The convoy effect in atomic multicast

Tarek Amhed-Nacer, Pierre Sutra, Denis Conan

The Convoy Effect in Atomic Multicast

Tarek Ahmed-Nacer, Pierre Sutra, and Denis Conan
SAMOVAR, Télécom SudParis, CNRS, Université Paris-Saclay
Évry, France
e-mail: firstname.lastname@telecom-sudparis.eu

***Abstract*—Atomic multicast is a group communication primitive that allows disseminating messages to multiple distributed processes with strong ordering properties. As such, atomic multicast is a widely-employed tool to build large-scale systems, in particular when data is geo-distributed and/or replicated across multiple locations. However, all the most efficient atomic multicast algorithms suffer from a convoy effect that slows down the delivery of messages. In this paper, we study the impact of this phenomenon in detail. To this end, we first capture the convoy effect in the critical section problem with a timed automaton. We then extend this approach to the seminal atomic multicast solution of Skeen. Our analytical model shows that the convoy effect quickly degrades the latency of messages. We confirm this claim by fitting our model with empirical data from literature. To sidestep this performance degradation, we advocate the use of message semantics in atomic multicast. In particular, we present a simple protocol that reduces the convoy effect by a factor ρ, where ρ is the probability that two messages commute.**

I. Introduction

Cloud computing is a recent paradigm for the dynamic provisioning of Internet-based services. Typically supported by state-of-the-art data centers containing ensembles of networked Virtual Machines (VMs), the Cloud delivers infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS), and Data as a Service (DaaS). Using such services, enterprises may offload their computing infrastructure to right-size their expenditure and reduce the time-to-market of their products.

Typical cloud applications are distributed among several (virtualized) machines. As a consequence, building such applications require the ability to disseminate messages among the infrastructure. To this end, Cloud developers usually rely on an underlying group messaging system, such as Apache Kafka[1], RabbitMQ[2], or JGroups[3]. Core properties of a group messaging system includes performance (in message delay and bandwidth usage), dependability (fault-tolerance), security, and ordering.

Message ordering guarantees that the order in which messages are received among recipients satisfies some property. For instance, this order can be first-in first-out, causally consistent or total. In particular, to totally order messages processes have to agree on a common gap-free delivery sequence. This agreement, or consensus, is a well-studied problem in the case where messages are always delivered to the same set of processes. An efficient solution is the acclaimed Paxos protocol [1], a core building block of many Cloud services such as Google App Engine, or Amazon Web Services [2]. On the other hand, when the set of recipients varies, processes have to implement an atomic multicast abstraction [3]. Unfortunately, in the state of our knowledge, existing atomic multicast solutions suffer all from various drawbacks. In this paper, we focus on the problem caused by the convoy effect [4]. When using group communication in a distributed environment, the convoy effect is the fact that one or more message deliveries are delayed by other ones, e.g., the delivery of local messages is delayed by as much as the latency of global messages [5].

The convoy effect is exacerbated in the Cloud landscape of services that are increasingly becoming global. Indeed, the Cloud is today migrating more and more to the edge of the network, where routers themselves become the virtualization infrastructure, in an evolution labelled as "Fog computing" [6]. Future Clouds are also expected to aggregate a high number of diverse and geographically distributed data centers and future data stores will consist of hundreds or even thousands of geo-distributed sites [7]. In this context, the convoy effect may reveal a burden when disseminating messages at the scale of multiple geo-distributed sites.

This paper makes a first step in the direction of understanding and circumventing the impact of the convoy effect in the atomic multicast primitive. We articulate our approach as follows. First, we identify the convoy effect in the critical section problem with a simple timed automaton. We then extend this approach to the seminal atomic multicast solution of Skeen. Our analytical model shows that the convoy effect quickly degrades the latency of messages. We confirm this claim by fitting our model with empirical data from literature. To sidestep this performance degradation, we propose to leverage the semantics of messages. Our last contribution is a simple variation of Skeen's protocol that reduces the convoy effect by a factor ρ, where ρ is the probability that two messages commute at the application level.

Outline. In Section II, we characterize analytically with a timed automaton the convoy effect in the case of the shared access to a critical section. Then, in Section III, we extend our approach to a well-known atomic multicast solution, and validate it using empirical data. In Section IV, we propose to inject the message semantics known at the application level inside the atomic multicast primitive in order to reduce the convoy effect. We survey related work in Section V and conclude in Section VI.

[1] http://kafka.apache.org
[2] http://jgroups.org
[3] http://www.rabbitmq.com

II. The case of System R

Blasgen et al. [4] study the convoy effect in System R, an early database management design that supports both the relational model and transactions [8]. To the best of our knowledge, this is the first systematic study of this phenomenon in a concurrent system. In this section, we recall the notion of convoy effect as proposed by Blasgen et al. [4], then we present a timed automaton to capture analytically this effect.

A. First observations

System R employs locks to orchestrate transactions that access the shared resources. To execute an operation on a resource, a process executing a transaction locks the resource, uses it, then unlocks the resource. As observed by the authors of [4], processes applying this discipline tend to "bump into one another" when contending for shared resources, forming on each lock a "convoy of waiters".

With more details, Blasgen et al. [4] model the convoy effect as a queue attached to the resource and that represents the waiting processes. Following the terminology in [4],

- The *duration of a lock* (d) is the average number of instructions executed while the lock is held;
- The *execution interval of a lock* (i) is the average number of instructions executed between two successive requests to the lock by a process; and
- The *collision cross section of a lock* (CCS) is the fraction of time during which the resource is granted. In a uniprocessor, the collision cross section equals $d/(d+i)$, ignoring the waiting time and the task switching time.

At the light of such definitions, the authors of [4] conclude that the higher the CCS ratio is, the more likely a convoy appears on the lock. Below, we refine these observations to obtain an analytical value of the convoy effect.

B. Refinements

Our analysis builds upon the decomposition of [4]. Figure 1 presents a timed automaton that models the concurrent execution of a set Π of n processes accessing a lock. Nodes in Figure 1 are tuples of the form (Q, D, I), where $Q, D, I \in \mathbb{N}$ indicate the number of processes respectively in the queue, holding the resource (critical section), and inside the execution interval of the lock. In Figure 1, labels of the form "$d/[i = d]$" means "after d units of time or instructions, if the condition $i = d$ holds, then the transition is triggered".

Initially, all the processes await in the queue, modeled by the state $(n, 0, 0)$ at the top of Figure 1. Then, some process p is granted the resource. The transition to the next state is immediate and unrestricted. Since all the other processes are stacked in the queue, the next state is $(n-1, 1, 0)$. Then, the transition to the next state occurs when the lock is released after d instructions. When p releases the resource, another process q gains access to the resource and the system reaches the state $(n-2, 1, 1)$.

For the sake of clarity, we assume that i and d are congruent with $\frac{i}{d} < n - 1$.[4] Under this hypothesis, the transition to the next state occurs at the time process q releases the lock, i.e., after d units of time. We then have to consider two cases:

- ($i = d$) Process p leaves the execution interval of the lock at the same time and it makes another request for the resource. The system thus stays in $(n-2, 1, 1)$.
- ($i > d$) Process p is still in the execution interval of the lock when process q releases the lock and another process accesses the resource. The system moves to $(n-3, 1, 2)$.

By iterating the above reasoning, we deduce that the system reaches a stable state verifying the condition $n = min(n-1, \frac{i}{d}) + 1 + Q$. It follows from the definition of CCS, that the convoy effect $\mathcal{CE}$ satisfies:

$$\mathcal{CE} = d \times Q = d \times (n - \frac{1}{CCS}) \quad (1)$$

Equation (1) confirms the informal arguments in [4]. Precisely, it tells us that the convoy effect is linearly proportional to the inverse of the CCS ratio.

In the next section, we follow the same approach to deduce the convoy effect in the case of atomic multicast.

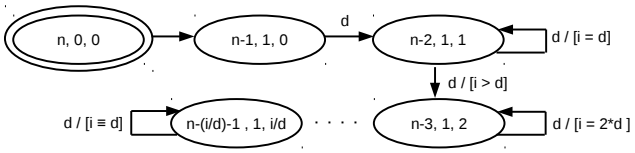


Fig. 1: Convoy effect in System R

III. Transposition to Atomic Multicast

Atomic multicast allows to propagate messages in an ordered manner to any number of processes in the system. In what follows, we first recall the definition of this group communication primitive, then we study the convoy effect in the algorithmic solution proposed by Skeen [9, 10]. As we shall see, our reasoning also extends easily to other atomic multicast protocols in literature.

A. Atomic Multicast

Let us note Msg some set of messages. Atomic multicast is defined by the operations AM-Cast(m) and AM-Deliver(m), where $m \in Msg$. Operation AM-Cast(m) allows a process to *multicast* message m to some set $dst(m)$ of processes. A process *delivers* message m when it executes AM-Deliver(m). During every run of atomic multicast, the following properties are verified:

- **Integrity.** For any process p and any message m, p delivers m at most once and only if p belongs to $dst(m)$ and m was previously multicast.
- **Validity.** If a process p multicasts a message m, eventually every process in $dst(m)$ delivers it.
- **Ordering.** Given two messages m and m', we write $m \prec m'$ when some process p delivers m before m'. The transitive closure of relation $\prec$ is a strict partial order over Msg.

The validity and integrity properties define reliable multicast, and together with the ordering property, they define atomic

[4] In the general case, the timed automaton slightly differs from the one we present in Figure 1. The system eventually oscillates between two states, and our results would vary by an additive constant of 1.

Algorithm 1 Skeen's algorithm – code at process p

```
1: Variables:
2:     clock                                                    // Initially, 0
3:     Pending                                                  // Initially, ∅
4:     Delivering                                               // Initially, ∅
5:     Delivered                                                // Initially, ∅
6:
7: AM-Cast(m) :=
8:     eff: forall q ∈ dst(m)
9:              send ⟨m⟩ to q
10: assignTimestamp(m) :=
11:    pre: received ⟨m⟩
12:    eff: clock ← clock + 1
13:         ts ← clock
14:         Pending ← Pending ∪ {(m, ts)}
15:         send ⟨m, ts⟩ to coord(m)
16: computeSeqNumber(m) :=
17:    pre: ∀q ∈ dst(m) : received ⟨m, _⟩ from q
18:    eff: sn = max({ts : received ⟨m, ts⟩})
19:         forall q ∈ dst(m)
20:           send ⟨m, sn⟩ to q
21: assignSeqNumber(m) :=
22:    pre: (m, _) ∈ Pending
23:         received ⟨m, sn⟩ from coord(m)
24:    eff: clock ← max({clock, sn})
25:         Pending ← Pending \ {(m, _)}
26:         Delivering ← Delivering ∪ {(m, sn)}
27: doDeliver(m) :=
28:    pre: (m, x) ∈ Delivering
29:         ∀(m′, y) ∈ Pending ∪ Delivering : (x, m) < (y, m′)
30:    eff: Delivering ← Delivering \ {(m, x)}
31:         Delivered ← Delivered ∪ {m}
32:         AM-Deliver(m)
```

multicast. We note here that our definition is for failure-free system. Other definitions appear in literature, in particular in the case where processes may crash [11, 12].

B. A Classical Solution

Algorithm 1 depicts the pseudo-code of Skeen's solution. This algorithm requires some arbitrary global ordering $<$ over the set of messages. Given $(x, m), (y, m') \in \mathbb{N} \times Msg$, we define $(x, m) < (y, m')$ as $x < y \vee (x = y \ \wedge \ m < m')$. In addition, given a message m, Algorithm 1 assumes some *coordinator* for m, denoted $coord(m)$. This coordinator is usually taken among the recipients of m.

Algorithm 1 consists of a set of actions, each having some effects (**eff**), guarded by one or more preconditions (**pre**). When all the preconditions in the **pre** block are true, the instructions in the corresponding **eff** block are triggered.

The algorithm makes use of three variables: a local clock ($clock$) and three buffers ($Pending$, $Delivering$ and $Delivered$). We detail their roles in what follows.

To atomic multicast a message m to $dst(m)$, a process p sends iteratively m to the recipients in $dst(m)$. Every process $q \in dst(m)$ that receives such a message computes a timestamp for m. To this end, q first increases its local clock and assigns it to variable ts. Then, q sends the pair (m, ts) to $coord(m)$, the process in charge of coordinating the delivery of m. Once process $coord(m)$ knows all the timestamps attributed by the processes in $dst(m)$, it defines the sequence number for m (variable sn) as the maximum of such timestamps, and sends it to $dst(m)$. Every process $q \in dst(m)$ that receives sn, removes m from its $Pending$ buffer and stores (sn, m) in the $Delivering$ buffer. Process q also updates its local clock with the maximum of its current value and sn. At some later point

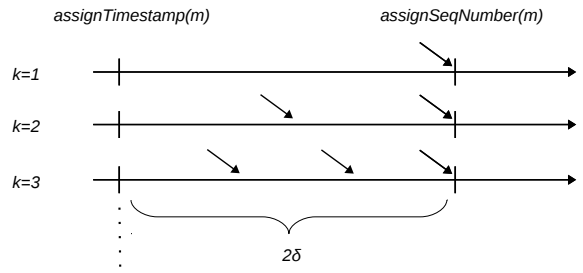


Fig. 2: Arrival of concurrent messages in Algorithm 1 *(δ is the average message delay)*

in time, q delivers message m with sequence number x when for every message m' in either $Pending$ or $Delivering$ with a timestamp or a sequence number y, $(x, m) < (y, m')$ holds. Message m is then moved into the $Delivered$ buffer.

The original solution of Skeen does not advance the local clock with the sequence numbers attributed to messages (line 24). Let us notice that, without this step, an arbitrary long convoy effect may hinder the protocol. For instance, if process p having $clock = 1$ receives a message m with $sn = 100$, then up to 99 messages may delay the delivery of m. To the best of our knowledge, line 24 appears first as a by-product of the fault-tolerant variation proposed by Fritzke et al. [13]. This step tempers the convoy effect in Algorithm 1 yet, as we shall see next, does not completely remove it.

C. Convoy Effect

In Algorithm 1, for some message m having a sequence number sn, action $doDeliver(m)$ does not trigger as long as there exists a message not yet delivered with a timestamp (or a sequence number) lower than sn. This dependency among messages creates a convoy effect. Similarly to Section II, we can model this phenomenon with a timed automaton.

To this end, we first observe that a process p can tag a message m' with a lower timestamp only in the interval between actions $assignTimestamp(m)$ and $assignSeqNumber(m)$. We note k the number of messages received between these two events, and for the sake of simplicity we assume that when k messages are received, they split evenly the interval in k parts. Figure 2 illustrates this situation for $k = 1..3$, where we denote δ the average message delay between processes.

As in Section II, we employ a timed automaton to model the convoy effect in Skeen's solution. We depict the result in Figure 3. Each state of Figure 3 corresponds to a tuple (P, D, L), where P, D, and L denote respectively the cardinals of variables $Pending$, $Delivering$, and $Delivered$. In this figure, a message m enters first the $Pending$ buffer. Then, after 2δ units of time, m is moved into the $Delivering$ buffer. Message m transits to $Delivered$ after a certain amount of time depending on the number of concurrent messages received between actions $assignTimestamp(m)$ and $assignSeqNumber(m)$. In Figure 3, this corresponds to the guards of the form "$[k = X]$".

With more details, consider that k messages are received in the interval, and note m' the last such message. All the

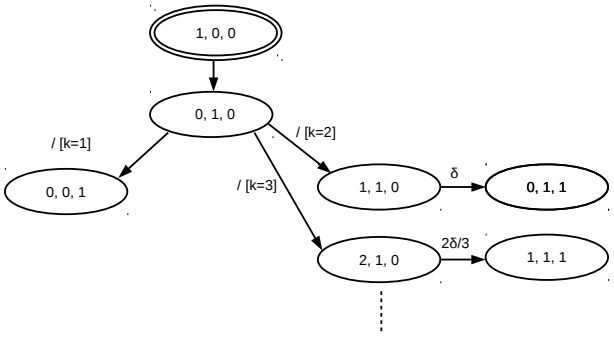


Fig. 3: Convoy effect in Algorithm 1

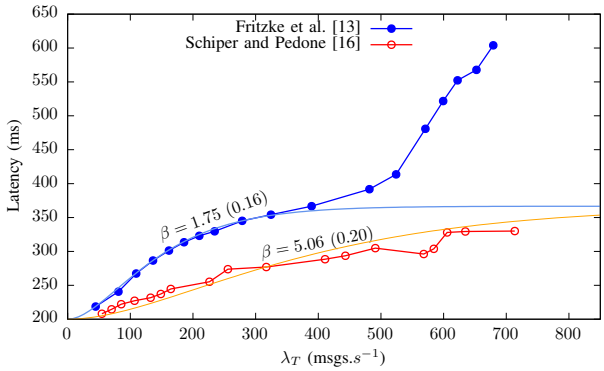


Fig. 4: Fitting the model with experimental data from [5] ($n = 4, s = 2, \delta = 100$)

messages received after $assignSeqNumber(m)$ have a higher timestamp than m. Therefore, since m' is the last message in the interval, when $assignSeqNumber(m')$ takes place either *(i)* m' has a higher sequence number than m, or *(ii)* m' is immediately delivered. In both cases, m waits at most $\frac{k-1}{k} \times 2\delta$ units of time.

Let us note P_c the probability of conflict between two messages, i.e., the probability that two messages have some recipient in common. In addition, let us consider that the arrival of messages follows a Poisson distribution. From Figure 3, we may evaluate the convoy effect in Algorithm 1 as follows:[5]

$$\begin{aligned}\mathcal{CE} &\geq \sum_{k=1}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \times \frac{k-1}{k} \times 2\delta \times P_c \\ &= 2\delta P_c e^{-\lambda} \left[\sum_{k=1}^{\infty} \frac{\lambda^k}{k!} - \frac{\lambda^k}{k(k!)} \right] \\ &= 2\delta P_c e^{-\lambda} \left[e^{\lambda} - 1 - \sum_{k=1}^{\infty} \frac{\lambda^k}{k(k!)} \right] \\ &= 2\delta P_c (1 - e^{-\lambda}(1 + Ei(\lambda) - \log(n) - \gamma)) \end{aligned} \quad (2)$$

where Ei is the exponential integral and γ the Euler–Mascheroni constant [14].

To obtain the value of P_c, we compute instead $1 - P_{\overline{c}}$, where $P_{\overline{c}}$ is the probability that two messages do not conflict. Let us note s the average size of $dst(m)$. The computation of $P_{\overline{c}}$ goes as follows:

$$P_{\overline{c}} = \frac{\binom{s}{n} \times \binom{s}{n-s}}{\binom{s}{n}^2} = \frac{\prod_{i=s}^{2s-1}(n-i)}{\prod_{i=0}^{s-1}(n-i)} \quad (3)$$

D. Latency

From the fact that in Algorithm 1 a message freshly multicast reaches a process in one message delay before action $assignTimestamp()$ triggers, the average latency of Skeen's algorithm is given by:

$$Latency([9]) = 3\delta + \mathcal{CE} \quad (4)$$

When using this analytical model, we have to keep in mind two considerations: First, parameter λ is obtained by dividing the total message rate in the system, say λ_T, with the size of the interval between actions $assignTimestamp()$ and $assignSeqNumber()$, i.e., 2δ. Second, most atomic multicast protocols employ batching, that is they group under the same timestamp multiple messages sent to the same location. Batching tends to synchronize (closed loop) clients as they receive at the same time the notification that their previous messages were delivered. If we note β the average batching size, we define parameter λ in Equation (2) as:

$$\lambda = \frac{\lambda_T}{2\delta\beta} \quad (5)$$

Equation (4) easily extends to variations of Algorithm 1. For instance, the optimization in [3] sends the timestamp to all the processes in $dst(m)$, and each such process computes locally the sequence number. This skips the need for a coordinator, thus improving latency at the cost of message complexity. In such a case the algorithm exhibits the following latency:

$$Latency([3]) = 2\delta + \mathcal{CE} \quad (6)$$

Our model also easily extends to algorithms that add a fault-tolerant mechanism based on groups (e.g., [5, 13, 15, 16]). At core, such algorithms emulate a process in Algorithm 1 by running consensus between processes in the same group ([15]). In particular, our approach directly applies to a geo-distributed system where the intra-group latency (at a site) is negligible over the inter-group latency (between sites).

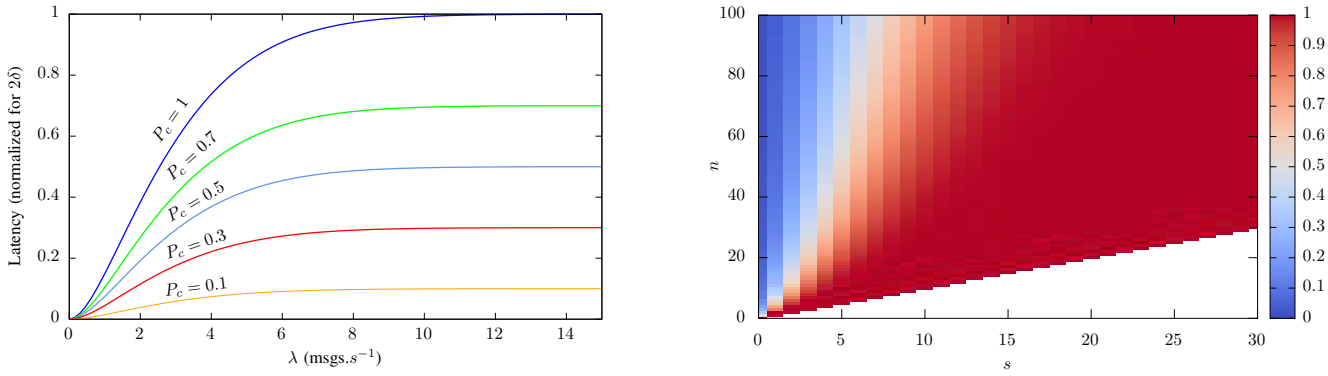
E. Validation and Impact

This section validates our model of the convoy effect in atomic multicast, and studies its impact both using experimental data and analytically.

In Figure 4, we fit our model with the experimental results reported in [5, Figure 3(b)]. These experiments take place in an emulated geo-distributed system of four sites. The average message delay (δ) across sites equals 100ms; at a site it equals 0.1ms. A client sends either global messages to two random sites (75%), or local messages to its local site (25%).

Figure 4 depicts the latency of global messages when considering the algorithms of Schiper and Pedone [16] and Fritzke et al. [13]. For each algorithm, the plain curve indicates

[5]Our result is a lower bound since if the two messages do not collide, we do not consider the contribution of the remaining $k-2$ messages.



(a) Convoy effect when varying P_c

(b) P_c when varying n and s

Fig. 5: Simulation results

the results obtained with Equation (6). In both cases, we used a least square regression method to fit the average batching rate (β) before the saturation point. We report β for each curve, together with the standard deviation error between the model and the data (in brackets).

Figure 4 tells us that our model matches precisely the behavior of the algorithm of [13] before it saturates. Regarding the algorithm of [16], the results reported in [5] are more noisy and as a consequence our model fitting is less precise.

In Figure 4, we observe a large difference between the two algorithms regarding β (precisely, a factor 2.84). This comes from the fact that the two algorithms do not handle similarly messages addressed to the local site. The algorithm of Fritzke et al. [13] delivers such messages after consensus. In [16], they follow the same path as global messages, i.e., they are timestamped and go through two consensus instances. As a consequence, local messages are delayed by global ones and delivered at the same time. This improves the positive effect of batching.

To further understand how the convoy effect impacts atomic multicast, we conduct two simulations with the help of our analytical model. The results are presented in Figure 5. Figure 5a plots how the convoy effect evolves according to the message arrival rate for various values of P_c. This figure tells us that the convergence toward $2\delta P_c$ is fast.

In Figure 4, concurrent messages collide with a probability $P_c = 5/6$ (see Equation (3)). Figure 5b simulates how this probability evolves when varying parameters n and s. To this end, we consider $n < 100$, a number that makes sense in a geo-distributed setting for a system that consists of a few dozens of sites. At the light of the results reported in Figure 5b, we can observe that the probability of conflict is non-negligible, even if each message is addressed to a few processes.

Overall, our evaluation shows that the convoy effect in atomic multicast has a significant impact on the latency of messages, even if they are targeting a small number of recipients. In the next section, we propose to leverage the semantics of messages in order to partly avoid this undesirable phenomenon.

IV. Primer of a solution

Atomic multicast assumes that all the pairs of messages conflict and should be ordered as soon as they have a common destination. On the contrary, in reliable multicast this conflict relation is empty. Similarly to the work of Pedone and Schiper [17], we propose to consider the generic multicast problem, where we would have some binary relation $\asymp$ defining that two messages conflict, i.e., do not commute at the application level. In what follow, we specify this distributed task then we propose a variation of Algorithm 1 as a solution.

We state the generic multicast problem as follows:

- **Integrity and Validity.** Identical to the definitions given in Section III-A.
- **Ordering.** The transitive closure of relation $\prec$ reduced to $\asymp$ is a strict partial order.

Any algorithm that solves atomic multicast trivially solves any instance of generic multicast, but it orders more messages than necessary. To solve efficiently generic multicast, we observe that we may adapt the precondition of action $doDeliver(m)$ in Algorithm 1, line 29, as follows:

$$\forall (m', y) \in Pending \cup Delivering : \\ (x, m) < (y, m') \vee (m \not\asymp m')$$

Let us note ρ the proportion of messages that commute at the application level. With the above modification, the convoy effect of Skeen's algorithm now equals $\rho \times \mathcal{CE}$, where $\mathcal{CE}$ is given by Equation (2).

V. Related work

The use of application semantics in group communication was originally introduced for consensus [17, 18], It allows to solve this distributed task optimally in two messages delays [19]. Guerraoui and Schiper [20] propose a tunable multicast primitive to either take a distributed lock, or commit a global transaction.

Several works [21, 22] observe the impact of the convoy effect on transactional systems. They try to reduce it, notably between global and local transactions. One solution consists in reordering transactions that commute after their deliveries by the group communication primitive.

We may classify atomic multicast algorithms proposed in literature into three categories: *non-genuine*, *quasi-genuine* and *genuine* algorithms. In the case of non-genuine algorithms, the

base idea is to execute an atomic broadcast protocol, pruning upon reception the messages that are not addressed to the local process. Due to its large overhead, the scalability of this type of algorithm is inherently limited [5].

A genuine algorithm solely allows a process in $dst(m)$ to execute steps when delivering message m. Almost all algorithms in this category are variations of Skeen's solution. One notable exception is the work of Delporte-Gallet and Fauconnier [15]. This algorithm assumes some ordering $<_\Pi$ over the processes. Upon AM-Cast(m), the first process in $dst(m)$ receives message m and forwards it to the next process in the order $<_\Pi$, then blocks. The last process in the chain sends an acknowledgment to $dst(m)$, allowing the delivery of m. At the light of this mechanism, the algorithm suffers from a large convoy effect, that we believe can also be characterized with our model.

Quasi-genuine algorithms offer a middle ground solution between the two previous categories. Multi-Ring Paxos [23] is a fault-tolerant atomic multicast algorithm that organizes processes in groups, each group executing a Paxos consensus algorithm. A client interested in some set of groups, e.g., $dst(m) = \{g_1, g_2, \ldots\}$, installs a Paxos learner to receive the messages from each of the group. This learner is in charge of delivering the messages in some arbitrary order once it receives some fixed value M of consensus instances from each group. To cope with varying message rates, Multi-Ring Paxos can skip several consensus instances at a time. This algorithm can be viewed as an extension of the deterministic merge broadcast algorithm of Aguilera and Strom [24]. To keep a low overhead, a Multi-Ring Paxos client has to install a learner for each new value of $dst(m)$. Hence, the number of consensus learners quickly grows with the number of groups. This limits the scalability of the deterministic merge approach.

VI. Conclusion

This paper studies the convoy effect, a perturbation that occurs in parallel systems when concurrency on shared resources increases. We first study this phenomenon in the critical section problem, from which we derive a simple model based on a timed automaton and a general formula, in line with the conclusions of Blasgen et al. [4]. Then, we transpose our approach to the case of atomic multicast, and the seminal algorithm of Skeen. We observe that the convoy effect quickly degrades the latency of messages, a claim that we assess by fitting our model with empirical data from literature. To sidestep the loss of performance due to the convoy effect, we propose to leverage the semantics of messages in atomic multicast. To this end, we specify the generic multicast problem and propose a simple variation of Skeen's solution that reduces the convoy effect by a factor ρ, where ρ is the probability that two messages commute at the application level.

As a future work, we plan to refine our model, in particular by including multiple classes of messages, with various arrival rates and probabilities of collision. This should allow us to fit more empirical data from literature. We are also interested in designing a solution to generic multicast that boils down to reliable multicast when all the messages commute.

Acknowledgment

The authors thank Pascal Hennequin for his fruitful discussion on Section III.

References

[1] L. Lamport, "The Part Time Parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.
[2] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos Made Live: An Engineering Perspective," in *PODC'07*, August 2007, pp. 398–407.
[3] R. Guerraoui and A. Schiper, "Genuine atomic multicast in asynchronous distributed systems," *Theoretical Computer Science*, vol. 254, no. 1, pp. 297–316, March 2001.
[4] M. Blasgen, J. Gray, M. Mitoma, and T. Price, "The convoy phenomenon," *ACM SIGOPS Operating Systems Review*, vol. 13, no. 2, pp. 20–25, April 1979.
[5] N. Schiper, P. Sutra, and F. Pedone, "Genuine versus Non-Genuine Atomic Multicast Protocols for Wide Area Networks: An Empirical Study," in *SRDS'09*, September 2009, pp. 166–175.
[6] L. M. Vaquero and L. Rodero-Merino, "Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 27–32, October 2014.
[7] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," in *Proceedings of the first workshop on Mobile Cloud Computing*. ACM, August 2012, pp. 13–15.
[8] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: Relational Approach to Database Management," *ACM Transactions Database Systems*, vol. 1, no. 2, pp. 97–137, June 1976.
[9] K. P. Birman and T. A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Transactions on Computers Systems*, vol. 5, no. 1, pp. 47–76, January 1987.
[10] R. Guerraoui and A. Schiper, "Total order multicast to multiple groups," in *ICDCS'97*, May 1997, pp. 578–585.
[11] V. Hadzilacos and S. Toueg, in *Distributed Systems (2nd Ed.)*, S. Mullender, Ed. ACM Press/Addison-Wesley Publishing Co., 1993, ch. Fault-tolerant Broadcasts and Related Problems, pp. 97–145.
[12] X. Défago, A. Schiper, and P. Urbán, "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey," *ACM Computing Survey*, vol. 36, no. 4, pp. 372–421, December 2004.
[13] U. Fritzke, P. Ingels, A. Mostefaoui, and M. Raynal, "Fault-tolerant Total Order Multicast to asynchronous groups," in *SRDS'98*, October 1998, pp. 228–234.
[14] C. M. Bender and S. A. Orszag, *Advanced Mathematical Methods for Scientists and Engineers*, ser. International series in pure and applied mathematics. McGraw-Hill, 1978.
[15] C. Delporte-Gallet and H. Fauconnier, "Fault-Tolerant Genuine Atomic Multicast to Multiple Groups," in *OPODIS'00*, December 2000, pp. 107–122.
[16] N. Schiper and F. Pedone, "On the Inherent Cost of Atomic Broadcast and Multicast in Wide Area Networks," in *ICDCN'08*, 2008, pp. 147–157.
[17] F. Pedone and A. Schiper, "Generic broadcast," in *DISC'99*, 1999, pp. 94–106.
[18] L. Lamport, "Generalized Consensus and Paxos," Microsoft, Tech. Rep. MSR-TR-2005-33, March 2005.
[19] L. Lamport, "Future Directions in Distributed Computing," A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, Eds. Springer-Verlag, 2003, ch. Lower Bounds for Asynchronous Consensus, pp. 22–23.
[20] G. Guerraoui and A. Schiper, "A generic multicast primitive to support transactions on replicated objects in distributed systems," in *FTDCS'95*, August 1995, pp. 334–342.
[21] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *SRDS'10*, September 2010.
[22] D. Sciascia and F. Pedone, "Geo-replicated storage with scalable deferred update replication," in *DSN'13*, June 2013, pp. 1–12.
[23] P. J. Marandi, M. Primi, and F. Pedone, "Multi-Ring Paxos," in *DSN'12*, June 2012, pp. 1–12.
[24] M. K. Aguilera and R. E. Strom, "Efficient Atomic Broadcast Using Deterministic Merge," in *PODC '00*, July 2000, pp. 209–218.