Practical Implementation of EpTO: An Epidemic Total Order Algorithm

Jocelyn Thode
Université de Fribourg
Fribourg, Switzerland
jocelyn.thode@unifr.ch

Ehsan Farhadi
Université de Neuchâtel
Neuchâtel, Switzerland
ehsan.farhadi@unine.ch

***Abstract*—Total ordering has always been of interest in large-scale distributed systems. EpTO is one of the recently introduced total order algorithms for large-scale distributed systems and claims to provide total order and scalability at the same time. In this paper, we verify this claim by implementing EpTO [1] and evaluate its reliability in real-world conditions. We then compare it to a deterministic total order algorithm named JGroups.**

I. Introduction

Creating an algorithm providing scalability, integrity and validity, along with a total ordering for the events through all peers in a distributed system has been one the hot topics in distributed systems research for many years. One of the recently designed algorithms on this topic is EpTO [1]. EpTO is an algorithm that claims to provide integrity, validity and total order in a large-scale distributed system. In addition, EpTO is designed to work without a global clock, removing the need to synchronize clocks precisely on every peer and thus is well-suited for dynamic large-scale distributed systems.

There are many other algorithms for disseminating and ordering events in a distributed system. There are some deterministic algorithms, which guarantee total order, agreement or other strong properties. Unfortunately, these types of algorithms are not scalable enough to be used in a large-scale distributed system [2, 3].

The problem with existing deterministic total ordering protocols is that they need some sort of agreement between all peers in the system. 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 [4, 5]. Due to faults and churn in large-scale distributed systems, the failure detector turns into a bottleneck for the system and thus limits the algorithm's scalability.

As an alternative to deterministic algorithms, there are probabilistic algorithms, focusing on scalability and resiliency against failures using a probabilistic dissemination approach [6, 7, 8, 9, 10, 11, 12, 13]. These algorithms guarantee the dissemination of events in the system with high probability. This way, there is no need for failure detectors and redundant traffic, making these algorithms highly scalable. As these algorithms focus on reliability of dissemination, they often have to ignore other properties such as total ordering.

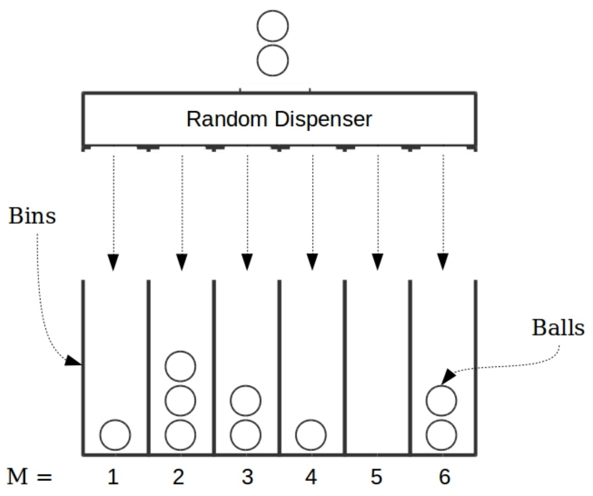

Figure 1. Balls-and-Bins [14].

This is where EpTO comes into light. EpTO, by mixing these two types of algorithms, provides total order along with scalability, validity and integrity. EpTO consists of two distinct parts. The first part is probabilistic dissemination. EpTO guarantees that all peers will receive an event with arbitrarily high probability. The second part of EpTO is deterministic ordering. Once peers have received all events, they deterministically order them using the events timestamp, and in case of a tie use the event's broadcaster id.

To model the first part, EpTO is using a *balls-and-bins* approach [13]. The balls-and-bins problem is a basic probabilistic problem: consider *n* balls and *m* bins where we consequently throw balls into a bin uniformly at random and independently from other balls. In this scenario, one of the natural questions that comes to mind is: what is the minimum number of balls that should be thrown, so that every bin has at least one ball with high probability?

Using the balls-and-bins approach we model processes as bins and events as balls and calculate how many balls need to be *thrown* such that every bin contains at least one ball with arbitrarily high probability. With 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. Thanks to these approaches, EpTO becomes highly scalable and resilient, while still providing total order.

A. Contribution

Until now, the creators of EpTO have only tested this algorithm in a simulated environment. In this work, we implement EpTO in pure Kotlin using a modified version of the NeEM library [15] and show that EpTO is suitable for real-world large-scale distributed systems. The modification to NeEM includes a dumbed-down gossip dissemination as seen in [1]. We then evaluate EpTO by comparing it to the deterministic total order algorithm provided by JGroups [16] in both stable (no-churn) and unstable systems. These comparisons help us verify if EpTO is actually performing as expected in a real-world scenario.

Here we will present our conclusion briefly

In our experiments we assume each peer knows every other peers. We implement EpTO using this assumption and test it against JGroups. The details concerning the methodology are explained in subsection IV-A. In future work, we want to test EpTO when the membership is not entirely known.

II. Ordering Algorithms

Distributed systems, like centralized systems, need to preserve the temporal order of events produced by concurrent processes 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 [3]: 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 $b \leq a$.

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. However, in some cases the order of all events is important, for example when we need to know exactly which operations have been invoked in which order. We then have to use total order, otherwise we can end up in an inconsistent state.

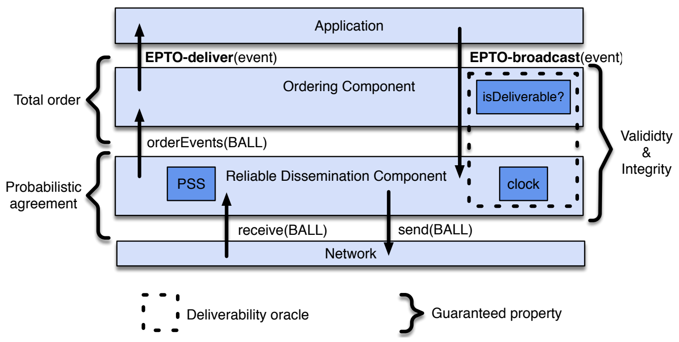

Figure 2. EpTO architecture [1].

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 system, exchange events, and order them together.

An event is defined as data sent at a given time by a peer. For example, we could imagine a system where each process publishes some data to other peers. The moment where we publish data combined with the data is called an event.

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.

We define EpTO scaling well as it was defined in [1]: "The number of messages transmitted per process per round is logarithmic in the number of processes, ...". The number of rounds needed to deliver an event stays low as well.

Since EpTO uses 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 instance there will be a hole in the sequence of delivered events but even in this case, the order of the delivered events will be protected by EpTO's deterministic ordering algorithm, thus the total order property is preserved.

B. Dissemination Component

The Dissemination component bridges EpTO with the rest of the network. As we can see in Figure 2, it receives balls, opens them, passes them to the Ordering component, and stores their events. Then, in the next round, it sends all the events received in the previous round 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 were 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 are 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 corrects itself when we receive new events, it generates no network traffic. This means it does not impact the scalability of EpTO.

IV. Performance Comparisons

In the following subsections, we will present the results obtained for the different tests ran and explain their meanings.

A. *Methodology*

Our tests are run on a cluster provided by the University of Neuchâtel. For setting up our testbed we use OpenNebula [17] to manage our cluster of virtual machines. We were planning on using Docker [18], more specifically Docker-compose and docker-swarm in order to orchestrate and manage our cluster of peers on the OpenNebula but due to some issues with docker-swarm, we decided to deploy our application directly on the OpenNebula virtual machines, and control it using parallel ssh sessions.

We deployed our application on a small network of 4 virtual machines and we distribute EpTO and JGroups peers equally over the network.

To measure the throughput, we transmit a certain number of events by certain number of peers in a defined period of time, and then we measure the number of packets transmitted over the network.

To measure the latency of both protocols, we measure the average time difference on a peer between the broadcast of an event and the time it delivers this same event. This way we can have an estimation of the approximate delay, and compare JGroups and EpTO. We execute EpTO and JGroups with 60, 100, 120 and 160 peers each peer broadcasting 12 events for 12 seconds (one event per second).

B. *Peer membership known*

As we measured and compared the latency of EpTO and JGroups we realize that JGroups is performing better than EpTO with smaller number of peers. As we increased the number of peers, our results showed that JGroups is not as scalable as EpTO, but considering our limited resources we couldn't fully observe the difference between JGroup and EpTO on this matter. Although we should bear in mind that in this experiment EpTO has a relatively large delay due to ordering of the events without a central node (as in JGroups). But despite of this delay, we expected that EpTO performs much better and our analysis is that due the insufficient resource on the testbed. Epto could not show its advantages.

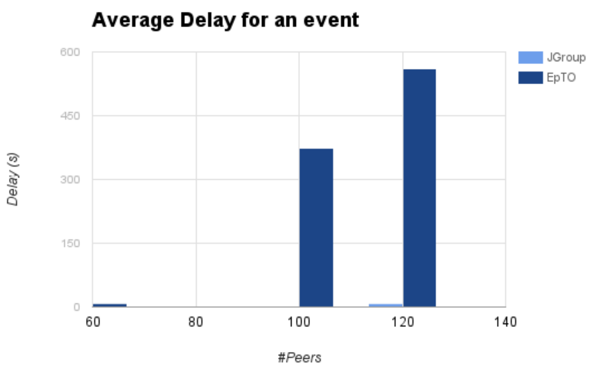

Figure 3. Approximate delay of delivering each event.

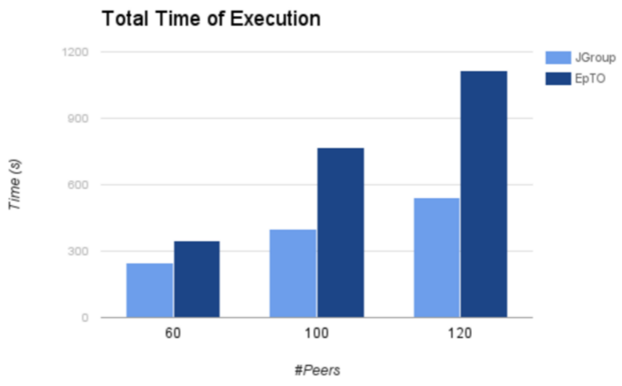

Figure 4. total run time of EpTO peers in order to deliver all events.

We perceive this issue again when we measured the total time of execution of EpTO for delivering a certain number of events.

For the throughput of both protocols we used a network sniffing tool, TCPdump. TCPdump is a tool for observing the network traffic. Using this tool we count number of packets (TCP packet for EpTO and UDP packet for JGroups) transmitting over the network for 10 minutes. And again, number of packets sent by EpTO peers are much higher than JGroups. This issue can be explain due the differences of the using protocols, TCP and UDP. TCP uses much more packets and data over the network for a connection to secure and guarantee the delivery of data. that's why in a perfect and lossless network, JGroups has better performance from perspective of network throughput.

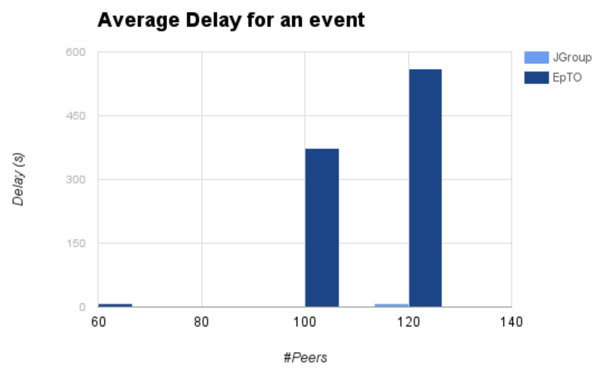

Figure 5. number of events transmitted packets over the network in 10 minutes.

V. Future Work

One of the students working on this project will do his master thesis on this project.

Future work will include testing EpTO while having unknown peer membership. To account for that, we will need to modify NeEM to support the use of a Peer Sampling Service. We plan on implementing the CYCLON Peer Sampling Service [19]. For now, we plan on using a tracker as in torrents to give each peer its initial view. This tracker is already implemented and works as a python web server that gets all alive peers and randomly select K peers and sends this result to the EpTO peer.

Our first implementation of EpTO focuses on correctness rather than fastness. Later work should optimize EpTO in regards to concurrency and network latency. To address the second point, we want to modify NeEM again to use UDP instead of TCP, to reduce latency and overhead in the network.

VI. Conclusion

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

Acknowledgment

This research was under supervision and help of two of the creators of EpTO, Dr. Hugues Mercier and Dr. Miguel Matos. We would like to thank them both for the help, insight and expertise they provided during this project.

We would also like to thank Valerio Schiavoni for providing access to the cluster of University of Neuchâtel as well as Sebastien Vaucher for the help with setting up Docker, Docker-Compose and Docker Swarm.

References

[1] M. Matos, H. Mercier, P. Felber, R. Oliveira and J. Pereira, "EpTO: an epidemic total order algorithm for large-scale distributed systems", in *Proceedings of the 16th Annual Middleware Conference*, ACM, 2015, pp. 100–111.

[2] 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.

[3] 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.

[4] T. D. Chandra, V. Hadzilacos and S. Toueg, "The weakest failure detector for solving consensus", *Journal of the ACM (JACM)*, vol. 43, no. 4, pp. 685–722, 1996.

[5] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems", *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[6] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu and Y. Minsky, "Bimodal multicast", *ACM Transactions on Computer Systems (TOCS)*, vol. 17, no. 2, pp. 41–88, 1999.

[7] N. Carvalho, J. Pereira, R. Oliveira and L. Rodrigues, "Emergent structure in unstructured epidemic multicast", in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, IEEE, 2007, pp. 481–490.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart and D. Terry, "Epidemic algorithms for replicated database maintenance", in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, ACM, 1987, pp. 1–12.

[9] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov and A.-M. Kermarrec, "Lightweight probabilistic broadcast", *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 4, pp. 341–374, 2003.

[10] P. Felber and F. Pedone, "Probabilistic atomic broadcast", in *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, IEEE, 2002, pp. 170–179.

[11] M. Hayden and K. Birman, "Probabilistic broadcast", Cornell University, Tech. Rep., 1996.

[12] C. Kim, J. Ahn and C. Hwang, "Gossip based causal order broadcast algorithm", in *Computational Science and Its Applications–ICCSA 2004*, Springer, 2004, pp. 233–242.

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

[14] Balls-and-bins figure, [Online]. Available: https://ned.ipac.caltech.edu/level5/Berg/Berg2.html (visited on 27/04/2016).

[15] NeEM : network-friendly epidemic multicast, [Online]. Available: http://neem.sourceforge.net/ (visited on 24/04/2016).

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

[17] OpenNebula : the simplest cloud management experience, [Online]. Available: http://opennebula.org/ (visited on 06/06/2016).

[18] Docker : an open platform for distributed applications for developers and sysadmins, [Online]. Available: https://www.docker.com/ (visited on 06/06/2016).

[19] S. Voulgaris, D. Gavidia and M. van Steen, "Cyclon: inexpensive membership management for unstructured p2p overlays", *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005, ISSN: 1573-7705. DOI: 10.1007/s10922-005-4441-x. [Online]. Available: http://dx.doi.org/10.1007/s10922-005-4441-x.