EpTO TO DEFINE

X

Master Thesis

Jocelyn Thode

Université de Fribourg

22 December 2016

***Abstract*—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 by comparing it to a deterministic total order algorithm named JGroups. We first compare them in a perfect environment assessing the scalability in terms of events throughput and peer numbers. We then compare them using a synthetic churn trace and a real one using the same metrics.**
Jocelyn ▶*Speak about results*◀

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.

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 broadcaster id of the events.

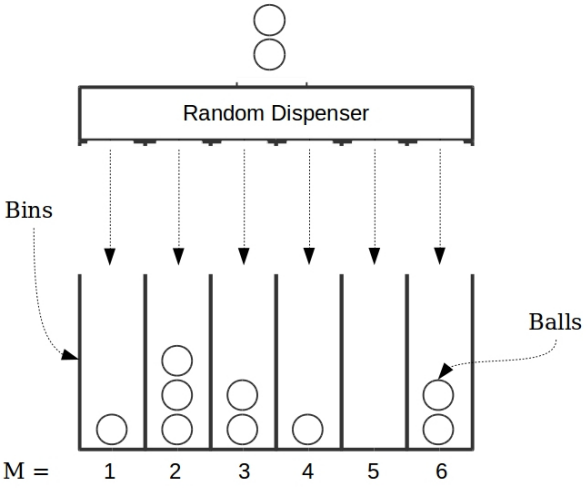

Figure 1. Balls-and-Bins [14].

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.

Until now, the creators of EpTO have only tested this algorithm in a simulated environment. In this work, we implement EpTO in pure Kotlin and introduce a benchmarking solution to show that EpTO is suitable for real-world large-scale distributed systems. We compare EpTO to the deterministic total order algorithm provided by JGroups [15] in both stable and unstable environments. EpTO uses a PSS CYCLON to obtain a random view, coupled with a tracker that keeps track of dead and alive peers to obtain a first initial view. These benchmarks can easily be launched and scaled on a cluster using Docker and Docker Swarm. Furthermore, we can inject synthetic

or real churn[1] in the system.

The next sections are dedicated to present our benchmarking solution and to present the results obtained. Section II presents the different kind of ordering. Section IV presents EpTO and the architecture of our implementation. Section V shows and explains the results obtained. We then argue about future work in section VII. We finally draw conclusions in section VIII.

II. Ordering Algorithms

Jocelyn ▶*I'm not sure we still need this section*◀ 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 cannot 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.

III. 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.

[1]Using the Failure Trace Archive databases

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.

We write the cluster parameters as (p, e), where p designates the total number of peers and e designates the global event throughput. We tested EpTO and JGroups with three different cluster parameters:

- $(100, 100)$: 50 peers with a global event throughput of 50 events per seconds.
- $(100, 100)$: 50 peers with a global event throughput of 100 events per seconds.
- $(100, 100)$: 100 peers with a global event throughput of 50 events per seconds. This parameter was also used for all synthetic churns.

IV. Framework

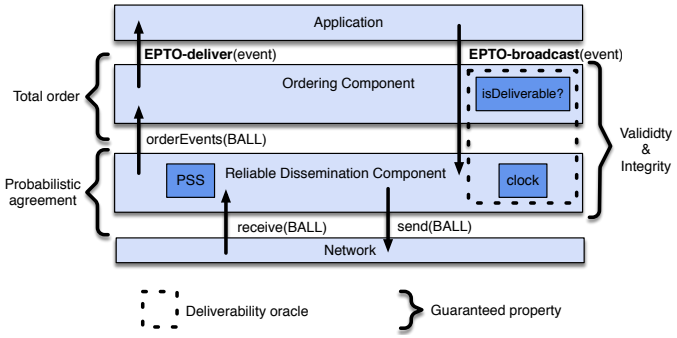

Figure 2. EpTO architecture [1].

EpTOTester is a practical implementation of EpTO designed to assess the claims made in [1]. Although the implementation was written with benchmarking in mind, the code could be adapted to be used in a real application with only minimal changes to the sources.

A. Architecture

Figure 2 illustrates the architecture of a single replica. An application extending our Application class can broadcast and delivers events. Events broadcasted to the Dissemination component are sent over the network every δ period. Every events received from the network are analyzed by the Dissemination component to find out whether

they need to be propagated further or not according to their Time To Live (TTL).They are then sent to the ordering component so that EpTO can determine whether to deliver these events or not and in which order. The order is based on the timestamp of the events given by the logical clock and their broadcaster ID in case of a tie. The network layer communication is done using UDP and a PSS CYCLON to gather a random view of peers. To obtain an initial random view, we contact an independent tracker that keeps track of dead and alive nodes in the system. We want to emphasize that the tracker is not obligatory. In practice it works well, but using DHTs is certainly a possibility.

B. Implementation

We implemented the data sent as randomly generated UUIDs. We implemented our own PSS CYCLON operating on its own port. When implementing JGroups we had to use TCPGOSSIP instead of the traditional MULTICAST option to coordinate peers. This is not a problem as in a real WAN JGroups could not rely on MULTICAST. The framework is open-sourced on Github.

C. Deployment

To deploy EpTO effectively we effectively need two different services. One containing the tracker and one containing all EpTO replicas. This is achieved using Docker and especially the new Docker Swarm introduced in Docker 1.12. This lets us have a unified way of deploying our benchmarks locally or remotely on vastly different clusters with minimal tweaks. Every benchmark is launched through a Python script available on the master mode. This script handles everything, from starting benchmarks, scaling the cluster during a churn and collecting results. All EpTO parameters are customizable by using options provided through this script. Gradle is used to automate the image building in the python scripts. Finally, a script is provided to push the images to a repository accessible to the remote cluster and to push the benchmarking script on the master node. JGroups deployment is much the same. **Jocelyn** ▶*I should create a figure showcasing the complete architecture*◀

D. Fault Injection

Our framework supports the ability to inject synthetic and real traces thanks to the work done in [16], although the synthetic churn was improved to support adding and removing nodes at the same time.

V. Evaluation

In this section we present the results of the benchmarks.

A. Testbed

Jocelyn ▶*Write about benchmarks*◀

B. Parameter

EpTO uses a constant c to set the probability of a hole appearing. We set $c = 4$ in all our benchmarks so as to not experience any holes as JGroups does not produce holes under normal conditions. We use a δ period of 100 ms for tests with no churn or synthethic churn and 250 ms for tests following a real trace.

We used specific settings recommended for a big cluster available in the JGroups library.

C. Bandwidth

Jocelyn ▶EpTO *was run with* $c = 4$ *to prevent any holes and a* δ *period of 100ms. Churn was run for 17minutes. 3min at the end with no churn, to see stabilization. Churn starts 30seconds after protocol*◀ **Jocelyn** ▶*We should define acronyms for the different benchmark variants*◀

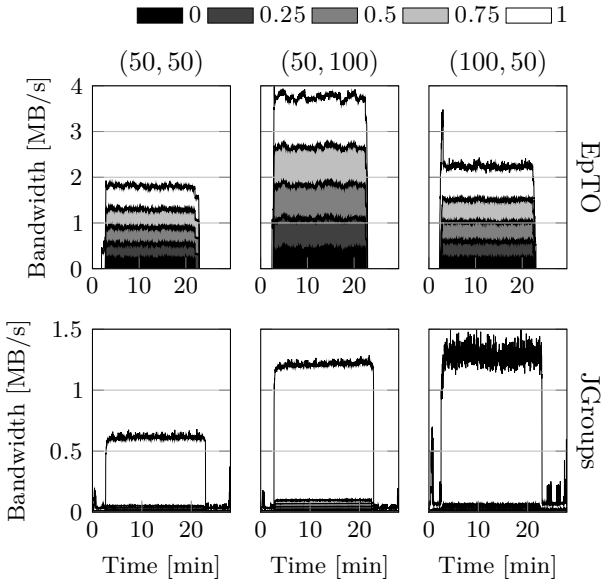

Figure 3. Throughput percentiles of a node during an experiment

In Figure 3 we can see EpTO has a worse baseline compared to JGroups. It uses a median bandwidth of approximately 1 MB/s for $(50, 50)$ whereas JGroups uses a median bandwidth of less than 0.20 MB/s. However, in JGroups most of the work is done solely by the coordinator. We can clearly see this as the 100th percentile is much higher than the rest and uses approximately 0.60 MB/s.

Comparing EpTO and JGroups in terms of bandwidth when we increase the number of events sent per second, we can see the bandwidth doubling in both cases.In lower peers scenario such as the ones presented in Figure 3 JGroups is clearly at an advantage. Since EpTO has a worse baseline we will reach the maximum bandwidth possible much quicker when increasing the event throughput.

Comparing EpTO and JGroups in terms of bandwidth when we increase the number of peers, EpTO is performing better than JGroups. Where JGroups basically has to double the bandwidth usage of the coordinator, EpTO

only increases it by 50% or less. Jocelyn ▶*logarithmic I think as was expected*◀. Thus in a scenario where we have many peers EpTO will be more efficient than JGroups at not reaching the maximum bandwidth.

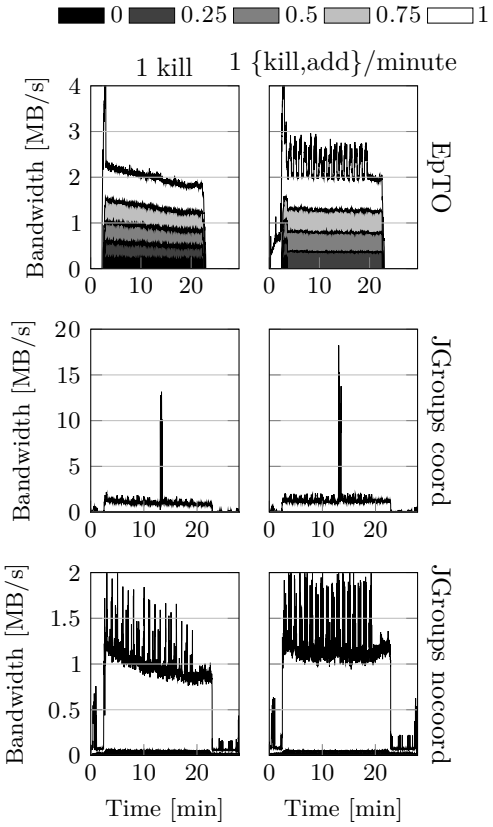

Figure 4. Throughput percentiles of a node during an experiment with churn

In Figure 4 We analyze two different synthetic churns. In the first one we kill one node per minute. In the second one, we still kill one node per minute, but we immediately create a new one. For JGroups we ran the benchmarks once without killing the coordinator and once killing it.

We can see that the churn doesn't affect EpTO at all when there are only nodes leaving. We have small peaks when adding a node to 3 MB/s or less. Probably due to running the PSS initialization method on top of having one more node spreading rumors in the system. This is confirmed at the end of the plot where EpTO goes back to a normal Bandwidth after stabilization.

On the other hand, when killing the coordinator in JGroups we can see a huge spike in bandwidth, going from 1.20 MB/s to more than 15 MB/s. This is due to how JGroups operates when selecting a new coordinator.

Even when not killing the coordinator, JGroups suffers from the churn. We can see that each time the view changes, it generates an almost 100% increase in bandwidth usage. This is due to JGroups having to update the view and propagate it to every peer.

D. Total GigaBytes sent/received

Table I
Total GB sent/received in a stable system

Protocol		Cluster parameters 50P50E	50P50E	100P50E
EpTO	Receive	10.85 ± 0.16	22.31 ± 0.39	26.01 ± 0.28
	Sending	10.85 ± 0.16	22.31 ± 0.39	26.01 ± 0.28
JGroups	Receive	0.81 ± 0.03	1.48 ± 0.01	2.00 ± 0.01
	Sending	0.80 ± 0.03	1.47 ± 0.01	1.95 ± 0.01

In Table I, EpTO has a worse baseline than JGroups. This is expected as EpTO sends $c*n*\log_2 n$ messages per events and JGroups sends at least n messages per event so we should have at least $c * \log_2 n$ more messages sent in EpTO if JGroups is perfect. Here we are well within this ratio.

Table II
Total GB sent/received with a synthetic churn

Protocol		Churn parameters {1kill}/minute	{1kill,1add}/minute
EpTO	Receive	21.00 ± 0.24	26.32 ± 0.32
	Sending	21.21 ± 0.25	26.57 ± 0.32
JGroups-coord	Receive	1.55 ± 0.02	1.85 ± 0.02
	Sending	1.51 ± 0.02	1.79 ± 0.02
JGroups-nocoord	Receive	1.54 ± 0.01	1.83 ± 0.02
	Sending	1.49 ± 0.01	1.78 ± 0.02

In Table II we can see that JGroups total bandwidth usage is smaller when there is churn. One hypothesis for this is that a JGroups replica takes a longer time to start up thus the overall benchmark has a longer time with less than 100 replicas. We also do not see a difference whether we kill the coordinator or not. This can be explained by the fact that before the detection of the faulty coordinator JGroups is forced to a stop for time up to 20 s. The big spike afterwards compensates for this hole.

E. Local Times

Jocelyn ▶*Having a table for each percentile figure might be a good idea to put numbers for key percentiles (min,50th,max) for example*◀

In Figure 5, JGroups delivers all events quicker than EpTO in all scenarios, even when churn is involved as is shown in Figure 6. However, EpTO is not too far behind. The difference between EpTO and JGroups is likely to be even smaller when running them in a real WAN network due to the latency. EpTO in our configuration has a δ period of 100 ms and is thus handicapped against JGroups in a LAN environment, because it only increments the TTL of an event every 100 ms.

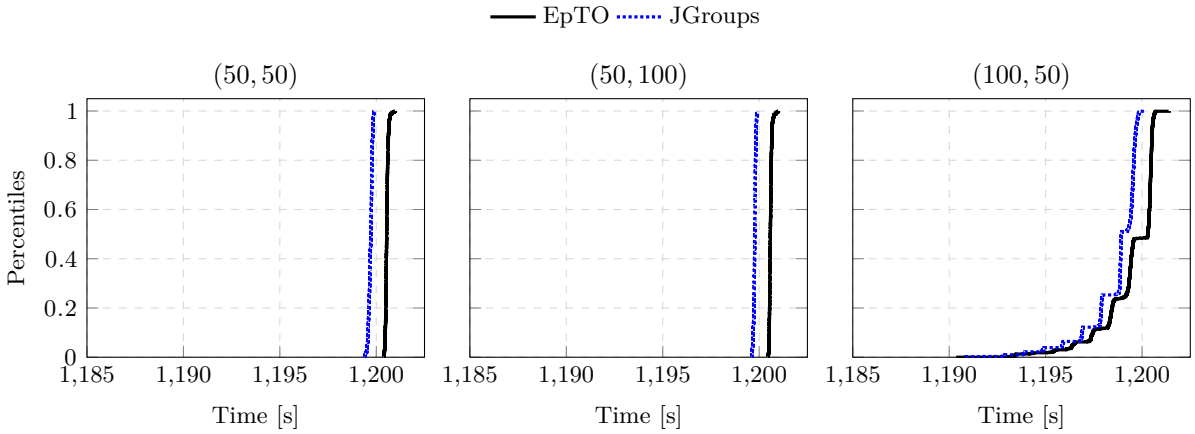

Figure 5. Local dissemination times

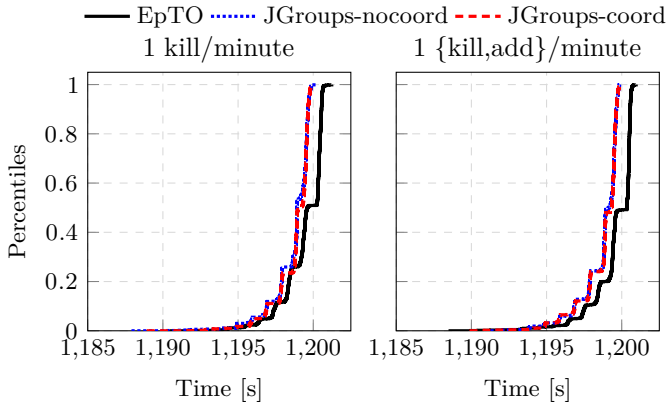

Figure 6. Local dissemination times with churn

F. Global Times

We computed global times as well. They are represented in Figure 7 and Figure 8. These global times are of less interest than their local counterpart as the differences in clocks between hosts can skew this measurement.

Nonetheless, here too we can see that EpTO is consistently slower than JGroups for the same reason as stated in subsection VI-C.

G. Local Dissemination stretch

In Figure 9, We can see the percentiles of the local dissemination stretch. The local dissemination stretch is the time measurement between the sending of an event by a peer and the delivery of this event locally.

JGroups is usually much faster than EpTO in a perfect environment. This is expected as the benchmarks involve a small number of nodes and are performed in a LAN environment with minmimal latency. The median dissemination stretch of JGroups is around 7 ms where as the median dissemiantion stretch of EpTO is around 630 ms for $(100, 50)$. When increasing the number of peers, JGroups starts to have long delivery times for some outliers.

In Figure 10 We can see a completely different picture. When under churn, the 95th percentile of JGroups is at 31 ms compared to 14 ms when there is no churn. The highest percentiles are at more than 10 s. This effect is due to the coordinator dying as we clearly see that it does not happen when we do not kill it.

The median is bigger at around 9 ms, whether we kill the coordinator or not. This shows that there are some degradation in JGroups local dissemination stretch when under churn.

On the contrary, EpTO performs very well under churn. The median degrade a bit at 650 ms with the 99th percentile being at 1030 ms compared to 982 ms when no churn is happening.

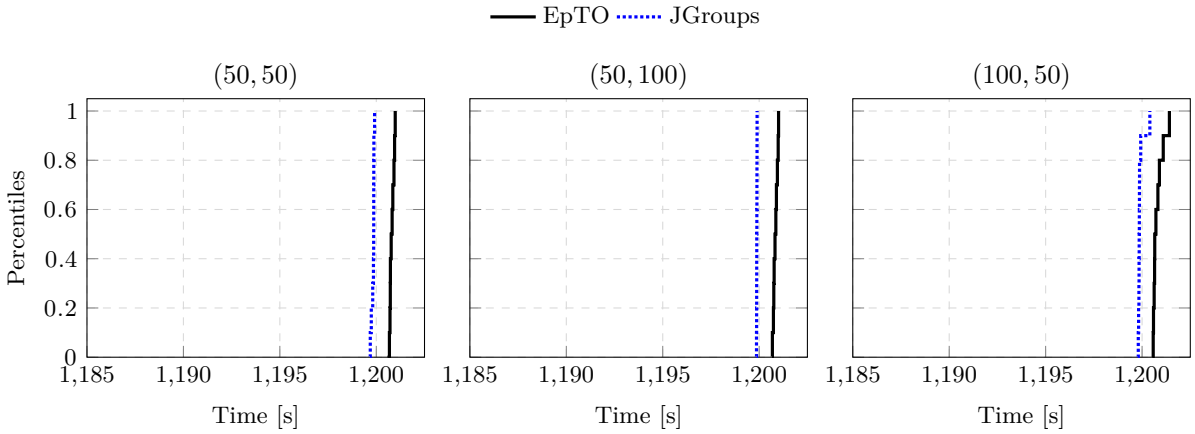

Figure 7. Global dissemination times

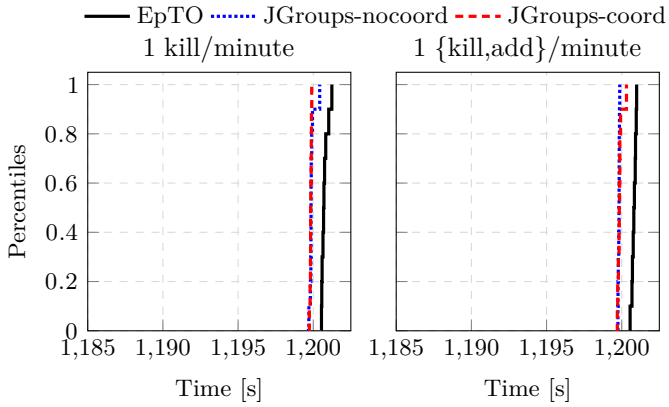

Figure 8. Global dissemination times with churn

H. Events sent

Jocelyn ▶*Representing as a table might be better*◀

Table III
TOTAL EVENTS SENT IN A STABLE ENVIRONMENT

	Cluster parameters		
Protocol	50P50E	50P50E	100P50E
EPTO	59 993.8 ± 3.3	119 898.2 ± 9.7	59 913.0 ± 164.3
JGROUPS	59 961.9 ± 10.9	119 885.7 ± 5.0	60 023.1 ± 287.1

In Table III we can see that both EPTO and JGROUPS deliver the same amount of events. This is expected in a perfect environment.

Table IV
TOTAL EVENTS SENT WITH A SYNTHETIC CHURN

	Cluster parameters	
Protocol	{1kill}/minute	{1kill,1add}/minute
EPTO	53 898.5 ± 133.9	59 798.6 ± 140.1
JGROUPS-coord	53 834.7 ± 175.5	59 507.9 ± 240.9
JGROUPS-nocoord	53 830.5 ± 200.3	59 450.5 ± 175.1

In Table IV When only killing nodes, EPTO and JGROUPS again deliver the same amount of events. When killing and adding nodes, JGroups seems to deliver a smaller amount of nodes, however it does not look significant **Jocelyn** ▶*I didn't run any statistical analysis*◀

VI. REAL TRACE

A. Bandwidth

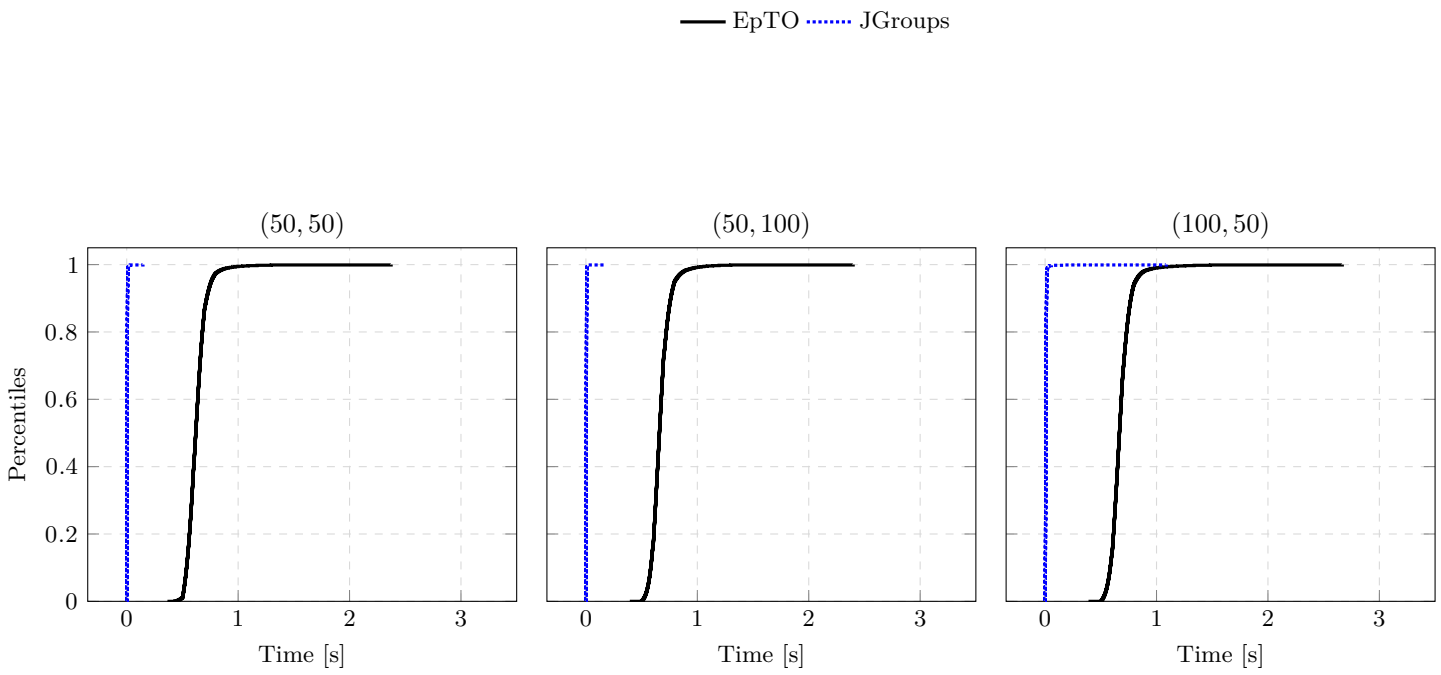

Figure 9. Local dissemination stretch

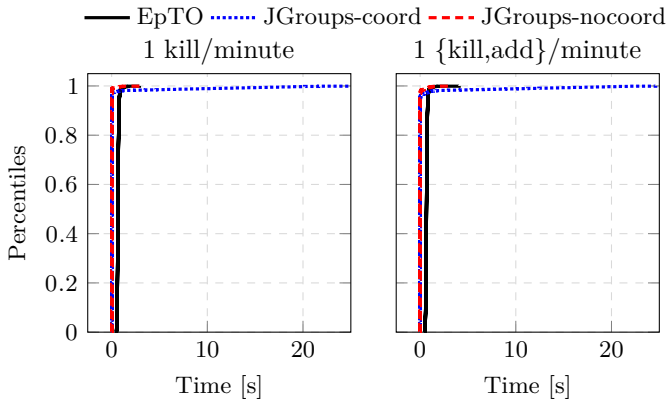

Figure 10. Local dissemination stretch with churn

B. Total Bytes sent/received

C. Local Times

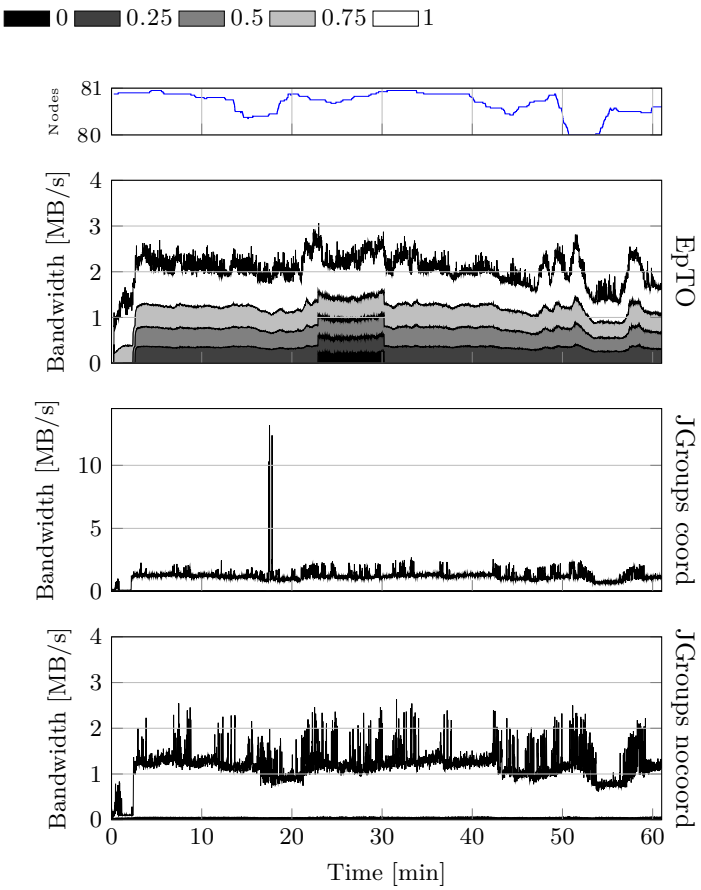

Figure 11. Throughput percentiles of a node during an experiment with churn

Table V
TOTAL GB SENT/RECEIVED

Protocol		Churn parameters: Real Trace
EPTO	Receive	81.41 ± 1.08
	Sending	82.67 ± 1.08
JGROUPS-coord	Receive	5.61 ± 0.08
	Sending	5.45 ± 0.08
JGROUPS-nocoord	Receive	5.63 ± 0.05
	Sending	5.48 ± 0.05

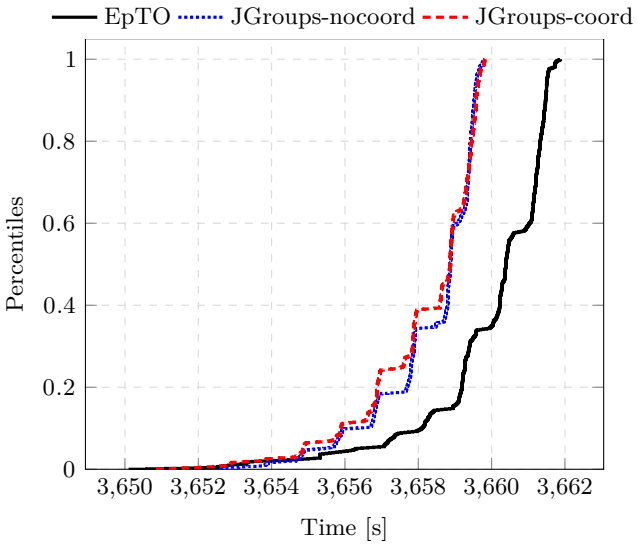

Figure 12. Local dissemination times

D. Global Times

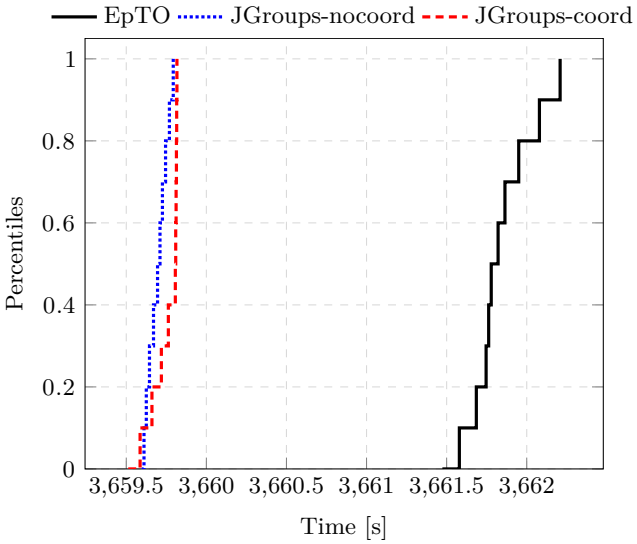

Figure 13. Global dissemination times with churn

E. Local Dissemination stretch

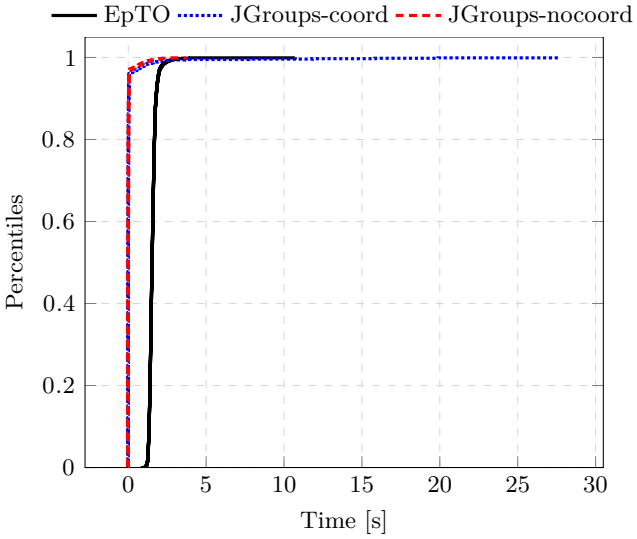

Figure 14. Local dissemination stretch with churn

F. Events sent

Table VI
TOTAL EVENTS SENT DURING A REAL TRACE

Protocol	
EPTO	$165\,844.2 \pm 210.2$
JGROUPS-coord	$166\,183.0 \pm 1368.1$
JGROUPS-nocoord	$166\,585.8 \pm 824.9$

VII. FUTURE WORK

VIII. CONCLUSION

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 04/27/2016).

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

[16] S. Vaucher, H. Mercier, and V. Schiavoni, "Have a seat on the erasurebench: Easy evaluation of erasure coding libraries for distributed storage systems," in *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, 2016, pp. 55–60.