EpTO: An Epidemic Total Order Algorithm for Large-Scale Distributed Systems

Miguel Matos
HASLab – INESC TEC &
University of Minho
Portugal
miguelmatos@di.uminho.pt

Hugues Mercier
Institute of Computer Science
University of Neuchâtel
Switzerland
hugues.mercier@unine.ch

Pascal Felber
Institute of Computer Science
University of Neuchâtel
Switzerland
pascal.felber@unine.ch

Rui Oliveira
HASLab – INESC TEC &
University of Minho
Portugal
rco@di.uminho.pt

José Pereira
HASLab – INESC TEC &
University of Minho
Portugal
jop@di.uminho.pt

ABSTRACT

The ordering of events is a fundamental problem of distributed computing and has been extensively studied over several decades. From all the available orderings, total ordering is of particular interest as it provides a powerful abstraction for building reliable distributed applications. Unfortunately, deterministic total order algorithms scale poorly and are therefore unfit for modern large-scale applications. The main contribution of this paper is EpTO, a total order algorithm with probabilistic agreement that scales both in the number of processes and events. EpTO provides deterministic safety and probabilistic liveness: integrity, total order and validity are always preserved, while agreement is achieved with arbitrarily high probability. We show that EpTO is well-suited for large-scale dynamic distributed systems: it does not require a global clock nor synchronized processes, and it is highly robust even when the network suffers from large delays and significant churn and message loss.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; D.4.7 [**Organization and Design**]: Distributed Systems

General Terms

Algorithms, Performance, Theory

Keywords

large-scale distributed systems, data dissemination, total order, epidemic algorithm

Middleware '15 December 07-11, 2015, Vancouver, BC, Canada
 ISBN 978-1-4503-3618-5/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2814576.2814804.

1. INTRODUCTION

The ordering of events is one of the most fundamental problems in distributed systems, and over the last decades a large body of research has been dedicated to the design of ordering abstractions with different guarantees and tradeoffs [9, 20]. Most of these abstractions focus on providing strong deterministic guarantees that enable distributed applications to solve various problems, such as synchronization, agreement or state machine replication. Unfortunately, such strong guarantees are expensive to obtain and the algorithms that implement them do not scale well and perform poorly under less than ideal network conditions. This results in a mismatch between what is expected by distributed applications and the achievable properties in the large-scale systems encountered in the real world. For instance, large-scale systems are prone to failures and partitions, and the CAP theorem [3, 14] states that one cannot simultaneously achieve consistency, availability, and partition tolerance. Many systems thus target relaxed forms of these properties, like deterministic algorithms with probabilistic delivery guarantees [12]. The most studied of these properties is eventual consistency [1, 27], which states that the system will reach a consistent state given a sufficiently long period of time during which no changes occur. Unfortunately, most large-scale systems deployed today seldom exhibit the behavior that leads to eventual consistency.

Instead of optimizing what properties can be obtained with deterministic algorithms, we argue that in large-scale settings, it is unrealistic to strive for the guarantees they provide since they result in prohibitively expensive or non-scalable algorithms. This is the same reason that led to the emergence of probabilistic dissemination algorithms based on epidemic principles as an alternative to deterministic dissemination algorithms [2, 4, 10, 11, 13, 16, 18, 19]. These algorithms provide probabilistic guarantees, typically achieving convergence with high probability in finite time. Existing epidemic dissemination protocols are highly scalable and resilient to adversarial network conditions, which are desirable properties in real world deployments. However they mostly focus on the probabilistic reliability of the dissemination, often overlooking stronger properties such as ordering. The ab-

sence of ordering properties, and in particular total order, renders epidemic dissemination algorithms unsuitable for a wide range of applications.

Existing Total Order algorithms, on the other hand, require some sort of agreement property[1]. Furthermore, Total Order and Consensus are equivalent problems [6], and to solve consensus in an asynchronous system, one needs to explicitly maintain a group and have access to a $\diamond\mathcal{S}$ failure detector[2] [5, 6]. Due to faults and churn, the stability period required by the $\diamond\mathcal{S}$ failure detector becomes prohibitive as the network grows, hence the scalability of consensus is intrinsically limited. Recent work strove to overcome these and increase the scalability of consensus by relying on probabilistic protocols [21]. Total Order algorithms thus share the same scalability limitations.

1.1 Contributions

In this paper, we present and analyze EpTO, a new *total order* epidemic dissemination algorithm with *probabilistic reliability* guarantees. EpTO guarantees that processes eventually agree on the set of received events with high probability and deliver these events in *total order* to the application. This is a substantial improvement over existing optimistic total order algorithms based on spontaneous order [23–25], which require reliable deterministic dissemination and suffer from the limitations mentioned above. It also improves upon existing probabilistic algorithms, which either have no or weak ordering guarantees [2, 4, 11, 18, 19], assume static and fully synchronous networks [16] or have scalability issues with concurrent broadcasts [13].

The main insight behind EpTO is a *balls-and-bins* approach to dissemination [19]. A balls-and-bins model abstracts processes as bins and messages (events) as balls, and studies how many balls need to be *thrown* such that each bin receives at least a ball with arbitrarily high probability. Our algorithm disseminates events in a small number of rounds, and the average load on each process is uniform. The number of messages transmitted per process per round is logarithmic in the number of processes, and the total number of messages transmitted in the network before an event is delivered is low and uniform over all processes. EpTO is conceptually simple and fully decentralized, not requiring any form of coordination among processes. It does not require sub-protocols, acknowledgments nor retransmissions. More importantly, it is highly robust and works even when processes rely only on logical time and are desynchronized, as well as when the networks suffers from large delays, churn and message loss.

The EpTO architecture allows us to guarantee total order while making sure that every process delivers every event with a probability arbitrarily close to one. Under challenging but realistic churn, delays, asynchronicity and message loss conditions, the probability of having holes in the sequence of delivered events can be made orders of magnitude smaller than the probability of a catastrophic hardware or network failure. Furthermore, EpTO can always insure that the well-behaving parts of the network work smoothly, circumscribing rare holes in the sequence of delivered events to parts of the network experiencing extreme adversarial conditions. To the best of our knowledge, no existing protocol exhibits these properties. EpTO easily scales to networks with tens of thousands of nodes. As potential applications we target very large scale systems that can leverage stronger ordering properties. For instance, DataFlasks [22] is a very large scale data store maintained exclusively with epidemic algorithms which, due to the absence of ordering, delegates important tasks such as version control to the client. Extending DataFlasks with EpTO would allow stronger ordering properties and hence provide richer abstractions to clients.

[1]With the notable exception of [20] which does not require agreement but suffers from other limitations such as a static membership.

[2]The weakest failure detector to solve consensus in an asynchronous system is usually referred in the literature as $\diamond\mathcal{W}$ which, as shown in [6], is equivalent to $\diamond\mathcal{S}$.

Integrity: For any event e, every process EpTO-delivers e at most once, and only if e was previously EpTO-broadcast.
Validity: If a correct process EpTO-broadcasts an event e, then it eventually EpTO-delivers e.
Total Order: If processes p and q both EpTO-deliver events e and e', then p EpTO-delivers e before e' if and only if q EpTO-delivers e before e'.
Probabilistic Agreement: If a process EpTO-delivers an event e, then with high probability all correct processes eventually EpTO-deliver e.

Table 1: Total Order Specification.

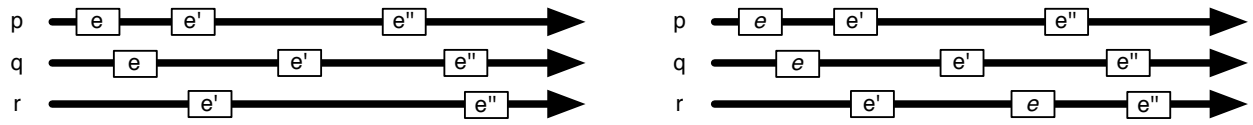
The rest of the paper is organized as follows. In Section 2, we formally state our problem and assumptions. The EpTO algorithm is described in Section 3 and analyzed in a simplistic scenario in Section 4. In Section 5, we show how to extend EpTO for systems with churn, message loss, large network delay, asynchronous processes and logical clocks. The theoretical analysis is supported by several simulations under realistic conditions in Section 6. Finally, we discuss related work in Section 7 and conclude the paper in Section 8.

2. PROBLEM AND ASSUMPTIONS

In this paper, we are interested in data dissemination with *reliability* and *ordering guarantees*. In particular, we want to ensure that any broadcast event is delivered with high probability to all correct processes and that the very same order of events is observed at all recipients. It is assumed that each process has a unique *id*. It is also assumed that processes have access to a peer sampling service (PSS) providing a uniform random sample of other processes [17]. A PSS is inexpensive to maintain, and PSS inaccuracies due to churn can be thought as messages losses and hence accommodated and analyzed appropriately.

The intuition behind EpTO is that events are available quickly at all nodes with high probability. Once events are thought to be available everywhere, we deterministically order them by timestamp, breaking ties with the id of the broadcasters, and deliver them to the application accordingly.

Processes use primitives EpTO-*broadcast* and EpTO-*deliver* to communicate, and the system must satisfy the properties described in Table 1. Besides agreement, which is probabilistic, the other properties closely follow those from traditional total order (or atomic) broadcast algorithms [9]. The integrity property precludes spurious messages by disallowing the delivery of duplicates and messages not previously sent, whereas *liveness* of the protocol is ensured by the validity property requiring correct processes to always deliver the messages they broadcast.



(a) Run A: order, but no agreement.

(b) Run B: agreement, but no order.

Figure 1: Order and agreement properties.

While the total order property is standard, its interplay with the probabilistic agreement guarantees of the protocol is of particular interest. Since the protocol reliability is probabilistic, holes may occur in the sequence of messages delivered (agreed upon) at each process, hopefully with arbitrarily low probability. While these sequences may differ for different processes, the order in which messages are delivered must be the same for all processes. Consider, for instance, the two runs depicted in Figure 1 with three processes and three events. In Figure 1a, the total order property is preserved but agreement is violated because process r did not receive event e. Therefore, this is a valid, although unlikely, run in EpTO. On the other hand, in the run of Figure 1b, agreement is preserved (all processes received all events), but total order is violated because process r delivered the events in a different order from the other two processes. Consequently, this run is not allowed in EpTO.

3. EPTO ALGORITHM DESCRIPTION

The EpTO algorithm is composed of two parts: a *dissemination component* responsible for satisfying the agreement property and an *ordering component* responsible for fulfilling the total order property. The validity and integrity properties are satisfied by the two components in tandem. The dissemination component handles the reception and retransmission of events. Received events are passed to the ordering component which orders and delivers them to the application. The two components and their interactions are depicted in Figure 2.

For the sake of explanation, we initially assume that processes have access to a *global clock*, e.g., as provided by a GPS or an atomic clock and used by Google's Spanner [8] (using the stability oracle of Algorithm 3). We emphasize that we only use this unrealistic assumption to ease the description of the algorithm. In Section 5, we show that this assumption is absolutely and completely unnecessary and relax it to regular logical clocks at little cost (using the stability oracle of Algorithm 4).

3.1 EpTO Dissemination Component

The EpTO dissemination component is depicted in Algorithm 1 for an arbitrary process p. It proceeds in rounds by periodically executing the task in lines 20 to 28. The algorithm assumes the existence of a peer sampling service (PSS) responsible for keeping p's *view* (line 2) up-to-date with a random stream of at least K deemed correct processes, allowing a fanout of size K. TTL (time to live) is a constant holding the number of rounds for which each event needs to be relayed during its dissemination. The *nextBall* set collects the events to be sent in the next round by process p.

The dissemination component consists of three procedures executed atomically: the event broadcast primitive, the event receive callback and the periodic relaying task. When p broadcasts an event (lines 6–10), the event is time-stamped with p's current clock, its ttl is set to zero, and it is added to the *nextBall* to be relayed in the next round. Upon reception of a ball (lines 11–19), events with $ttl < TTL$ are added to *nextBall* for further relaying. When a received event is already in *nextBall*, we keep the one with the largest ttl to avoid excessive retransmissions. Finally, the process clock is updated. The periodic relaying task is executed every δ time units (lines 20–28), in other words rounds last δ time units. Process p first updates the ttl of each event in *nextBall* and then sends it to K processes randomly chosen from its *view*. It then calls the procedure *orderEvents* of the ordering component (Algorithm 2) and resets the *nextBall*.

Algorithm 1: Dissemination component (process p)

```
1  initially
2      view ← ... // system parameter: set of uniformly random correct peers
3      K ← ...                                  // system parameter: fanout
4      TTL ← ...  // system parameter: nb times events need to be relayed
5      nextBall ← ∅     // set of events to be relayed in the next round
6  procedure EpTO-broadcast(event)
7      event.ts ← getClock()
8      event.ttl ← 0
9      event.sourceId ← p.id
10     nextBall ← nextBall ∪ (event.id, event)
11 upon receive BALL(ball)
12     foreach event ∈ ball do
13         if event.ttl < TTL then
14             if event.id ∈ nextBall then
15                 if nextBall[event.id].ttl < event.ttl then
16                     nextBall[event.id].ttl ← event.ttl
                       // update TTL
17             else
18                 nextBall ← nextBall ∪ (event.id, event)
19         updateClock(event.ts)        // only needed with logical time
20 task every δ time units
21     foreach event ∈ nextBall do
22         event.ttl ← event.ttl + 1
23     if nextBall ≠ ∅ then
24         peers ← Random (view,K)
25         foreach q ∈ peers do
26             send BALL(nextBall) to q
27     orderEvents(nextBall)
28     nextBall ← ∅
```

3.2 EpTO Ordering Component

The EpTO ordering component is depicted in Algorithm 2. Procedure *orderEvents* is called every round (line 27 of Algorithm 1) and its goal is to deliver events to the application (Algorithm 2, line 30). To do so, each process p maintains a *received* map of $(id, event)$ pairs with all known but not

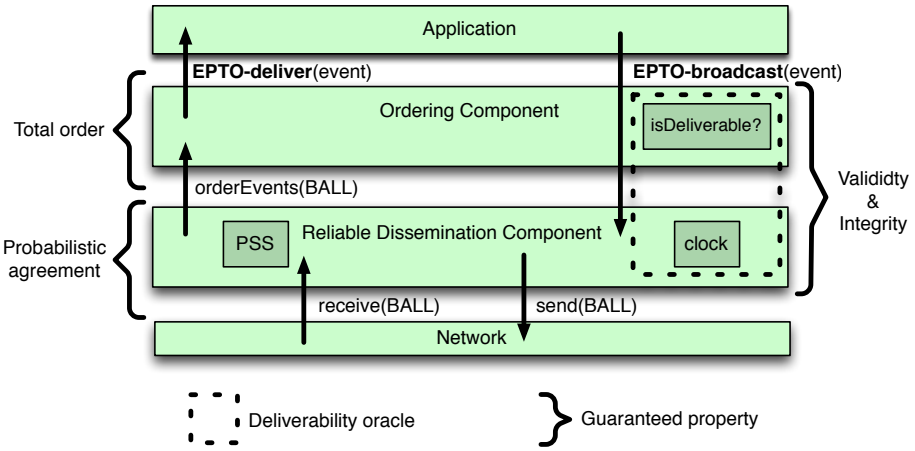


Figure 2: EpTO architecture.

yet delivered events and a *delivered* set with all the events already delivered to the application.

The main task of this procedure is to move events from the *received* set to the *delivered* set, preserving the total order of the events. This is done in several steps as follows. We start by incrementing the *ttl* of all events previously received (lines 6–7) to indicate the start of a new round. Then, in lines 8 to 14, all the events received in *ball* are processed. An event already delivered or whose timestamp is smaller than the timestamp of the last event delivered ($lastDeliveredTs$) is discarded (line 9). Delivering such an event in the former case would violate integrity due to the delivery of a duplicate, and in the latter case would violate total order. Otherwise, the event is added to *received* or, if already there, its *ttl* value is set to the largest of both occurrences. Note that the event's *ttl* is no longer used for dissemination but only for deliverability detection purposes. The next step (lines 15–26) is to build the set of events to be delivered in the current round ($deliverableEvents$): an event e becomes deliverable if it is deemed so by the *isDeliverable* oracle shown in Algorithm 3 and if its timestamp is smaller than any non deliverable event in the *received* set. Lines 15 to 21 collect the deliverable events in the $deliverableEvents$ set and calculate the minimum timestamp ($minQueuedTs$) of all the events that cannot yet be delivered. Next, lines 22 to 26 purge from $deliverableEvents$ all the events whose timestamp is greater than $minQueuedTs$, as they cannot yet be delivered without violating total order. The remaining events are ready to be delivered and thus are removed from the *received* set. Finally, in lines 27 to 30, the events in $deliverableEvents$ are delivered to the application in timestamp order.

4. EPTO ALGORITHM ANALYSIS

In this section, we prove that EpTO satisfies the Total Order specification described in Table 1: integrity, validity, total order and probabilistic agreement. To simplify the analysis, we first assume that processes have access to a global clock and that rounds are synchronous. We also assume that there is no churn, no message loss, and that the network latency is smaller than the round duration. We mention again that all these assumptions are unnecessary and we remove them in Section 5.

Algorithm 2: Ordering component (process p)

```
1  initially
2      received ← ∅   // map of received but not delivered events
3      delivered ← ∅                          // set of delivered events
4      lastDeliveredTs ← 0  // maximum timestamp of delivered events
5  procedure orderEvents(ball)
       // update TTL of received events
6      foreach event ∈ received do
7          received[event.id].ttl ← received[event.id].ttl + 1
       // update set of received events with events in the ball
8      foreach event ∈ ball do
9          if event.id ∉ delivered ∧ event.ts ≥ lastDeliveredTs
           then
10             if event.id ∈ received then
11                 if received[event.id].ttl < event.ttl then
12                     received[event.id].ttl ← event.ttl
13             else
14                 received ← received + (event.id, event)
       // collect deliverable events and determine smallest
          timestamp of non deliverable events
15     minQueuedTs ← ∞
16     deliverableEvents ← ∅
17     foreach event ∈ received do
18         if isDeliverable(event) then
19             deliverableEvents ← deliverableEvents ∪ event
20         else if minQueuedTs > event.ts then
21             minQueuedTs ← event.ts
22     foreach event ∈ deliverableEvents do
23         if event.ts > minQueuedTs then
               // ignore deliverable events with timestamp
                  greater than all non-deliverable events
24             deliverableEvents ← deliverableEvents \ event
25         else
               // event can be delivered, remove from received
                  events
26             received ← received − (event.id, event)
27     foreach event ∈ deliverableEvents sorted by (ts, srcId)
       do
28         delivered ← delivered ∪ event
29         lastDeliveredTs ← event.ts
30         deliver(event)        // deliver event to the application
```

Algorithm 3: Stability oracle — Global clock

```
1 initially
2     globalClock ← ...
3 procedure isDeliverable(m)
      // with TTL given by Lemma 3
4     return m.ttl > TTL
5 procedure getClock()
6     return globalClock.getTime()
7 procedure updateClock(ts)
      // nothing to do
```

Lemma 1. *Synchronous* EpTO *satisfies the Integrity, Validity and Total Order properties.*

Proof.

Integrity property: From the ordering component (Algorithm 2), only events in *received* can be added to *delivered* and delivered to the application, and events can be added to *received* only if they are contained in a ball. From the dissemination component (Algorithm 1), events can only be included in a ball in two ways: either they are broadcasted locally (lines 6–10) or they are received in a ball from an other process (lines 11–19). It follows that only broadcasted events can be delivered.

Since each event has a unique identifier, *received* and *delivered* cannot contain the same event more than once. Only events in *received* can be added to *delivered*, and events can be added to *received* only if they are not in *delivered* (Algorithm 2, lines 8–14). Furthermore, any event added to *delivered* is removed from *received* (Algorithm 2, lines 23–28), thus events are gradually moved from *received* to *delivered*. It follows that events are added to *delivered* and delivered to the application at most once.

Validity property: Consider an event e broadcasted by a process p. By construction, e is always added to p's *received*. An event becomes stable when its *ttl* is greater than TTL. From the ordering component (Algorithm 2), the *ttl* of each event in *received* increases in every round in two possible ways: by one unit by default (Algorithm 2, lines 6–7), or by one unit or more if a received ball has the same event with a larger *ttl* (Algorithm 2, lines 10–12). Since TTL is finite, an event will therefore become stable after at most TTL rounds, after which it will be moved from *received* to *delivered* and delivered to the application. This proves that if a process broadcasts an event e, then it eventually delivers it.

Total Order property: The broadcasted events can be sorted by timestamp and then by broadcast process identifier. Consider two processes p and q, both having events e and e' in *delivered*. Without loss of generality, assume that the timestamp of event e is smaller than the timestamp of e', and that p delivered e' before e whereas q delivered e before e'. This means that process p added e to *delivered* after adding e', despite the timestamp of e' being greater than the timestamp of e. This is a contradiction, since an event cannot be put in *received* if its timestamp is smaller than the timestamp of the last delivered event (Algorithm 2, line 9), and only events in *received* can be moved to *delivered*. □

The remaining property is Probabilistic Agreement. If the stability oracle was perfectly accurate (Algorithm 3 and Algorithm 4), the sequence of events delivered at each process would be exactly the same. However, as we use a probabilistic dissemination algorithm, the set of events delivered (agreed upon) by any two processes might differ. We show in the next two subsections that the probability of processes disagreeing on the set of delivered events can be made arbitrarily close to zero.

4.1 Gossiping with Balls and Bins

The insight behind EpTO is the adaptation of a balls-and-bins epidemic algorithm [19]. Processes are abstracted as bins, and to broadcast a rumor within a system with n processes, the following gossip protocol is used. The process starting a rumor sends balls to K other processes chosen uniformly at random, disseminating the rumor. For each round that follows, the processes which received one or more balls in the previous round send balls to K other processes chosen uniformly at random, further disseminating the rumor. The gossip protocol terminates after m rounds.

Theorem 2 (from [19]). *If the gossip protocol uses* $K = \lceil \frac{2e \ln n}{\ln \ln n} \rceil$ *balls per process and runs for* $m = (c+1)\log_2 n$ *rounds, where* $c > 1$ *is a constant, then at the end of the protocol each process has learned the rumor* with high probability. *More precisely, each bin contains at least one ball with probability* $1 - O\left(n^{-(c+1)}\right)$.

Intuitively, the idea behind the proof of this gossip protocol is that during the first $\log_2 n$ rounds, the number of balls disseminated doubles at each round until at least n balls are transmitted per round. The last $c \log_2 n$ rounds thus create at least $cn \log_2 n$ balls, which is sufficient to conclude that each process has received at least one ball with high probability. The gossip epidemic protocol has several desirable properties. First, the fanout K grows slowly enough to be practically useful. Second, the load is uniformly distributed across all processes. Third, the expected number of messages received by a process during a round is smaller than $O\left(\frac{\log_2 n}{\ln \ln n}\right)$.

4.2 EpTO Probabilistic Agreement Analysis

With no churn, access to global time, no lost messages, synchronous rounds and network latency smaller than the round duration, Theorem 2 can be applied in a straight forward manner.

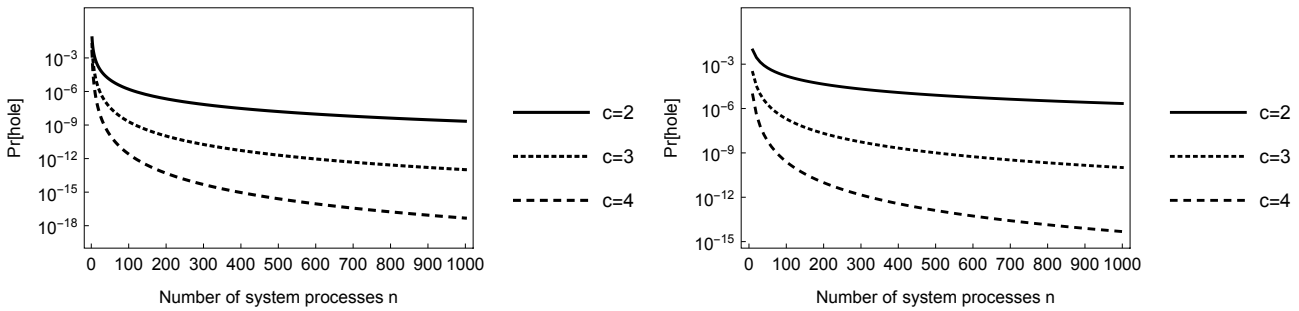
Lemma 3. *If* $K \geq \lceil \frac{2e \ln n}{\ln \ln n} \rceil$ *and* $TTL \geq \lceil (c+1)\log_2 n \rceil$ *where* $c > 1$*, then synchronous* EpTO *satisfies the Probabilistic Agreement property.*

Proof. **Probabilistic Agreement property:** Disseminating events with EpTO corresponds to *aging* the events until they can be delivered without holes with high probability. To do so, processes age events they have received but not yet delivered by incrementing their *ttl* (Algorithm 2, lines 6-7). When an event *ttl* reaches TTL, a process locally knows that this event has been in the system long enough to reach all other processes with high probability and can be delivered to the application.

With these parameters for K and TTL, Theorem 2 guarantees that every event broadcasted in the system will be transmitted to processes at random at least $cn \log_2 n$ times, thus every process receives and delivers every event with high probability.

□

To illustrate that the number of holes can be made arbitrarily close to 0, Figure 3a shows upper bounds for the



(a) Probability that a fixed process p has a hole for event e.

(b) Probability that event e has a hole for at least one process.

Figure 3: Probabilistic agreement upper bounds.

probability that a fixed process p has a hole for event e, and Figure 3b shows the probability that event e has a hole for at least one process. Both figures assume that an event is disseminated at random exactly $cn \log_2 n$ times for $c \in \{1, 2, 3\}$. In practice, the probability of hole is much lower than the bounds from the figures for two reasons. First, the K processes to which each ball is sent are different. More importantly, the bounds provided by Theorem 2 are quite loose and as a result the number of balls in the system for each event is much larger than $cn \log n$.

An advantage of our approach over prior work is that each process groups all the received events per round in the same ball. This limits traffic network when more that one event is broadcasted in a span of TTL rounds.

5. EXTENSIONS

So far, we described how EpTO was able to guarantee total order while decreasing the probability of event holes arbitrarily close to zero under optimistic assumptions: access to global time, synchronized rounds, small network latency and no churn or lost messages. In this section, we shed light on the robustness of EpTO and show that it works when all these constraints are lifted. For all these extensions, the Integrity, Validity, and Total Order properties are not affected, and the proofs of the corresponding results with global time and synchronous rounds can be applied without modification. However, in all cases, the Probabilistic Agreement property needs to be revisited and analyzed more carefully.

5.1 Logical Time

Algorithm 4: Stability oracle — Logical clocks

```
1  initially
2      logicalClock ← 0
3  procedure isDeliverable(m)
       // with TTL given by Lemma 4
4      return m.ttl > TTL
5  procedure getClock()
6      logicalClock ← logicalClock + 1
7      return logicalClock
8  procedure updateClock(ts)
9      if ts > logicalClock then
10         logicalClock ← ts
```

We now relax the assumption of a global clock and show how EpTO can use logical time. We still assume that the round duration is the same value δ for each process, although this assumption is also unnecessary and lifted in Subsection 5.2. We use a scalar logical clock implemented in a standard way: the local clock is incremented whenever an event is broadcasted and received with procedures getClock() and updateClock(ts) of Algorithm 4 (instead of Algorithm 3). By disambiguating concurrent events using the process identifiers, we are still able to totally order all events. However, the delivery of concurrent events with logical clocks might leave unnecessary holes. Consider the example depicted in Figure 4 with processes p and q. It is further assumed for this example that $TTL = 2$, that the initial logical clock of the processes is set to one, and that $p.id$ precedes $q.id$. It should be noted that rounds are labeled just for presentation purposes since EpTO does not require round synchronization nor labeling. Process q broadcasts e with timestamp 1 ($e, ts = 1$) at round zero. Process p receives event e in round two but just before the reception, it broadcasts event ($e', ts = 1$). Because p broadcasts e' before receiving e, the timestamp associated with e' still does not take into account the timestamp of e, and thus both events have the timestamp set to one. Simultaneously, e is deemed stable at q because $TTL = 2$ rounds have elapsed since its broadcast. If our only criterion was event stability, q would correctly deliver e. However, by doing so q would no longer be able to deliver e' as it would violate total order. This is because $p.id$ precedes $q.id$, and thus e' precedes e, which could well be the order in which p will deliver both events. This results in an unnecessary hole in the sequence of delivered events at q. However, if q waits for TTL more rounds it will be able to receive e' with high probability. Thus, it is necessary to double the number of rounds when using logical time.

Lemma 4. *If* $K \geq \lceil \frac{2e \ln n}{\ln \ln n} \rceil$ *and* $TTL \geq 2\lceil (c+1) \log_2 n \rceil$ *where* $c > 1$*, then* EpTO *with logical time satisfies the Probabilistic Agreement property.*

Proof. It is clear that doubling the number of rounds does not affect non-concurrent events, which are delivered to all processes with high probability.

Let $TTL = 2\lceil (c+1) \log_2 n \rceil$, which is twice the bound provided by Theorem 2. Let e and e' be concurrent processes with the same logical timestamp, and without loss of generality assume that e was created before e' in real time and that e' has precedence over e. From Theorem 2, it follows that the number of rounds between the broadcasts of e and e' is at most $\frac{TTL}{2}$ with high probability (otherwise they could

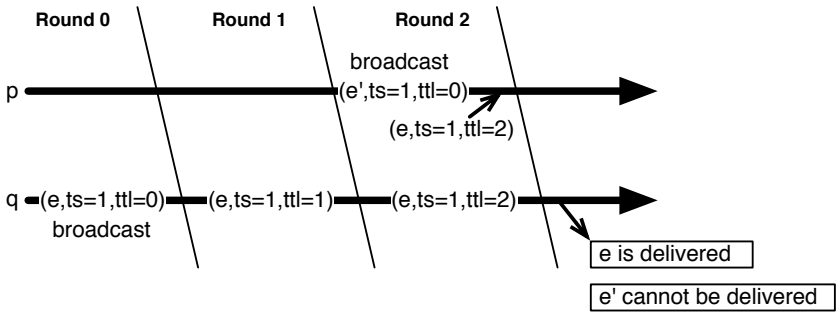


Figure 4: Concurrency hole with logical time.

not be concurrent). Still from Theorem 2, $\frac{TTL}{2}$ additional rounds are sufficient for every process to learn both e and e' with high probability, after which they can be delivered in the right order. □

We note that the worst case described in Figure 4 and requiring to double the number of rounds only applies for networks with minimal activity, since processes update their logical clocks every time they receive a ball. In all our simulations, the network activity was sufficient to update the logical clocks frequently and keep the processes tightly synchronized. We discuss this further is Section 8.

5.2 Process Drift

It is possible that the round duration is not exactly δ time units for every process, for instance due to contention at processes or the imprecision of the local clocks. Consider two degenerate cases. In the first case, a single process p is significantly slower than the other processes in the system. For high event publication rates, events created by p will take a long time to be injected in the system and thus newer events will already have been delivered by other processes, precluding the delivery of p's events by other processes. In the second case, suppose that p is significantly faster than the other processes in the system. An event e generated by p will be broadcasted and delivered by all other processes with high probability after TTL rounds. However, p will detect its own events as stable too early and thus will deliver them and preclude the delivery of concurrent events broadcasted by other processes. Although formalizing drift patterns is beyond the scope of this article, we mention that if some processes have very slow rounds, then their latency will increase (this is discussed in the next subsection). We also tested large random drifts numerically, and EPTO performed very well (consult Section 6). For now, we state and prove a simple result when the speeds of all processes remain bounded.

Lemma 5. *Let assume that the round duration δ of each process is always bounded by $\delta_{\min} \leq \delta \leq \delta_{\max}$, that $K \geq \lceil \frac{2e \ln n}{\ln \ln n} \rceil$, and that $c > 1$. If $TTL \geq \lceil (c+1) \log_2 n \rceil \cdot \frac{\delta_{\max}}{\delta_{\min}}$, then synchronous* EpTO *with global time satisfies the Probabilistic Agreement property. If $TTL \geq 2\lceil (c+1) \log_2 n \rceil \cdot \frac{\delta_{\max}}{\delta_{\min}}$, then* EpTO *with logical time satisfies the Probabilistic Agreement property.*

Proof. We first prove the result for synchronous EPTO with global time. The dissemination of an event in the network can be represented as a tree, where the root is the process which broadcasted the event, nodes at depth d for $d \geq 1$ are the processes which are d balls away from the root (a node can appear at several depths), and the leaves are the processes that delivered the event. Since the round duration of each process is constant and bounded by $\delta_{\min}$ and $\delta_{\max}$, if follows that all the leaves of the tree are at depth at least $\lceil (c+1) \log_2 n \rceil$. This is equivalent to a dissemination protocol with at least $\lceil (c+1) \log_2 n \rceil$ rounds, and from Theorem 2 we can conclude that every process learns every event with high probability. The result for EPTO with logical time follows from Lemma 4. □

5.3 Network Latency

The analysis done so far does not consider the network latency. However, this must be taken into account to avoid that some processes locally stabilize and deliver events before having received concurrent events from other processes. Intuitively, we need to approximate the aging of events with the dissemination periods. This approximation is controlled by the parameter δ which specifies the time between asynchronous rounds of EpTO. Setting δ correctly thus requires a good estimate of the end-to-end communication delay. Setting rounds too short will add an additional number of useless rounds and is a waste of network resources, whereas if the rounds are too long the delivery delays will be uselessly large.

Lemma 6. *Let N be the set of network processes, and let $N_1 \subseteq N$ be a subset of the network such that the latency within N_1 is always bounded by L_{max}. If $\delta > L_{\max}$, then EPTO satisfies the Probability Agreement property within N_1 with the following parameters:*

1. *Synchronous EPTO with global time: $K \geq \lceil \frac{2e \ln n}{\ln \ln n} \rceil$, $c > 1$ and $TTL \geq \lceil (c+1) \log_2 n \rceil + 1$;*
2. EpTO *with logical time: $K \geq \lceil \frac{2e \ln n}{\ln \ln n} \rceil$, $c > 1$ and $TTL \geq 2\lceil (c+1) \log_2 n \rceil + 1$.*

Proof. We consider only the processes in N_1. Since the latency within N_1 is smaller than the round duration, it follows that messages will always reach their destination at most one round after they were transmitted. The proofs of Lemmas 3 and 4 can therefore be applied directly with one additional round. □

If the entire network latency is small and bounded, then we can guarantee that every process will deliver every event in total order with high probability. However, in most large-scale systems, assuming a small and bounded end-to-end communication delay is unrealistic, and guaranteeing that

all processes deliver all events with high probability might incur prohibitively large delays. In practice, deterministic dissemination protocols typically use failure detectors that ping processes and discard (consider as failed) all those whose latency is above a certain threshold [15]. These protocols are very sensitive to latency changes within the network. EPTO does much better: by setting the round duration based on the latency of the well-behaving nodes (or on the latency under good network conditions), we can guarantee that the well-behaving part of the network will satisfy the Probabilistic Agreement property. The processes with large latency can remain in the network, however they might fail to deliver some events from other processes and to transmit some of their events to other processes for delivery. To the best of our knowledge no other dissemination protocol has this property. Still, in practice, and as we show in Section 6, even processes with latencies much larger than δ are able to deliver all events in total order.

5.4 Churn and Message Loss

To illustrate the robustness of our algorithm against churn and message loss, we assume for simplicity purposes that at each round of the protocol, there are α processes leaving the network and α new processes joining the network, thus the number of nodes after each round remains constant. Furthermore, we assume that the network suffers from a message loss rate of ϵ.

Since we want events to be delivered by all processes with high probability, we must send enough balls in the network so that the total number of received balls per event is at least $cn \log_2 n$. However, failed processes and lost messages mean that some of the balls are eliminated before they can be transmitted to other processes. This can be countered by increasing the fanout, as shown next.

LEMMA 7. *Let α and ϵ define the churn and message loss rate of the network, and let $K = \left\lceil \frac{2e \ln n}{\ln \ln n} \cdot \frac{n}{n-\alpha} \cdot \frac{1}{1-\epsilon} \right\rceil$ If $TTL \geq \lceil (c+1) \log_2 n \rceil$, then synchronous* EPTO *with global time satisfies the Probabilistic Agreement property. If $TTL \geq 2\lceil (c+1) \log_2 n \rceil$, then* EPTO *with logical time satisfies the Probabilistic Agreement property.*

PROOF. On average, after churn and lost messages, $K = \left\lceil \frac{2e \ln n}{\ln \ln n} \right\rceil$ balls transmitted by each process will be received. The lemma follows from Theorem 2 and Lemmas 3 and 4. □

The increased fanout is sufficient to guarantee that all the processes that are present in the network when an event is broadcasted will deliver it with high probability as long as they are still in the network at delivery time. A process entering the network should deliver all the events created after it joined with high probability, although no such guarantee can be made for events already created but not yet delivered. Another advantage of our technique is that the fanout is still logarithmic in the number of processes for any churn rate. EPTO is the first algorithm of practical interest that guarantees total ordering and good performance in the presence of significant churn and message loss for large-scale applications.

If the amount of churn varies at each round, an upper bound $\alpha_{\max}$ can be used at little cost. The same thing can be said if the number of network processes varies: since both the number of rounds and the fanout are logarithmic, using a reasonable upper bound $n_{\max}$ has very little effect on the performance of EPTO. Finally, we mention that if the churn rate is very high, then very few processes remain in the network long enough, and any approach with a logarithmic fanout will fail. However, deterministic algorithms will certainly not fare better and such extreme networks are too unstable to be of any practical interest.

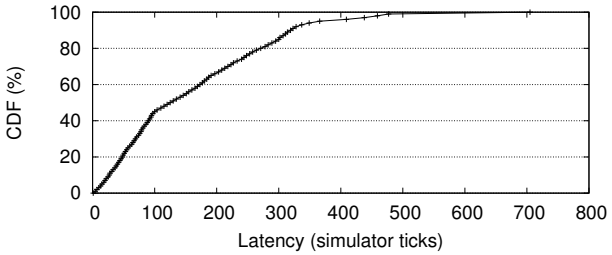


Figure 5: Sample latency distribution from 226 geographically dispersed PlanetLab nodes.

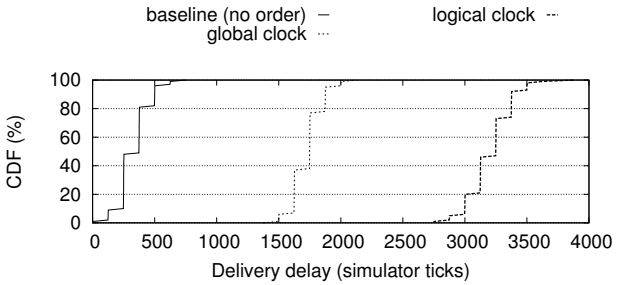


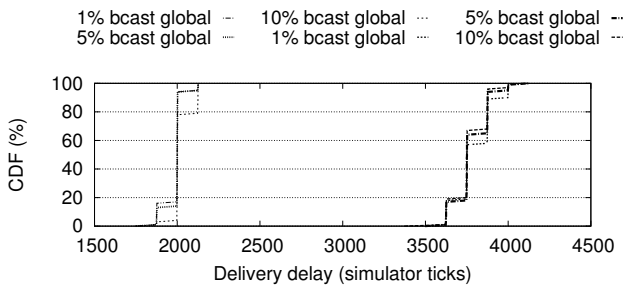
Figure 6: Baseline delivery delay for reliable (non-ordered) delivery. 100 processes; 5% prob. broadcast.

6. EVALUATION

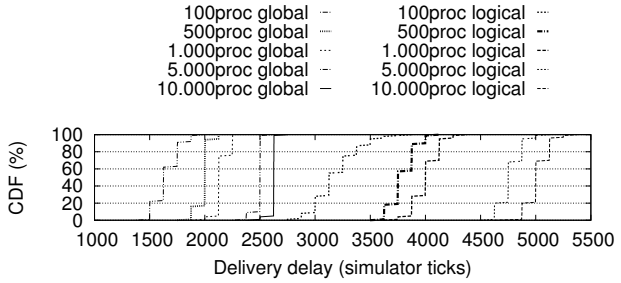
In this section, we evaluate the performance of EPTO. To this end, we developed a realistic discrete simulator that models network asynchrony, process drift, churn and message loss. The simulator uses a priority queue and a monotonically increasing integer to represent the passage of time, i.e., a tick. Processes execute at time $now() + \delta \pm \Delta$[3], balls sent are delivered at processes at time $now() + networkLatency$ and processes may be added/removed from the system at a rate $churnRate$.

We focus our experimental analysis on the delivery delay under different conditions. The delivery delay corresponds to the time elapsed between an event creation and its reception. The objectives of the simulations are to study how the delivery delay evolves as the system grows, both in the number of processes and of concurrent events, as well as how it evolves in adverse conditions. In all the experiments, unless otherwise stated, we use $\delta = 125$ simulator ticks for the process round period with a uniformly random drift of 1%. The end-to-end latency distribution used is drawn from a sample of geographically dispersed PlanetLab nodes and is depicted in Figure 5. The mean latency is ≈ 157, the standard deviation is ≈ 119, and the 5^{th} , 50^{th} and 95^{th} percentiles are 15, 125 and 366 simulator ticks, respectively. As expected, most of the processes are connected with rea-

[3]The process drift.



(a) Increasing the brodcast rate from 1% to 10% (500 processes). Note that the some lines overlap.



(b) Increasing the system size from 100 to 10.000 processes (5% broadcast rate).

Figure 7: Delivery delay with increasing system size and event publication rate.

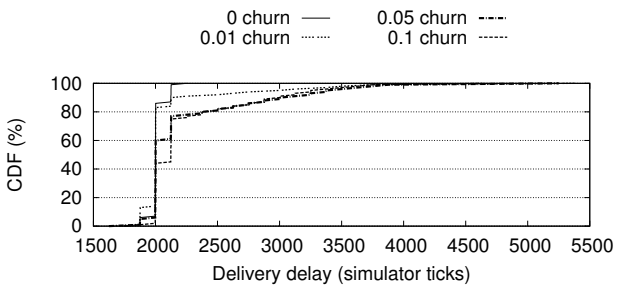


Figure 8: Delivery delay under churn for 500 processes, global clock and 5% broadcast rate.

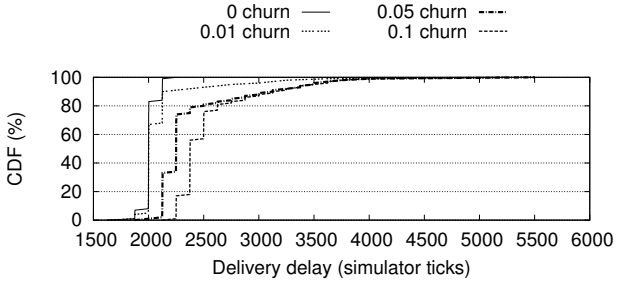


Figure 9: Delivery delay under churn for 500 processes, global clock and 5% broadcast rate using Cyclon [28] as PSS.

sonably low latency while some processes have a very large latency, up to six times the round duration in the worst case. **In all the experiments that follow, we have not observed a single hole in the sequence of delivered events.**

We start by observing the baseline delivery delay which corresponds to a pure balls-and-bins dissemination (i.e., Algorithm 1) without order guarantees essentially showing the time required for an event to infect all processes. Figure 6 shows the delivery delay of events in simulator ticks from broadcast to delivery for 100 processes and a probability of broadcast of 5%. We expect other epidemic dissemination protocols in the literature [2, 4, 10, 11, 16, 19] to yield similar values. When using EpTO with a global clock and the TTL given by the theoretical analysis (TTL=15), the cost of obtaining a totally ordered delivery of events is about three to five times that of reliable delivery. In practice however, we observed that it is possible to reduce the TTL and still have all processes receive all events in the same order. For instance, with a TTL as small as 5, EpTO was still able to deliver all events in total order to all processes, which results in a substantial improvement of the delivery delay. This hints that the theoretical analysis is conservative and the TTL can be relaxed to much lower values. This is discussed further in Section 8.

Next, we analyze how the broadcast rate and the number of processes impact the delivery delay by respectively increasing the broadcast rate from 1% to 10% (Figure 7a), and increasing the number of processes from 100 to 10.000 (Figure 7b). As expected, the broadcast rate has little impact on delivery delay when using either global or logical clocks (Figure 7a). EpTO scales also very well with the system size, as the delivery delay increases logarithmically with the number of processes (Figure 7b). For instance, growing the system by two orders of magnitude (from 100 to 10000 processes) less than doubles the delivery delay for global and logical clocks. This comes from the fact that the number of rounds increases logarithmically with the number of processes.

We also study the behavior of EpTO under churn by observing the evolution of the delivery delay. To this end, we subject the system to a given churn rate by removing $churnRate$ percent nodes uniformly at random and adding $churnRate$ percent nodes every δ simulator ticks. We then analyze the ordering of events for processes that remained in the system long enough. Results are presented in Figure 8. As we can observe, the impact of churn on the delivery delay is small for most processes, and we suspect that the delivery tail is due to processes whose latency is higher than the round duration. Note however that the churn magnitude used in the experiments is significantly larger than what is observed in real systems [26] so the real impact of churn in EpTO will be much smaller. Furthermore, even at this magnitude, we observed no hole in the sequence of delivered events.

In practical systems, churn degrades performance for two additional reasons. First, since maintaining perfect view of the system is unattainable at very large scales, the view made available to EpTO by the PSS will inevitably contain references to failed processes that have not yet been purged. This implies that if such failed processes are selected as targets for the balls, there will be less balls in the system. Conversely, new processes can take a long time before appearing in the view of other processes. To assess the impact of these, we run an additional simulation where the view used by EpTO is maintained by an implementation of Cyclon [28]. Results are shown in Figure 9. As expected, there is a performance degradation due to the above factors. This impact could be minimized by further increasing the size of the fanout, but

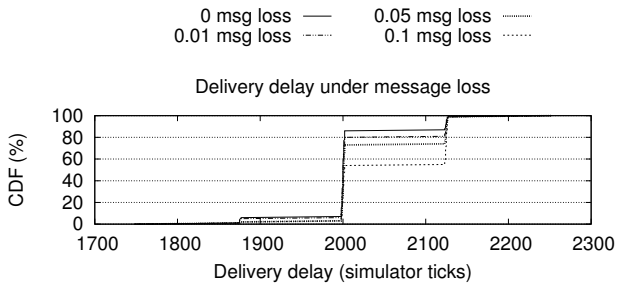


Figure 10: Delivery delay with message loss for 500 processes, global clock and 5% broadcast rate.

more importantly by adjusting the PSS properties to favour freshness as discussed in [17]. However such study is out of the scope of our current work.

Finally, we study the impact of message loss on the delivery delay. This is depicted in Figure 10. The results show that the impact on the delivery delay is limited even at a high loss rate of 10%. This confirms the robustness of EpTO in yet another hostile environment.

7. RELATED WORK

There is a vast amount of work on deterministic total order algorithms, much of which has been analyzed and categorized in [9]. Optimistic replication was surveyed in [24]. Optimistic total order algorithms rely on the real time required for an event to reach all nodes by exploiting the spontaneous network order [23, 25]. The goal is to decrease the delivery latency of deterministic protocols and allow applications to process events optimistically. These protocols still require a deterministic delivery of the final order and thus are subject to the same scalability constraints of traditional deterministic protocols. Besides, they are sensitive to network fluctuations and thus prone to mistakes in the optimistic delivery. Unconscious eventual consistency was studied in [1], where messages can be delivered out of order, in which case rollbacks are performed. The algorithm tries to create coalitions of processes with the objective of decreasing the delivery of out of order messages and thus the number of rollbacks.

A deterministic algorithm with probabilistic delivery guarantees was studied in [12]. The system periodically monitors the delays and message losses, and based on these observations, adjusts the timing parameters and the number of redundant transmissions to obtain acceptable delivery and latency probabilities. The authors mention that probabilistic order properties could be provided by adding a layer on top of their system but do not provide more detail. Furthermore, the system is not scalable without probabilistic dissemination.

The first probabilistic total order algorithm, *Pbcast*, was presented in [16]. It uses an epidemic dissemination protocol, and like EpTO waits for messages to become stable before delivering them. However, unlike EpTO, it is based on a fully synchronous model, the network is static, and the dissemination protocol has weaker guarantees than a balls and bins dissemination. Pbcast was extended in [2]. The improved algorithm can run asynchronously and is composed of two sub-protocols: an unreliable best-effort broadcast, followed by a two-phase "anti-entropy" protocol based on epidemic dissemination that detects and retransmits lost messages. However, the extended algorithm no longer considers ordering: messages are delivered in FIFO order, and it is unclear how to achieve total order in an asynchronous setting.

The PABCast protocol proposed in [13] proceeds in asynchronous rounds during which processes can either broadcast an event or vote for other processes' events. Processes communicate through gossip and exchange the set of events and the list of processes that voted for these events. A round terminates when processes collect $n - f$ votes (n being the system size and f the number of faulty processes) and deterministically deliver all events. PABCast provides probabilistic safety and liveness properties, whereas EpTO provides deterministic safety (integrity and total order are always preserved) and probabilistic liveness (validity is preserved and agreement is achieved w.h.p). The basic version of PABCast only allows for processes to either broadcast a single event or place a vote for an event of another process. This can be overcome with several extensions, but as the authors point out, the number of concurrent broadcasts makes the protocol more prone to out of order deliveries. A broadcast algorithm with causal order was presented in [18]. Like EpTO, it is based on epidemic dissemination, however no implementation was provided. The algorithm requires solicitations, retransmissions, and does not group multiple processes into balls, which probably results in prohibitive network traffic.

8. CONCLUSION AND FUTURE WORK

The ordering of events is one of the most fundamental and studied problems in distributed systems, and until recently, the main research focus was on the construction of primitives with strong guarantees. However, the impressive growth of large-scale distributed systems exposed the practical weaknesses of these approaches: poor scalability, unacceptable degraded behavior under churn, and increased latency. These practical weaknesses led to the development of alternative formulations with weaker yet quantifiable guarantees such as eventual consistency and epidemic dissemination protocols themselves.

In this article, we presented EpTO, an epidemic total order algorithm for large-scale distributed systems. EpTO provides deterministic integrity and total order, and thus safety is deterministic.[4] Validity is also deterministic and agreement is ensured with high probability, thus liveness is ensured with high probability. The probabilistic nature of agreement allows an arbitrarily small probability of hole in the sequence of messages delivered at each process.

EpTO offers many advantages over prior work. It is conceptually simple, does not require any centralized coordination and thus can scale to a large number of processes and events. Furthermore, as agreement is ensured with high probability, the guarantees offered by EpTO are similar to those of deterministic algorithms. For instance, it is practically feasible to configure EPTO such that the probability of hole is orders of magnitude smaller than the probability of catastrophic network failure. Finally, EpTO remains robust even under hostile network conditions. We can extend and improve the work presented in this article is various ways; the rest of this section describes these avenues for future work.

[4]We consider the regular definitions of safety and liveness proposed by [9] instead of the *pure* variants proposed in [7].

8.1 Theoretical Bounds

We believe EpTO can be improved and extended in several ways. For instance, the bounds from [19] for the fanout and number of rounds required to guarantee a sufficient number of balls are very loose, and as a result our bounds for the Probabilistic Agreement property are also very loose. This resulted, as we have shown in Section 6, in way too many balls in the system and thus a waste of network resources. We are currently working on tighter bounds, which will be presented in the extended version of this work.

Moreover, for logical time, we need to double the number of rounds in the worst case, but in practice if the network activity is more than barely minimal, there are enough transmitted messages to update the logical clocks quickly. Even though it is not shown in the article, we have confirmed this through simulations as well. We plan to study the conditions under which this worst-case constraint can be relaxed without sacrificing the Probabilistic Agreement guarantees, further improving the performance and resource usage of EpTO.

8.2 Tagged Delivery

In EpTO, processes do not necessarily know when a hole has occurred in their sequence of delivered events. This being said, processes are allowed to drop events whenever their delivery would result in an order violation, and in this specific case they know that dropped events are out of order. It is thus possible, instead of dropping them, to tag these events as "out-of-order" and to deliver them to the application. This could be of particular interest for perturbed processes, which are otherwise difficult to integrate to the well-behaving part of the network. This is a significant improvement over existing work using failure detectors that simply discards such perturbed processes.

8.3 Corrective Delivery

One can consider the delivery of corrective deliveries to fix mistakes as done in optimistic protocols. Note, however, that the absence of a final order in EpTO makes these corrective deliveries substantially different from the ones in optimistic protocols: in the latter, a corrective delivery as given by the final order is definitive, and thus enables the application to proceed accordingly. On the other hand, in EpTO the location of potential holes is unknown and as such it is not possible to issue a corrective delivery and inform the application that it is final. This is actually close to the notion of unconscious eventual consistency [1], where processes might receive corrective deliveries but are not aware (i.e., they are unconscious) if the delivery order they possess is definitive. Studying the suitability of EpTO to such a programing model is an interesting research avenue, although at this early stage, and considering the excellent performance of EpTO, the additional complexity layer required to implement corrective delivery does not appear to be a worthy tradeoff.

8.4 Delivery Tradeoffs

In complement to the unconscious programing model discussed above, one can also consider directly sharing the probabilistic nature of agreement to the application layer. More precisely, from the balls-and-bins model which underlies the dissemination guarantees, it is possible to go a step further and expose the notion of stability to the application by associating each known but not yet delivered event with the probability of being stable (and deliverable). For some types of applications, having weaker but quantifiable guarantees might be acceptable. For instance, knowing that a majority of processes have delivered a message may be sufficient. Therefore, the application could peek into the list of received events and decide, for each event, if the associated probability of stability and deliverability is acceptable and process it accordingly. We plan to formalize and develop this model in future work, potentially allowing a wide range of tradeoffs between latency and ordering probability.

8.5 Real System Implementation

Finally, while we are confident in the accuracy of the simulated results, a real implementation of the system would be quite valuable both in practical terms and to further assess EpTO's behavior under real conditions. As a matter of fact, implementing such protocols in a real system is far from trivial as we have witnessed first-hand [21]. Leveraging this experience, we plan to provide a real implementation of EpTO and evaluate it on a real setting in future work.

9. REFERENCES

[1] R. Baldoni, R. Guerraoui, R. R. Levy, V. Quéma, and S. T. Piergiovanni. Unconscious eventual consistency with gossips. In A. K. Datta and M. Gradinariu, editors, *Stabilization, Safety, and Security of Distributed Systems, 8th International Symposium, SSS 2006, Dallas, TX, USA, November 17-19, 2006, Proceedings*, volume 4280 of *Lecture Notes in Computer Science*, pages 65–81. Springer, 2006.

[2] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, 1999.

[3] E. A. Brewer. Towards robust distributed systems. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, New York, USA, July 2000. ACM Press.

[4] N. A. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues. Emergent structure in unstructured epidemic multicast. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 481–490, Washington, DC, USA, 2007. IEEE Computer Society.

[5] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[7] B. Charron-Bost, S. Toueg, and A. Basu. Revisiting safety and liveness in the context of failures. In *Concurrency Theory*, pages 552–565. Springer-Verlag, Aug. 2000.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013.

[9] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing surveys*, 36(4):372–421, 2004.

[10] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. E. Sturgis, D. C. Swinehart, and D. B. Terry. Epidemic algorithms for replicated database maintenance. In F. B. Schneider, editor, *PODC*, pages 1–12. ACM, 1987.

[11] P. Eugster, R. Guerraoui, S. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, 21(4):341–374, 2003.

[12] P. Ezhilchelvan, D. Clarke, and A. Di Ferdinando. Near certain multicast delivery guarantees amidst perturbations in computer clusters. Technical Report CS-TR-1267, Newcastle University, July 2011.

[13] P. Felber and F. Pedone. Probabilistic atomic broadcast. In *SRDS*, pages 170–179. IEEE Computer Society, 2002.

[14] S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[15] N. Hayashibara, X. Défago, R. Yared, and T. Katayama. The Φ accrual failure detector. In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil*, pages 66–78. IEEE Computer Society, 2004.

[16] M. Hayden and K. Birman. Probabilistic broadcast. Technical Report TR96-1606, Cornell University, 1996.

[17] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[18] C. Kim, J. Ahn, and C. Hwang. Gossip based causal order broadcast algorithm. In A. Laganà, M. L. Gavrilova, V. Kumar, Y. Mun, C. J. K. Tan, and O. Gervasi, editors, *Computational Science and Its Applications - ICCSA 2004, International Conference, Assisi, Italy, May 14-17, 2004, Proceedings, Part IV*, volume 3046 of *Lecture Notes in Computer Science*, pages 233–242. Springer, 2004.

[19] B. Koldehofe. Simple gossiping with balls and bins. In *Studia Informatica Universalis*, pages 109–118, 2002.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[21] F. Maia, M. Matos, J. Pereira, and R. Oliveira. Worldwide consensus. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 257–269. Springer-Verlag, June 2011.

[22] F. Maia, M. Matos, R. Vilaca, J. Pereira, R. Oliveira, and E. Riviere. Dataflasks: Epidemic store for massive scale systems. In *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*, pages 79–88, Oct 2014.

[23] F. Pedone and A. Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, 2003.

[24] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[25] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *IEEE Symposium on Reliable Distributed Systems*, 2002.

[26] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 189–202, New York, NY, USA, 2006. ACM.

[27] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[28] S. Voulgaris, D. Gavidia, and M. van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.