{max}}$), we assume $t_{\text{max}}$ as the maximum inter-arrival time observed ($t_{\text{max}}$) considering a safety margin ($\beta > 0$), which leads to minimum message overhead (see Line 21 of Algorithm 4).

For estimating $t_{\text{max}}$, the transducing task first computes $t$ as the difference between the current and the last arrival times ($A_t[j]$ and $A_{t-1}[j]$) of messages sent by a process $p_j$—see Line 17 of Algorithm 4. Subsequently, if $t > t_{\text{max}}$, then the task assigns $t$ to $t_{\text{max}}$ (see Lines 18–19 of Algorithm 4). Lastly, when the characteristics of the system workload change, $t_{\text{max}}$ also can change. Thus, we apply a forgetting factor ($\phi$) in the estimation of $t_{\text{max}}$—see Line 20 of Algorithm 4.

4.1.2. The controller component

The controller component is the main element in the protocol regulation. We implemented such a component by using two basic tasks (see Fig. 5): dynamic set-point estimation and time-silence regulation.

Set-point estimation task. The set-point estimation task suggests dynamic set-points using the user requirements and the protocol cost—both defined in terms of resource consumption. As such, we consider the following rules: if the estimated resource consumption ($rc$) is larger than the user-defined requirement ($rc_D$), then the protocol must work in lower consumption mode, decreasing message overhead; otherwise, if the estimated resource consumption is lower than the user-defined requirement, then the protocol must work in lower latency mode, properly increasing message overhead (see Fig. 6).

The set-point estimation task is presented in Algorithm 5.

**Algorithm 5: Dynamic Set-Point Estimation Task**

```plaintext
1 on event delivery of m at p_i do
2 get $rc_D$; /*get desired resource consumption from user*/
3 get $rc$; /*get resource consumption from Transducing Task*/
4 $rc_P = rc_D - rc$; /*compute residual resource consumption*/
5 $\alpha \cdot h_P = rc_P + \alpha \cdot h_{max}$; /*compute dynamic set-point*/
```

In Algorithm 5, on each delivered message, we obtain the user-defined resource consumption ($rc_D$) and get the current estimated resource consumption ($rc$) from the transducing task; see Lines 2–3. Recall that $rc$ represents actually a metaphor and results from the combination of the system workload with the protocol message overhead.
The message overhead represents an additional cost imposed by the protocol execution, then it must be adapted to allow for the user expectations \((r_c)\). The variable \(r_c\) represents the protocol operation mode in terms of resource consumption: if \(r_c > 0\), then the protocol can increase message overhead; on the other hand, \(r_c < 0\) indicates that the protocol must decrease message overhead—see Line 4.

In order to estimate the protocol set-point in terms of message overhead, we assume the variation of resource consumption as proportional to the variation of message overhead, i.e., the higher message overhead, the higher the percentage of resource consumption (see Fig. 7).

Thus, assuming the maximum and minimum percentages of the resource consumption as \(r_{c_{\text{max}}} = 1\) and \(r_{c_{\text{min}}} = 0\), respectively, we estimate the desired set-point on message overhead as (see Algorithm 5, Line 5):

\[
\text{o\text{vh}}_{\text{p}} = r_c \times \text{o\text{vh}}_{\text{max}}.
\]

Such set-point on desired message overhead \((\text{o\text{vh}}_{\text{p}})\) is then used as a reference to calculate the dynamic time-silence values, as below.

**Time-silence regulation task:** The time-silence regulation task makes continuous adjustments in time-silence to meet the desired message overhead \((\text{o\text{vh}}_{\text{p}})\), considering dynamic changes in the computing environment or in the application workload. To achieve that, we assume a linear function, say \(F\), which determines as output a time-silence value from the input message overhead:

\[
ts = F(\text{o\text{vh}}).
\]

Note that \(F\) is linear, thus the desired variation of time-silence \((\text{ts}_{\text{var}})\) can be expressed as a function of the desired variation on message overhead. That is, \(\text{ts}_{\text{var}} = F(\text{o\text{vh}}_{\text{p}} - \text{o\text{vh}})\), where \(\text{o\text{vh}}\) is the current message overhead. Regarding \(F\), we assume the following linear relation between message overhead and time-silence: the higher the time-silence (or its variation), the lower the message overhead (or its variation)—see Fig. 8.

As a result, we can compute \(F(\text{o\text{vh}}_{\text{p}} - \text{o\text{vh}})\) as:

\[
F(\text{o\text{vh}}_{\text{p}} - \text{o\text{vh}}) = -(\frac{\text{o\text{vh}}_{\text{p}} - \text{o\text{vh}}}{\text{o\text{vh}}_{\text{max}}}) \times \text{ts}_{\text{max}}.
\]

Note that the definition of \(F\) uses information about the main factors which impact the performance of the protocol: number of members (from \(\text{o\text{vh}}_{\text{max}}\)); sending rate of the application (from \(\text{ts}_{\text{max}}\)); user-defined requirement and end-to-end delays (from \(\text{o\text{vh}}_{\text{p}}\)).

Then, we use the following control action \((\text{ts}_{\text{ctrl}})\) to determine the time-silence adjustment:

\[
\text{ts}_{\text{ctrl}} = K_p \times \text{ts}_{\text{var}}
\]

where \(K_p\) is a parameter which represents the gain of the controller and weighs the effects of the variation of message overhead in the time-silence adjustments. Since \(\text{ts}_{k+1} = \text{ts}_k + \text{ts}_{\text{ctrl}}\), we can express the time-silence adaptation as:

\[
\text{ts}_{k+1} = \text{ts}_k + K_p \times \text{ts}_{\text{var}}
\]

Algorithm 6 shows the time-silence regulation task. On each delivered message, this task obtains the dynamic set-point \((\text{o\text{vh}}_{\text{p}})\); see Line 2. Then, \(\text{o\text{vh}}\) is used to estimate the desired variation in the time-silence, i.e., \(\text{ts}_{\text{var}} = F(\text{o\text{vh}}_{\text{p}} - \text{o\text{vh}})\) – see Line 3. Lastly, time-silence is computed and bounded between zero and \(\text{ts}_{\text{max}}\) – see Lines 4–7.

### 4.2. Control loop analysis

This section presents details of the control loop analysis used to determine the parameter \(K_p\) of the controller (see Eqs. (3) and (4), and Algorithm 6, Line 4). Determining a value for \(K_p\) (actually, the range of possible values) is required for the correct tuning of the controller performance, which is perceived from the following properties: control stability, control accuracy, and overshoot. Roughly speaking, stability is related to the controller capability in choosing values for time-silence \((\text{ts})\) that make the protocol overhead \((\text{o\text{vh}})\) (plant output) converge to a desired operational region which includes the desired operational set-point or desired output (i.e., the reference overhead, \(\text{o\text{vh}}_{\text{p}}\)). Once the control loop is verified stable, the control accuracy property is used to characterize the controller response in steady-state, which defines the controller capability in minimizing (or eliminating) errors when working in steady-state (steady-state error).

The analysis of the time-silence regulation loop to find out the proper values for \(K_p\) was made in two steps: (i) finding the transfer function which represents such a closed loop; and (ii) using the closed loop transfer function to analyze the control performance.

Lastly, we note that the control loop analysis presented in this section relies on a linear relation between protocol overhead and time-silence. Such linear relation is intuitive since reducing...
time-silence will likely produce new null messages which increase overhead. Observe, though, that in some cases null messages may not be sent when application messages are transmitted before the expiration of the related time-silence. In the latter situation, overhead will be maintained – but never decreases. The same reasoning applies in the other direction when time-silence values are increased. Therefore, the linear relation is an appropriated overhead will be maintained – but never decreases. The same expiration of the related time-silence. In the latter situation, overhead. Observe, though, that in some cases null messages may increase time-silence will likely produce new null messages which increase overhead. Observe, though, that in some cases null messages may not be sent when application messages are transmitted before the expiration of the related time-silence. In the latter situation, overhead will be maintained – but never decreases. The same reasoning applies in the other direction when time-silence values are increased. Therefore, the linear relation is an appropriated overhead will be maintained – but never decreases. The same expiration of the related time-silence. In the latter situation, overhead. Observe, though, that in some cases null messages may time-silence and message overhead between the (k – 1)th and the kth samples as:

\[ \Delta ovh(k) = ovh(k – 1) = – \Delta ts(k – 1) \times \frac{ovh_{\text{max}}}{ts_{\text{max}}} \]  

(10)

Applying z-transform in Eq. (10), we obtain:

\[ ovh(z) \ast (1 – z^{-1}) = – ts(z) \ast (1 – z^{-1}) \times \frac{ovh_{\text{max}}}{ts_{\text{max}}} \]  

(11)

Thus, from Eq. (11), we obtain:

\[ CBP(z) = \frac{ovh(z)}{ts(z)} = \frac{ovh_{\text{max}}}{ts_{\text{max}}} \]  

(12)

\[ H_1 \text{ can be obtained rewriting the Eq. (9) as:} \]

\[ \Delta ts = - \Delta ovh \times \frac{ts_{\text{max}}}{ovh_{\text{max}}} \]  

(13)

As result of the Eq. (13), we can use z-transform to define the transfer function of H_1 as:

\[ H_1(z) = - \frac{ts_{\text{max}}}{ovh_{\text{max}}} \]  

(14)

The transfer function of the control law gets the desired time-silence variation (i.e., \( ts_{\text{var}} \)) as input and produces a time-silence value as output—this is:

\[ C(z) = \frac{ts(z)}{ts_{\text{var}}(z)} \]  

(15)

The control law is expressed as (see Algorithm 6):

\[ ts(k) = ts(k – 1) + K_P \times ts_{\text{var}}(k) \]  

(16)

As result of the Eq. (16), we can express:

\[ ts(z) \ast (1 – z^{-1}) = K_P \times ts_{\text{var}}(z) \]  

(17)

Thus, we can find:

\[ C(z) = \frac{ts(z)}{ts_{\text{var}}(z)} = \frac{K_P \times z}{z – 1} \]  

(18)

Lastly, from Eq. (18), we can compute G(z) as:

\[ G(z) = \frac{- ts_{\text{max}}}{ovh_{\text{max}}} \times \frac{K_P \times z}{z – 1} \times \left( \frac{- ovh_{\text{max}}}{ts_{\text{max}}} \right) \]  

(19)

That is:

\[ G(z) = \frac{K_P \times z}{1 + \frac{K_P \times z}{z – 1}} = \frac{K_P \times z}{z \ast (K_P + 1) – 1} \]  

(20)

Stability analysis. The system stability is an important property because it ensures that the system output converges to an equilibrium state after it has been subjected to external disturbances or transient input variations [16]. In a discrete control system, a typical criteria used to ensure system stability is [29]:

Stability criteria. A system represented by a transfer function T(z) is stable if all the poles of T(z) are inside the unit circle.

Hence, the stability of G(z) is ensured if all its poles (i.e., roots of \( z \ast (K_P + 1) – 1 \)) have their magnitudes strictly less than 1 [29]. From Eq. (20), the poles of G(z) can be computed as:

\[ z = \frac{1}{K_P + 1} \]  

(21)
Therefore, the poles of $G(z)$ have magnitudes strictly less than 1 if $K_p > 0$.

**Accuracy analysis.** The control system is accurate if it is stable and if its measured output converges to a value close to the reference input (i.e., set-point). How close the measured output should be to the set-point is dependent on the application requirements.

We can observe the system accuracy using Eq. (20), as follows:

$$G(1) = \frac{K_p \ast 1}{1 \ast (K_p + 1) - 1} = 1.$$  \hspace{1cm} (22)

As $G(1) = 1$, then the steady state error is zero and the system is therefore accurate—see [16].

**Overshoot analysis.** The maximum overshoot ($M_P$) refers to the maximum amount by which the transient response of the control system exceeds the set-point as a result of a change in an input—see [16]. Our control system has real poles, then we can compute $M_P$ as [29]:

$$M_P = \left| \frac{1}{K_p + 1} \right| \ast 100\%.$$  \hspace{1cm} (23)

Thus, from Eq. (23), we conclude that: the greater $K_p$, the smaller $M_P$. This conclusion combined with the result of the stability analysis were used to choose the value of $K_p$ during the performance evaluation in Section 5.

4.3. **Applying the self-manageable approach to the Amoeba case**

In order to illustrate the generality of our approach we applied it to an uniform delivery version of Amoeba’s group communication protocol [19], hereafter called simply Amoeba. The uniform delivery version was developed here to provide the same message delivery guarantees as our causal blocks based group communication approach. Amoeba was chosen due to its simplicity and because it represents a typical sequenced-based approach, distinct from our symmetric group protocol.

The original Amoeba protocol uses a fixed sequencer process to produce message ordering—i.e., messages are first passed to the sequencer that multicast them to the group. In its uniform agreement on message delivery version, processes should acknowledge to the sequencer which in turn waits for a quorum of acknowledgment messages before sending a control message allowing delivery. The price paid for this version (in contrast with the non-uniform version) is a two-phase protocol, which introduces an extra communication step.

In order to save resources, acknowledgments can be piggybacked on application messages, and to ensure that there are regular acknowledgments even when members are not originating multicasts, each member is expected to send periodic acknowledgments. We can then configure the inactivity time interval ($t_u$) used for triggering these control messages for a desired trade-off between message delivery latency and message overhead.

We implemented an autonomic version of the above protocol, merging this protocol with a feedback control loop (see Fig. 10)—it is similar to the one used in the causal blocks based approach, but we regulate $t_u$ instead of $t_s$.

5. **Performance evaluation**

In this section, we evaluate the performance of the self-manageable approach comparing it with the non-self-manageable version of the group communication protocol. To realize that, we conducted several experiments using simulations and measurements.

We use simulation to explore a larger number of different scenarios, varying factors such as workloads, group sizes, faulty conditions, and changes in user-defined requirements. The measurement is used to observe the protocol performance in a real network environment with unknown traffic.

In the simulated experiments, we used a simulation tool that we had previously developed. In such a tool, distinct fault models can be simulated, when necessary, by defining related probability density functions. The simulation tool allows also modifying the simulation specification on-the-fly, for instance, by changing end-to-end latencies, modifying topologies, switching protocol’s behavior, modifying distribution functions for faulty behaviors etc.

We simulate processes that are hosted in sites that communicate over the Internet, and we assume that message delays convey resource usage, increasing with the amount of messages transmitted. In order to consider such a resource-usage dependent delays, we simulated the effects of a subjacent communication infrastructure with core routing, and modeled queuing effects on end-to-end delays (core routing simulates 10 Mbps throughput in a Network with links with MTU of 64 KB). Therefore, in our simulations, end-to-end delay is calculated from two parts: core routing delay and the delays from a site on border to the core, and, after the routing, from the core to another site. For the latter part, we assume a log-normal probabilistic distribution that expresses inter-site communication delays. More precisely, we assumed a delay log-normal distribution with mean and standard deviation of 10 ms and 5 ms, respectively.

For the measurement in a real network environment, we used a group of 5 processes distributed across a metropolitan-area Ethernet scenario. In this scenario, each process was located in a distinct machine which also runs other distributed services. Four machines were in a local area network connected with the fifth machine hosted in another local area network at a distance of 10 km apart. The connections between the machines were of 1 Gbps. The group communication primitives were implemented over UDP. The group application used a synthetic workload and the workload of the other distributed applications were unknown.

5.1. **Context of the performance evaluation**

In the evaluation, the main performance metrics are blocking time (BT) and message overhead (ovh). Blocking time for a message $m$ received at process $p_i$ is defined as the latency from the reception to the delivery of $m$ at $p_i$. Thus, blocking time isolates the network latency and measures how much time the protocol holds a message after its reception from the network until its delivery to the application, in order to assure the desired group properties. In the experiments, we compute the average and standard deviation values for BT over a number of protocol executions.

It should be noted that the actual blocking time observed depends on distinct factors, such as system and network workloads, application activity (i.e., how often application messages are sent) and group sizes. Therefore, we must observe the behavior of our protocol under a representative variation of these factors.

3 We measured delays between different sites of the Brazilian Academic Research Network (RNP), and computed maximum of 221.855 ms, mean of 10.321 ms and standard deviation of 4.796 ms for 2000 packets of 15 KB.
The other performance metric, message overhead ($ovh$), measures how many protocol messages are created in face of the total number of messages, as discussed in Section 4.1.1. That is, $ovh = n_{ct}/(n_{ap} + n_{ct})$, where $n_{ct}$ and $n_{ap}$ are, respectively, the total number of control messages and total number of application messages. We can push the protocol to work faster (i.e., to produce faster message delivery), if we decrease the time-silence values, forcing more protocol activity and more protocol messages being transmitted. On the other hand, by doing that we increase message overhead and resource consumption (i.e., use of network). This time-silence effect must also be observed, as well the effect of different workloads. Notice that more protocol activity from application messages will likely to produce fewer null messages, decreasing the protocol overhead.

In very dynamic and elastic computing environments, such as cloud computing, users are billed by the amount of computing resources their applications consume (e.g., networking, processing and storage facilities etc.). In these contexts, underlying mechanisms (like group communication protocols) must be able to self-adjust their resource usage according to changes in the cloud workload or in the cloud's application requirements. We built our experiments targeting such environments. For instance, suppose a cloud environment where a module named SLA Manager defines the SLA negotiated for a distributed application over the cloud. As an illustration, consider the sketch of a Resource Cloud Manager (see Algorithm 7), which receives dynamic defined SLAs and triggers change requests for resource consumption set-points on a self-manageable group communication protocol used by all replicas that run the distributed application. A change in the set-point could be launched for increasing resource consumption which will allow for faster message delivery, or, otherwise, to reduce resource consumption for the related applications, slowing down message delivery speed. Therefore, we also must evaluate the capability of our approach to correctly respond to dynamic changes in the resource consumption set-points.

Algorithm 7: Resource Cloud Manager adapting to new SLA

1. let cm be a instance of ResourceCloudManager;
2. let rm be a instance of ReplicationManager;
3. on event receive of (SLARenegotiation) request by cm do
   
   for all the (rm ∈ CloudApplication) do
   
   trigger (GroupProtocol, ChangeSetPoint);
   
   do other stuff;

5.2. Evaluation scenarios

Following the previous discussion, we evaluated our protocol for a number of representative scenarios, varying the number of processes ($n$), the workload profile ($Load$), and the occurrence of process failures. Below we further discuss each of the evaluated scenarios.

- Distinct-fixed-workloads: In this scenario, we compared the non-self-manageable (i.e., fixed time-silence) and the self-manageable versions varying group sizes and workload profiles between each series of experiments. In this kind of experiment, we can observe, how different workload profiles define different operation points for resource consumption on group communication, and how the self-manageable version adapts itself for these different workloads, to follow desired resource consumption (resource consumption set-point).
- Variable-workload: In this scenario, we experimented with dynamic workloads for different group sizes. We then evaluated how the self-manageable version follows the operation point (resource consumption) in a very dynamic environment.

- changing set-point: In this scenario, we experimented with a fixed group size and dynamically varied the desired operating set-point (resource consumption). We then evaluated how the self-manageable version of group communication adapts to these new set-points and its impact on message delivery speed.
- Faulty scenario: In this scenario, for a fixed group size, we simulated crashes, and observed how our self-manageable version of group communication adapts itself to the occurrence of these crashes, keeping the operational set-point in terms of resource consumption.
- amoeba-comparison: In this experiment, we evaluated the generality of the autonomic approach applying it to a self-manageable version of the Amoeba’s group communication protocol, and comparing its performance with the non-self-manageable counterpart. We then observed how both versions behave in presence of different workloads.
- real network: In this scenario, we experimented on a real network environment. The experiment compares the non-self-manageable and the self-manageable versions for a group of 5 processes located in 5 distinct machines.

5.3. Evaluation results

In order to simulate the generation of application messages, each process uses a Bernoulli probability density function to decide when to send a message, following three levels of transmission rates: 100, 150 and 50 messages per second. These transmission rates are named medium, high and low, respectively.

In the experiments, we measured delivery blocking time and protocol overhead for each configuration of the evaluation factors (time is represented in ms). The simulation for each configuration of the simulation factors was run for 60 s (the number of transmitted messages per process varies from 3,000, for low workload, to 9,000 messages for high workload), and the same simulation was replicated three times for calculating the related statistics. Each generated application message has 4 Kbytes.

For the self-manageable approach, we use the following configuration for control parameters:

- smoothing factor $\alpha$ = 0.99 which was computed considering a smoothing window $w_1$ = 100 messages—i.e., $\alpha = \frac{w_1 - 1}{w_1}$;
- safety margin $\beta = 0.1$;
- forgetting factor $\phi = 0.99999$ which was computed considering a forgetting window $w_2 = 100,000$ messages—i.e., $\phi = \left(\frac{w_2}{w_2 - 1}\right)$;
- proportional gain $K_p = 1,000$ which ensures stability and a maximum overshoot $M_p = 0.99$ (see Section 4.2).

Following we present the performance results for each of the six scenarios just described.

5.3.1. Distinct-fixed-workloads

In this evaluation, the simulation factors are the number of processes (10, 20, 30 and 40) and the time-silence period $ts$, for each load profile. The basic causal blocks mechanism with a fixed time-silence ($ts = 20$ ms and $ts = 200$ ms) is compared with the self-manageable version where the time-silence is dynamically adjusted to follow a set-point of relative resource consumption ($rc_D = 25\%$).

Table 1 shows the mean values and related confidence intervals (c.i.—confidence level of 95%) for protocol overhead, and mean and standard deviation (std. dev.) for delivery blocking time. In order to better visualize the variation of overhead in relation with the load variation, we measured the relative percent difference between high and low loads for each configuration (relative percent
difference, r.p.d., is calculated by dividing the absolute difference of two values by the average value of the same two values).

In scenarios with distinct number of nodes (from 10 to 40) for fixed timed-silence values, we can observe a higher variation in overhead (measured by r.p.d. metric). On the other hand, the self-manageable approach was able to achieve a much smaller variation (from 0.45% to 9.00%), almost meeting and never being greater than the desired overhead. Notice that the desired overhead ($h_{des}$) is calculated from Eq. (1) ($h_{des}$ = $c_{fp}$ * $h_{max}$) and because the set-point for percentage of resource consumption was set to 25% in these experiments (i.e., $c_{fp}$ = 25%), the desired overhead is equal to 22.5%, 23.75%, 24.167% and 24.375% for the configurations with 10, 20, 30 and 40 processes, respectively. This successful result is due to the fact that the controller component tries to balance the trade-off between blocking time and message overhead also as a function of the number of processes (see Algorithm 6).

5.3.2. Variable-workload

For different group sizes, we ran simulations where the load varies dynamically between low and high workload profiles. We observed that the operation point varies dynamically during the simulation. These results are presented in Table 2. Even in these dynamic scenarios, our self-manageable version followed the desired operation point for desired message overhead, despite the dynamic changes in workloads and group sizes. The results are quite similar to those of Table 1.

5.3.3. Changing set-point

Fig. 11 shows the trace of message overhead and blocking time for a simulation with $n = 20$ and medium workload where the desired percentage of resource consumption is dynamically modified from 25% to 40%. We can observe that the self-manageable protocol initially adjusts itself to a percentage of resource consumption equal to 25% ($c_{fp} = 0.25$) — that is equivalent to 23.75% in terms of message overhead (i.e., $c_{fp} * h_{max} = 0.25 * 19/20 = 0.2375$). After that, at time instant $t = 10,000$ ms, the desired percentage of resource consumption was increased to 40% which in turn caused a change in the desired message overhead to 38% (i.e., $c_{fp} * h_{max} = 0.40 * 19/20 = 0.38$). As can be observed, the self-manageable approach worked properly self-tuning its operation according to the dynamically defined set-points: changing to faster message delivery with an increase in message overhead.

5.3.4. Faulty scenario

We simulated the occurrence of process failures in a group with $n = 5$. To realize that, we forced the crash of 2 processes at $t = 1000$ ms which caused the execution of the group membership procedure to exclude these processes from the group. As we can observe in the Fig. 12, there is an increasing on blocking time and protocol overhead at this point due to the messages exchanged to converge to a new view, but our self-manageable approach, thanks to the dynamically adjustment of time-silence, produces a faster convergence to a new operating point, slightly improving blocking time, as explained in the following. When the processes crash, null messages for the completion of causal blocks are not generated by such faulty processes, thus messages are not delivered, which explains the peaks after $t = 1,000$ ms in the graph of Fig. 12(b). On other hand, when the crash is detected, the messages generated by the membership procedure cause an increasing in the overhead of the protocol, which explains the peaks in the curves of the graph in Fig. 12(a).

After the new group view is installed, the self-manageable approach senses the change in the number of group members and the controller executes two simultaneous actions. First, as the number of processes decreases from 5 to 3, the maximum overhead estimated by the controller decreases from 80% to 66.7% (i.e., from 4/5 to 2/3). This fact forces the decreasing of the operation point in terms of the protocol overhead. Second, the decrease in the number of group members causes a decrease in the mean resource consumption in the computing environment. This second fact forces the increase of the operation point in terms of the protocol overhead because there are more available resources in the computing environment (observe that, for a given message transmission rate, 3 processes consume less resources than 5 processes). The algorithm implemented by the controller balances these two effects and promotes a moderate adjustment in the operation point in terms of overhead from 15% to 10%, which explains the behavior in terms of mean overhead presented by the self-manageable approach in Fig. 12(a).

In addition, the decrease of resource consumption due to the decrease of the number of group members leads the controller to also decrease time-silence, providing faster message delivery, which consequently causes a decrease in message blocking time—see Fig. 12(b).
5.3.5. Amoeba-comparison

First we experiment with a group of size $n = 3$, with two distinct intervals for periodic messages and the low and high workload profiles. We observed that the performance and resource consumption of the non-autonomic version of the Amoeba protocol can vary significantly according to load profile. Table 3 presents these results.

Afterwards, we experimented with the self-manageable version of the Amoeba protocol, for the operation point $r_{CB} = 50\%$, with the consequent desired message overhead of $33.33\% = 50\% \times 2/3$ (worst case situation when no acknowledgment is piggybacked) and observed that the protocol self-configures itself following the provided operation point. The results for low and high load profiles are presented in Table 4.

5.3.6. Real network

In this scenario, the basic causal blocks mechanism with a fixed time-silence ($ts = 10$ ms and $ts = 500$ ms) is compared with the self-manageable version where the time-silence is dynamically self-adjusted to follow a set-point of relative resource consumption $r_{CB} = 40\%$. The workload was generated based on Burst Traffic in the following way. Each process continually alternates between sending a burst of a random number of application messages and sleeping for a random period of time. The number of messages for each burst was defined by a uniform distribution with mean of 100 messages and standard deviation of 10 messages. For the sleeping time, it was also utilized a uniform distribution with mean of 14 ms and standard deviation of 4 ms. In each experiment (it was replicated 3 times as explained earlier) each process transmitted at least 10,000. Therefore, it was transmitted 150,000 messages for calculating the statistics.

Table 5 shows the mean values and related confidence intervals (c.i.—confidence level of 95%) for protocol overhead as well as the maximum observed protocol overhead, and mean and standard deviation values (std. dev.) for delivery blocking time.

We observe that, as expected, larger time-silence values decreased message overhead. For the autonomic approach, we defined a desired resource consumption of 40\%, which is equivalent to a message overhead set-point of $40\% \times 4/5 = 32\%$. In this evaluation we can observe that despite the variation in the workloads,
the self-manageable protocol adapts itself to maintain the message overhead under the specified set-point, following the same behavior observed in the simulations.

5.4. Discussion

From the experiments we can observe two main advantages of the new self-manageable group communication approach. First, it allows the upper-layer application or user to dynamically control message blocking time and message overhead, by adjusting resource consumption set-points at runtime: if one application can afford more resources to attain faster message delivery, it can increase the percentage of resource consumption, or decrease it, otherwise. The second advantage is that for a fixed resource consumption (say, 25%), the autonomic protocol dynamically adjusts itself to varying system loads, application activity, and group configuration, keeping the desired balance between message blocking time and message overhead. It should be noticed that if system workloads increase, shortening time-silence might have the contrary (and undesired) effect on message delivery that can become slower in such conditions. However, the autonomic version is capable of sensing the new high load condition, increasing the time-silence value to maintain the right balance between cost and speed of the group communication protocol. These two characteristics are not possible in existing group communication protocols, because their configurations are done either offline or online assuming that the system load variations are known a priori. So, if such assumptions turn to be not valid at runtime (as it is the case in very dynamic environments such as cloud computing), the performance of the non-autonomic approaches will degrade as we saw in distinct-fixed-workloads and variable-loads scenarios.

Moreover, our approach allow us not only to self-adapt to system loads but also to new group configurations. For instance, if more group members are added, the protocol will naturally increase time-silence values to maintain a given protocol overhead (in terms of resource consumption).

6. Conclusions

Traditional group communication approaches for distributed systems do not support dynamic self-configuration of its operating parameters from user-defined requirements. However, when the behavior of the computing environment is unknown and can change over time, or when the application requirements can dynamically change, self-configuring is a basic issue that is required to deal with the trade-off between performance requirements such as speed (e.g., message delivery latency) and cost (e.g., protocol overhead). Self-configuring ability requires the modeling of the dynamic behavior of the distributed system which is a great challenge when the computing environment can change at runtime. To address these challenges, this paper presented a novel self-manageable group communication protocol based on feedback control theory, which is capable of self-configuring its operation parameters at runtime from previously specified performance requirements. Our approach integrates failure detection, message ordering and group reconfiguration, which facilitated the construction of the self-configuring mechanism. That was the reason for the chosen design with causal blocks.

We carried out a series of experiments to evaluate the performance of our self-manageable group communication approach in terms of speed (blocking time) and cost (overhead). These evaluations showed the protocol performance under distinct scenarios, and demonstrated how the protocol can dynamically adjust its operation according to the desired set-point (i.e., the user-defined resource consumption). When compared with the non-self-manageable version, the self-manageable approach produced better performance in terms of message delivery time and message protocol overhead. Although the autonomic or self-manageable approach has been presented for a specific group communication protocol, its basic principles and mechanisms can be applied to any existing group communication protocols, as the control actions are carried out on the frequency of transmission of monitoring messages (which is required by any group communication protocol in order to update membership views).

In future works, we plan to evaluate our self-manageable group communication approach in a cloud computing platform and integrate it with other mechanisms related to the management of autonomic applications under development at LaSiD [1,11].

References


<table>
<thead>
<tr>
<th>Table 5</th>
<th>Simulation results for the real network scenario.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factors</td>
<td>Results</td>
</tr>
<tr>
<td>ts period</td>
<td>Blocking time (ms)</td>
</tr>
<tr>
<td>Mean ± std.dev.</td>
<td>Mean ± c.i.</td>
</tr>
<tr>
<td>10</td>
<td>154.59 ± 70.38</td>
</tr>
<tr>
<td>500</td>
<td>313.91 ± 171.24</td>
</tr>
<tr>
<td>SELF</td>
<td>273.95 ± 233.65</td>
</tr>
</tbody>
</table>