2015 International Conference on Protocol Engineering (ICPE) and International Conference on New Technologies of Distributed Systems (NTDS).

TRAINS: A Throughput-Efficient Uniform Total Order Broadcast Algorithm

Michel Simatic*, Arthur Foltz, Damien Graux, Nicolas Hascoët, Stéphanie Ouillon, Nathan Reboud and Tiezhen Wang
Télécom SudParis
Institut Mines-Télécom, Évry, France
Email: [First name].[Last name]@telecom-sudparis.eu

***Abstract*—Within data centers, many applications rely on a uniform total order broadcast algorithm to achieve load-balancing or fault-tolerance. In this context, achieving high throughput for uniform total order broadcast algorithms is an important issue: It contributes to optimize data center resources usage and to reduce its energy consumption. This paper presents TRAINS, a throughput-efficient uniform total order broadcast algorithm. The paper estimates TRAINS performance. It evaluates the prediction-oriented throughput efficiency ($POTE$) — i.e. the theoretical ratio between bytes delivered and bytes transmitted on the network. TRAINS $POTE$ improves the $POTE$ of the best algorithm of the literature. For 5 processes, the $POTE$ improvement reaches a peak of 250% for 10 bytes messages. Experimental evaluation confirms TRAINS high throughput capabilities. The trade-off of this throughput improvement is the alteration of the latency. The worst alteration is in the case of 2 processes: 125%.**

I. INTRODUCTION

Many distributed applications require stronger delivery guarantees than those provided by a best-effort network broadcast. For instance, some web servers have to be replicated for load-balancing or fault-tolerance. Video game industry has another motivation for having messages delivered in the same order on all the replicas: It provides a realistic user experience in multiplayer video games such as Age of Empire [1]. For all of these applications, the replicas broadcast their state changes to the other replicas. Broadcast messages delivery must be guaranteed. And the delivery order must be the same on all replicas in order to be able to apply a state machine approach [2], [3]. These requirements have led to the specification of uniform total order (UTO-) broadcast [4]. A UTO-broadcast algorithm ensures the following properties: 1) Validity: if a correct process UTO-broadcasts message m, then it will eventually UTO-deliver m; 2) Uniform agreement: if a replica UTO-delivers a message m, then all correct replicas eventually UTO-deliver m; 3) Integrity: for any message m, a replica UTO-delivers m at most once, if and only if m was previously UTO-broadcast by a process; 4) Total order: for any message m and m', if a replica UTO-delivers m without having UTO-delivered m', then no replica UTO-delivers m' before m.

In parallel of this specification activity, an important algorithmic activity took place. Between 1984 when Chang and Maxemchuk published the historical first algorithm [5] and the survey of Défago, Schiper, and Urbán in 2004 [6], more than sixty algorithms were proposed. And, this research field remains active with recent proposals such as LCR [7], Ring Paxos [8] and FastCast [9]. The vast majority of proposals tackles performance problems. They can be classified into two categories: those targeting low latency, and those targeting high throughput [10]. Latency measures the time required to complete a single message broadcast. Sequencer-based algorithm establishes a first record [11], improved by Isis V3 [12]. Throughput measures the number of broadcasts that the processes can complete per time unit. Totem uses a token on a virtual ring to establish a first record [13]. Ring Paxos combines Paxos algorithm with a virtual ring to improve this record [8]. LCR combines vector clocks and a virtual ring to establish the current throughput record [7].

Carrying on a research activity on throughput of UTO-broadcast algorithms is important. For instance, Spread (an industrial middleware offering UTO-broadcast) has released version 4.4.0 in May 2014. The release notes emphasize that Spread "is tailored for data center networks and can provide 30%–50% higher throughput [. . .] in modern local area networks" [14]. Indeed, UTO-broadcast algorithms are used within data centers, thus in the context of clustered environments. In this context, CPU is the limiting factor. Improving throughput has an impact on CPU usage. For instance, if we pack messages to avoid duplication of some UTO-broadcast protocol data, we improve the throughput. We also reduce the number of CPU interruptions related to message exchange: This boosts the performance [15]. This reduces energy consumption of the data center. Moreover, UTO-broadcast algorithms are used between data centers — e.g. in the context of geo-replicated databases. In this context, network is the limiting factor. If we improve the throughput, we improve the network usage: Cloud applications can handle more requests.

LCR is considered as the best algorithm from a throughput point of view (see Line 2 of Table I) [7]. Nevertheless, taking a look at the maximum throughput efficiency (MTE) — i.e. the rate between the maximum achieved throughput per receiver and the nominal transmission capacity of the system per receiver [8] — is revealing. We can evaluate MTE_{LCR}, because we know that the nominal capacity of the system used is 116 Mb/s [7]: We obtain Line 3 of Table I. For 100 bytes UTO-broadcasts, MTE_{LCR} is only 28% — i.e. a loss of 72%. This makes us think that LCR is not efficient enough when dealing with short messages.

This paper presents TRAINS, a throughput-efficient uniform total order broadcast algorithm. TRAINS has better throughput efficiency than LCR because it requires less overhead bytes to

TABLE I. THROUGHPUT PERFORMANCE OF LCR FOR 5 PROCESSES

UTO-broadcast size (bytes)	100	1 000	10 000
Throughput of LCR (Mb/s)	32	88	112
MTE_{LCR}	28%	76%	96%

carry UTO-broadcasts.

This paper makes the following contributions. First, it proposes TRAINS, a new UTO-broadcast algorithm derived from Train algorithm [16]: Participating processes are dispatched on a virtual ring. Several trains carrying wagons rotate simultaneously on this ring. Each wagon belongs to one of the processes. It carries one or several messages UTO-broadcast by this process. Second, the paper presents the flow control used in TRAINS. This flow control requires no additional messages, nor piggybacked data on messages. Third, the paper estimates the performance of TRAINS. The paper evaluates the prediction-oriented throughput efficiency ($POTE$) — i.e. the theoretical ratio between bytes delivered and bytes transmitted on the network. $POTE_{\text{TRAINS}}$ and $POTE_{\text{LCR}}$ can be compared because, unlike MTE_{TRAINS} and MTE_{LCR}, they do not depend on the experimental setup. $POTE_{\text{TRAINS}}$ improves $POTE_{\text{LCR}}$. For 5 processes, the $POTE$ improvement reaches a peak of 250% for 10 bytes messages. Experimental evaluation confirms TRAINS high throughput capabilities. The trade-off of this throughput improvement is the alteration of the latency. The paper estimates the theoretical latency L_{TRAINS} of TRAINS. L_{TRAINS} alters L_{LCR}. The worst alteration is in the case of 2 processes: 125%.

The remainder of the paper is structured as follows. Section II describes our system and performance model. Section III presents TRAINS algorithm. Section IV describes TRAINS flow control. Section V evaluates the performance of TRAINS and compares it to LCR. Section VI comments on related work. Section VII concludes the paper.

II. MODEL

We use the same model as the model used by LCR [7].

Concerning the system model, we assume a small cluster of homogeneous machines interconnected by a local area network. Each machine hosts a process participating to the algorithm. Moreover, we assume that a process stays on the same machine. It does not migrate from one machine to another. TRAINS integrates a membership service[1] [18]. This service implements the abstraction of a perfect failure detector (P) [19] to which each process has access.

Concerning the performance model, we assume that our LAN is based on a switch. Thus, we use the round-based model used in [7]. In one round: 1) a network card can send a message and simultaneously receive one; 2) a process can send a message to all or a subset of processes; 3) the network is able to carry several messages simultaneously.

III. ALGORITHM

In TRAINS, participating processes are dispatched on a virtual ring. Several trains carrying wagons rotate simultaneously on this ring. Each wagon belongs to one of the processes. It carries one or several messages broadcast by this process. Section III-A presents all of the data structures used in TRAINS. Section III-B describes straightforward procedures and functions. Section III-C focuses on algorithms executed in the absence of failures. It explains how TRAINS ensures UTO-delivery. Section III-D describes the algorithm that is executed when the virtual ring changes. Finally, Section III-E gives an example.

A. Data structures

This section presents *wagon* and *train*, the two data structures exchanged between TRAINS processes. Then, it presents data structures local to each process.

1) Wagon: In TRAINS, application messages are aggregated inside *wagons*. A wagon contains the following fields:

- `sender`: address of the process sending the wagon,
- `rotat`: each wagon is attached to a train in order to rotate on the virtual ring. `rotat` field contains the identifier of the rotation made by this train on the virtual ring when the wagon is attached to this train;
- `msgs`: ordered list of application messages broadcast by the `sender` process.

2) Train: Wagons are themselves aggregated inside *Trains*. Each train rotates between processes of the virtual ring. A train contains the following fields:

- `id`: identifier of the train (coded as an integer),
- `lc`: logical clock used to avoid train duplication when recovering from a process failure,
- `rotat`: identifier of the rotation made by the train on the virtual ring,
- `wag`: ordered list of the wagons carried by the train.

3) Local Data: Each participating process p_i uses the following variables or constants local to p_i:

- `DELAY`: (constant) maximum time process p_i will wait when it fails in its first tentative to participate to TRAINS;
- `NB_RO`: (constant) minimum number of rotations (done by each train) that we need to distinguish to guarantee that TRAINS is a UTO-broadcast algorithm. Section III-C proves that the value of `NB_RO` is 3;
- `NB_TR`: (constant) number of simultaneous trains rotating on the virtual ring;
- `idLast`: identifier of the last train sent by process p_i;
- `initDone`: boolean set to `true` when TRAINS initialization is done for process p_i;
- `lastTrs`: array containing the last `NB_TR` trains sent by process p_i;
- `lastTrsView`: array containing `NB_TR` views, each one corresponding to the view when process p_i sent one of the `NB_TR` trains;

[1]The presentation of the algorithms of TRAINS membership service is outside the scope of the paper. See [17] for details.

- `nbJoin`: number of times process p_i tried to participate in TRAINS;
- `rcvdWag`: bi-dimensional array containing (`NB_TR`$\times$ `NB_RO`) ordered lists of received wagons that process p_i cannot yet deliver (because p_i has no guarantee that uniformity and total order properties are verified);
- `view`: ordered list containing the last view of participants to the membership protocol.
- `wagToSnd`: wagon containing the messages that process p_i wants to broadcast. These messages will be added to the next train sent by p_i;

B. Straightforward procedures and functions

This section presents the different procedures and functions used by algorithms of sections III-C and III-D, which do not need to be detailed:

- `append(aList,anElement)`: adds element `anElement` at the end of list `aList`;
- `Fsend(aMsg)` to p_j: sends `aMsg` in FIFO order (this includes reliability). TCP protocol is an example of `Fsend()`;
- `goneSet(`p_i`,oldView,newView)`: Returns the set of gone processes between `oldView` and `newView`, preceding p_i on the virtual ring;
- `succ(`p_i`,aView)`: returns the address of p_i's successor in view `aView` (or $\perp$ if `aView` is `[]`).
- `UTO-deliver(aListOfMsgs)`: delivers the different messages contained in `aListOfMsgs`;

C. Failure-free behavior

This section presents the algorithms that are executed in the absence of failure.

When a process p_i wants to use TRAINS, the `initialize` procedure is executed (see Algorithm 1). Once `initialize` procedure is done, p_i can broadcast a message `aMsg` by invoking the `UTO-broadcast` procedure (see Algorithm 2). This message is added to the wagon `wagToSnd`. When the uniformity and total order properties are guaranteed for a wagon, `UTO-deliver` procedure is called with the list of messages contained in this wagon: `aMsg` is delivered.

To get uniformity and total order guarantees, there are two cases to consider. In the first case, the sending process p_i is the only participant: `UTO-broadcast` calls immediately `UTO-deliver` (Line 3 of Algorithm 2). In the second case, the sending process p_i is not alone. Train messages are exchanged between processes participating in TRAINS (see Algorithm 3). So p_i receives a train tr. Let ι be the value of `tr.id` and θ the value of `rotat` field. `wagToSnd.rotat` receives the value θ; `wagToSnd` is appended to tr and to `rcvdWag[`ι`][`θ`]` (Lines 25–27 of Algorithm 3). p_i sends the updated tr. When p_i receives tr one rotation later, we have the guarantee that p_i has received all of the wagons w transported by tr with `w.rotat` equal to θ. When p_i receives tr another rotation later, we have the guarantee that all of the other processes have received all of the wagons w transported by tr with `w.rotat` equal to θ. Therefore, p_i UTO-delivers the wagons in `rcvdWag[`ι`][`θ`]` (Lines 10–13 of Algorithm 3).

Moreover, we can determine the value of `NB_RO`. During one rotation of a train tr, one process p_j executes Line 7 of Algorithm 3: `tr.rotat` is incremented by one during each rotation. Previously, we have seen that p_i waits 2 rotations of the train tr before delivering the wagons in `rcvdWag[`ι`][`θ`]`. So, p_i needs to distinguish the values θ, $\theta+1$ and $\theta+2$. By definition of `NB_RO`, we conclude that the value of `NB_RO` is 3.

Algorithm 1 Procedure `initialize` for any process p_i

// Initialize global variables
`nbJoin` $\leftarrow 1$
`idLast` $\leftarrow$ `NB_TR` -1
`lastTrs[0...NB_TR-1]` $\leftarrow \{\perp, \ldots, \perp\}$
`lastTrsView[0...NB_TR-1]` $\leftarrow \{$`[]`$, \ldots,$ `[]`$\}$
`rcvdWag[0...NB_TR-1][0...NB_RO-1]` $\leftarrow$ $\{\{$`[]`$,\ldots,$`[]`$\},\ldots,\{$`[]`$,\ldots,$`[]`$\}\}$
`wagToSnd.sender` $\leftarrow p_i$
`wagToSnd.rotat` $\leftarrow 0$
`wagToSnd.msgs` $\leftarrow$ `[]`
`view` $\leftarrow$ `[]`
`initDone` $\leftarrow$ `false`
// Join participants (see section III-D for more information)
Join membership service
Wait for `initDone` to become `true`

Algorithm 2 Procedure `UTO-broadcast(aMsg)` for any process p_i

1: `append(wagToSnd.msgs, aMsg)`
2: **if** `size(view) == 1` **then**
3:   `UTO-deliver(wagToSnd.msgs)`
4:   `wagToSnd.msgs` $\leftarrow$ `[]`
5: **end if**

D. Taking care of virtual ring changes

To build our virtual ring and to manage changes in its members, TRAINS integrates a membership service [18]. This service handles *joins* (requests to join the group of processes) and *leaves* (requests to leave the group). This service also excludes processes that are suspected to have crashed. Finally, this service provides a view of the group members. TRAINS membership service guarantees that the relative order of processes in consecutive views is the same. For instance, let view v_i be $[p_A, p_B, p_D]$. If process p_C joins, new view v_{i+1} can be $[p_A, p_B, p_C, p_D]$ or $[p_C, p_A, p_B, p_D]$, but not $[p_C, p_B, p_A, p_D]$.

Upon view change, a process p_i that has a new successor sends all of the last `NB_TR` trains p_i sent to its previous successor before view change (Lines 24–35 of Algorithm 4). However, p_i may not have yet received all of the `NB_TR` trains — e.g. because p_i joined recently the membership service. So, p_i is not able to send all of the last `NB_TR` trains. This may lead to a deadlock: If a train tr was lost between p_i and p_i's successor *before* view change, p_{i+1} — the successor of p_i *after* view change — is waiting for p_i to send tr; The successor of p_{i+1} is waiting for p_{i+1} to send tr; …; p_i is

Algorithm 3 Receiving a train for any process p_i

```
1:  upon Freceive(tr) do
2:    local id ← tr.id
3:    if initDone then
4:      if id == (idLast + 1) mod NB_TR
             and tr.lc ⩾ lastTrs[id].lc then
5:        local rotat ← tr.rotat
6:        if rotat == lastTrs[id].rotat then
7:          rotat ← (rotat + 1) mod NB_RO
8:        end if
9:        // p_i delivers pending wagons.
10:       local r ← (rotat + 1) mod NB_RO
11:       for all w ∈ rcvdWag[id][r] do
12:         UTO-deliver(w.msgs)
13:       end for
14:       rcvdWag[id][r] ← []
15:       // p_i prepares the new train and saves
          // received wagons.
16:       lastTrs[id].lc ← tr.lc + 1
17:       lastTrs[id].rotat ← rotat
18:       lastTrs[id].wag ← []
19:       for all w ∈ tr.wag so that
            w.sender ∈
             lastTrsViews[id] \ ({p_i}∪
              goneSet(p_i,lastTrsViews[id],view))
          do
20:         append(rcvdWag[id][w.rotat],w)
21:         if w.sender != succ(p_i,view) then
22:           append(lastTrs[id].wag,w)
23:         end if
24:       end for
25:       wagToSnd.rotat ← rotat
26:       append(lastTrs[id].wag,wagToSnd)
27:       append(rcvdWag[id][rotat],wagToSnd)

28:       wagToSnd.msgs ← []
29:     end if
30:   else
31:     lastTrs[id] ← tr
32:     initDone ←
                 (∀ i ∈ ⟦0,NB_TR⟦, lastTrs[i] ≠⊥)
33:   end if
34:   Fsend(lastTrs[id]) to succ(p_i,view)
35:   lastTrsView[id] ← view
36:   idLast ← id
```

waiting for its predecessor to send tr. Therefore, all of the processes experience a deadlock. And, p_i is the cause of this deadlock. To break this deadlock, p_i leaves the membership service, waits for a random period, and joins again (Lines 39–45 of Algorithm 4).

Once a process p_i has received all of the NB_TR trains, p_i will never leave spontaneously the membership service: The initialization of the algorithm is done (Line 32 of Algorithm 3).

E. Example

To illustrate the behavior of TRAINS, consider the message sequence chart of Figure 1. Process p_A joins the membership service. Since p_A is the only member of the view, p_A sets

Algorithm 4 View change management for any process p_i

```
1:  upon viewChange(newView) do
2:    if size(newView) == 1 then
3:      if initDone then
4:        // p_i is left alone: It delivers pending wagons.
5:        for j = 1 to NB_RO do
6:          for i = 1 to NB_TR do
7:            local id ← (idLast + i) mod NB_TR
8:            local r ←
                (lastTrs[id].rotat+j) mod NB_RO
9:            for all w ∈ rcvdWag[id][r] do
10:             UTO-deliver(w.msgs)
11:           end for
12:           rcvdWag[id][r] ← []
13:         end for
14:       end for
15:       UTO-deliver(wagToSnd.msgs)
16:       wagToSnd.msgs ← []
17:     else
18:       // p_i is the first participant to TRAINS.
19:       initDone ← true
20:     end if
21:     view ← newView
22:   else if succ(p_i,newView) != succ(p_i,view)
                                  and view ≠⊥ then
23:     if initDone then
24:       // p_i sends again all of its last sent trains, in case
          // some did not reach its previous successor.
25:       for i = 1 to NB_TR do
26:         local id ← (idLast + i) mod NB_TR
27:         if size(view) == 1 then
28:           // p_i was alone before view change.
29:           lastTrs[id].lc ←
                       lastTrs[id].lc + 1
30:           lastTrs[id].wag ← []
31:           lastTrsView[id] ← newView
32:         end if
33:         Fsend(lastTrs[id]) to
            succ(p_i,newView)
34:         lastTrsView[id] ← newView
35:       end for
36:       view ← newView
37:     else
38:       // p_i is missing trains. It cannot resend the NB_TR
          // trains: p_i leaves and tries to join again later.
39:       Leave membership service
40:       nbJoin ← nbJoin + 1
41:       Wait for a random time in [0, nbJoin × DELAY]
42:       lastTrs[0...NB_TR-1] ← {⊥, . . . , ⊥}
43:       lastTrsView[0...NB_TR-1] ←
                                   {[], . . . , []}
44:       view ← []
45:       Join membership service
46:     end if
47:   else
48:     view ← newView
49:   end if
```

`initDone` to true. Then, process p_B joins the membership service. p_A notices it has a new successor: It sends trains t_{000} and t_{100}. Upon receiving these trains, p_B forwards them to its own successor — i.e. p_A. Moreover, since p_B has received all of the trains that rotate on the virtual ring, p_B sets `initDone` to true. Upon receiving train t_{000}, process p_A stores wagon w_{A1m0} in `rcvdWag[0][1]` and sends train t_{011} containing wagon w_{A1m0} to process p_B. Upon receiving $t_{011}(w_{A1m0})$, as p_B wants to UTO-broadcast message $m1$ in wagon w_{B1m1}, p_B stores w_{A1m0} and w_{B1m1} in `rcvdWag[0][1]`. Then p_B sends $t_{010}(w_{B1m1})$ to p_A. In parallel, p_A receives train t_{100}: p_A stores w_{A1m2} in `rcvdWag[1][1]` and sends $t_{111}(w_{A1m2})$. Trains go on rotating and carrying new wagons. Upon receiving $t_{042}(w_{B2m5})$, process p_A increments the `rotat` field of this train: `rotat` was 2; It is now 0. Thus, p_A knows that all of the wagons with rotation 1, carried by train with id 0, have been received by all of the processes: p_A UTO-delivers the wagons contained in `rcvdWag[0][1]` — i.e. w_{A1m0} and w_{B1m1}. Afterwards, p_A stores w_{A0m8} in `rcvdWag[0][0]` and sends $t_{050}(w_{A0m8})$ to p_B. Etc.

IV. Flow control

Our flow control regulates processes that want to send bursts of messages. In addition, our flow control allocates more network bandwidth (if available) for these processes. For this purpose, we introduce one new constant value, two new global variables and two new algorithms. `OPTIM_TR_SIZE` is a constant containing the optimal train size with respect to the number of participating processes and `NB_TR`. Its value is determined thanks to simulation or dedicated performance tests. `wagToSndMaxSize` is a global variable containing the maximum size of `wagToSnd`. `wagToSndMaxSize` is initialized to 0 during TRAINS initialization. `lastWagSizeDic` is a global dictionary. It associates the sender of each wagon received in the last train to the size of this wagon. `lastWagSizeDic` is initialized to {} during TRAINS initialization.

To enforce flow control on any participating processes p_i, p_i now UTO-broadcasts its messages with Algorithm 5. Lines 1–2 of Algorithm 5 regulate p_i when p_i wants to broadcast more messages than TRAINS can convey.

Algorithm 5 Procedure `utoBroadcastWithFlowControl(aMsg)` for any process p_i

```
1: if size(wagToSnd) + size(aMsg) >
                              wagToSndMaxSize
   then
2:    Wait until wagToSnd.msgs == []
3: end if
4: utoBroadcast(aMsg) // See algorithm 2
```

To compute the value of `wagToSndMaxSize`, we take advantage of a TRAINS feature: When a process p_i receives a recent train, this train informs implicitly p_i about how all of the other processes fill up their wagon, and thus use the bandwidth. Algorithm 6 does this computation. In order to call Algorithm 6 each time a process receives a recent train, we insert a call to Algorithm 6 between Line 4 and Line 5 of Algorithm 3. Algorithm 6 classifies participating processes into two categories. On the one hand, there are *greedy* processes. The size of their wagon is larger than the average size of a wagon. Or, it is larger than the previously sent wagon (Line 10 of Algorithm 6). We need to know how many greedy processes there are among participants. With the number of greedy processes, we know how many processes have to share extra bytes available in a train of `OPTIM_TR_SIZE` bytes. On the other hand, there are *sober* processes. For these processes, we need only to know how many bytes they are sending. With the number of bytes used by sober processes, we know how many extra bytes are available for greedy processes. Since we want the current process to send as many messages as possible, we assume that the current process is greedy: We initialize `nbGreedy` with 1 (Line 4 of Algorithm 6). Then, we determine the number of greedy processes and the number of bytes used by sober processes (Lines 5–18 of Algorithm 6). Finally, we compute the updated value of `wagToSndMaxSize` (Line 26 of Algorithm 6).

Notice that we need to make two adjustments because our method counts the wagons of all n processes whereas a train carries at most $n-1$ wagons. So, if we find that n processes are greedy, we adjust the number of greedy processes to take into account that the rotating train will always contain only $n-1$ wagons of greedy processes (Lines 20–22 of Algorithm 6). Moreover, if there are sober processes, during the rotation of the train, this train will not contain the wagon of the most sober process. This is why we evaluate the value of `minSober` in Algorithm 6. It influences the computation of the updated value of `wagToSndMaxSize` (Line 26 of Algorithm 6).

Algorithms 5 and 6 are fully local. They require neither piggybacked data on existing messages, nor additional messages.

V. Performance evaluation

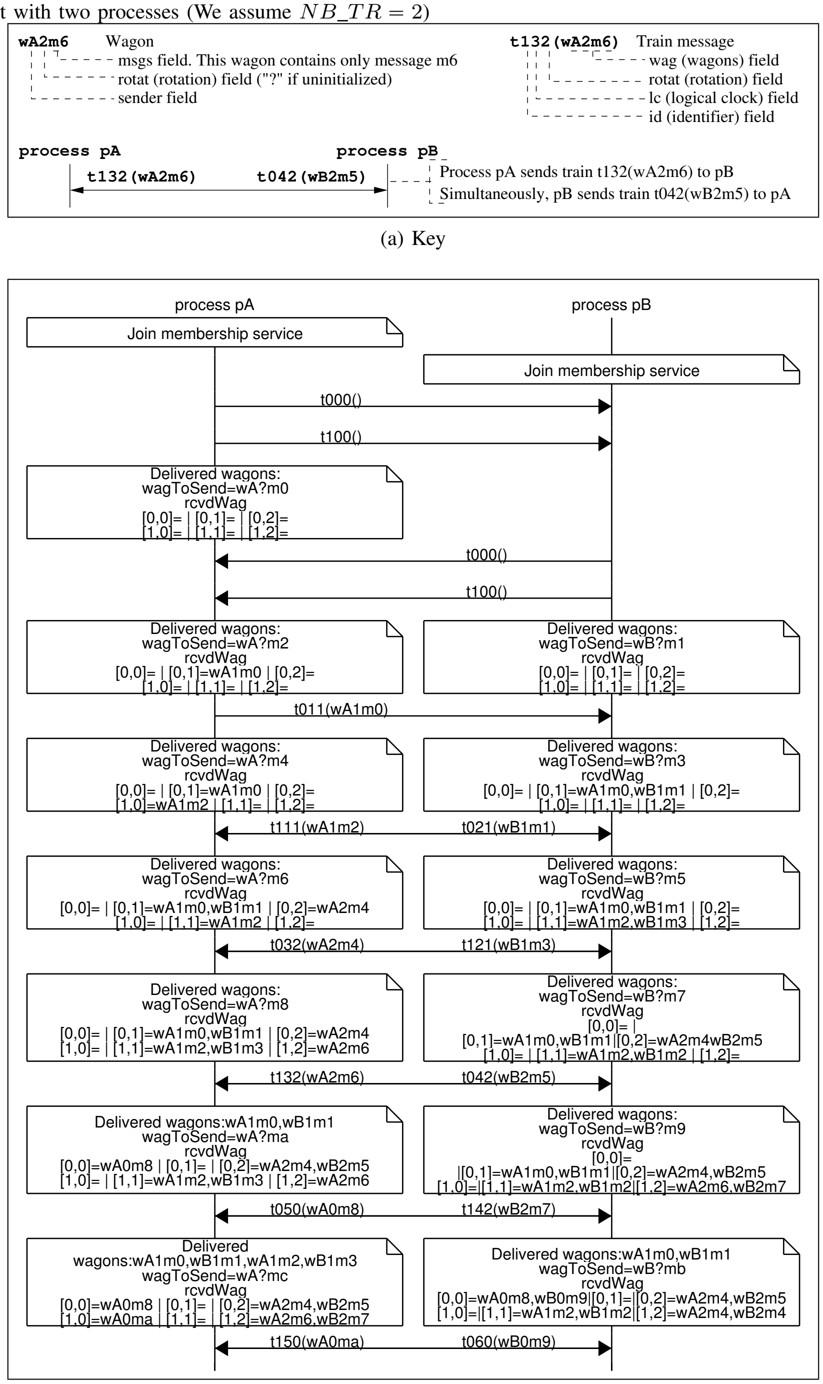
This section evaluates TRAINS performance and compares it to LCR performance. Section V-A focuses on throughput. Section V-B analyzes latency.

A. Throughput

TRAINS uses the same system model as LCR. Moreover, like LCR, TRAINS does not use physical broadcast. But, TRAINS sends UTO-broadcast messages around a virtual ring. As a result, we can predict that TRAINS should have at least the same throughput as LCR.

To compare more precisely the throughput of both algorithms, we could compare their maximum throughput efficiency (MTE) — i.e. the measured rate between the maximum achieved throughput per receiver and the nominal transmission capacity of the system per receiver [8]. But, MTE_{TRAINS} cannot be compared to MTE_{LCR}, because of different experimental setup. For instance, the frequency of our processors is 2.80 GHz, whereas the frequency of processors used for LCR experiments is 1.66 GHz. In addition, MTE is sensitive to optimizations in implementation. Therefore, we define prediction-oriented throughput efficiency ($POTE$) — i.e. the theoretical ratio between the number of bytes UTO-delivered per message and the number of bytes of the message. To compute $POTE$, we assume that there are n processes participating to the algorithm, each of them UTO-broadcasting messages of an average size of s bytes. Moreover, in the case

Fig. 1. Message sequence chart with two processes (We assume $NB\_TR = 2$)



(a) Key

(b) Message Sequence Chart

of TRAINS, each wagon contains an average of u messages. We demonstrate that $POTE_{\text{TRAINS}} = \frac{(n-1)us}{10+(n-1)[7+u(5+s)]}$ (see Appendix A) and $POTE_{LCR} = \frac{s}{24+4n+s}$ (see Appendix B). Lines 2–4 of Table II synthetizes the results for $n = 5$ processes and trains with an optimal size of 4 KiB (thus, $u = 1014/(5+s)$ when $s < 1010$ and $u = 1$ otherwise). $POTE_{\text{TRAINS}}$ improves $POTE_{\text{LCR}}$ according to the size of the broadcast messages. The $POTE$ improvement reaches a peak of 250% for 10 bytes messages.

To confirm that TRAINS high throughput efficiency means high throughput, we implement TRAINS. Then, we run performance tests on $n = 5$ Dell Precision T3500 computers, equipped with processor Intel Xeon W3530 (2.80 GHz) and 4 GiB of RAM. These computers run Linux 3.14.9 SMP kernel. They are interconnected with an HP ProCurve 2610-24 switch. 10 trains rotate in parallel. Each train has an optimal size of 4 KiB (see constant OPTIM_TR_SIZE in Section IV). Line 5 of Table II contains TRAINS results. Finally, we determine MTE_{TRAINS}. We do not want to compare it to MTE_{LCR}, but to $POTE_{\text{TRAINS}}$. This comparison gives us an idea of the quality of the optimizations in our implementation. To

Algorithm 6 Procedure `updateWagToSendMaxSize(tr)` for any process p_i

```
local wagSizeDic ← {}
local bytesSober ← 0
local minSober ←⊥
local nbGreedy ← 1
for all w ∈ tr.wag so that
        w.sender ∈ lastTrsViews[id] \ ({p_i}∪
          goneSet(p_i, lastTrsViews[id], view))
  do
  wagSizeDic{w.sender} ← size(w)
  if w.sender not in lastWagSizeDic.keys then
    lastWagSizeDic{w.sender} ← 0
  end if
  if size(w) > OPTIM_TR_SIZE/size(view)
     or (size(w) ⩾ lastWagSizeDic{w.sender}
          and lastWagSizeDic{w.sender} > 0 )
  then
    nbGreedy ← nbGreedy +1
  else
    bytesSober ← bytesSober + size(w)
    if minSober ==⊥ or size(w) < minSober
    then
      minSober ← size(w)
    end if
  end if
end for
lastWagSizeDic ← wagSizeDic
if nbGreedy == size(view) then
  nbGreedy ← nbGreedy −1
end if
if minSober ==⊥ then
  minSober ← 0
end if
wagToSndMaxSize ← (OPTIM_TR_SIZE -
    bytesSober + minSober) / nbGreedy
```

determine MTE_{TRAINS}, we are missing the nominal capacity of our system. With NetPerf [20], we measure that the TCP point-to-point throughput is 94 Mb/s. The nominal capacity of our system is the point-to-point throughput multiplied by $n/(n-1)$ [7]. Therefore, the nominal capacity of our system is $94 \times n/(n-1) = 117.5$ Mb/s. MTE_{TRAINS} is a few percent below $POTE_{\text{TRAINS}}$ (see Line 6 of Table II): Our implementation is well optimized.

TABLE II. THROUGHPUT EVALUATION OF TRAINS AND LCR FOR 5 PROCESSES

UTO-broadcast size (bytes)	10	100	1 000	10 000
$POTE_{\text{TRAINS}}$	66.0%	94.3%	98.6%	99.9%
$POTE_{\text{LCR}}$	18.5%	69.4%	95.8%	99.6%
Improvement of $POTE_{\text{LCR}}$	257%	35.8%	2.9%	0.3%
Throughput of TRAINS (Mb/s)	76.1	108.7	113.6	113.9
MTE_{TRAINS}	64.8%	92.5%	96.7%	96.9%

B. Latency

The theoretical latency of broadcasting a single message is defined as the number of rounds that are necessary from the initial broadcast of message m until the last process delivers m [7]. Appendix C demonstrates that the latency of TRAINS is $L_{\text{TRAINS}} = \frac{5}{2}n - \frac{1}{2}$. The latency of LCR is $L_{\text{LCR}} = 2n - 2$ [7]. L_{TRAINS} alters L_{LCR} decreasingly with the number of participating processes (see Table III). This higher latency is due to the inherent trade-off that exists between throughput and latency [21]: Since TRAINS improves throughput, it alters latency. The worst alteration is in the case of 2 processes: 125%.

TABLE III. LATENCY EVALUATION OF TRAINS AND LCR

Number of processes	2	4	6	8	∞
L_{LCR} (rounds)	2	6	10	14	∞
L_{TRAINS} (rounds)	4.5	9.5	14.5	19.5	∞
Alteration of L_{LCR}	125.0%	58.3%	45.0%	39.3%	25%

VI. RELATED WORK

UTO-broadcast algorithms are classified into five families: fixed sequencer, moving sequencer, privilege-based, communication history, and destinations agreement [6]. Since train messages of TRAINS can be considered as tokens, TRAINS is member of the privilege-based family.

TRAINS principles are inspired by the Train algorithm where a single train rotates on a virtual ring [16]. In TRAINS, to improve throughput, several trains rotate in parallel on the virtual ring. Total order is preserved thanks to the `rotat` field in trains and wagons. Moreover, we apply the technique suggested by [7] to save bandwidth: Whenever a process p_i sends a train tr to its successor p_{i+1}, p_i discards p_{i+1}'s wagon from tr. In case of failure, the recovery of lost trains is inspired by Totem's technique to recover a lost token [13]: We resend trains that may have been potentially lost.

Concerning flow control, Ring Paxos does not propose any flow control [8]. Totem and LCR propose a mechanism that requires piggybacked data [7], [13]. FastCast proposes a mechanism that requires additional messages [9].

VII. CONCLUSION

This paper presents TRAINS, a token-based UTO-broadcast algorithm. We propose a flow control that requires no additional messages, nor piggybacked data on messages. This paper estimates TRAINS performance. For 5 processes, we evaluate that $POTE_{\text{TRAINS}}$ improves $POTE_{\text{LCR}}$. The $POTE$ improvement reaches a peak of 250% for 10 bytes messages. Experimental evaluation confirms TRAINS high throughput capabilities. The trade-off of this throughput improvement is the alteration of the latency. The worst alteration is in the case of 2 processes: 125%. The main perspective of our work is to improve TRAINS latency in order to make TRAINS suitable for the context of geo-replicated databases. A C implementation of TRAINS is available at https://github.com/simatic/TrainsProtocol. A Java implementation of TRAINS is available at https://github.com/simatic/TrainsProtocolJava.

ACKNOWLEDGMENT

The authors would like to thank Denis Conan (Télécom SudParis) and Eric Gressier-Soudan (CNAM-CÉDRIC) for their careful rereading and fruitful suggestions, and Vivien Quéma and Gautier Berthou (Grenoble University) for giving access to their implementation of LCR.

References

[1] P. Bettner and M. Terrano, "1500 Archers on a 28.8: Network Programming in Age of Empire and Beyond," in *Proceedings of the 2001 Game Developer Conference*. San Jose, California, USA, March 2001.

[2] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer Networks*, vol. 2, no. 2, pp. 95–114, May 1978.

[3] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, pp. 299–319, December 1990.

[4] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," Cornell University, Ithaca, NY, USA, Tech. Rep. TR94-1425, 1994.

[5] J.-M. Chang and N. F. Maxemchuk, "Reliable broadcast protocols," *ACM Trans. on Comput. Syst.*, vol. 2, no. 3, pp. 251–273, 1984.

[6] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, pp. 372–421, December 2004.

[7] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Trans. on Comput. Syst.*, vol. 28, pp. 5:1–5:32, July 2010.

[8] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: A High-Throughput Atomic Broadcast Protocol," in *Proceedings of 40th International Conference on Dependable Systems and Networks (DSN 2010)*, 2010.

[9] G. Berthou and V. Quéma, "FastCast: a Throughput- and Latency-efficient Total Order Broadcast Protocol," in *Proceedings of the International Middleware Conference (Middleware)*, ser. Middleware '13, 2013.

[10] X. Défago, A. Schiper, and P. Urbán, "Comparative performance analysis of ordering strategies in atomic broadcast algorithms," *IEICE Trans. Inf. Syst. E86-D*, pp. 2698–2709, December 2003.

[11] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal, "An efficient reliable broadcast protocol," *SIGOPS Oper. Syst. Rev.*, vol. 23, pp. 5–19, October 1989.

[12] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. on Comput. Syst.*, vol. 5, pp. 47–76, January 1987.

[13] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella, "The Totem single-ring ordering and membership protocol," *ACM Trans. on Comput. Syst.*, vol. 13, pp. 311–342, November 1995.

[14] Spread, "The Spread Toolkit," http://www.spread.org/, October 1998.

[15] R. Friedman and R. van Renesse, "Packing messages as a tool for boosting the performance of total ordering protocols," in *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing 1997*, August 1997, pp. 233–242.

[16] F. Cristian, "Asynchronous atomic broadcast," *IBM Technical Disclosure Bulletin*, vol. 33, no. 9, pp. 115–116, February 1991.

[17] M. Simatic, "Contributions au rendement des protocoles de diffusion à ordre total et aux réseaux tolérants aux délais à base de RFID (Contributions to efficiency of total order broadcast protocols and to RFID-based delay tolerant networks, in French)," Ph.D. dissertation, Centre d'Étude et De Recherche en Informatique du Cnam (CÉDRIC) / Conservatoire National des Arts et Métiers (CNAM) / École Doctorale ÉDITE, 2012.

[18] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*. New York, NY, USA: ACM, 1987, pp. 123–138.

[19] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of ACM*, vol. 43, pp. 225–267, March 1996.

[20] R. Jones, "Netperf," http://www.netperf.org/, 2007.

[21] P. Urbán, X. Défago, and A. Schiper, "Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms," in *Proceedings of the Nineth International Conference on Computer Communications and Networks 2000*, October 2000, pp. 582–589.

Appendix

A. $POTE_{\text{TRAINS}}$

Upon receiving a train, a process is able to deliver the UTO-broadcast contained in the wagons attached to the train received two rotations before. Therefore, to calculate $POTE_{\text{TRAINS}}$, we analyze the structure of a train message. A UTO-broadcast message is stored inside a wagon with the length of the message (4 bytes), a message type (1 byte) and the message itself (s bytes). Each wagon contains a length (4 bytes), the address of the sender (reduced to 2 bytes [17]), the `rotat` field of this wagon (1 byte) and an average of u messages stored in this wagon. A train contains a length (4 bytes), a message type (1 byte), a field related to integrated membership service (2 bytes [17]), an `id` field (1 byte, since we assume there will be no more than 256 trains circulating in parallel on the virtual ring), a logical clock (shrunk to 1 byte [17]), a `rotat` field (1 byte), and the wagons. We conclude: $POTE_{\text{TRAINS}} = \frac{(n-1)us}{10+(n-1)[7+u(5+s)]}$

B. $POTE_{\text{LCR}}$

Each LCR message m_j contains piggybacked information concerning a previously received message m_k. So, when a process receives m_j, it is able to deliver the UTO-broadcast contained in m_k. Therefore, to calculate $POTE_{\text{LCR}}$, we analyze the structure of LCR messages implemented by LCR authors. Each LCR message contains the following fields: the type of the message (coded as a C++ `enum`: 4 bytes), the address of the sender (4 bytes), the identifier of the message (4 bytes), the piggybacked acknowledgement of a received message m made of m's sender address (4 bytes) and m's identifier (4 bytes), the vector clocks (n participating processes $\times$ 4 bytes), the size of carried UTO-broadcast message (4 bytes), and the UTO-broadcast message itself (s bytes). We conclude: $POTE_{LCR} = \frac{s}{24+4n+s}$

C. *Latency of* TRAINS

This appendix details the computation of TRAINS latency.

Let n be the number of participating processes. Let $p_0, \ldots, p_{n-1}$ be these processes. Let p_0 be the process that increments the rotation field of a received train. We assume that there are enough rotating trains in parallel so that, when a process wants to UTO-broadcast a message, a train is immediately available to take the wagon containing this message.

When $p_{i,i\in[\![0,n[\![}$ UTO-broadcasts a message m, it is put in a wagon w. w is attached to train tr that is immediately available. Moreover, `w.rotat` is set to `t.rotat`. p_0 is the process incrementing `t.rotat`. So, the last process that sees the incremented value of `t.rotat` is p_{n-1}. Train tr requires $n-i-1$ rounds to go from p_i to p_{n-1}. Afterwards, p_{n-1} has to wait for two rotations of the train before p_{n-1} receives train tr with $\texttt{w.rotat} == (\texttt{t.rotat} + 1) \mod \texttt{NB\_RO}$. So, it takes $n-i-1+2n$ rounds for p_{n-1} to UTO-deliver w and thus m. In other words, latency for messages UTO-broadcast by $p_{i,i\in[\![0,n[\![}$ is $L_{UTO}(p_i) = n-i-1+2n$. In average, $L_{\text{TRAINS}} = \frac{1}{n}\sum_{i=0}^{n-1} L_{UTO}(p_i)$. We conclude that $L_{\text{TRAINS}} = \frac{5}{2}n - \frac{1}{2}$.