**Compas'2023** : Parallélisme/ Architecture/ Système
LISTIC/USMB - Annecy France, 4-7 juillet 2023

# Agrégation opportuniste de messages pour les algorithmes à phases

Célia Mahamdi, Jonathan Lejeune, Julien Sopena, Pierre Sens et Mesaac Makpangou

Sorbonne Université - CNRS
4, place Jussieu,
F-75005, Paris, France
Email : firstname.lastname@lip6.fr

**Résumé**
Dans le cloud computing, plusieurs applications s'exécutent simultanément sur une même infrastructure physique sous-jacente. Les algorithmes à phases sont essentiels pour de nombreuses applications distribuées telles que les systèmes de gestion de base de données ou les services de validation de transactions. En effet, ces applications se basent sur des problèmes fondamentaux des systèmes distribués tels que le consensus ou la validation atomique et sont résolues via des algorithmes à phases (Paxos, ZAB, Two-phase commit, etc.).
À chaque phase, au moins un participant diffuse un message aux autres et attend les réponses d'un sous-ensemble d'entre eux avant de commencer la phase suivante. Pour un algorithme à phases, il est alors possible de prédire pour chaque nœud ses communications futures. Sur la base de cette observation, nous proposons une solution générique et peu intrusive pour économiser la bande passante dans un contexte cloud en agrégeant de manière opportuniste les messages envoyés par les applications.
Le cœur de la nouvelle API repose sur la surcharge de la primitive *send*, où les utilisateurs peuvent définir un compromis entre le gain en messages et la dégradation de la latence. Nous évaluons notre mécanisme à travers plusieurs instances du même algorithme (3 variantes de Paxos et l'algorithme de diffusion atomique Zookeeper) s'exécutant simultanément. Nos résultats montrent qu'un bon réglage de la nouvelle primitive *send* permet d'économiser une grande quantité de bande passante avec peu de dégradation de la latence.

**Mots-clés :** algorithmes à phases, aggrégation de messages, évaluation expérimentale, Paxos

## 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. These applications often exchange many small control messages that contain very little data. Many studies have shown that a large part of the bandwidth was used by the message headers [2, 4, 11] and network bandwidth becomes one of the main bottlenecks for the core network infrastructure of data centers [7].
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 [13]), atomic validation protocols (Zookeeper Atomic Broadcast usually named as ZAB [10], or two-phase commit [15]) are widely used 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 [13], the 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 [9] [14] [6] 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 one 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 sending or whether it is better to send it immediately. 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 reduces 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.
We evaluated our solution with multiple instances of 4 phase-based algorithms (3 variants of Paxos consensus and the ZAB commit protocol). However, due to the lake of space, only Paxos is presented here. We show that a good tuning of our mechanism saves up to 30% percent of bandwidth with a little latency degradation.
The paper is organized as follows. Section 2 outlines-related work, in Section 3 we present our aggregation mechanism, and Section 4 describes our experimental evaluation. Finally, Section 5 concludes the paper and introduces some future research directions.

## 2. Related Work

This section presents existing aggregation mechanisms.

### 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 [17] which is enabled by default in most of TCP implementations. Since TCP/IP packets have a 40-bytes header, when chunk of small messages are sent, the overhead is huge and may lead to network congestion. The main idea behind the Nagle's algorithm is data buffering until acknowledgement receiving or until that 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 overload, many protocols aggregate them into a single packet [12]. 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.

### 2.2. Messages aggregation in application layers

Another way to aggregate messages is to take into account the application's protocol. We can cite HotStuff [18] which is a leader-based Byzantine fault-tolerant replication protocol. The 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.

## 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 message buffer. Such a mechanism must then answer two questions :

— Should a new application message be buffered (and so delayed) or sent immediately ?
— When buffered messages have to be sent on the network ?

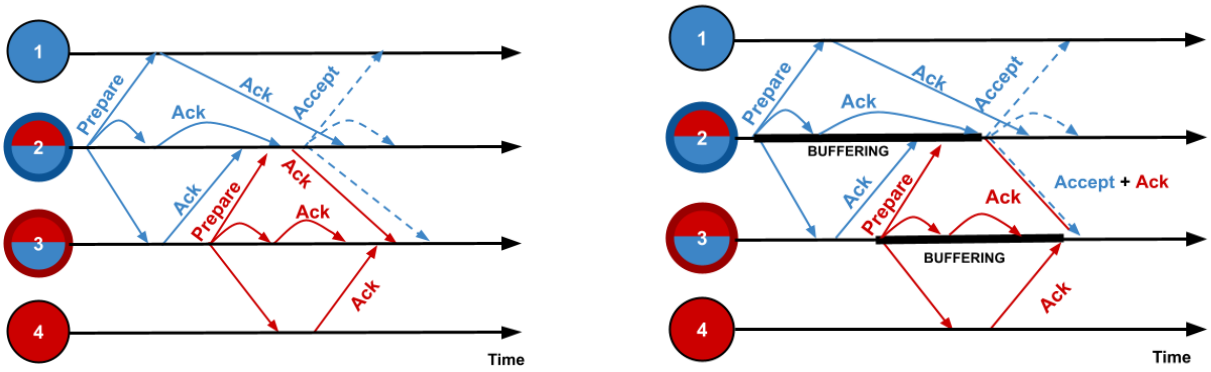
In Omaha, the answer to the first question depends on the criticality of the message for the liveness of the application algorithm. To answer the second question, we leverage the phase-based protocol to delay messages without slowing down the application. Anyway, the buffering time must be bounded to 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 4). Next, we present our opportunistic approach by applying it to the Paxos protocol.

### 3.1. 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 method is simple to implement and increases bandwidth savings by significantly reducing message sending. However, this approach is application agnostic and thus delays every application messages. As a result, it does not take into account the criticality of messages, which can lead to buffering of some blocking messages and hugely degrade the application latency. Moreover, it does not take into account the system load. Thus, it is possible to buffer (and delay) a message even if no future sending is planned to the same recipient, which is useless.
This time-based approach is particularly efficient in an environment where the message load is high. However, it is no longer suitable when the load decreases or varies.

### 3.2. Our opportunistic approach

The idea behind our approach is to exploit the knowledge of future message sending for a 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 [13] in a multiple application context (description of Paxos in A.1). Each application is independent of another one and runs on its own set of physical nodes. However, we suppose that these sets intersect.
In this example, we consider two sets of nodes (each one represents a running application). 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




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

FIGURE 1 – Beginning of two Paxos instances

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 undertakes to send 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.

The pseudo-code of the pledge mechanism is given in A.2 which can be applied to any phase-based algorithms.

## 4. Experimental study

### 4.1. Experimental testbed and configuration

**Infrastructure settings.** To ease the analysis, the experiments were first carried out with Peersim [16], a discrete events simulator. In order to validate our results in real infrastructure, we then test Omaha on the Grid'5000 platform (a.k.a. g5k) [1]. In both platforms, we deployed 15 nodes. In g5k platform, each node is deployed on a dedicated physical host [1]. 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 have 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 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 the impact of the aggregation mechanism (we study in Section 4.4 the impact of an unreliable network with different message loss rates).

**Aggregation mechanisms and parameters.** We compare Omaha to the original algorithm without any aggregation mechanism, and the time-based aggregation mechanism (cf. 3.1) 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 A.2). In all experiments, we consider that these two parameters are

1. 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

statically set and never change during the experiment execution.

**Workload.** The performance of the aggregation mechanisms is directly related to the network load and thus 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. Concurrent instances are not synchronized, i.e., one instance can start running the protocol regardless of the state of the other running instances. Therefore, they are 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.

**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 consensus algorithms)

— 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 in the measurements 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 greater value the better).

We observed a low variation of the latency between each instance, the highest standard deviation (3 for a 130.1 average latency) has been measured in high load configuration when `probaBuf` has a value of 100.

### 4.2. Impact of the `probabuf` parameter

Tables 1 and 2 present the impact of Omaha on classical Paxos instances by varying workload and `probaBuf` parameters while keeping a constant `timeout` parameter equals to one RTT.

In Table 1, we observe that the bandwidth saving increases linearly with the pledge probability, regardless 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 larger. In this case, many instances of Paxos are running concurrently and more pledges can be done. Conversely, in a low load pattern, few Paxos instances are running concurrently and Omaha is not able to predict future communications, so the bandwidth saving is lower.

| 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 | 0.1% | 1.6% | 4.6% | 6.1% | 7.6% | 9.1% | 10.5% | 12.0% | 13.4% | 14.7% |
| High | 0.2% | 3.4% | 9.5% | 12.6% | 15.9% | 18.9% | 22.2% | 25.3% | 28.1% | 30.2% |

TABLE 1 – Impact of the `probabuf` parameter on the classical Paxos algorithm : bandwidth saving

Table 2 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, in high load patterns, many pledges can be done. This can result in additional delay in message transmission. Nevertheless, it is possible to save bandwidth up to 30.2% while having a low latency degradation (5.4%).

| 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 2 – Impact of the probabuf parameter on the classical Paxos algorithm : latency degradation

### 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 3(b) and 3(a). We vary the value of the timeout with a constant buffering probability equal to 90%.

| 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% |

(a) 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% |

(b) Latency degradation

TABLE 3 – Impact of the timeout parameter on the classical Paxos algorithm

In Table 3(a), we can see that the bandwidth saving follows globally 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 higher than one RTT will never expire because message sending is mainly due to the pledge mechanism.
In Table 3(b), we observe very little latency degradation, especially in the low load pattern where few messages can be aggregated. At high load, more messages can be saved, so the latency degradation is more significant.
Experiments conducted on the g5k platform (see section 4.1 for the platform settings) corroborate those of the simulations.

### 4.4. 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.
Experiments show 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 ( 5% and 10% loss rate), 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.

## 5. Conclusion and future works

This paper proposes Omaha an opportunistic message aggregation mechanism allowing to find a trade-off 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. Our mechanism exploits the knowledge of underlying phase-based algorithms to anticipate future message exchanges between application nodes. 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.

## Bibliographie

1. The grid 5000 platform. – `https://www.grid5000.fr`.
2. Alqahtani (J.), Alanazi (S.) et Hamdaoui (B.). – Traffic behavior in cloud data centers : A survey. – In *2020 International Wireless Communications and Mobile Computing (IWCMC)*, pp. 2106–2111. IEEE, 2020.
3. Badrinath (B.) et Sudame (P.). – Gathercast : the design and implementation of a programmable aggregation mechanism for the internet, 2000.
4. Benson (T.), Akella (A.) et Maltz (D. A.). – Network traffic characteristics of data centers in the wild. – In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 267–280, 2010.
5. Burrows (M.). – The chubby lock service for loosely-coupled distributed systems, 2006.
6. Chand (S.), Liu (Y. A.) et Stoller (S. D.). – Formal verification of multi-paxos for distributed consensus, 2016.
7. Chkirbene (Z.), Hadjidj (R.), Foufou (S.) et Hamila (R.). – Lascada : A novel scalable topology for data center network. *IEEE/ACM Transactions on Networking*, vol. 28, n5, 2020, pp. 2051–2064.
8. Corbett (J. C.), Dean (J.), Epstein (M.), Fikes (A.), Frost (C.), Furman (J. J.), Ghemawat (S.), Gubarev (A.), Heiser (C.), Hochschild (P.) et al. – Spanner : Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, vol. 31, n3, 2013, pp. 1–22.
9. García-Pérez (Á.), Gotsman (A.), Meshman (Y.) et Sergey (I.). – Paxos consensus, deconstructed and abstracted, 2018.
10. Junqueira (F.), Reed (B.) et Serafini (M.). – Zab : High-performance broadcast for primary-backup systems. *2011 IEEE/IFIP 41st Int. Conference on Dependable Systems & Networks (DSN)*, 2011, pp. 245–256.
11. Kandula (S.), Sengupta (S.), Greenberg (A.), Patel (P.) et Chaiken (R.). – The nature of data center traffic : measurements & analysis. – In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pp. 202–208, 2009.
12. Khanna (S.), Naor (J. S.) et Raz (D.). – Control message aggregation in group communication protocols, 2002.
13. Lamport (L.). – The part-time parliament, may 1998, 1998.
14. Lamport (L.). – Fast paxos. *Distributed Computing 19, pages79–103(2006)*, 2006.
15. Mahmoud (H.), Nawab (F.), Pucher (A.), Agrawal (D.) et El Abbadi (A.). – Low-latency multi-datacenter databases using replicated commit. *Proc. of the VLDB Endowment*, vol. 6, n9, 2013, pp. 661–672.
16. Montresor (A.) et Jelasity (M.). – PeerSim : A scalable P2P simulator, septembre 2009.
17. Nagle (J.). – Congestion control in IP/TCP internetworks. *RFC*, vol. 896, 1984, pp. 1–9.
18. Yin (M.), Malkhi (D.), Reiter (M. K.), Gueta (G. G.) et Abraham (I.). – Hotstuff : Bft consensus with linearity and responsiveness, 2019.

## A. Appendix

### A.1. Paxos algorithm

The Paxos algorithm [13] 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.

### A.2. A new network API

The pseudo-code of the pledge mechanism is given in A.2 which can be applied to any phase-based algorithms. Thus, 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 of 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.

**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 ∈ 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], d) at bufs[d].deadline
31                 end
32                 flushing ← false
33             end
34         end
35         if flushing == true then
36             sendBuff(bufs[d], d)
37         end
38     end
39 end
```