OMAHA: Opportunistic Message Aggregation for pHase-based Algorithms (extended version)

Celia Mahamdi, Jonathan Lejeune, Julien Sopena, Pierre Sens, Mesaac Makpangou

OMAHA: Opportunistic Message Aggregation for pHase-based Algorithms

CELIA MAHAMDI, CNRS, LIP6, DELYS, Sorbonne Université, Paris, France
JONATHAN LEJEUNE, CNRS, LIP6, DELYS, Sorbonne Université, Paris, France
JULIEN SOPENA, CNRS, LIP6, DELYS, Sorbonne Université, Paris, France
PIERRE SENS, CNRS, LIP6, DELYS, Sorbonne Université, Paris, France
MESAAC MAKPANGOU, CNRS, LIP6, DELYS, Inria, Sorbonne Université, Paris, France

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, etc.). 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 several 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 tradeoff between message saving and latency degradation. We evaluate our mechanisms using multiple instances of the same algorithm (three 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 up to 30% of bandwidth with only a 5% degradation in latency.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Computing methodologies → Distributed algorithms;**

Additional Key Words and Phrases: Phase-based distributed algorithms, message aggregation, experimental evaluation, Paxos

ACM Reference Format:
Celia Mahamdi, Jonathan Lejeune, Julien Sopena, Pierre Sens, and Mesaac Makpangou. 2024. OMAHA: Opportunistic Message Aggregation for pHase-based Algorithms. *Form. Asp. Comput.* 36, 4, Article 26 (December 2024), 23 pages. https://doi.org/10.1145/3698593

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).
Authors' Contact Information: Celia Mahamdi, CNRS, LIP6, DELYS, Sorbonne Université, Paris, France; e-mail: celia.mahamdi@lip6.fr; Jonathan Lejeune, CNRS, LIP6, DELYS, Sorbonne Université, Paris, France; e-mail: jonathan.lejeune@lip6.fr; Julien Sopena, CNRS, LIP6, DELYS, Sorbonne Université, Paris, France; e-mail: julien.sopena@lip6.fr; Pierre Sens, CNRS, LIP6, DELYS, Sorbonne Université, Paris, France; e-mail: pierre.sens@lip6.fr; Mesaac Makpangou, CNRS, LIP6, DELYS, Inria, Sorbonne Université, Paris, France; e-mail: mesaac.makpangou@lip6.fr.

ACM 1433-299X/2024/12-ART26
https://doi.org/10.1145/3698593

1 Introduction

In a 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 [2, 4, 13]. Indeed, Benson et al. [4] 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 [7].

Moreover, in data centers, many applications are bandwidth hungry and congest the underlying network, resulting in an increase in their completion time. Recent papers tackle this issue by distributing network bandwidth more evenly based on an application-aware approach [14] or at the MAC level [11]. Another way to address this issue is to use aggregation mechanisms 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 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 step $i + 1$ starts after the completion of all or a part of step i. These steps involve exchanges of messages between nodes in the system. Consensus (Paxos [16]) and atomic validation protocols (Zookeeper Atomic Broadcast, usually named as ZAB [12] or two-phase commit [18]) 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 [16], one of the most implemented consensus algorithms [5], 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 [6, 10, 17, 22], 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. Taking these considerations into account, we can determine whether it is relevant to buffer a message, delaying its sending for the purpose of aggregation. Based on this observation, we have developed a generic, opportunistic, and non-intrusive message buffering mechanism that 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. Such a reduction could decrease the overall congestion of the networks and, as a side effect, decrease the completion time.

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 tradeoff between bandwidth saving and latency degradation according to their needs.

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

The article is organized as follows. Section 2 outlines related work, Section 3 details our aggregation mechanism, and Section 4 describes our experimental evaluation. Finally, Section 7 concludes the article and introduces some future research directions.

Table 1. Summary of the Protocols for a System with f Faulty Nodes

	Classical Paxos [16]	Fast Paxos [17]	Fast Byzantine Paxos [19]	ZAB [12]
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 pledge periods	1	1	3	1

2 Related Work and Background

This section presents existing aggregation mechanisms and gives an overview of four well-known phase-based protocols that we implemented to evaluate OMAHA in Section 4.

2.1 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 [21], 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 acknowledgment is received or the buffer is full.

Badrinath and Sudame introduce another aggregation mechanism called Gathercast [3]. 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 [15]. Gathercast aggregates small control packets (as TCP ACK messages) addressed to the same host in a similar way to the TCP Nagle 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. OMAHA, our aggregation mechanism, features a multiplexing and demultiplexing mechanism enabling the aggregation of 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.

2.2 Messages Aggregation in Application Layers

Another way to aggregate messages is to take into account the application's protocol. For instance, HotStuff [23], 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 that can be activated opportunistically and that multiplexes messages of any running application.

2.3 Studied Algorithms

In this article, we study four phase-based algorithms that are representative of protocols widely used on cloud platforms. However, OMAHA can be applied to any phase-based algorithm.

Table 1 summarizes the characteristics of the four algorithms.

Paxos. The Paxos algorithm [16] 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:

— *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.

Note that the loss of messages may block the execution of the protocol, especially when a node waits for a quorum to initiate the next phase. To tolerate message losses, a timer can be armed for the *prepare* and *accept* messages.

Many works have been published to extend the Paxos algorithm by tolerating Byzantine faults (Fast Byzantine Paxos [19]) or by optimizing it, especially to reduce messages' complexity (Fast Paxos [17], Single-Decree Paxos [10], Multi-Paxos [9], etc.). These algorithms optimize a single Paxos instance, but they do not consider multiple instances that run concurrently on the same physical infrastructure. These works are therefore complementary to our contribution.

Fast Paxos. Fast Paxos [17] is an optimized version of the original algorithm by removing the preparation phase: each participant sends its proposal directly to the other ones. However, Fast Paxos requires a larger quorum size, notably $2f + 1$, compared to Paxos, which requires $f + 1$.

Fast Byzantine Paxos. Fast Byzantine Paxos [19] solves Byzantine consensus in only two communication steps in the common case. In the common case, there is a unique correct leader, all correct acceptors agree on its identity, and the system is in a period of synchrony. The leader proposes its value to all participants. The participants accept and forward this value to the other ones. To learn a value, a participant has to receive $4f + 1$ acceptances. At this point, nodes decide. When communications are asynchronous, the protocol requires more communication steps. Indeed, participants need to be able to follow the leader's progress and vice versa. This is why each participant sends acknowledgments to the others, in addition to the leader. When a participant receives enough confirmations, it notifies the leader. The latter stops its proposal sending when it receives $2f + 1$ notifications from participants. If the participants do not hear from $2f + 1$ participants after a timeout, they start to suspect the leader, and if $2f + 1$ participants suspect the leader, a new one is elected.

ZooKeeper Atomic Broadcast. ZAB (ZooKeeper Atomic Broadcast) is a crash-recovery atomic broadcast algorithm designed for the ZooKeeper coordination service. ZooKeeper achieves availability and reliability through replication, and ZAB is a key component to manage atomic updates to the replicas. ZAB has three phases: discovery, synchronization, and broadcast. A ZAB process can undertake two roles: leader and follower. A leader carries out the primary role, proposing transactions according to a sequence of primary broadcast calls. The followers accept transactions according to the protocol steps. Moreover, a leader also fulfills the responsibilities of a follower. In the ZAB protocol, an "epoch" refers to a unit of time or sequence used to identify and order various events or transactions.

In the discovery phase, followers communicate with their prospective leader. The leader gathers information about the most recent transactions that the followers have accepted. This phase aims to identify the most recent sequence of accepted transactions within a quorum Q and initiate a new epoch.

The synchronization phase aims to synchronize the replicas using the transactions accepted by the leader, referred to as the history. The leader proposes transactions from its history to the followers in Q. Then, the followers acknowledge the proposals if they have not joined a more recent epoch. When the leader receives a set of acknowledgments from the quorum Q, it sends a commit message to all followers. The leader becomes established and is no longer prospective.

Last but not least, we have the broadcast phase. The leader proposes a new transaction to the quorum Q of followers. Upon receiving a quorum of acknowledgments, the leader sends a commit message to all followers, validating the transaction. When no crash occurs, the processes remain in this phase indefinitely, broadcasting transactions whenever a ZooKeeper client initiates a writing request.

3 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 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?

A simple way to address these questions is to systematically buffer messages for a fixed duration of time. This mechanism, agnostic to the application, will be used as the comparative baseline for the experimental study (Section 4). 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. The buffering time must be bounded to ensure that messages will be sent eventually.

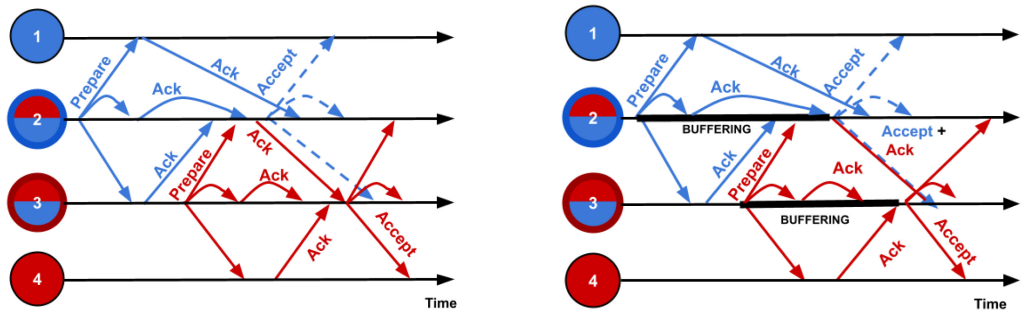
We first describe the application agnostic time-based approach. Next, we present our opportunistic approach by applying it to the Paxos protocol. Finally, we detail our new API.

3.1 The Application Agnostic 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.

3.2 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 tradeoff between latency degradation and bandwidth gain for any load.



(a) With no aggregation mechanism (b) With the opportunistic aggregation mechanism

Fig. 1. Beginning of two Paxos instances.

Figure 1 illustrates the principle of our approach by applying it to the Paxos protocol [16] 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.

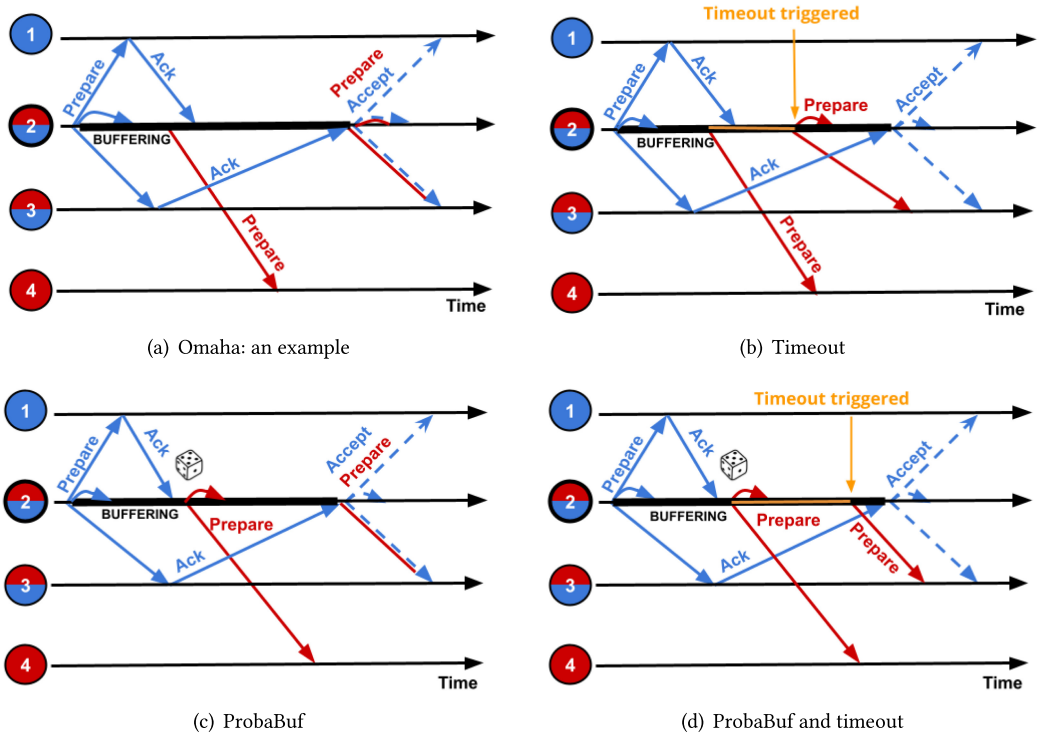
In this example, we consider two sets of nodes (each one represents a running application). The blue set and the red set are composed of nodes 1, 2, 3 and 2, 3, 4, respectively. Each set, according to its needs, can launch instances of the Paxos algorithm over time. Here, nodes 2 and 3 are the leaders 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 phase 1), it will broadcast an `accept` message as soon as it receives a quorum of `ack` messages (beginning of phase 2). The protocol ensures that any participant that sends a `prepare` message will contact the same set of recipients in the 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 both sets. Then, the pledge mechanism identifies the opportunity to buffer the `ack` message intended for node 3 from the red set, enabling its aggregation with the `accept` message from the blue set at the commencement 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 application 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.

3.3 OMAHA Mechanism: Timeout and probaBuf

To limit latency degradation, we introduced two additional mechanisms: *timeout* and *probaBuf*. We present them through an example. Figure 2(a) presents an execution of the OMAHA mechanism. The leader (node 2 for both sets) begins a blue Paxos and waits for a majority of blue acks. It then proceeds with a red Paxos. Node 4, which is not involved in the blue Paxos, receives its prepare message normally. However, for nodes 2 and 3, we can delay the red prepare to aggregate it with the blue accept. To mitigate latency degradation, we have added two other mechanisms. These mechanisms are summarized in Figure 2.



(a) Omaha: an example

(b) Timeout

(c) ProbaBuf

(d) ProbaBuf and timeout

Fig. 2. OMAHA mechanism: *timeout* and *probaBuf*.

Timeout. First, we introduce a timeout (Figure 2(b)). As mentioned, the blue prepare messages for nodes 2 and 3 can be delayed. Here, the blue ack from node 3 takes a long time to arrive. To avoid excessive latency, each buffered message arms a timeout. If the timeout is triggered, the buffered messages are sent.

Probabuf. If a message can be pledged, it is assigned a probability of being buffered, called *probaBuf*. In Figure 2(c), node 2's prepare is promptly sent, but the prepare of node 3 is delayed.

These two parameters can be coupled (Figure 2), allowing the user to find a compromise between bandwidth savings and latency degradation. Therefore, the configuration of OMAHA is essential for its performance. This point will be discussed in Section 6.

3.4 A New Network API

OMAHA can be used as a library. To improve performance, OMAHA can be deployed in the kernel or at the 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 of buffering the message when a pledge is possible. 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 mitigate latency degradation. Indeed, this parameter acts like a priority, indicating the importance of the message for the progress of the algorithm. The higher this value is, the more likely it is to buffer messages and thus increase latency.

ALGORITHM 1: The pledge mechanism algorithm

```
1  Local variables :
2  begin
3      curPledges : {(appId_1 : set_of_node_ids_1), (appId_2 : set_of_node_ids_2), (appId_3 :
         set_of_node_ids_3)...}
       /* 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 : {(nodeid_1 : ({set of msgs}, deadline_1)), (nodeid_2 :
         ({set of msgs}, deadline_2)), (nodeid_3 : ({set of msgs}, deadline_3)), ...}
       /* 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 pledgedSend(msg, dests, probaBuf, t, app):
11 begin
12     removeEntry(app) in curPledges
13     send(msg, dests, probaBuf, t, app)
14 end

15 Primitive sendBuff(buff_dest, dest):
16 begin
17     cancel any timeout related to buff_dest
18     networkSend(buff_dest.msgs) to dest
19     clear buff_dest
20 end

21 Primitive send(msg, dests, probaBuf, t, app):
22 begin
23     for all d ∈ dests do
24         flushing ← true
25         add msg to bufs[d].msgs
26         if ∃(app', nodes) ∈ curPledges where d ∈ nodes and app ≠ 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 ← now + t
30                     schedule sendBuff(bufs[d].msgs, d) at bufs[d].deadline
31                 end
32                 flushing ← false
33             end
34         end
35         if flushing == true then
36             sendBuff(bufs[d].msg, d)
37         end
38     end
39 end
```

— `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 the "*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)`: This indicates the end of a pledge period for the application `app` (line 10) and the sending of the `msg` message with any messages present in the buffer. 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). Buffered messages are application-level messages. Therefore, sending the buffer constitutes a single network message. 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 18).
— If `probaBuf > 0`, then the decision to buffer the message or not depends on the `probaBuf` value and if the sending node is in the "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 18).
  — 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.

3.5 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 the 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 7) and that each message type is associated with a maximum buffering time (line 8) and a buffering probability (line 9).

As explained in Section 2 and illustrated in Figure 1(a), when the leader starts a new instance of Paxos (line 10), it sends a prepare message (line 13) 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 14). When the leader receives a majority of *ack* messages (line 23), 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 30).

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 $\perp$

initOMAHA(app_id, $timeouts$, $probabuf$)
/* function that initializes OMAHA variables */

Dedicated local variables to OMAHA:
app_id
/* unique id for each application */
$timeouts : \{(message\_type\_1 : timeout\_1), (message\_type\_2 : timeout\_2), (message\_type\_3 : timeout\_3)...\}$
/* timeout value for each type of message */
$probaBuf : \{(message\_type\_1 : probaBuf\_1), (message\_type\_2 : probaBuf\_2), (message\_type\_3 : probaBuf\_3)...\}$
/* probaBuf value for each type of message */

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

3.6 Implementation Details

The implementation of OMAHA has been designed for TCP/IP, chosen for its widespread usage in phase-based applications like Apache ZooKeeper, Apache Kafka, Google Spanner, and more.

Figure 3 shows an implementation of nodes without OMAHA. Application messages are sent from one node to another via the application port. Applications can also encrypt messages, as shown by app 2 in red.

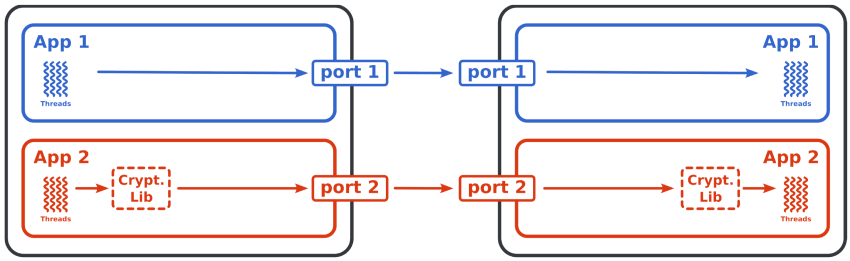


Fig. 3. Node implementation diagram without OMAHA.

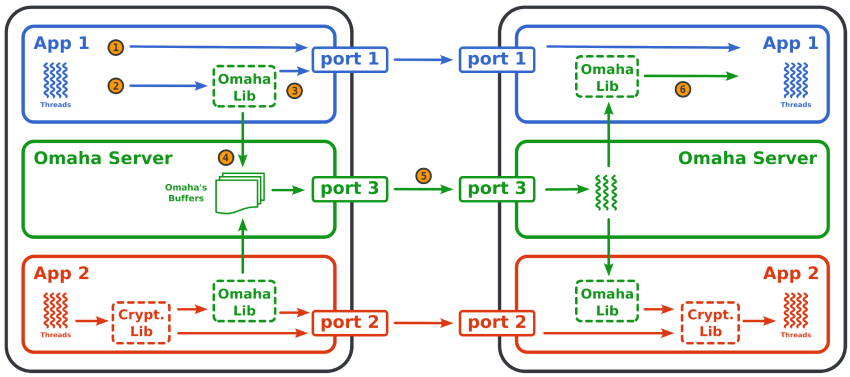


Fig. 4. OMAHA: Node implementation diagram.

Figure 4 depicts how OMAHA is integrated into a node. Each application still has its own legacy ports for transmission, regardless of OMAHA. Applications leveraging OMAHA encapsulate its library to use the overloaded send and beginPledge primitives. The OMAHA server keeps buffers for each node. Additionally, the OMAHA server has its own port. It allows OMAHA to multiplex multiple ports simultaneously.

Figure 4 depicts the message flow:

(1) The application directly transmits messages that cannot be buffered.
(2) If a message can be buffered, it goes through OMAHA.
(3) Depending on the value of *probaBuf*, the message can be sent directly.
(4) Or it can be transmitted to the OMAHA server via shared memory to be stored in the destination node's buffer.
(5) If the end of a pledge period is detected or a timeout expired, the contents of the buffer are sent as a single network message.
(6) Upon receiving this message, the OMAHA server handles distributing the messages to the relevant applications.

4 Experimental Study

4.1 Experimental Testbed and Configuration

This section describes our experiment setup environment.

4.1.1 Infrastructure Settings. To investigate various combinations of the metrics, the experiments were first carried out with Peersim [20], a discrete events simulator (Sections 4.2 to 4.5).

To validate our results in a real infrastructure, we then test OMAHA on the **Grid'5000 platform (g5k)** [1] (Section 4.6). In both platforms, we deployed 15 nodes. In the g5k platform, each node is deployed on a dedicated physical host.[1] We consider an **average round-trip time (RTT)** 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 4.2 through 4.4, and 6, a reliable system (all nodes never crash and execute the protocols correctly, there is no loss or 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 4.5, 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.

4.1.2 Considered Application Phase-based Algorithms. We consider and implement four application phase-based algorithms presented in Section 2.3: three variants of the Paxos algorithm and the ZAB algorithm.

4.1.3 Aggregation Mechanisms and Parameters. We compare OMAHA to the original algorithm without any aggregation mechanism and the time-based aggregation mechanism (cf. Section 3.1), which systematically buffers messages.

The impact of the two following parameters is 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 OMAHA API of the *send* primitive, Section 3.4). In all experiments, we consider that these two parameters are statically set and never change during the experiment execution. In Sections 4.2 through 4.5, we consider that these parameters are constant whatever the phase and whatever the message type. However, in Section 6 we consider different values of $probaBuf$ depending on the criticality of the message type.

4.1.4 Workload. 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.

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

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

Table 2. 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	0.6%	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 3. Impact of the *probabuf* Parameter on the Classical Paxos Algorithm: Latency Degradation

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%

— The **average latency,** 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 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 3,000th 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 the value, the better) and bandwidth consumption saving (the higher the 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.

4.2 Impact of the *probabuf* Parameter

Tables 2 and 3 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 2, 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 3 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 be 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%).

Table 4. Impact of the Timeout Parameter on the Classical Paxos Algorithm: Bandwidth Saving

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 5. Impact of the Timeout Parameter on the Classical Paxos Algorithm: Latency Degradation

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%

4.3 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 4 and 5. We vary the value of the *timeout* with a constant buffering probability equal to 90%.

In Table 4, 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 5, 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.

4.4 Study of the Tradeoff between Bandwidth Saving and Latency Degradation

In this section, we study the tradeoff between bandwidth saving and latency degradation with different settings of the two parameters. Figure 5 shows the results considering a high load. The other loads will be discussed later.

For one simulation, the overall bandwidth consumption values for Paxos, Fast Paxos, Fast Byzantine Paxos, and ZAB without the aggregation mechanism are 123.525 Mo, 105.12 Mo, 302.22 Mo, and 22.04 Mo, respectively.

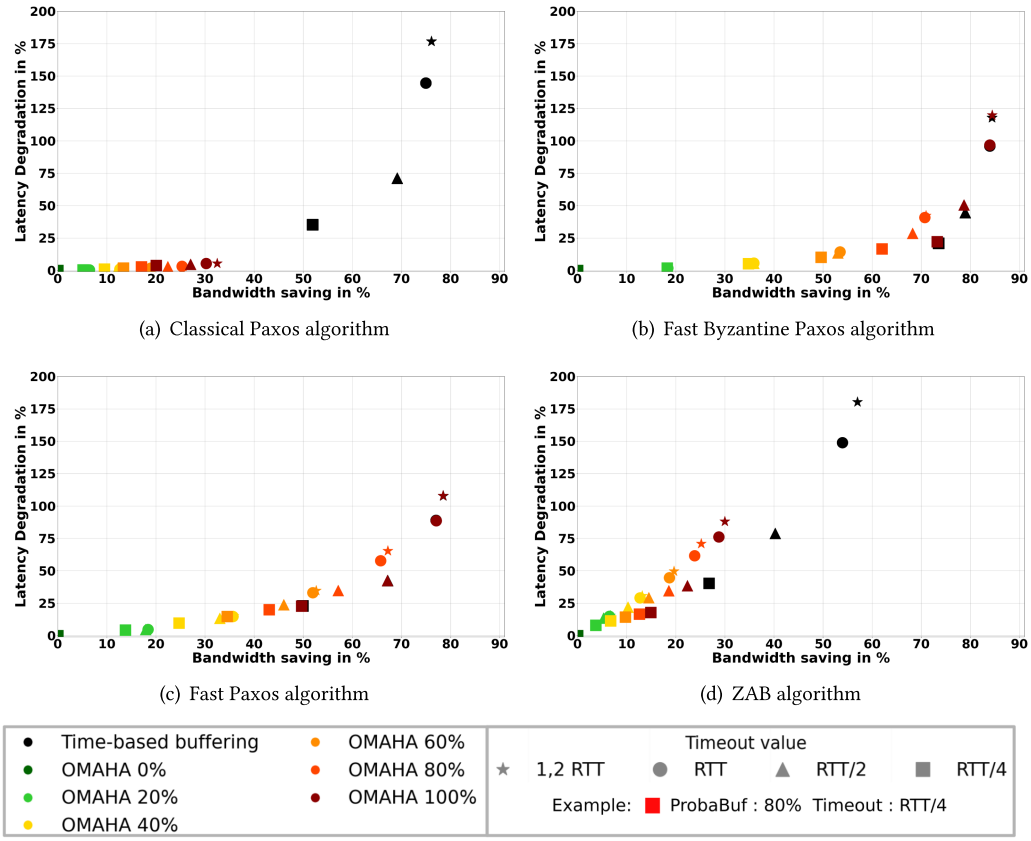
The x-axis and the y-axis correspond to bandwidth saving and latency degradation, respectively. The shape of dots represents a given timeout value, while the color represents the value of $probaBuf$. Note that the black color is dedicated to the time-based aggregation mechanism as a baseline.

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, 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 1).

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



(a) Classical Paxos algorithm

(b) Fast Byzantine Paxos algorithm

(c) Fast Paxos algorithm

(d) ZAB algorithm

Fig. 5. Pareto considering a high load.

(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 5(a) and 5(b), where we can see that OMAHA is more efficient with Fast Byzantine Paxos 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. In Fast Byzantine Paxos, on the other hand, all nodes can make pledges. The aggregation power is therefore well distributed between nodes.

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 message to each node. Delaying any messages sent by a ZAB quorum node would result in a significant degradation in latency, visible in Figure 5(d). By modifying the settings of OMAHA, it is possible to improve performance (see Section 6).

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.

4.5 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.

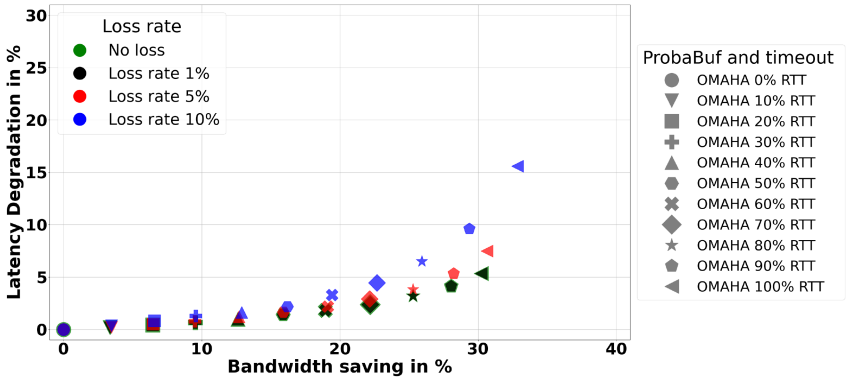


Fig. 6. Impact of message loss on Paxos.

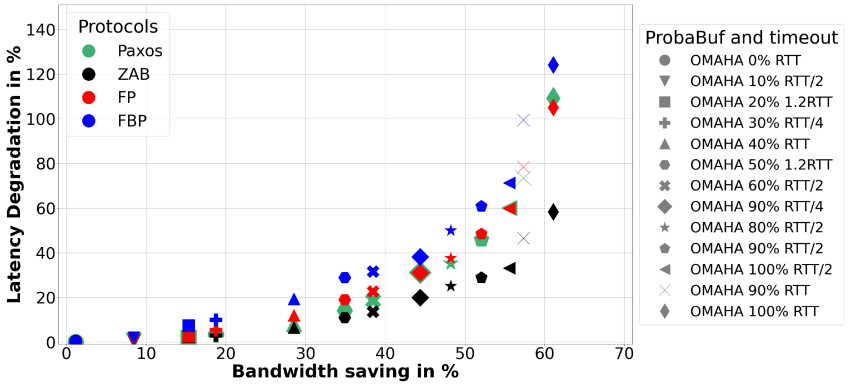


Fig. 7. Pareto for executions mixing different concurrent protocols on the g5k platform.

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.

4.6 Experiment on Grid'5000 Platform

This section presents results of experiments conducted on the g5k platform (see Section 4.1.1 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 the Pareto of each protocol running concurrently on the same infrastructure.

In Figure 5(d), we observed that OMAHA was less efficient for ZAB compared to the other protocols. As explained in Section 4.4, 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 tradeoff between bandwidth saving and latency degradation.

If we compare ZAB behavior in Figures 7 and 5(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 5(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.

5 OMAHA: An Offline Analysis Tool

Through OMAHA,we have developed a tool that helps to find the best configuration to save bandwidth for a given acceptable latency degradation rate set by a user. Using this tool, we initiated an extensive experimentation campaign. We considered the four previously mentioned algorithms: Paxos, Fast Paxos, Fast Byzantine Paxos, and ZAB. We evaluate each mechanism (OMAHA and time-based buffering) with three load patterns:

— Low: A request every 100 ms
— Medium: A request every 50 ms
— High: A request every 20 ms

We summarize the results in Table 6. For maximum latency degradation rates of 5%, 10%, 20%, and 50%, we documented the highest achievable bandwidth savings for both OMAHA and time-based buffering. The maximum rate of latency degradation is not consistently reached. The presented values correspond to the highest bandwidth gain achievable while maintaining latency degradation equal to or below the different rates considered. The boxes labeled "NR" indicate Not Reachable. This signifies that there is no configuration available to save bandwidth for such latency degradation. The results are displayed in green when OMAHA is the best option and in orange when time-based buffering provides better results.

We can observe initially that irrespective of the objective, algorithm, or load (except for one case with the ZAB algorithm), OMAHA is settable to save bandwidth. For all loads and a maximum latency degradation rate of

— 5%: the bandwidth savings range from 1% to 29.2% for a latency degradation ranging between 0.3% and 4.8%.
— 10%: the bandwidth savings range from 1.6% to 44.7% for a latency degradation ranging between 0.3% and 9.9%.
— 20%: the bandwidth savings range from 1.6% to 62% for a latency degradation ranging between 0.3% and 20.0%.
— 50%: the bandwidth savings range from 1.6% to 74.5% for a latency degradation ranging between 0.3% and 49.2%.

For degradation rates of 5%, 10%, and 20%, OMAHA consistently wins. The time-based buffering never manages to propose a configuration with a loss rate lower than or equal to these values.

Under moderate load, even with a 5% latency degradation, noticeable gains in bandwidth are observed. At 10% and 20%, they become significant, reaching up to 27.2% and 45.8%, respectively. At a high degradation rate of 50%, OMAHA does not provide any benefit compared to an aggressive time-based buffering strategy that, while significantly compromising latency, achieves greater bandwidth savings.

Table 6. Summary Table of the Experimental Campaign

				5%	10%	20%	50%
PAXOS	Low	OMAHA	Latency Degradation	0.3%	0.3%	0.3%	0.3%
			Bandwidth Gain	5.7%	5.7%	5.7%	5.7%
		Time based	Latency Degradation	NR	NR	NR	42.7%
			Bandwidth Gain	NR	NR	NR	8.2%
	Medium	OMAHA	Latency Degradation	1.5%	1.5%	1.5%	1.5%
			Bandwidth Gain	15.3%	15.3%	15.3%	15.3%
		Time based	Latency Degradation	NR	NR	NR	38.6%
			Bandwidth Gain	NR	NR	NR	32.2%
	High	OMAHA	Latency Degradation	4.1%	5.3%	5.3%	5.3%
			Bandwidth Gain	29.2%	32.5%	32.5%	32.5%
		Time based	Latency Degradation	NR	NR	NR	35.2%
			Bandwidth Gain	NR	NR	NR	52.0%
FAST PAXOS	Low	OMAHA	Latency Degradation	4.8%	6.1%	6.1%	43.4%
			Bandwidth Gain	1.4%	8.2%	8.2%	19.4%
		Time based	Latency Degradation	NR	NR	NR	27.9%
			Bandwidth Gain	NR	NR	NR	4.1%
	Medium	OMAHA	Latency Degradation	3.0%	9.6%	19.3%	46.0%
			Bandwidth Gain	9.6%	18.2%	26.5%	53.3%
		Time based	Latency Degradation	NR	NR	NR	46.0%
			Bandwidth Gain	NR	NR	NR	53.2%
	High	OMAHA	Latency Degradation	4.8%	9.0%	20.0%	42.5%
			Bandwidth Gain	18.4%	27.1%	43.1%	67.2%
		Time based	Latency Degradation	NR	NR	NR	42.5%
			Bandwidth Gain	NR	NR	NR	67.2%
FAST BYZANTINE PAXOS	Low	OMAHA	Latency Degradation	4.2%	7.0%	16.8%	49.2%
			Bandwidth Gain	2.5%	9.7%	18.6%	41.6%
		Time based	Latency Degradation	NR	NR	NR	27.3%
			Bandwidth Gain	NR	NR	NR	43.5%
	Medium	OMAHA	Latency Degradation	4.8%	9.4%	19.4%	47.3%
			Bandwidth Gain	16.0%	27.2%	45,8%	67.3%
		Time based	Latency Degradation	NR	NR	NR	47.2%
			Bandwidth Gain	NR	NR	NR	73.2%
	High	OMAHA	Latency Degradation	3.3%	9.2%	16.7%	38.3%
			Bandwidth Gain	27.2%	44.7%	62.0%	74.5%
		Time based	Latency Degradation	NR	NR	NR	44.7%
			Bandwidth Gain	NR	NR	NR	79.0%
ZAB	Low	OMAHA	Latency Degradation	3.2%	9.7%	9.7%	9.7%
			Bandwidth Gain	1%	1.6%	1.6%	1.6%
		Time based	Latency Degradation	NR	NR	NR	43.2%
			Bandwidth Gain	NR	NR	NR	4.1%
	Medium	OMAHA	Latency Degradation	NR	9.6%	16.7%	36.9%
			Bandwidth Gain	NR	2.9%	6.8%	14.0%
		Time based	Latency Degradation	NR	NR	NR	42.1%
			Bandwidth Gain	NR	NR	NR	12.7%
	High	OMAHA	Latency Degradation	4.7%	9.9%	17.9%	38.5%
			Bandwidth Gain	2.9%	5.3%	15.0%	22.4%
		Time based	Latency Degradation	NR	NR	NR	40.3%
			Bandwidth Gain	NR	NR	NR	26.8%

Table 7. Simulation versus Grid5000

	Prediction		Real Measurements	
	Latency	Bandwidth	Latency	Bandwidth
Low	7,0%	9,7%	3,5%	10,1%
Medium	9,3%	27,3%	10,9%	27,1%
High	9,2%	44,7%	13,4%	43,4%

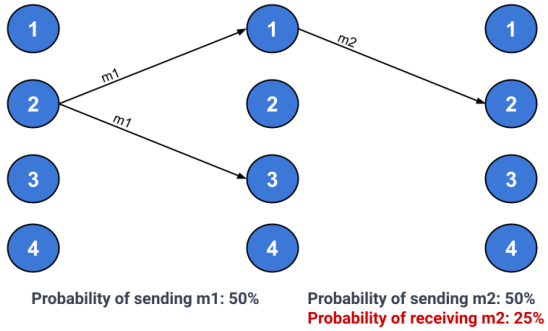


Fig. 8. Conditional probabilities example.

We verify our tool's quality by comparing our predictions with real measurements for the Fast Byzantine Paxos, the most bandwidth-efficient algorithm. We reproduce the same experiments on g5k. For each load, we use the OMAHA configuration that saves the most bandwidth while limiting latency degradation to 10%. The results are summarized in Table 7.

We observe that the results obtained match the predictions made offline. However, we notice a discrepancy in latency results for low throughput between predictions and real environments (7.0% vs. 3.5%), attributed to entropy. To mitigate this effect, the experiment should be repeated a large number of times.

6 Optimization: Parameter Settings

Users must define the parameters of the *send* primitive. Choosing the right parameter combination is crucial as it plays a key role in OMAHA's performance. We first propose a theoretical analysis to define a value of the *probaPledge* parameter in order to maximize bandwidth gains and minimize latency degradation when all messages have the same importance and can be buffered. Next, we study how to define the *probaPledge* parameter in real applications where messages could have different importance depending on their semantic.

6.1 Theoretical Analysis Uniform Probabilities

The selection of *probaPledge* based on messages is important to avoid disruption in the algorithm execution. Indeed, when determining the probability of a message, it is crucial to consider the probabilities of preceding messages.

Figure 8 illustrates a scenario involving four nodes. Node 2 sends a message, m1, to all nodes, but there is a 50% chance of buffering m1. Upon receiving m1, nodes respond using message m2, which also has a 50% chance of being buffered. In this case, the probability of node 2 receiving a response is 25%, not 50%. Indeed, the probability of receiving message m2 is expressed by Equation (1):

$$P(receive\ m2) = P(send\ m1) \times P(send\ m2|\ receive\ m1) = 0,5 \times 0,5 = 0,25. \quad (1)$$

Here, we are considering the worst-case scenario where each message can be pledged. The sending of the message will rely each time on the associated probability. As a result, some messages will be buffered, while others will not. To transition from one phase to another, it is necessary to gather a quorum of messages. However, to reach this quorum as quickly as possible, a sufficient number of messages must be received. Waiting for the arrival of buffered messages to assemble a quorum proves counterproductive.

More formally, let's consider an algorithm that requires a quorum q. The quorum indicates the minimum threshold of responses expected by a node c to make a decision. We want to calculate the probability of c receiving q messages. Let $m_1, .., m_n$ be the set of messages sent by the algorithm to c, each message being associated with a probability to be buffered, denoted as $p_1, \ldots, p_n$. The probability that c receives the messages is expressed as Equation (2):

$$P(c\ receive\ messages) = \prod_{i=1}^{n} p_i. \tag{2}$$

To ensure that a quorum of messages is received, this probability must be higher than the quorum. We then have the following equation:

$$P(c\ receive\ messages) = q/n \Leftrightarrow \prod_{i=1}^{n} p_i = q/n. \tag{3}$$

Now, assuming that the p_is are uniform with p_1=p_2=...=p_n=p, we can calculate the values of the p, resulting in

$$p^n = q/n \Leftrightarrow p = \sqrt[n]{q/n}. \tag{4}$$

This value is theoretical and only applies if all the p_is are equal, without considering the importance of the messages. However, it ensures the reception of a quorum of messages for each message, thus avoiding blocking the execution of the algorithm.

6.2 No Uniform Probabilities

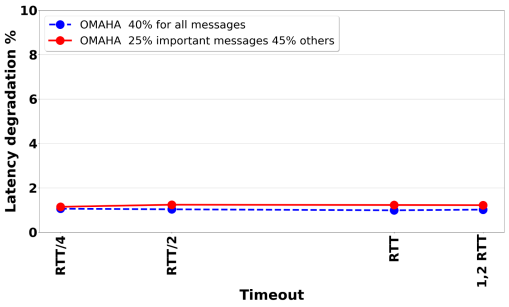
In the theoretical calculation, we considered uniform probabilities, where all messages could be buffered. However, in practice, this scenario rarely occurs. Therefore, to configure OMAHA, we decided to set the buffering probability based on the importance of the messages.

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.

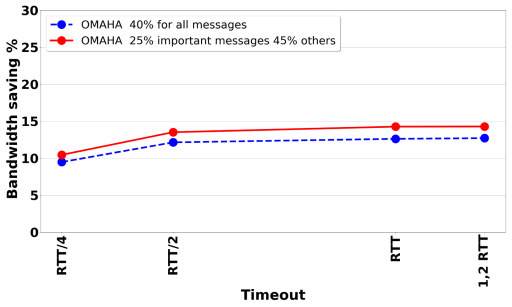
Response messages must have a probability close to $1 - \frac{size(quorum)}{\#nodes}$ to ensure enough message receiving. Thus, for a large quorum, e.g., two-thirds of the nodes, the response messages will have a low probability of being delayed, e.g., 33%.

Figure 9 shows the results when we adapt the buffering probability for the Paxos, Fast Byzantine Paxos, and ZAB algorithms considering three load patterns.

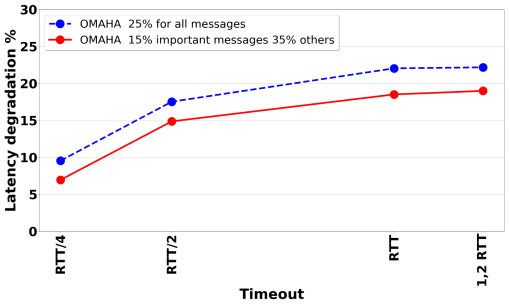
In Paxos (Figures 9(a) and 9(b)), we choose a value of 45% for response messages. This value is slightly lower than the quorum size (50% of nodes). For the critical messages, we assign the buffering probability to 25%. We observe that adapting the buffering probability has no significant impact on performance here. Nevertheless, there is a slight improvement in bandwidth with nearly equivalent latency degradation.



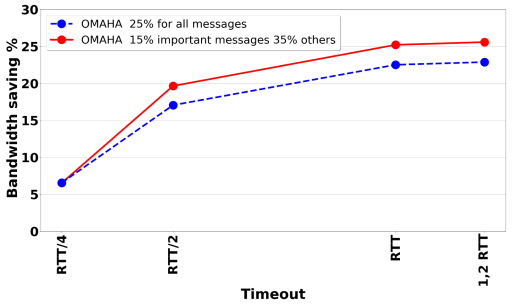
(a) Paxos high load latency degradation



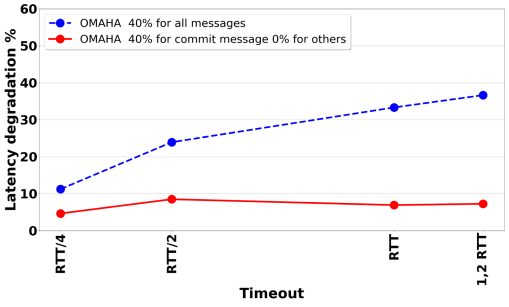
(b) Paxos high load bandwidth saving



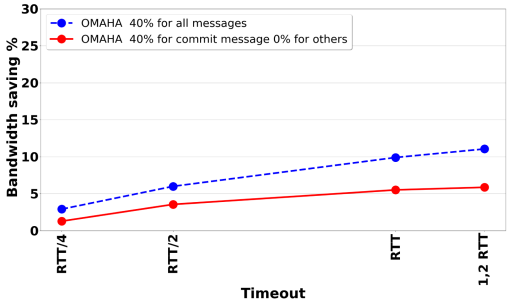
(c) FBP low load latency degradation



(d) FBP low load Bandwidth saving



(e) ZAB medium load latency degradation



(f) ZAB medium load Bandwidth saving

Fig. 9. Adaptation of buffering probabilities.

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 Figures 9(c) and 9(d) 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 5(d). Thus, we opt not to buffer the critical message and instead buffer the remaining messages with a probability of 40%. With this setting, we observe in Figures 9(e) and 9(f) a high reduction in the latency degradation but a limited bandwidth saving. However, for the same 5.9% bandwidth gain, the degradation rate varies significantly. Indeed, this bandwidth gain corresponds to a timeout value of $RTT/2$ on the blue curve and $1.2RTT$ for the red one. The associated latency degradation is 24.0% and 7.2%, respectively. Thus, for the same

bandwidth gain, we observe a difference of 16.8 percentage points. The latency degradation is therefore significantly reduced.

7 Conclusion and Future Works

This article proposes OMAHA, an opportunistic message aggregation mechanism allowing to find a tradeoff between 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. We demonstrate that fine-tuning our mechanism can save up to 30% of bandwidth with only a 5% degradation in latency for the Paxos protocol. 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, etc.), 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.

Acknowledgment

Part of the experiments were conducted on the Grid'5000 experimental testbed: http://www.grid5000.fr/.

References

[1] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Perez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. 2013. Adding virtualization capabilities to the Grid'5000 testbed. In *Cloud Computing and Services Science, volume 367 of Communications in Computer and Information Science*. Springer International Publishing, 3–20.

[2] Jarallah Alqahtani, Sultan Alanazi, and Bechir Hamdaoui. 2020. Traffic behavior in cloud data centers: A survey. In *2020 International Wireless Communications and Mobile Computing (IWCMC'20)*. IEEE, 2106–2111.

[3] B. R. Badrinath and Pradeep Sudame. 2000. Gathercast: the design and implementation of a programmable aggregation mechanism for the internet. In *9th International Conference on Computer Communications and Networks*. Las Vegas, NV, 206–213.

[4] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. 267–280.

[5] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, USA, 335–350.

[6] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. 2016. Formal Verification of Multi-Paxos for Distributed Consensus. 119–136. https://arxiv.org/pdf/1606.01387.pdf

[7] Zina Chkirbene, Rachid Hadjidj, Sebti Foufou, and Ridha Hamila. 2020. LaScaDa: A novel scalable topology for data center network. *IEEE/ACM Transactions on Networking* 28, 5 (2020), 2051–2064. https://doi.org/10.1109/TNET.2020.3008512

[8] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (August 2013), 22 pages. https://doi.org/10.1145/2491245

[9] Hao Du and David J. St Hilaire. 2009. *Multi-Paxos: An Implementation and Evaluation*. Department of Computer Science and Engineering, University of Washington, Tech. Rep. UW-CSE-09-09-02 (2009). https://dada.cs.washington.edu/research/tr/2009/09/UW-CSE-09-09-02.PDF

[10] Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. 2018. Paxos Consensus, Deconstructed and Abstracted. 912–939. https://software.imdea.org/~gotsman/papers/paxos-esop18.pdf

[11] David Georgantas and Peristera A. Baziana. 2023. Traffic burstiness study of an efficient bandwidth allocation MAC scheme for WDM datacenter networks. In *IEEE International Mediterranean Conference on Communications and Networking (MeditCom'23)*. IEEE, 169–174. https://doi.org/10.1109/MEDITCOM58224.2023.10266597

[12] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN'11)*. 245–256.

[13] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. 2009. The nature of data center traffic: Measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*. 202–208.

[14] M. R. Siavash Katebzadeh, Paolo Costa, and Boris Grot. 2023. Saba: Rethinking datacenter network allocation from application's perspective. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys'23)*, Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (Eds.). ACM, 623–638. https://doi.org/10.1145/3552326.3587450

[15] Sanjeev Khanna, Joseph Naor, and Danny Raz. 2002. Control message aggregation in group communication protocols. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP'02)*. 135–146.

[16] Leslie Lamport. 2019. The part-time parliament. Concurrency: The Works of Leslie Lamport. Association for Computing Machinery, New York, NY, USA, 277–317. https://doi.org/10.1145/3335772.3335939

[17] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (2006), 79–103. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-112.pdf

[18] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment* 6, 9 (2013), 661–672. http://www.vldb.org/pvldb/vol6/p661-mahmoud.pdf

[19] J.-P. Martin and Lorenzo Alvisi. 2006. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 202–215. https://www.cs.cornell.edu/lorenzo/papers/Martin06Fast.pdf

[20] Alberto Montresor and Márk Jelasity. 2009. PeerSim: A scalable P2P simulator. *IEEE Ninth International Conference on Peer-to-Peer Computing*. IEEE.

[21] John Nagle. 1984. Congestion Control in IP/TCP internetworks. *RFC* 896 (1984), 1–9. https://doi.org/10.17487/RFC0896

[22] Harald Ng, Seif Haridi, and Paris Carbone. 2023. Omni-Paxos: Breaking the barriers of partial connectivity. In *Proceedings of the 18th European Conference on Computer Systems*. 314–330.

[23] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC'19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

Received 15 January 2024; revised 7 August 2024; accepted 24 September 2024