EpTOTester

Implementation of Large-Scale Epidemic Total Order Algorithms

Master Thesis

Jocelyn Thode

Université de Fribourg

Supervisor: Dr. Hugues Mercier

Co-Supervisor: Dr. Miguel Matos

30 December 2016

***Abstract*—EpTO [1] is one of the recently introduced total order algorithms for large-scale distributed systems, and provides total order and scalability at the same time. In this paper, we verify this claim by implementing EpTO 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. Our results show EpTO is performing has expected, although further testing on a bigger cluster is required to evaluate how it performs when JGroups no longer does.** Hugues ►*General comment: if possible, maybe we should rent resources on Amazon to test with larger networks.*◄

I. Introduction

Creating an algorithm providing scalability, integrity and validity, along with a total ordering of the events through all peers in a distributed network has been one the hot topics in distributed computing 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. Furthermore, 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 these kind of systems.

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

The problem with existing deterministic total ordering protocols is that they need some sort of agreement between all peers in the network. 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 structure and thus limits the algorithm's scalability. Hugues ►*The first three paragraphs use the word system a lot. You could rework this a little bit.*◄ Jocelyn ►*I tried replacing some 'system' by synonyms such as 'network' or 'structure'.*◄

As an alternative to deterministic algorithms, there are probabilistic algorithms, focusing on scalability and resiliency against failures using a probabilistic dissemination approach [6]–[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.

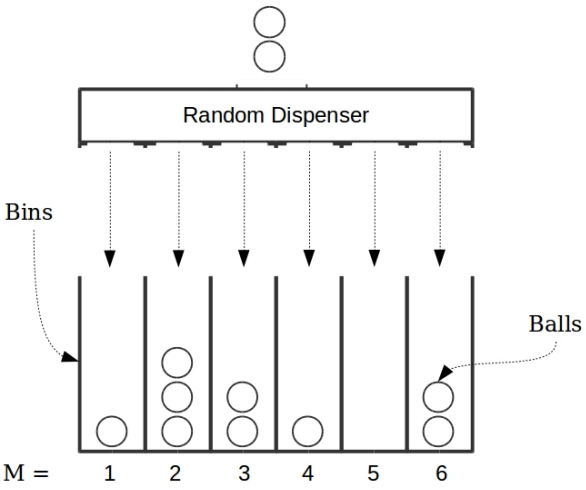

Figure 1. Balls-and-Bins[1]

EpTO claims to solve these seemingly irreconcilable differences. By using a probabilistic dissemination together with deterministic ordering, EpTO provides total order along with scalability, validity and integrity. EpTO consists of two distinct components. EpTO's dissemination component guarantees that all peers will receive an event with arbitrarily high probability. Once peers have received all events, EpTO's ordering component orders them using the events timestamp, and in case of a tie use the broadcaster ID of the events.

To model the first component, 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 is highly resilient with imperfect networks and highly scalable as the networks grow, while always 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[2] and introduce a bench-

[1]Figure inspired from [14]
[2]https://kotlinlang.org/

marking 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 [16] 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[3] in the system.

The next sections present our benchmarking solution and the results obtained. Section II presents the different kinds of ordering. Section III defines the terms used in the paper. Section IV presents EpTO and the architecture of our implementation. Section V shows and explains the results obtained as well as the limitations our project faced. We then present possible future work in Section VI. We finally conclude in Section VII.

II. Ordering Algorithms

Distributed systems and centralized systems alike, 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. **Hugues** ►*It would be enlightening to be more precise here and add a large-scale distributed application for which total order is important.*◄

[3]Using the Failure Trace Archive databases

III. Definitions

A process or peer is defined as an actor in our system running the application that needs total order. Each process communicates with other processes in the distributed system, exchange events, and order them together. **Hugues** ►*I changed it, but you switched from the present to the future tense in the same sentence. You might as well check the rest of the document for other instances, if any.*◄ **Jocelyn** ►*I checked the whole document again I think it is fixed now*◄

An event j is defined as $e_j = (broadcasterID, timestamp, payload)$, where the timestamp is the local time at the process when the event is created and is used with the broadcasterID to order events. The payload represents the data sent.

A ball is an abstraction taken from the balls-and-bins problem. In practice, we bundle sets of events together and send them as one packet to reduce the network traffic overhead. **Hugues** ►*Not clear. A ball is an abstraction. We reduce traffic by bundling events.*◄ **Jocelyn** ►*Is this better?*◄

We define EpTO scaling well as it is 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 is logarithmic as well. **Hugues** ►*This paragraph is confusing. Is it necessary?*◄ **Jocelyn** ►*I remember having questions about what scaling well meant during the workshop that is why I added this definition. If you feel it is not needed we can remove it*◄

Since EpTO uses a probabilistic agreement instead of a deterministic agreement, there is a nonzero probability that a peer does not receive an event. In this instance there will be a hole in the sequence of delivered events, but the order of delivered events is protected by EpTO's deterministic ordering algorithm, thus the total order property is preserved. The probability that a peer does not receive an event is controlled by EpTO and depends on various parameters such as the fan-out, the number of rounds (time to live), the number of nodes, and the network conditions. In particular, this probability can be made smaller than the probability of catastrophic network failure.

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

- $(50, 50)$: 50 peers with a global event throughput of 50 events per second.
- $(50, 100)$: 50 peers with a global event throughput of 100 events per second.
- $(100, 50)$: 100 peers with a global event throughput of 50 events per second. These parameters were also used for all synthetic churns.

IV. EpToTester

EpTOTester is a practical implementation of EpTO designed to assess the claims made in [1]. Although the

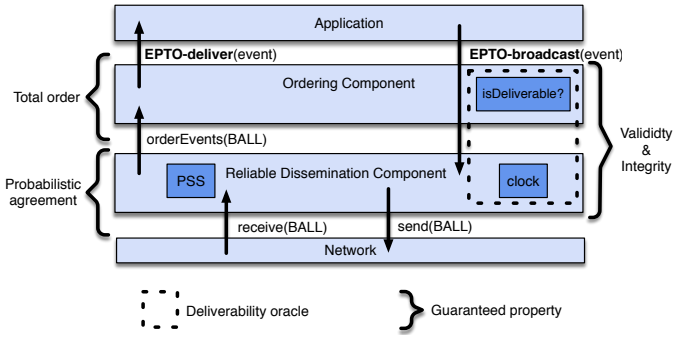

Figure 2. EpTO architecture [1].

implementation is written with benchmarking in mind, the code can be adapted for real applications with only minimal changes to the sources. EpTOTester is open-sourced on Github[4].

A. Architecture

Figure 2 illustrates the architecture of a single replica. An application extending our Application class can EpTO-broadcast and EpTO-deliver events. Events broadcasted to the Dissemination component are sent over the network every δ period, where δ is a unit of time. Every ball received from the network is unwrapped and its events 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 obtain a random view of peers. To obtain an initial random view, we contact an independent tracker implemented as a simple Python web server that keeps track of dead and alive nodes in the system. We want to emphasize that the tracker is not required. In practice it works well, but using a DHT is certainly a possibility.

B. Implementation

We implement the payload as randomly generated UUIDs. We implement our own PSS CYCLON operating on its own port. We compare EpTO to the deterministic total order algorithm provided by JGroups. We use JGroups 3.6.11. When implementing JGroups we use TCPGOSSIP provided by the JGroups library instead of the traditional MULTICAST option to coordinate peers as Docker does not support MULTICAST. This is not a problem as in a real WAN JGroups could not rely on MULTICAST.

C. Deployment

[4]https://github.com/jocelynthode/EpTOTester
[5]This figure is partially inspired from a figure in [17]

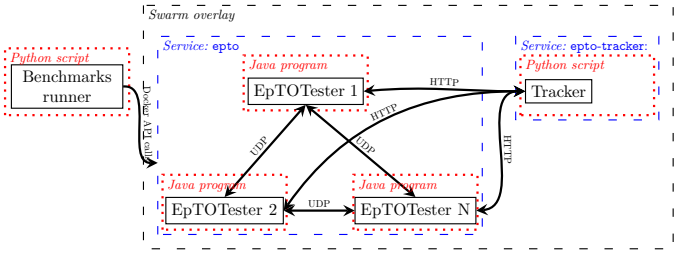

Figure 3. EpTOTester architecture[5]

To deploy EpTO effectively we need two different Docker Swarm services: one containing the tracker and one containing all EpTO replicas. Both of these services use the same network overlay to communicate. 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 modifications. Every benchmark is started through a Python script available on the master node. 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[6] is used to automate the project building. 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. The entire EpTOTester architecture is shown in Figure 3.

D. Fault Injection

Our framework supports the ability to inject synthetic and real traces thanks to the work done in [17]. The synthetic churn provided by it was improved to support adding and removing nodes at the same time.

V. Evaluation

In this section we present our results. In the following benchmarks 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 by JGroups for a big cluster.

Every JGroups test run with churn is run once killing the coordinator and once not killing it. We use the following nomenclature to differentiate both tests:

JGroups-coord: The coordinator is killed once.
JGroups-nocoord: The coordinator is purposely kept alive throughout the experiment.

[6]https://gradle.org/

Testbed. Jocelyn ▶*Most of this paragraph is copied from ErasureBench as we use the same cluster and Docker as well. Is this fine as is?*◀Our cluster consists of 20 machines interconnected by a 1 Gb/s switched network. Each machine has an 8-core Intel Xeon CPU and 8 GB of RAM. We deploy 12 virtual machines on these hosts. Each virtual machine has access 4 VCPUs as well as 7 GB of RAM. We use KVM as our hypervisor. Each VM uses Debian as its O/S. EpTOTester is packaged as docker images. We use Docker 1.12 and its new functionality Docker Swarm to orchestrate our services. Each docker container has a memory restriction of 300 MB of RAM to be able to run them all together.

Experiment parameters. Every benchmark except the one following a real trace are run 10 times, each during 20 min. When there is synthetic churn, the churn starts 30 s after the benchmark and run for 17 min. The benchmarks following a real trace are run 5 times. The trace is speed-up 2x, which means we follow 2 h worth of trace in 1 h. Every benchmark run with churn is run with the $(100, 50)$ parameters. When we have a global event throughput lower than the number of actual peers, we use a uniform random number generator to artificially reduce the global event throughput. For example, using $(100, 50)$ parameters, each peer has a 50% probability of sending an event at each second.

We compare EpTO and JGroups based on their bandwidth, the local and global time to deliver every expected events, their local dissemination stretch and finally the number of events sent on average.

A. Bandwidth

The initial bandwidth peaks observed for EpTO are due to the PSS initialization. When a node starts, it receives an initial view and then quickly runs the PSS active thread four times to jump-start itself. In Figure 4 we observe that 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 4 JGroups is clearly at an advantage. Since EpTO has a worse baseline we will reach the maximum bandwidth available much quicker when increasing the event throughput.

Comparing EpTO and JGroups in terms of bandwidth when we increase the number of peers, EpTO scales better than JGroups. Where JGroups basically has to double the bandwidth usage of the coordinator, EpTO only increases it marginally 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 available bandwidth on a given node.

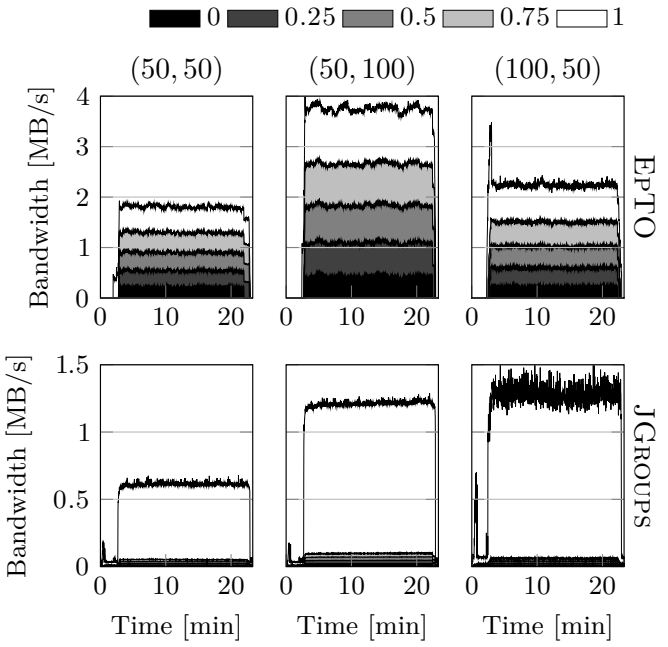

Figure 4. Bandwidth percentiles of nodes during a stable experiment

In Figure 5 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 run the benchmarks once without killing the coordinator and once killing it.

We can see that the churn does not 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.

B. Total GigaBytes sent/received

In all cases, JGroups receives more bytes than it sends. This is due to the fact that we do not measure the bandwidth on the TCPGOSSIP, which is in charge of view changes.

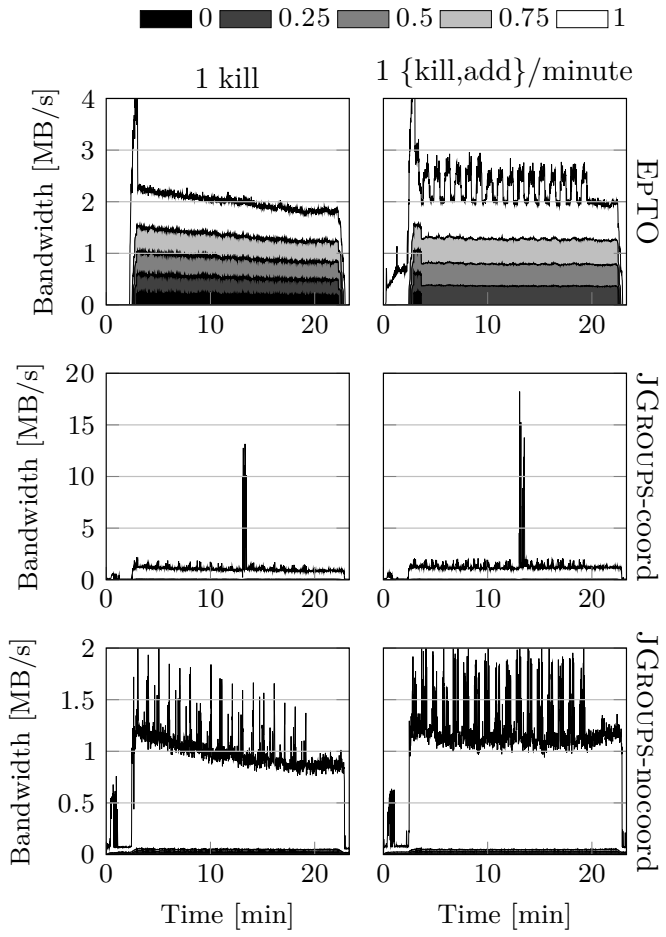

Figure 5. Bandwidth percentiles of nodes during an unstable experiment with synthetic churn

Table I
Total GB sent/received in a stable system

Protocol		Cluster parameters		
		(50, 50)	(50, 100)	(100, 50)
EpTO	Receive	10.84 ± 0.16	22.31 ± 0.39	26.01 ± 0.27
	Sending	10.84 ± 0.16	22.31 ± 0.39	26.01 ± 0.27
JGroups	Receive	0.78 ± 0.03	1.45 ± 0.01	1.88 ± 0.01
	Sending	0.77 ± 0.03	1.44 ± 0.01	1.84 ± 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. In Table II we 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 compared to stopping a replica. Therefore 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

Table II
Total GB sent/received with a synthetic churn

Protocol		Churn parameters	
		1 kill/minute	1{kill,add}/minute
EpTO	Receive	21.00 ± 0.24	26.32 ± 0.32
	Sending	21.21 ± 0.25	26.57 ± 0.32
JGroups-coord	Receive	1.47 ± 0.02	1.75 ± 0.02
	Sending	1.43 ± 0.02	1.70 ± 0.02
JGroups-nocoord	Receive	1.45 ± 0.01	1.73 ± 0.02
	Sending	1.41 ± 0.01	1.68 ± 0.02

that before the detection of a faulty coordinator JGroups is forced to a halt for period of up to 20 s. The big spike afterwards compensates for this hole.

C. Local Times

In Figure 6, JGroups delivers all events quicker than EpTO in all scenarios, even when churn is involved as is shown in Figure 7. 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. We expect EpTO to outperform JGroups in this situation when we have a high number of peers in the system or when the network has a higher latency.

D. Global Times

Global times are represented in Figure 8 and Figure 9. 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 V-C.

E. Local Dissemination stretch

In Figure 10, We 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 minimal latency. The median dissemination stretch of JGroups is around 7 ms where as the median dissemination stretch of EpTO is around 630 ms for (100, 50). When increasing the number of peers, JGroups starts to have longer delivery times for some outliers. A small portion of the local dissemination stretches are really fast (1 ms or lower). This happens when the coordinator itself sends an event. Since it has to deliver it to himself

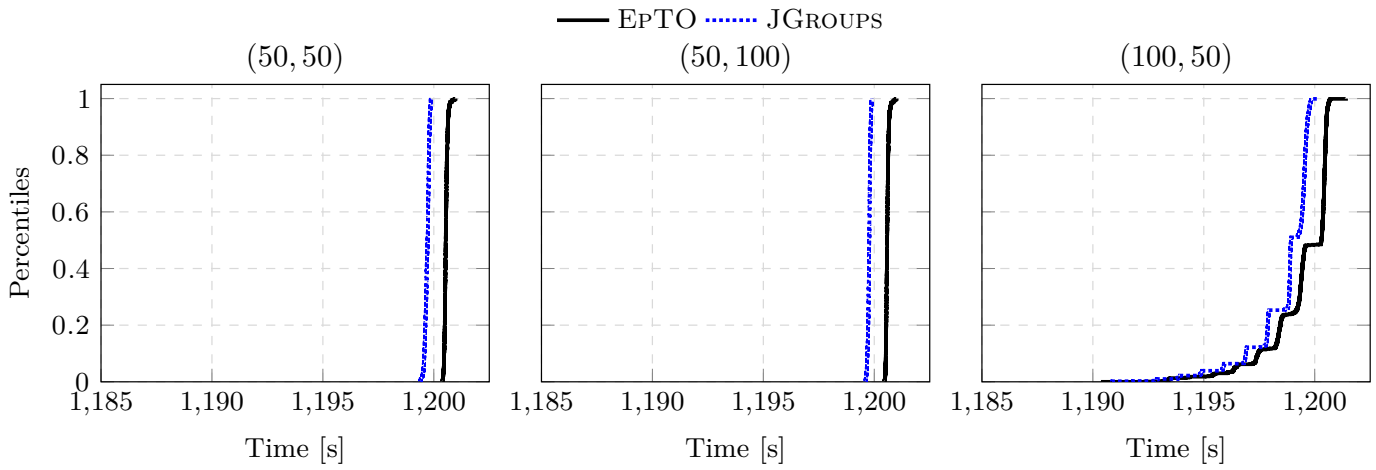

Figure 6. Local dissemination times

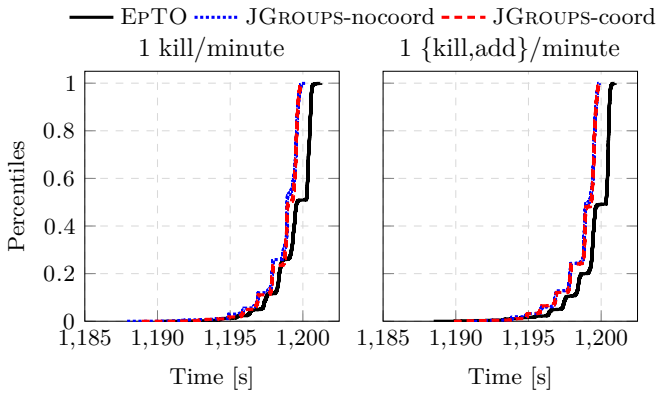

Figure 7. Local dissemination times with churn

the local dissemination stretch is intra-process and thus extremely fast.

In Figure 11 We 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 20 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 degrades a bit at 650 ms with the 99th percentile being at 1030 ms compared to 982 ms when no churn is happening.

F. Events sent

The variation observed in the different tests are due to the fact that we run an experiment for a period of time and that in some configurations a randomness decides whether an event is sent or not.

Table III
Total events sent in a stable environment

	Cluster parameters		
Protocol	(50, 50)	(50, 100)	(100, 50)
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 see that both EpTO and JGroups deliver the same amount of events. This is expected in a perfect environment. In Table IV When only killing nodes,

Table IV
Total events sent with a synthetic churn

	Cluster parameters	
Protocol	1 kill/minute	1{kill,add}/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

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 enough to draw any conclusion. **Jocelyn** ►*I didn't run any statistical analysis*◄

G. Real Trace

Figure 12 confirms the results from the synthetic churn. EpTO is not affected by the churn. As before, JGroups is affected in the same situations as when following a synthetic churn. The spikes visible on JGroups-coord and JGroups-nocoord show the different view changes occuring.

Table V also confirms our earlier results. EpTO is still within the $c * log_2(n)$ ratio fixed earlier.

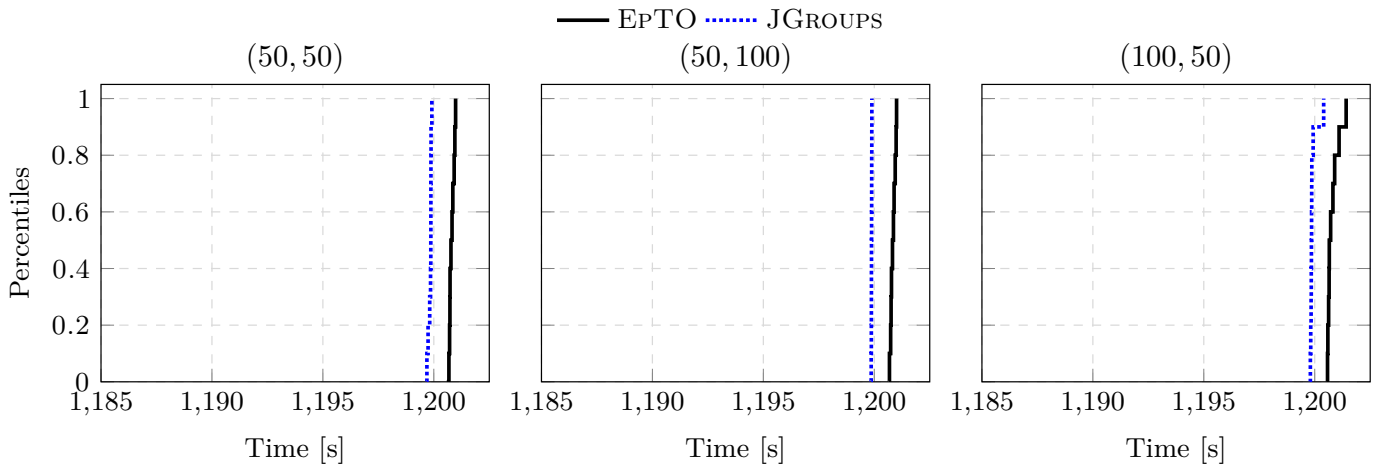

Figure 8. Global dissemination times

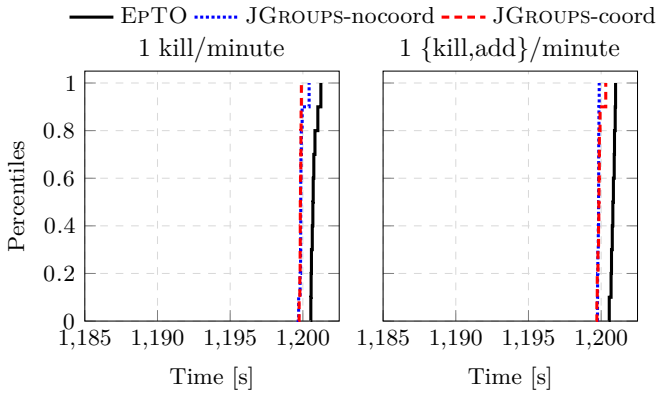

Figure 9. Global dissemination times with churn

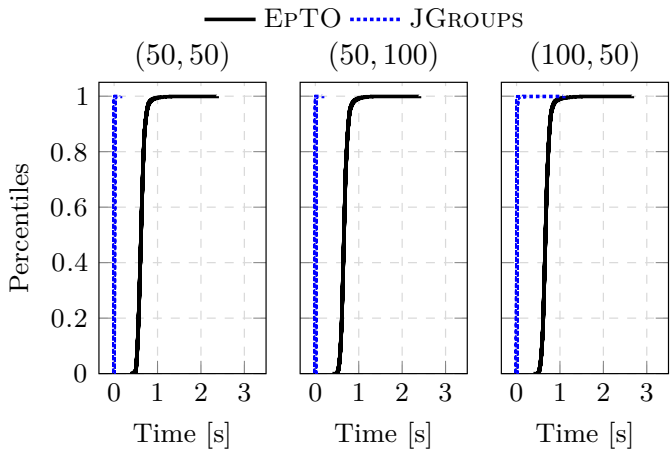

Figure 10. Local dissemination stretch

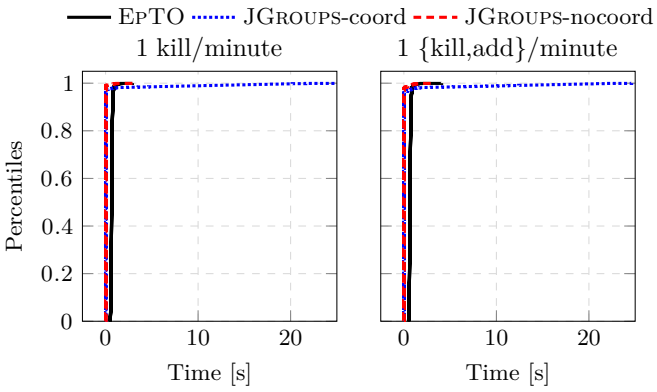

Figure 11. Local dissemination stretch with churn

Table V
Total GB sent/received

Protocol		Churn parameters Real Trace
EpTO	Receive Sending	81.41 ± 1.08 82.67 ± 1.08
JGroups-coord	Receive Sending	5.56 ± 0.08 5.40 ± 0.08
JGroups-nocoord	Receive Sending	5.58 ± 0.05 5.43 ± 0.05

The local and global dissemination times shown in respectively Figure 13 and Figure 14 still show JGroups to outperform EpTO in this scenario. Again, we want to emphasize that it is expected for JGroups to perform better than EpTO with the cluster size we use.

In Figure 15, EpTO as worse outliers than before and so does JGroups-coord. Table V is interesting. We see that on average JGroups seems to deliver more events than EpTO, but its standard deviation is way higher than

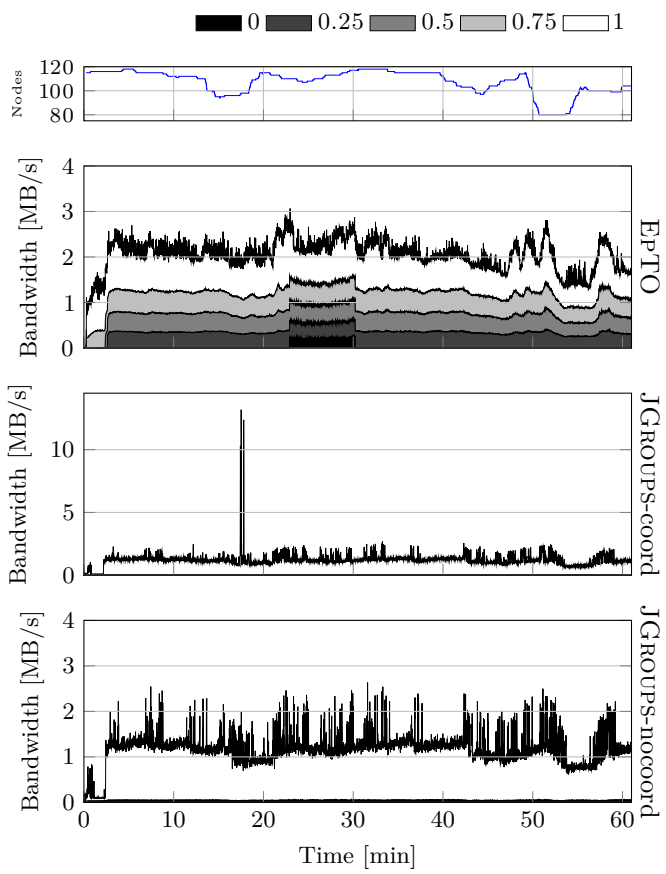

Figure 12. Bandwidth percentiles of a node during an experiment with a real churn

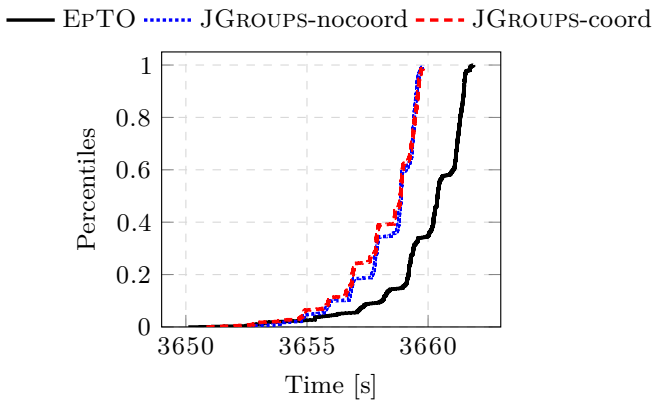

Figure 13. Local dissemination times

Table VI
TOTAL EVENTS SENT DURING A REAL TRACE

Protocol	
EpTO	165 844.2 ± 210.2
JGroups-coord	166 183.0 ± 1368.1
JGroups-nocoord	166 585.8 ± 824.9

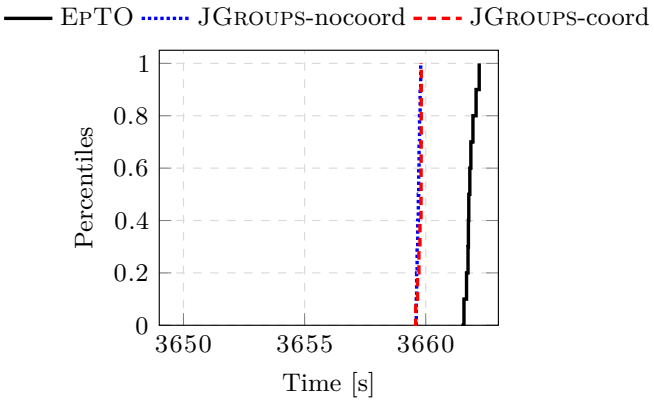

Figure 14. Global dissemination times with a real churn

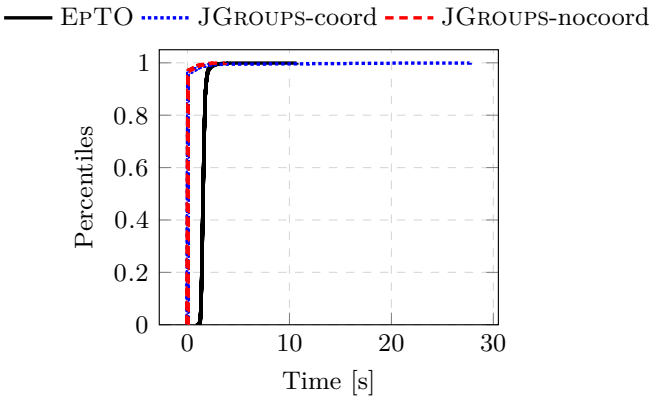

Figure 15. Local dissemination stretch with a real churn

that of EpTO, suggesting EpTO to be more stable and consistent.

H. Problems encountered

The EpTO simulation and theory is assuming we can have balls of infinite size. In practice, packet loss happens. We must therefore limit the ball size to a practical number. We use balls with a maximum size of 1432 B. We choose this number as the default MTU on Ethernet is 1500 B. We then subtract the bytes required for the IPv4 and UDP headers.

When running JGroups we sometimes get failed runs, where either some peers will experience holes or peer will deliver out of order events. This only happens when there is churn happening and when the coordinator dies. We have not investigated these problems further. This points to either JGroups failing when there is too much churn or a bug present in the SEQUENCER implementation.

VI. Future Work

Jocelyn ▶*Do we have Push-Pull references?*◀ Future work should focus on testing EpTO on a bigger cluster, preferably at a point where JGroups can no longer function, as EpTO is designed to work on very large number of

peers. As it is, EpTO does indeed work as intended in our infrastructure and under churn, but a larger test would showcase EpTO's ability to handle large peer numbers.

Future work also includes working on an alternate EpTO dissemination algorithm. This algorithm would use a Push-Pull approach to further reduce the needed number of balls in the system and thus reduce the network load.

VII. Conclusion

Jocelyn ►*Do later*◄ Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

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] (2016). Balls-and-bins figure, [Online]. Available: https://ned.ipac.caltech.edu/level5/Berg/Berg2.html (visited on 12/30/2016).

[15] (2016). JGroups - a toolkit for reliable messaging, [Online]. Available: http://jgroups.org/ (visited on 12/23/2016).

[16] 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.

[17] 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.