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

Jocelyn Thode*, Ehsan Farhadi†
Université de Neuchâtel
Neuchâtel, Switzerland
Email: *jocelyn.thode@unifr.ch, †ehsan.farhadi@unine.ch

***Abstract*—One of the fundamental problems of distributed computing, is the ordering of events through all peers. Among different types of ordering, total ordering is of particular interest as it provides a powerful abstraction for building reliable distributed applications. unfortunately, existing algorithms can not provide reliability, scalability, resiliency and total ordering in one package. EpTO is a total order algorithm with probabilistic agreement that scales both in the number of processes and events. EpTO uses a probabilistic agreement to disseminate events through the system with high probability, and also provides integrity, total order and validity. We are going to implement EpTO using the NeEM library and show that EpTO is well-suited for large-scale dynamic distributed systems. Afterwards we will evaluate EpTO by comparing it to the deterministic total order algorithm provided by JGroups.**

I. Introduction

The ordering of events is one of the most fundamental problems in distributed systems and it has been studied over the past few decades. Many researches have been done to create algorithms with different properties and trade-offs such as synchronization, agreement or state machine replication [1, 2]. Unfortunately, due the fact that these properties are strong guarantees, the algorithms that implement them do not scale as expected because of churn and failures in an non-ideal environment. Because of the scalability problem inherent to deterministic approaches, probabilistic protocols have emerged. Nowadays, existing probabilistic protocols are highly scalable and resilient, but they do not provide total ordering.

Maybe add a paper to justify this claim

The problem with existing deterministic total ordering protocols, is that they need some sort of agreement between all peers in the system on the order of messages. This causes a massive amount of network traffic and overhead on the system. Moreover, an agreement feature for an asynchronous system requires to explicitly maintain a group and have access to a failure detector. Due to the faults and churn in large-scale distributed systems, failure detector turns into bottleneck of the system and thus, limits the scalability of the algorithm.

A. Contribution

EpTO [3] guarantees that all processes eventually agree on the set of received events with high probability and deliver these in total order to the application. The intuition behind EpTO is to make events available in all processes quickly

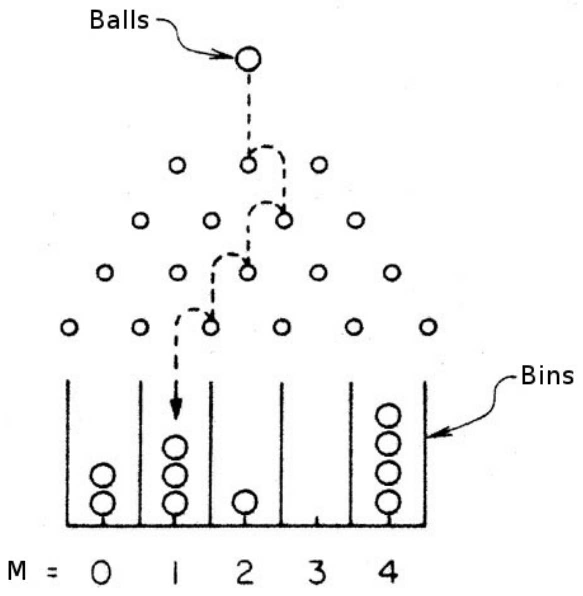

Figure 1. Balls-and-Bins

with high probability. Once events are thought to be available in every peer, each peer deterministically orders them using the timestamp of each event and breaking ties with the id of the broadcaster peer, delivers them to the application accordingly.

The main insight behind EpTO dissemination protocol is a *balls-and-bins* approach [4]. The *Balls-and-bins* problem is a basic probabilistic problem: consider *n* balls and *m* bins where we consequently throw balls into a bin, completely random and independent from other balls.

Using the balls-and-bins approach we model processes as bins and events (or set of events) as balls. Based on this model we will calculate how many balls need to be *thrown* such that each bin contains at least one ball with arbitrarily high probability. Using this approach the number of messages transmitted per process per round is logarithmic in the number of processes, therefore the number of messages sent in the system is low and uniform over all processes. Using these approaches, EpTO becomes highly scalable and resilient, having in mind that it also provides total order.

In this paper, we will implement EpTO and compare it against a known deterministic total order algorithm, namely JGroups [5].

These tasks will be split in two tasks. The first task will assume that all peers are known by all peers. We will implement EpTO using this assumption and test it against JGroups. The second task will assume unknown membership on all peers. We will update EpTO and use a PSS[1]. We will also update the underlying layer NeEM [6] to use UDP instead of TCP and switch to NIO.2[2].

II. Ordering Algorithms

We will explain why Ordering algorithms are necessary, the difference between partial order and Total order and why partial order might not be sufficient in some cases

Distributed systems, like centralized systems need to preserve the temporal order of event produced by concurrent process in the system. When there are separated processes that can only communicate through messages, you can't easily order these messages. Therefore we need ordering algorithms to overcome this problem.

We have two types of ordering algorithms [2]: the partial order algorithms and the total order algorithms.

A. Partial Order Algorithms

Assuming S is partially ordered under $\leq$, then the following statements hold for all a, b and c in S:

- Reflexivity: $a \leq a$ for all $a \in S$.
- Antisymmetry: $a \leq b$ and $b \leq a$ implies $a = b$.
- Transitivity: $a \leq b$ and $b \leq c$ implies $a \leq c$.

B. Total Order Algorithms

A totally ordered set of events is a partially ordered set which satisfies one additional property:

- Totality (trichotomy law): For any $a, b \in S$, either $a \leq b$ or $b \leq a$.

Explain advantages and usefulness of both systems

In other words, total order is an ordering that defines the exact order of every event in the system. On the other hand, partial ordering only defines the order between certain key events that depend on each other. Partial order can be useful since it is less costly to implement but, in some cases where the order of all events is important, for example in order to have an exact image of the state of the system at all time, we have to use total order.

III. EpTO

A. Definitions

A process or peer is defined as an actor in our system running the application that needs total order. Each process will communicate with other processes in the distributed systems and exchange events and order them together.

An event is defined as something that happens at a given time. For example, we could imagine a system where each process publish some sort of data. The moment where this data is published combined with the data is called an event.

[1]Peer Sampling Service

[2]Non-blocking I/O available from Java 7

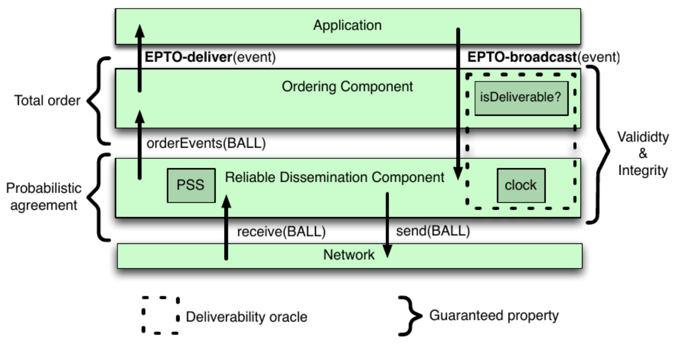

Figure 2.

We need a better quality figure (SVG)

4.

A ball is a set of events bundled together and sent as one package. We use balls in EpTO to reduce network traffic and make it scalable in terms of processes and events.

EpTO is scalable in the sense that

Explain the logarithmic scalability

Since EpTO is using a probabilistic agreement instead of a deterministic agreement, there might be a situation where a peer does not receive an event (with a very low arbitrary probability). In this case there will be a hole in the sequence of delivered events but even in this case, the order of the delivered event will be protected by EpTO's deterministic ordering algorithm and total order property is preserved.

B. Dissemination Component

The Dissemination component is the component that bridges EpTO with the rest of the network. As we can see in Figure 2, it is in charge of receiving balls, open them , pass them to the Ordering component and then forward them to *K* other processes, where K denotes the gossip fan-out.

When an application wants to publish an event, it will broadcast this event to the Dissemination component.

C. Ordering Component

The Ordering component is responsible for ordering events before delivering them to the application. To achieve this, the Ordering component has a *received* hash table of (*id*, *event*) pairs containing all the events which are received by the peer, but not yet delivered to the application and a *delivered* hash table containing all the events which are delivered.

In brief, the Ordering component first increments the timestamp of all the events which have been received in previous rounds to indicate the start of a new round. Then, it processes new events in the received ball by discarding events that have been received already (delivered events or events with timestamp smaller than the last delivered event). This is done to prevent delivering duplicate events. The remaining events in the received ball will be added to the *received* hash table, and wait to be delivered based on the Stability oracle.

D. Stability Oracle

The Stability oracle is the component that outputs timestamps. It will increment its local clock every round as well as synchronize itself using timestamps of newly received events, to make sure the local clock does not drift too much.

This Stability oracle offers an API to the Ordering component letting it know when an event is mature enough to be delivered to the application.

As this component is local and only correct itself when we receive new events, it generates no network traffic. This means it does not impact the scalability of EpTO.

IV. Performance Comparisons

Here we will present the different test we ran, the methodology used and the result we obtained

A. Methodology

Here we explain how we ran our tests and on what type of machines

B. Peer membership known

We will present the results obtained and try to explain the results

C. Peer membership unknown

We will present the results obtained and try to explain the results

V. Conclusion

He we will summarize the results we found and present some future tasks that could be accomplished on EpTO

Acknowledgment

Write acknowledgment

References

[1] X. Défago, A. Schiper and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey", *ACM Computing Surveys (CSUR)*, vol. 36, no. 4, pp. 372–421, 2004.

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[3] M. Matos *et al.*, "Epto: An epidemic total order algorithm for large-scale distributed systems", in *Proceedings of the 16th Annual Middleware Conference*, ACM, 2015, pp. 100–111.

[4] B. Koldehofe, "Simple gossiping with balls and bins", in *Studia Informatica Universalis*, 2002, pp. 109–118.

[5] (). Jgroups - a toolkit for reliable messaging, [Online]. Available: http://jgroups.org/ (visited on 24/04/2016).

[6] (). Neem : Network-friendly epidemic multicast, [Online]. Available: http://neem.sourceforge.net/ (visited on 24/04/2016).