OMAHA: Opportunistic Message Aggregation for pHase-based Algorithms

Célia Mahamdi*, Jonathan Lejeune*, Julien Sopena*, Pierre Sens* and Mesaac Makpangou*†
*Sorbonne Université, CNRS, LIP6, DELYS
F-75005 Paris, France
firstname.lastname@lip6.fr
†Inria

***Abstract*—In the cloud computing context, several applications run concurrently over the same underlying physical infrastructure. Phase-based algorithms are key building blocks for many distributed applications such as DBMS or transaction validation services. Indeed, these applications rely on consensus or atomic validation solved by phase-based algorithms (Paxos, ZAB, two-phase commit ...). In each phase, at least one participant broadcasts a message and waits for the responses from a subset of the recipients before starting the next phase. For a given phase-based algorithm, it is then possible to predict future communications for each node. Based on this observation, we propose a generic and low-intrusive solution to save network bandwidth in a cloud context by aggregating messages sent by applications in an opportunistic way. We propose a new API to easily apply our mechanism with applications using phase-based algorithms. The core of this API is the overloading of the *send* primitive where the users can define a trade-off between message saving and latency degradation. We evaluate our mechanisms using multiple instances of the same algorithm (3 variants of the Paxos consensus and the Zookeeper Atomic Broadcast algorithm) running concurrently. Our results show that a good tuning of the new *send* primitive saves a large amount of bandwidth with little latency degradation.**

***Index Terms*—phase-based distributed algorithms, message aggregation, experimental evaluation, Paxos**

I. Introduction

In a the cloud context, applications are deployed on a large-scale distributed infrastructure. Thanks to virtualization, many parallel applications run concurrently on the same shared physical support. Several articles reveal a high proportion of small messages in data centers, with a significant portion of bandwidth dedicated to message headers [1], [2], [3]. Indeed, Benson et al. [2] show that 50% of packets are less than 300 bytes. Furthermore, studies highlight network bandwidth as one of the primary bottlenecks for the core network infrastructure of data centers [4].

To address this issue, aggregation mechanisms can be activated in the lower layers of the network stack. The principle is to multiplex several messages addressed to the same destination. However, this strategy is application agnostic, which prevents smart message aggregation. Indeed, the network layer does not know if the sending of a message is blocking and critical for the liveness of the application or if it can be delayed without performance degradation.

Many distributed applications use *phase-based* algorithms, especially in data management. These algorithms execute a sequence of steps, where the step $i+1$ starts after the completion of all or a part of the step i. Consensus (Paxos [5]), atomic validation protocols (Zookeeper Atomic Broadcast usually named as ZAB [6], or two-phase commit [7]) are widely used protocols that rely on phase-based algorithms. Although these algorithms are essential for many applications (DBMS, Google Spanner [8]), they have nonetheless a high message complexity implying a non-negligible degradation of the bandwidth. For instance, the Paxos algorithm [5], one of the most implemented consensus algorithms [9], relies on a set of broadcasts to all participants where each step is synchronized by the waiting for a quorum in response messages. There are many versions of the Paxos protocol [10][11][12][13] but, in its most widespread version, the message complexity is quadratic with the number of participants.

In phase-based algorithms, the steps are known in advance. Thus, it is possible to know if a node will communicate with another in the near future, regardless of the application. Considering this knowledge, we are able to determine whether it is relevant to buffer a message and thus delay it sending. Based on this observation, we have developed a generic, opportunistic and non-intrusive message buffering mechanism which is applicable in a context where several applications run concurrently and independently on the same infrastructure. Our mechanism can significantly reduce message complexity and network bandwidth while limiting latency degradation. It provides an intermediate layer between applications and the network stack, overloading the traditional communication API. The core of this new API is the specification of a new *"send"* primitive that offers users the ability to tune a trade-off between bandwidth saving and latency degradation according to their needs.

We evaluated our solution with multiple instances of 4 phase-based algorithms (3 variants of Paxos consensus and the ZAB commit protocol). We show that a good tuning of our mechanism saves up to 30% percent of bandwidth with as little as 5% latency degradation.

The paper is organized as follows. Section II outlines-related work, in Section III we present our aggregation mechanism, and Section IV describes our experimental evaluation. Finally, Section V concludes the paper and introduces some future research directions.

II. Related Work and background

This section presents existing aggregation mechanisms and gives an overview of the Paxos protocol as an example of phase-based algorithm. The Paxos protocol will be used in Section III-D to illustrate a use case of our mechanism and in Section IV for the experimental study. [1].

A. Messages aggregation at network layers

Traditional network aggregation techniques are based on message piggybacking. The TCP protocol provides an aggregation mechanism based on the Nagle algorithm [14] which is enabled by default in most TCP implementations. Since TCP/IP packets have a 40-byte header, sending a large number of small messages can lead to network overhead and congestion. The main idea behind Nagle's algorithm is to buffer data until the acknowledgement is received or the buffer is full.

Badrinath and Sudame introduce another aggregation mechanism called Gathercast [15]. Acknowledgment is often used in reliable communications to ensure that any message has been delivered correctly to the recipient. Since acknowledgments cause communication overhead, many protocols aggregate them into a single packet [16]. Gathercast aggregates small control packets (as TCP ACK messages) addressed to the same host in a similar way to the TCP Nagle's algorithm. Packets are delayed until a timer expiration.

However, at the network layer, applications are isolated from each other by using different ports. It is then impossible to aggregate messages from different applications. Moreover, all these mechanisms are application agnostic which leads to negative effects on time-sensitive applications (e.g. real-time applications such as chat, streaming, etc.) that do not tolerate message delaying.

B. Messages aggregation in application layers

Another way to aggregate messages is to take into account the application's protocol. For instance HotStuff [17], a leader-based Byzantine fault-tolerant replication protocol,saves messages by piggybacking phases of consecutive consensus instances. The message aggregation mechanism is only applied to a single application instance. To the best of our knowledge, we are not able to find an aggregation mechanism which can be activated opportunistically and which multiplexes messages of any running application.

C. Paxos algorithm

The Paxos algorithm [5] is a leader-based, fault-tolerant algorithm that solves the consensus problem where correct processes must agree on some proposed values. It assumes asynchronous (i.e., no bounds on the transmission delay) and unreliable communications. The set of participants is statically fixed and tolerates f participant crashes.

A Paxos instance begins when the leader starts a new ballot. An instance execution is divided into three phases:

[1]ZAB and 2 other Paxos variants are studied too but not described here due to lack of space

- *Preparation phase:* the leader sends a *prepare* message with a new ballot number b to all participants. When a participant p receives a *prepare* message, it agrees to join the ballot b if and only if b is greater than the most recent ballot number in which p has already participated. An *ack* message is then sent to the leader.
- *Acceptance phase:* when the leader learns that a quorum of $f + 1$ participants has accepted to join its ballot, it sends an *accept* message to all participants. When a participant receives an *accept* message, it broadcasts to all participants an *accepted* message only if it does not take part in another more recent ballot.
- *Decision phase:* when a participant receives a quorum of *accepted* messages for the same ballot number, then it decides the definitive value.

III. Aggregation mechanism

This section presents Omaha, our contribution to aggregate opportunistically messages sent by applications using phase-based algorithms. We assume that each node and application have a unique identifier.

In order to achieve any aggregation mechanism, we assume that each node maintains for each destination a message buffer. Such a mechanism must then answer two questions:

- Should a new application message be buffered (and therefore delayed) or sent immediately?
- When should buffered messages be sent over the network?

In Omaha, the answer to the first question depends on the criticality of the message for the liveness of the application algorithm. To answer the second question, we leverage the phase-based protocol to delay messages without slowing down the application. Anyway, the buffering time must be bounded to ensure that messages will be sent eventually.

We first describe a time-based approach which will be our baseline comparison in the experimental study (Section IV). Next, we present our opportunistic approach by applying it to the Paxos protocol. Finally, we detail our new API.

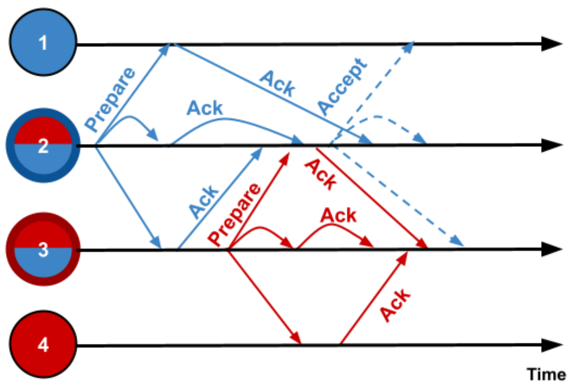
A. The classic time-based approach

The time-based approach systematically aggregates messages for a static period of time or until the buffer is full. Therefore, the answer to the first question is to always buffer messages. The answer to the second question is an arbitrary choice of buffering time. This approach is simple to implement and saves bandwidth by significantly reducing the number of messages sent. However, it is application agnostic and therefore delays all application messages. As a result, it does not take into account the criticality of messages, which can lead to some blocking messages being buffered and application latency being severely degraded. Furthermore, it does not take the system load into account. Thus, it is possible to buffer (and delay) a message even if no future sending to the same recipient is planned, which is useless.

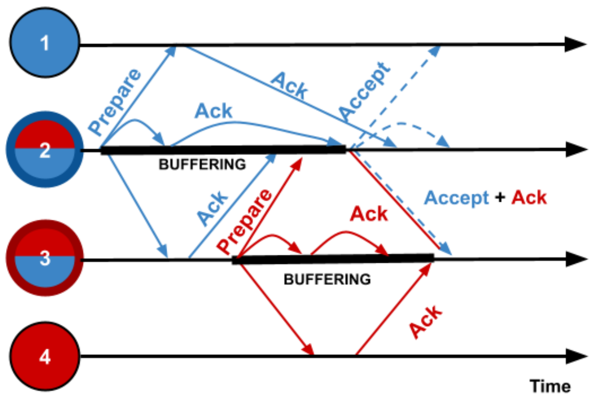
B. Our opportunistic approach

The idea behind our approach is to exploit the knowledge of future message sending for smart buffering. Thus, we are able to find a good trade-off between latency degradation and bandwidth gain for any load.

Figure 1 illustrates the principle of our approach by applying it to the Paxos protocol [5] in a multiple application context. Each application is independent of another one and runs on its own set of physical nodes. However, we assume that these sets intersect.



(a) With no aggregation mechanism



(b) With the opportunistic aggregation mechanism

Fig. 1. Beginning of two Paxos instances

In this example, we consider two sets of nodes (each one represents a running application). The blue set and the red set are composed of nodes $1, 2, 3$ and $2, 3, 4$ respectively. Each set, according to its needs, can launch instances of the Paxos algorithm over time. Here, nodes 2 and 3 are the leaders of the blue and red set respectively. Figure 1(a) shows an execution without aggregation mechanism and where each set runs independently. Node 2 first runs a Paxos instance for the blue set, and then node 3 runs an instance for the red set.

To apply our opportunistic mechanism, it is first necessary to know when it is relevant to delay a message. We observe that once node 2 broadcasts a `prepare` message to the blue set (beginning of the phase 1), it will soon broadcast an `accept` message as soon as it receives a quorum of `ack` messages (beginning of the phase 2). The protocol ensures that any participant which sends a `prepare` message will contact the same set of recipients in a short term (once the quorum is reached). It is therefore possible to exploit this knowledge to buffer messages from another application addressed to the nodes belonging to the first set of recipients. We call this mechanism a **pledge**: a node can buffer and delay a message if it commits to sending it eventually. In Figure 1(b), we can see that node 2 intersects the two sets. Then, the pledge mechanism detects that it is possible to buffer the `ack` message of red set addressed to node 3 and aggregates it with the `accept` message of the blue set at the beginning of its phase 2.

To summarize, we are able to predict when a node A will send a message to a node B thanks to knowledge of the algorithm's phases. So, for a given instance, during the waiting time to start the next phase, it is possible to aggregate any message from A to B issued from any other application. As this waiting time is inherent to the algorithm, it is possible to overlap this mandatory latency with message aggregation.

C. A new network API

Omaha can be used as a library. To improve performance, Omaha can be deployed in the kernel or at hypervisor level. In this way, we avoid the bottleneck associated with a potential middleware layer. The pseudo-code of the pledge mechanism is given in Algorithm 1 which can be applied to any phase-based algorithms. We overload the network API by adding to the *send (msg, dests)* primitive three new arguments (line 21):

- `probaBuf`: the probability to aggregate the message. Zero means that the message will be sent immediately (as in the original *send*) while 1 (100%) means that the message will be delayed and buffered. Considering a probability rather than a boolean value allows to take into account more precisely the criticality of the message. This parameter therefore controls the latency degradation. The higher this value is, the more likely it is to buffer messages and thus increase latency.
- `timeout` (denoted `t` in pseudo-code): the maximum time that the message will remain in the buffer. This ensures that a delayed message following a pledge will eventually be sent, thus ensuring the safety and liveness properties of the application's algorithm.
- `app`: the id of the application.

During a waiting period (e.g., quorum waiting), the node is in "*pledge period*". Two primitives allow the application to declare the beginning and the end of a pledge period in its algorithm:

- `beginPledge(app, futureDests)`: this primitive declares the beginning of a pledge period for the application `app` (line 6). `futureDests` is the set of nodes that will be contacted at the end of the pledge period. It is then possible to buffer any message addressed to these nodes.

- `pledgedSend(msg,dests,probaBuf, timeout, app)` : indicates the end of a pledge period for the application `app` (line 16) and the sending of the `msg` message. This message follows the same sending rules as the others by calling the overloaded *send* primitive.

These primitives specify an API for a new intermediate layer between the network and the application layers. In this layer, each node maintains a buffer for each recipient node (line 4). Each buffer has a deadline indicating when the buffer will be flushed to send messages to the destination node. This deadline is computed according to the timeout defined for each message (lines 28 and 29). When the *send* primitive is called, two cases occur:

- if `probaBuf = 0` then the message must be sent directly. Therefore, if the buffer associated with the recipients is not empty, then the message is aggregated with the other ones included in the buffer (line 25) and the whole is sent immediately (lines 36 and 13).
- if `probaBuf > 0` then the decision to buffer the message or not depends on the `probaBuf` value and if the sending node is in "pledge period" for the recipients (line 26). Thus :
  - if there is no pledge period for a recipient r, then the message is sent immediately to r ((lines 36 and 13)) .
  - otherwise with probability `probaBuf` the message is added to the buffers of each destination and their associated deadlines are updated (lines 25 to 30). In other words, the message and those already buffered are sent immediately with probability 1-`probaBuf`.

Finally, when a deadline of a buffer is met, all its messages are sent as a single network message.

D. An example applying the pledge mechanism to Paxos

This section shows how to link our Omaha layer to an application. Calls to the pledge API must be integrated into the application's algorithm by identifying the start and end of a pledge period, then using `beginPledge` and `pledgedSend` functions, respectively.

We illustrate this with the example of Paxos algorithm. For the sake of simplicity, we focus on the *preparation phase* up to the beginning of the *acceptance phase* of Algorithm 2. We assume that the application id is known (line 6) and that each message type is associated with a maximum buffering time (line 7) and a buffering probability (line 8).

As explained in Section II and illustrated in Figure 1(a), when the leader starts a new instance of Paxos (line 9), it sends a prepare message (line 12) to all participants of this application. Then, to initiate the next phase, the leader must wait for a majority of *ack* messages. This is notified to the Omaha layer by calling the `beginPledge` function (line 13). When the leader receives a majority of *ack* messages (line 22), it stops waiting and sends the *accept* message that notifies the Omaha layer of the end of the pledge period by calling `pledgeSend` function (line 29).

Algorithm 1: The pledge mechanism algorithm

```
1  Local variables :
2  begin
3      $curPledges$ : $Map$ of ($app$: application id, $nodes$:$Set$ of
         node ids)
       /* map associating an application id with a
          set of node ids for which we know they
          will be contacted in a near future      */
4      $bufs$ : $Map$ of ($nodeid$,($msgs$ : $Set$ of $Message$,
         $deadline$))
       /* map associating a recipient id with the
          list of buffered messages that are
          intended for it and the associated
          deadline of sending                     */
5  end

6  Primitive beginPledge($app$, $futureDests$) :
7  begin
8      put($app$,$futureDests$) in $curPledges$
9  end

10 Primitive sendBuff($buff\_dest$, $dest$):
11 begin
12     cancel any scheduling related to $buff\_dest$
13     networkSend($buff\_dest.msgs$) to $dest$
14     clear $buff\_dest$
15 end

16 Primitive pledgedSend($msg$, $dests$, $probaBuf$, $t$, $app$):
17 begin
18     removeEntry($app$) in $curPledges$
19     send($msg$, $dests$, $probaBuf$, $t$, $app$)
20 end

21 Primitive send($msg$, $dests$, $probaBuf$, $t$, $app$):
22 begin
23     for $all\ d \in dests$ do
24         $flushing \leftarrow true$
25         add $msg$ to $bufs[d].msgs$
26         if $\exists(app', nodes) \in curPledges\ where\ d \in nodes\ and$
             $app \neq app'$ then
27             if $random() < probaBuf$ then
28                 if $bufs[d].deadline\ does\ not\ exist\ or$
                     $bufs[d].deadline > now + t$ then
29                     $bufs[d].deadline \leftarrow now + t$
30                     schedule sendBuff($bufs[d]$, $d$) at
                         $bufs[d].deadline$
31                 end
32                 $flushing \leftarrow false$
33             end
34         end
35         if $flushing == true$ then
36             sendBuff($bufs[d]$, $d$)
37         end
38     end
39 end
```

IV. Experimental study

A. Experimental testbed and configuration

This section describes our experiment set up environment.

1) Infrastructure settings: To ease the analysis, the experiments were first carried out with Peersim [18], a discrete events simulator (Sections IV-B to IV-F). To validate our results in a real infrastructure, we then test Omaha on the Grid'5000 platform (a.k.a. g5k) [19] (Section IV-G). In both platforms, we deployed 15 nodes. In g5k platform, each node

Algorithm 2: Preparation phase of the Paxos algorithm for a participant p_i

```
Original local variables to Paxos :
$ballot$ : ($numBallot$, $p_k$) initially (0,$\perp$)
$acceptBal$ initially $\perp$
$acceptVal$ initially $\perp$
Additional local information required to use the Omaha layer
$app\_id$
$timeout$ : Map of (message type, timestamp)
$probaBuf$ : Map of (message type, probability)

Upon Propose (new_val) :
begin
    $ballot \leftarrow$ Ballot($ballot.numBallot$++, $p_i$ )
    send(<$Prepare$, $ballot$ >, $set_k$, $probaBuf[Prepare]$,
      $timeout[Prepare]$, $app\_id$)
    beginPledge($app\_id$, all participants of $app\_id$)
end

Upon reception of message Prepare($bal$) from $p_j$:
begin
    if $ballot \leq bal$ then
        $ballot \leftarrow bal$
        send($< Ack, bal, acceptBal, acceptVal >$ , $\{p_j\}$,
          $probaBuf[Ack]$, $timeout[Ack]$, $app\_id$)
    end
end

Upon reception of message Ack ($bal$, $acceptBal$, $acceptVal$)
  from a majority of participants :
begin
    if $all\ acceptVal$ = $\perp$ then
        $val \leftarrow new\_val$
    else
        $val \leftarrow$ the value associated with the biggest $bal$ for all
          $acceptBal$
    end
    pledgedSend($< Accept, ballot, val >$ , all participants of
      $app\_id$, $probaBuf[Accept]$, $timeout[Accept]$, $app\_id$)
end
```

is deployed on a dedicated physical host [2]. We consider an average round-trip time (denoted RTT below) of 60 milliseconds that follows a normal distribution with a standard deviation of 10%. In g5k, network latency has been injected to obtain the same setting. We consider a complete communication graph where any node is able to communicate with any other. In g5k, nodes communicate using TCP/IP sockets.

Although Paxos is compatible with an unreliable environment, we first assume, in Sections IV-B, IV-C, IV-D and IV-E, a reliable system (all nodes never crash and execute the protocols correctly, there is no loss nor duplication of messages) in order to ease the analysis of the results by focusing only on the impact of the aggregation mechanism. Next, in Section IV-F, we study the impact of an unreliable network with different rates of message loss. Indeed, as our mechanism aggregates several application messages into a single network message, it may be more sensitive to information loss.

2) Considered application phase-based algorithms: We consider four application phase-based algorithms: three variants of the Paxos algorithm and the Zookeeper Atomic Broadcast algorithm (ZAB). Their characteristics are summarized in Table I.

	Classical Paxos[5]	Fast Paxos[11]	Fast Byzantine Paxos (FBP) [20]	ZAB [6]
Steps	4	2	4	3
Nodes	$2f+1$	$3f+1$	$5f+1$	$2f+1$
Quorum size	$f+1$	$2f+1$	$3f + 1$ or $4f + 1$ (according to phases)	$f+1$
Number of pledges periods	1	1	3	1

TABLE I
SUMMARY OF THE PROTOCOLS FOR A SYSTEM WITH f FAULTY NODES

3) Aggregation mechanisms and parameters: We compare Omaha to the original algorithm without any aggregation mechanism, and the time-based aggregation mechanism (cf. III-A) which systematically buffers messages.

The impact of the two following parameters are investigated:

- The $timeout$ parameter which defines the maximum time a message can spend in a buffer before being sent. It applies to both Omaha and the time-based approach.
- The $probaBuf$ parameter which defines the probability that a message is buffered by the Omaha mechanism if a *pledge* can be applicable.

Note that these two parameters can be set by the user for each message (using the overloading of the *send* primitive, Section III-C). In all experiments, we consider that these two parameters are statically set and never change during the experiment execution. In Sections IV-B, IV-C, IV-D, and IV-F, we consider that these parameters are constant whatever the phase and whatever the message type. However, in Section IV-E we consider different values of $probaBuf$ depending on the criticality of the message type.

4) Workload: The performance of the aggregation mechanisms is directly related to the network load and therefore to the number of application algorithm instances running simultaneously. In the following experiments, we evaluate each mechanism with three load patterns: low, medium, and high. Each pattern corresponds to an average of 2, 5 and 10 concurrent running instances of the same algorithm, respectively. All 15 nodes participate in each instance. The concurrent instances are not synchronized, i.e., one instance can start running the protocol independently of the state of the other running instances. They are therefore not all in the same phase at the same time. Moreover, for each instance, we arbitrarily choose its leader node before running the Paxos protocol.

5) Metrics: To compare the efficiency of each aggregation mechanism, we define the following two metrics:

- The **average latency** to run an instance of an algorithm, *i.e.*, the time between the moment when a node initiates the protocol and the moment when a quorum of nodes is reached (i.e., nodes agree on a value in the case of Paxos

[2]Configuration of a g5k physical host : 2 CPUs Intel Xeon E5-2660 8 cores/CPU, 64GB RAM, 1863GB HDD, 1 x 10Gb Ethernet, running Linux 5.10.0-16-amd64 with Java 11

consensus algorithms, or a transaction commitment in the case of ZAB)

- The **bandwidth consumption** which is the total amount of data produced by the network layer (IP) during the whole experiment execution.

Each experiment ends when the 3000th instance of the algorithm is completed. The first 100 instances are discarded from the measurements in order to consider a stationary and stable load value. Note that in the following tables and plots, the values are not absolute but relative to the performance of a system without an aggregation mechanism. Thus, these two metrics are expressed respectively in terms of latency degradation (the lower value, the better) and bandwidth consumption saving (the higher value, the better).

We observed little variation of the latency between each instance, with the highest standard deviation (3 for a mean latency of 130.1) measured in high load configuration when $probaBuf$ has a value of 100.

B. Impact of the $probabuf$ parameter

Tables II and III present the impact of Omaha on classical Paxos instances by varying workload and $probaBuf$ parameters while keeping a constant $timeout$ parameter equal to one RTT.

In Table II, we observe that the bandwidth saving increases linearly with the pledge probability, whatever the load of the system. This was quite predictable since the probability value is the same for all types of messages.

Depending on the load, the bandwidth saving varies: in a high load pattern, we observe that the bandwidth saving is higher. In this case, many instances of Paxos are running concurrently and more pledges can be combined. Conversely, when the load is low, few Paxos instances are running concurrently and Omaha is unable to predict future communications, so the bandwidth saving is lower.

Table III shows the effect of the Omaha mechanism on the latency of the algorithm. We can see that the impact of the probability is not the same for different loads. As explained previously, when the load is high, many pledges can combined. This can lead to additional delays in message transmission. Nevertheless, bandwidth saving of up to 30.2% can be achieved with only a slight degradation in latency (5.4%).

C. Impact of the $timeout$ parameter

We now study the impact of the $timeout$ parameter on the classical Paxos algorithm and show the results in Tables V and IV. We vary the value of the $timeout$ with a constant buffering probability equal to 90%.

In Table IV, we can see that the bandwidth saving follows broadly the same evolution for all load values. First, between $RTT/4$ and RTT, the bandwidth gain increases. Then, the gain slows down and stabilizes. Indeed, since the duration of a phase is one RTT on average, a timeout value greater than one RTT will never expire because message sending is mainly due to the pledge mechanism.

In Table V, we observe very little latency degradation, especially in the low load pattern where few messages can be aggregated. At higher loads, more messages can be saved, so the latency degradation is greater.

D. Study of the trade-off between messages and latency

In this section, we study the trade-off between bandwidth saving and latency degradation with different settings of the two parameters. We consider only the high load pattern, since in the other cases, our mechanism has less impact on latency and the trade-off is easier to find. Figure 2 shows the results.

For one simulation, the overall bandwidth consumption values for Paxos, Fast Paxos, Fast Byzantine Paxos and ZAB without aggregation mechanism are 123.525 Mo, 105.12 Mo, 302.22 Mo, and 22,04 Mo respectively.

The x-axis and the y-axis correspond to bandwidth saving and latency degradation, respectively. The shape of dots represents a given timeout value while the color represents the value of $probaBuf$. Note that the black color is dedicated to the time-based aggregation mechanism as a baseline.

First, there is no linear correlation between the two metrics. We can observe a positive correlation, with points very close to a Pareto front. We also note the absence of outliers. This absence is explained by low standard deviation values. On average, a Paxos takes 120 ms, with a worst-case standard deviation of 1.8 ms. This, shows the stability of the aggregation mechanisms despite the jitter injected into the experiment.

Second, whatever the Paxos variant, latency degradation is limited as long as the bandwidth saving remains below 50%. We could think, in a first quick analysis, that this phenomenon is due to the size of the quorums. For example, if only 50% of responses are expected, it is possible to aggregate (and therefore potentially delay) 50% of the messages without degrading latency. Following this reasoning, a degradation should appear later for Fast Byzantine Paxos whose quorum size is larger (see Table I).

However, it should be noted that each type of message can be delayed, as the $probaBuf$ is the same for all messages. This means that messages essential to the progress of the algorithm will also be delayed, resulting in a degradation of the latency.

Even if the quorum size for Fast Byzantine Paxos is larger than that of Paxos, Fast Byzantine Paxos has more phases and therefore more quorums to collect, allowing more pledges to be made (and thus a greater bandwidth gain). While the size of the quorum has an impact on the mechanism, the efficiency of Omaha is due more to the presence of pledges and their fulfillment.

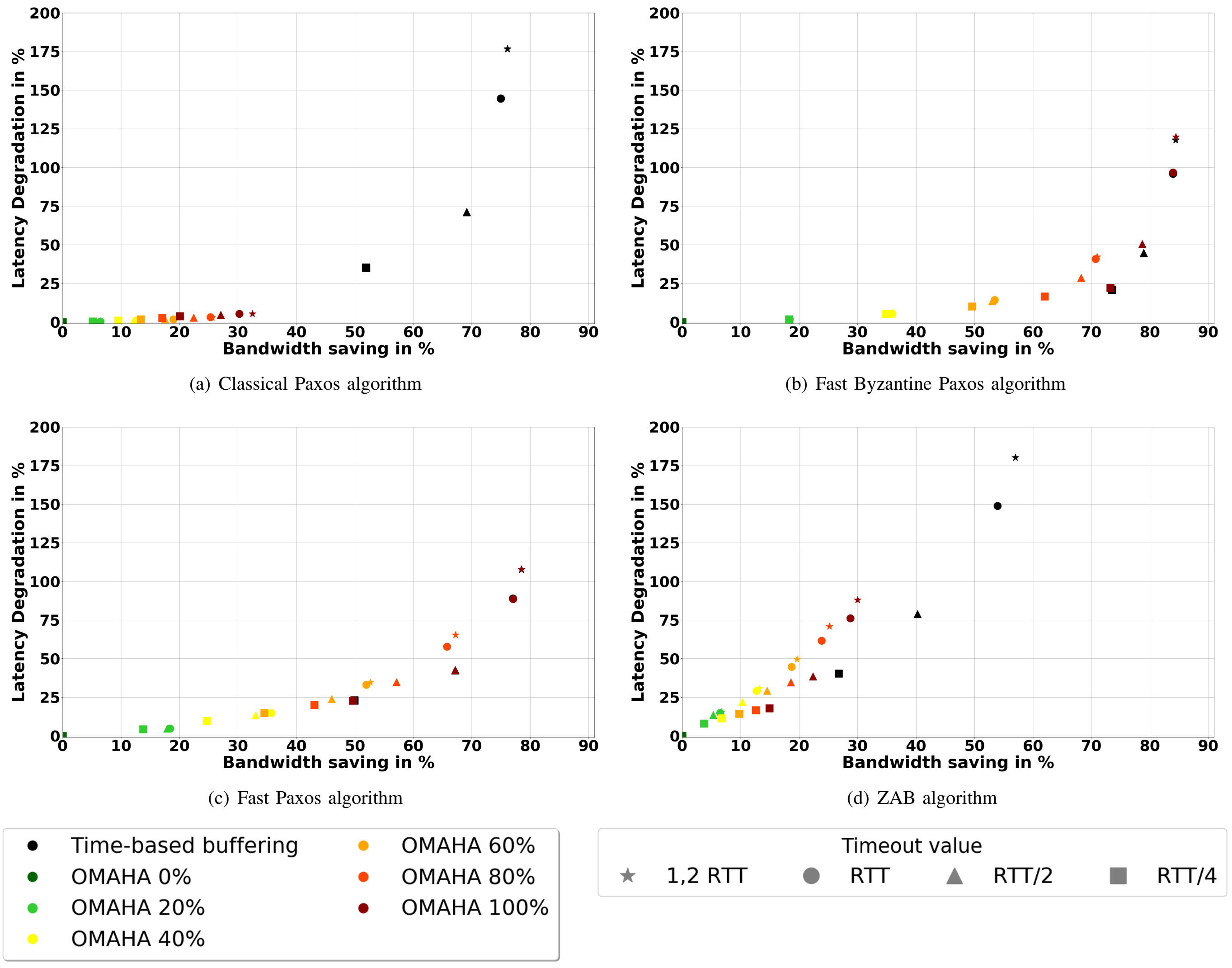
We can see the effect of the distribution of pledges by comparing the scatterplots of Figures 2(a) and 2(b), where we can see that Omaha is more efficient with FBP than Paxos. This can be explained by the centralized aspect of the original Paxos algorithm: most phases are executed by the leader, who centralizes the pledges. Inn FBP, on the other hand, all nodes can make pledges. The aggregation power is therefore well distributed between nodes.

Buffering probability Load	10	20	30	40	50	60	70	80	90	100
Low	0.6%	1.2%	1.8%	2.3%	2.9%	3.5%	4.0%	4.5%	5.1%	5.6%
Medium	1.6%	3.1%	4.6%	6.1%	7.6%	9.1%	10.5%	12.0%	13.4%	14.7%
High	3.4%	6.4%	9.5%	12.6%	15.9%	18.9%	22.2%	25.3%	28.1%	30.2%

TABLE II
IMPACT OF THE $probabuf$ PARAMETER ON THE CLASSICAL PAXOS ALGORITHM: BANDWIDTH SAVING

Buffering probability Load	10	20	30	40	50	60	70	80	90	100
Low	0.0%	0.0%	0.0%	0.1%	0.1%	0.2%	0.2%	0.3%	0.3%	0.3%
Medium	0.1%	0.3%	0.4%	0.5%	0.7%	0.8%	1.0%	1.2%	1.4%	1.6%
High	0.2%	0.4%	0.7%	1.0%	1.4%	1.8%	2.4%	3.2%	4.2%	5.4%

TABLE III
IMPACT OF THE $probabuf$ PARAMETER ON THE CLASSICAL PAXOS ALGORITHM: LATENCY DEGRADATION



(a) Classical Paxos algorithm

(b) Fast Byzantine Paxos algorithm

(c) Fast Paxos algorithm

(d) ZAB algorithm

Fig. 2. Pareto considering a high load

Similarly, ZAB is also a centralized algorithm but very few messages are sent. Unlike Paxos, the leader initially proposes a transaction to only a quorum of nodes (and not to all nodes). After receiving an ack from all nodes of the initial quorum, the leader sends a commit messages to each node. Delaying any messages sent by a ZAB quorum node would result in a significant degradation in latency, visible in Figure 2(d). By modifying the settings of Omaha, it is possible to improve

Timeout - Load	RTT/4	RTT/2	RTT	1.2RTT
Low	1.3%	3.7%	5.1%	5.2%
Medium	5.6%	9.9%	13.4%	13.8%
High	18.7%	24.9%	28.1%	29.2%

TABLE IV
IMPACT OF THE TIMEOUT PARAMETER ON THE CLASSICAL PAXOS ALGORITHM: BANDWIDTH SAVING

Timeout - Load	RTT/4	RTT/2	RTT	1.2RTT
Low	0.4%	0.3%	0.3%	0.2%
Medium	1.4%	1.3%	1.4%	1.3%
High	3.2%	3.8%	4.2%	4.1%

TABLE V
IMPACT OF THE TIMEOUT PARAMETER ON THE CLASSICAL PAXOS ALGORITHM: LATENCY DEGRADATION

performance (see Section IV-E).

In conclusion, we have shown that Omaha achieves a good bandwidth gain while keeping a reasonable latency degradation. Its efficiency depends on the phase patterns of the algorithm and a good parameter setting.

E. Parameter settings

Users must define the parameters of the *send* primitive. As a hint, we decided to set the buffering probability based on the importance of the messages.

Some messages are essential to the progress of the algorithm, and delaying them can lead to a significant degradation in latency. In the Paxos protocol, this is the case for the *prepare* and *accept* messages sent by the leader. On the other hand, some messages can be delayed without impacting the algorithm when nodes are trying to reach a quorum of responses. In the Paxos protocol, this is the case for the *ack* and *accepted* response messages sent by participants. Any additional message received after the quorum has been reached is useless because it has no impact on the liveness of the algorithm.

Response messages must have a probability close to $1 - \frac{size(quorum)}{\#nodes}$ to ensure enough message receiving. Thus, for a large quorum, e.g 2/3 of the nodes, the response messages will have a low probability of being delayed, e.g., 33%.

Figures 3, 4, and 5 show the results when we adapt the buffering probability for the Paxos, FBP and ZAB algorithms, respectively, considering 3 load patterns.

In Paxos (Fig. 3), we choose a value of 45% for response messages. This value is slightly lower than the quorum size (50% of nodes). For the critical messages, we assign the buffering probability to 25%. We observe that adapting the buffering probability increases the bandwidth saving with almost the same latency degradation.

Since Fast Byzantine Paxos has larger quorums (60% and 80% according to the phases), we choose a buffering probability of 35% for response messages and 15% for critical messages. Using this setting, we observe in Figure 4 a significant decrease of the latency for a larger gain in bandwidth.

As explained previously, the setting is crucial for ZAB. Before committing a transaction to every node, the leader only addresses a quorum. These messages cannot be delayed without significantly degrading the latency as seen in Figure 2(d). Thus, we decide not to buffer the critical message and to buffer the other messages with a probability of 40%. With this setting, we observe in Figure 5 a high reduction in the latency degradation but a limited bandwidth saving.

F. Unreliable network

Since we aggregate multiple application messages into single network messages, we study the impact of message losses on Omaha when running several instances of the Paxos algorithm.

Each time a prepare or accept message is sent, Paxos arms its own timeout to detect possible losses. When this timeout expires, the messages are resent. To avoid false detections, we set the Paxos timer to $RTT + 2 * timeout$. The RTT corresponds to the duration of the phase and the $timeout$ is the parameter of the *send* primitive.

In Figure 6, we present Pareto fronts considering different message loss rates. We can observe that message loss has a limited impact on Omaha performance.

There is no significant degradation of latency for a loss rate of 1%. When the loss rate increases, we observe a significant increase in latency proportional to the bandwidth saving. Indeed, a high bandwidth saving induces an increase in the number of buffered messages sent by multiple instances of Paxos. The more likely loss of a single message of the physical network can then slow down several applications.

G. Experiment on Grid'5000 platform

This section presents results of experiments conducted on the g5k platform (see section IV-A1 for the platform settings).

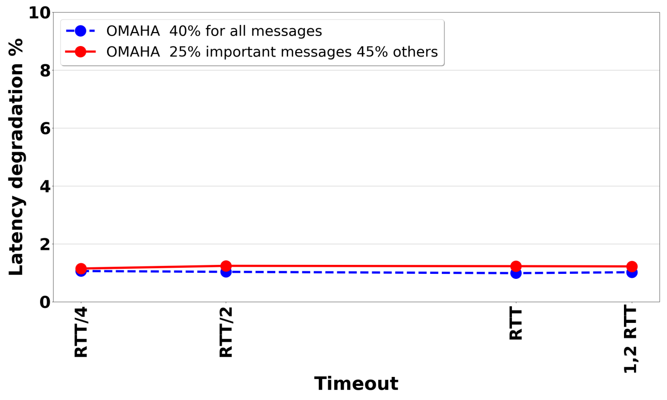
For this experiment, we mix the same number of instances of the Paxos, ZAB, Fast Paxos and Fast Byzantine Paxos protocols. When an instance starts, the protocol is chosen randomly with a uniform distribution.

Figure 7 shows Pareto of each protocol running concurrently on the same infrastructure.

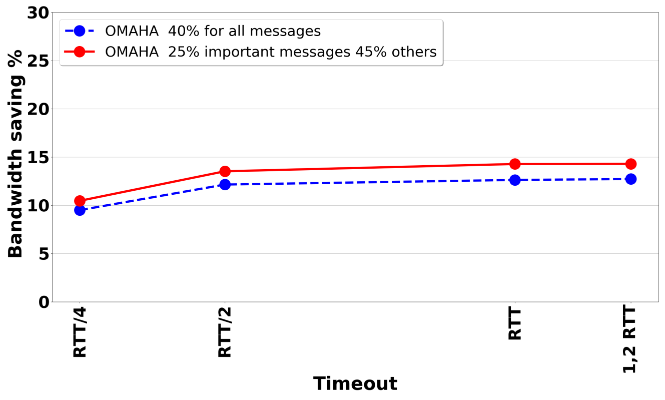
In the Figure 2(d), we observed that Omaha was less efficient for ZAB compared to the other protocols. As explained in Section IV-D, ZAB generates few pledge periods. However, this new experiment shows that ZAB is able to exploit pledge periods from other protocols to greatly decrease its bandwidth cost. Among all the protocols, ZAB achieves the best trade-off between bandwidth saving and latency degradation.

If we compare ZAB behavior in Figures 7 and 2(d), the results are very different. For the same configuration (probaBuf of 40%, one RTT timeout, black triangle in Figure 7 and yellow circle in Figure 2(d)), we observe that the bandwidth saving increases from 12.7% to 28.5% whereas latency degradation is reduced from 29.1% to 6.5%.

This last result is encouraging and shows that Omaha could benefit all phase-based applications currently running in the system.

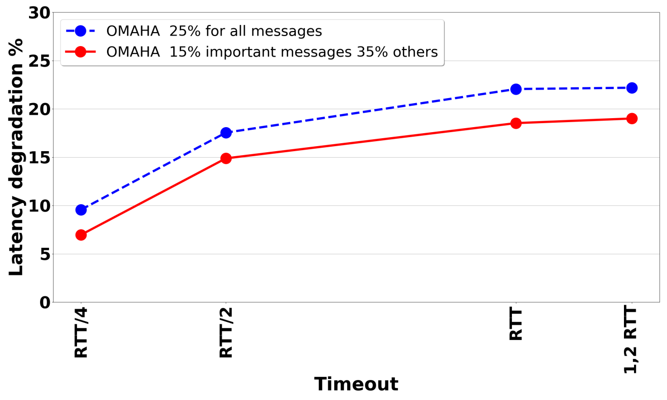


(a) Paxos high load latency degradation

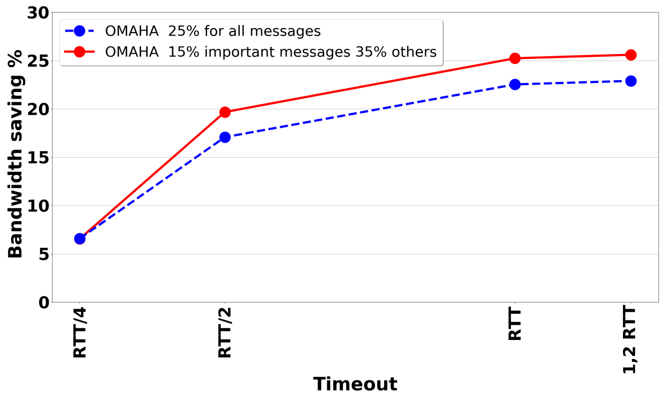


(b) Paxos high load bandwidth saving

Fig. 3. Paxos adaptation of buffering probabilities

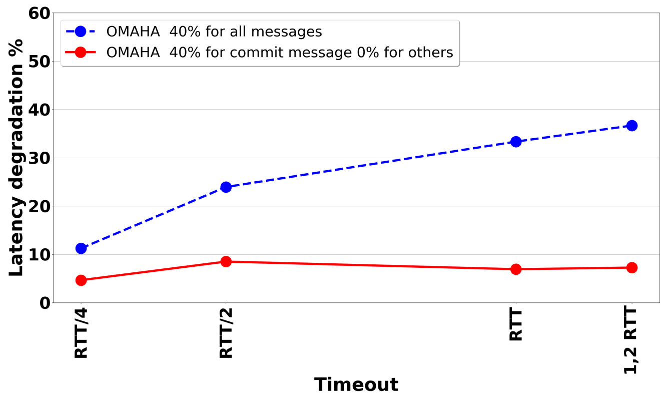


(a) FBP low load latency degradation

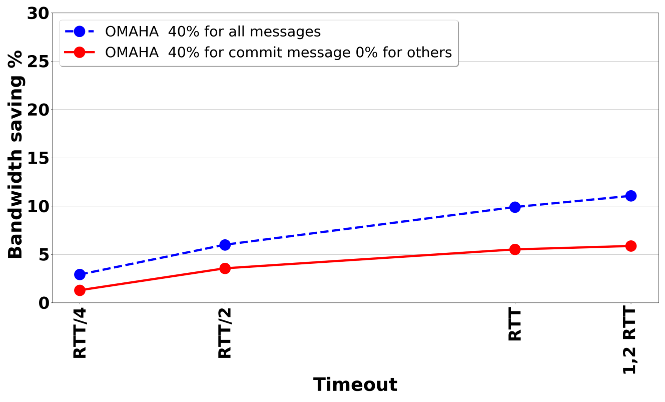


(b) FBP low load Bandwidth saving

Fig. 4. Fast Byzantine Paxos adaptation of buffering probabilities



(a) ZAB medium load latency degradation



(b) ZAB medium load Bandwidth saving

Fig. 5. ZAB adaptation of buffering probabilities

V. Conclusion and Future Works

This paper proposes Omaha an opportunistic message aggregation mechanism allowing to find a trade-off between

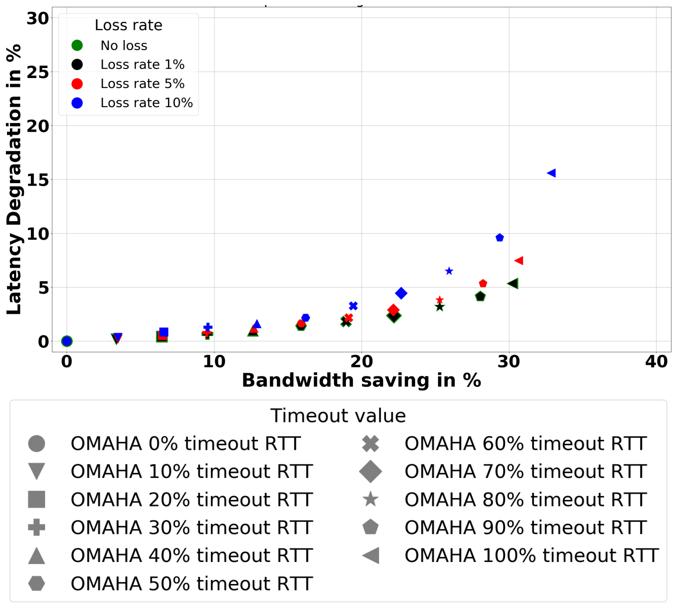


Fig. 6. Impact of messages loss on Paxos

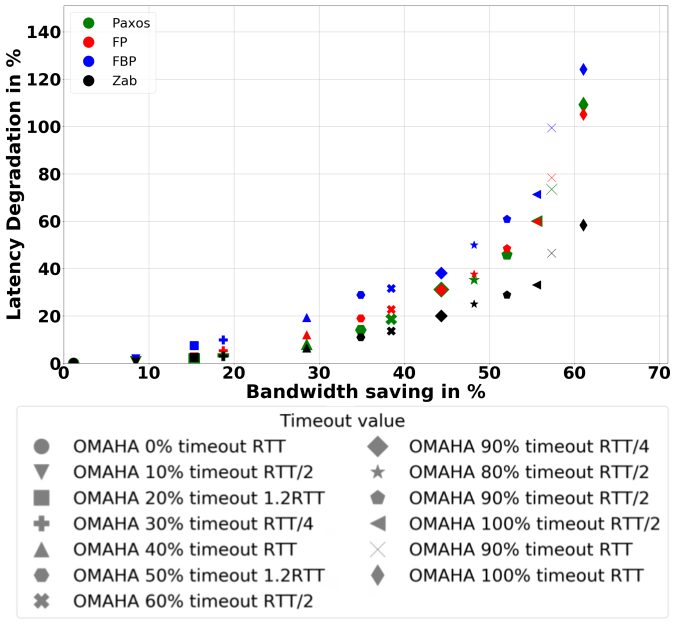


Fig. 7. Pareto for executions mixing different concurrent protocols on the g5k platform

latency degradation and bandwidth saving for parallel applications. We compared our mechanism with a time-based solution that buffers all messages and sends them periodically. Our mechanism exploits the knowledge of underlying phase-based algorithms to anticipate future message exchanges between application nodes. Omaha provides an API for a new intermediate layer between the network and the applications in order to be low intrusive and thus limit modifications of the algorithm specifications. We applied the aggregation mechanism to four widely used phase-based algorithms: three variants of the Paxos algorithm and the Zookeeper Atomic Broadcast algorithm. Omaha allows to reduce the number of messages exchanged while limiting the latency degradation. Its efficiency depends on the characteristics of the algorithm (number of phases, quorum size, type of message . . .) which must be known in order to have a good setting of the parameters of the API.

We will consider an extension of Omaha as a future work. Currently, the $probabuf$ parameter is static, which implies that the programmer has to set it manually for each type of message. By considering a dynamic and automatic setting, it will be possible to adapt the aggregation to the current state of the system in order to respect the constraints defined by the Service Level Agreements.

VI. Acknowledgment

The authors acknowledge the support of the French Agence Nationale de la Recherche (ANR), under grants ANR-20-CE25-0017 (SeMaFoR project) and ANR-22-CE25-0008-01 (SkyData project). Part of the experiments were conducted on the Grid'5000 experimental testbed http://www.grid5000.fr/.

References

[1] J. Alqahtani, S. Alanazi, and B. Hamdaoui, “Traffic behavior in cloud data centers: A survey,” in *2020 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE, 2020, pp. 2106–2111.

[2] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010, pp. 267–280.

[3] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, “The nature of data center traffic: measurements & analysis,” in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, 2009, pp. 202–208.

[4] Z. Chkirbene, R. Hadjidj, S. Foufou, and R. Hamila, “Lascada: A novel scalable topology for data center network,” *IEEE/ACM Transactions on Networking*, vol. 28, no. 5, pp. 2051–2064, 2020.

[5] L. Lamport, “The part-time parliament, may 1998,” 1998. [Online]. Available: https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf

[6] F. Junqueira, B. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” *2011 IEEE/IFIP 41st Int. Conference on Dependable Systems & Networks (DSN)*, pp. 245–256, 2011. [Online]. Available: https://marcoserafini.github.io/papers/zab.pdf

[7] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, “Low-latency multi-datacenter databases using replicated commit,” *Proc. of the VLDB Endowment*, vol. 6, no. 9, pp. 661–672, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p661-mahmoud.pdf

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, “Spanner: Google's globally distributed database,” *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013. [Online]. Available: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-16.pdf

[9] M. Burrows, “The chubby lock service for loosely-coupled distributed systems,” pp. 335–350, 2006.

[10] Á. García-Pérez, A. Gotsman, Y. Meshman, and I. Sergey, “Paxos consensus, deconstructed and abstracted,” Springer, Cham, pp. 912–939, 2018. [Online]. Available: https://software.imdea.org/~gotsman/papers/paxos-esop18.pdf

[11] L. Lamport, “Fast paxos,” *Distributed Computing 19, pages79–103(2006)*, 2006. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-112.pdf

[12] S. Chand, Y. A. Liu, and S. D. Stoller, “Formal verification of multi-paxos for distributed consensus,” Springer, pp. 119–136, 2016. [Online]. Available: https://arxiv.org/pdf/1606.01387.pdf

[13] H. Ng, S. Haridi, and P. Carbone, “Omni-paxos: Breaking the barriers of partial connectivity,” in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 314–330.

[14] J. Nagle, "Congestion control in IP/TCP internetworks," *RFC*, vol. 896, pp. 1–9, 1984. [Online]. Available: https://doi.org/10.17487/RFC0896

[15] B. Badrinath and P. Sudame, "Gathercast: the design and implementation of a programmable aggregation mechanism for the internet," pp. 206–213, 2000.

[16] S. Khanna, J. S. Naor, and D. Raz, "Control message aggregation in group communication protocols," Springer, pp. 135–146, 2002.

[17] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," New York, NY, USA, p. 347–356, 2019. [Online]. Available: https://doi.org/10.1145/3293611.3331591

[18] A. Montresor and M. Jelasity, "PeerSim: A scalable P2P simulator," Seattle, WA, pp. 99–100, Sep. 2009.

[19] [Online]. Available: https://www.grid5000.fr

[20] J.-P. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006. [Online]. Available: https://www.cs.cornell.edu/lorenzo/papers/Martin06Fast.pdf