JEpTO

a practical evaluation on optimistic total ordering

Nicola Gilberti* and Diego Lobba†
*nicola.gilberti@studenti.unitn.it, †diego.lobba@studenti.unitn.it

***Abstract*—In this report it is presented JEpTO, an implementation of the EpTo agreement protocol for the Java language, exploiting the Akka framework to handle message exchanges and actor behaviour. The correctness of the original algorithm, from which our work is based, is questioned by showing a wrong execution trace. A proposal to address the issue and how this affects the overall performance of the protocol are further discussed. Finally, a series of measures is presented to evaluate the behaviour of our approach.**

I. Introduction

EpTO [1] is an algorithm for optimistic total order exploiting the balls and bins model [2] to spread reliably a rumor with high probability.

Safety properties(Table I) – guaranteed by the protocol design – include Integrity, Validity and Total Order. On the other hand, agreement holds with high probability (Probabilistic Agreement). With the term high probability, it is denoted the result for which a property holds with a probability $1-O(n^{-1})$, for some system parameter n.

In case of EpTO, let n be the total number of processes within the system, K be the number of processes to which an event is spread concurrently, let c be a constant value greater than one, finally let the TTL be a value which determines the time a given event is in the system and after which it will be delivered, then EpTO with logical clocks will achieve probabilistic agreement with the following values:

$$K \geq \left\lceil \frac{2\,e\,\ln n}{\ln\ln n} \right\rceil \tag{1}$$

$$\text{TTL} \geq 2\left\lceil (c+1)\log_2 n \right\rceil \tag{2}$$

The algorithm, exploiting logical clocks, goes through the following steps. An event is generated and broadcast, the broadcast action consists simply of inserting the event into a *ball*, comprising events that will be later sent to K deemed correct processes. The process uses a logical clock to synchronise itself with other processes. The clock is incremented on two occasions:

1) whenever a new event is generated the clock is incremented by one;
2) whenever an event coming from a source with a higher clock value is found, the clock is set equal to that value.

In the generation process, an event is timestamped with the process clock value at the time of the broadcast and a TTL value is initialised to zero.

Time is divided into rounds of fixed length, whose duration is the same for every process. At the beginning of a new round the *dissemination component* of the algorithm lies. Each event in the ball gets its TTL incremented. The ball is then sent to a sample of K deemed correct processes randomly chosen. The ball is then passed to the ordering component and emptied.

The *ordering component* of the algorithm deals with the received set of events and with their delivery. In this phase, care must be taken in order not to deliver an event out of order and to avoid the presence of holes in the delivery chain. Two sets are used to distinguish among those events already seen in previous rounds and those that can be safely delivered, the *received* and *deliverable* sets, respectively.

First, every event already in the received set (previously put in preceding rounds) has its TTL value incremented. Then each event in the ball passed by the

Integrity	Any event e is delivered at most once by every process, and only if e was previously broadcast.
Validity	If a correct process broadcast an event e, then it will eventually deliver e.
Total Order	For any two processes p and q that both deliver events e and e', then p delivers e before e' if and only if q delivers e before e'.
Probabilistic Agreement	If a process delivers an event e, then with high probability all correct processes eventually deliver e.

Table I
Properties defined by the EpTO algorithm

disseminating component is considered: if the event has not been delivered yet or its timestamp is less than the one of the last delivered event then it is ignored. If this is not the case total order breaks. Otherwise the event is checked against the received set. In the scenario in which the event is already present in the set with a lower TTL value, it is updated with the newer information. On the other hand it is appended to the set.

From the received set it is searched the non-deliverable event having the lowest timestamp value. Let M be the timestamp value of such event. Events considered to be deliverable by the *stability oracle* are moved to the deliverable set. The stability oracle will consider an event as stable, hence deliverable, if its TTL is greater than or equal to the value given by Equation (2). Events in the deliverable set, whose timestamp value is lower than M, can later be safely delivered. Those with timestamp greater are removed from the deliverable set.

The events remained in the deliverable set are sorted both based on their timestamp and generator's process id, and finally delivered, storing the timestamp value of the last delivered event.

II. Implementation

In this section it is described how we adapted EpTO to the actor model provided by Akka. An overview of the functionalities provided is then briefly discussed.

Akka is a framework which allows to build distributed applications through the actor model. From the point of view of the programmer an actor is an entity with state and behaviour, similarly to an object. Actor's actions are triggered trough messages. An actor's behaviour cannot be directly called in any way. This abstraction makes easy to deal with concurrent tasks, since serialisation points lie within the message reception phase.

The unique type of actor required to implement EpTO is the *EptoActor*. An EptoActor extends a *CyclonActor*, which in turn implements Cyclon [3]. A CyclonActor is used to draw a random sample of processes in the algorithm dissemination component.

To start a set of EptoActors a JoinMessage has to be sent to each participant, in order to define a common process already in the system. In the provided main program a single process is chosen as unique tracker which is used by other processes to join the network. By joining the network, new actors will have the tracker in their cache and Cyclon will obtain fresh information about other participants connected.

To effectively make JEpTO start, an EptoStartMessage must be sent to processes. This message triggers process events generation and starts the round interval needed by EpTO. Cyclon and Epto are encapsulated into non-interfering methods. View changes are therefore totally transparent to the EptoActor, which only interacts with Cyclon when drawing a random set of processes from its cache.

A. Cyclon

The extraction of a random set of peers used to send an information has proved to be valuable not only to avoid scenarios in which each participant is aware of all the others, but also to reduce communication overheads. This allows to empty the view size from unresponsive entries that probably failed and replace them with newly arrived actors (i.e. the so called*churn*). Churn simulation is not in the scope of this report, it is therefore not examined how the implementation proposed deals with it.

The unique motivation behind implementing Cyclon is to provide a Peer Sampling Service (PSS) to JEpTO, comparable to the original work. Our Cyclon implementation assumes that nodes, when joining, have a reference to a participant already in the network. This is accomplished by sending a *Join Message* to new participants. When receiving such message, the reference is added to the receiver's cache. Within the actor's cache, entries describing neighbouring nodes are tagged with a timestamp used for aging purposes, which is incremented at every Cyclon round.

Periodically, a node selects the oldest entry from its cache as future target for the shuffling phase. Then $L - 1$ nodes randomly chosen are added in the *Request Message*, together with the sender's ID having age 0. Nodes sent within the request message are stored by the sender into a temporary buffer and considered later in the merge phase.

Once the request message is received, the target selects L random entries from its cache and generates a *Reply Message* to send it back. It then performs a merge operation. In this, the receiver removes any reference to itself and to any nodes it already knows about, replacing nodes sent within the reply message with new entries in case no space is left in cache.

The sender, receiving the reply message, checks whether the reply corresponds to the request previously sent (by means of an ID). If this is the case, it performs a merge by inserting new nodes' information. If no space is left in cache it removes entries related to nodes stored in the temporary buffer and adds those obtained from the reply message. In case the reply does not correspond to any request previously sent, new nodes are added as long as there is room for them within the actor's cache.

B. Functionalities

The package implementing JEpTO is distributed as a Maven project, simplifying its building and execution phases. Two main programs are offered: one for a networked deployment of a single node and the other for simulating the behaviour of several participants over one or more broadcasting events.

While the use of networked nodes is of great help in testing the algorithm in a real deployment environment, the multi-actor program is at the base of the evaluation metrics considered due to its small memory footprint when scaling the number of actors invoked.

When calling the multi-actor main, it is possible to define the system parameters defining both Cyclon and EpTO behaviours (Equation (1) and Equation (2)) through a configuration file.

The configuration file (listing 1) depicts the set of parameters that can be tuned. The `jepto.config.cyclon` namespace defines the view size of an actor, together with the amount of entries exchanged during a round and its length, expressed in milliseconds.

The second set of parameters are related to EpTO. The number of receivers to which an event is spread during a round resembles the parameter K found in Equation (1). The maximum TTL values determines the value after which an event is deemed stable by the stability oracle. It is possible to set an arbitrary duration to the round interval, and the total number of actors in the system. The simulation will last for the amount of time defined by `sim-time`. Defining the number of senders allows just a subset of the total number of actors to broadcast a single event during the simulation, allowing fine grained analysis.

The log level parameter determines the amount of information written to the log (stored by default at `$HOME/EpToLogs/execution.log`). When set to *INFO*, only broadcast and delivery events are tracked. When set to *DEBUG* instead, the content of the received set and the deliverable set are printed at every round, for any actor. This was valuable for instance when producing the wrong execution trace.

The last parameter to be discussed is `as-paper`. When set to true the *OrderEvents* function used is the one described in the original paper. When set to false it is the one described in Algorithm 1.

Listing 1. "Example of configuration file"

```
jepto.config {
    cyclon {
        view-size            = 100
        shuffle-length       = 30
        shuffle-period-millis = 100
    }
    num-receivers    = 17
    max-ttl          = 43
    round-interval   = 5000
    num-actors       = 100
    as-paper         = false
    # define the simulation time in seconds
    # if not defined, the system keeps going
    sim-time         = 20
    # if not defined, backup to "INFO"
    log-level        = "INFO"
    # comment num-senders for continuous
        event generation
    num-senders      = 1
}
```

III. Discussion

In this section it is argued the correctness of the original protocol, showing a wrong execution trace and proposing an alternative solution.

The pseudocode presented in the original article assumes the use of global clocks and its extension to logical clocks does not involve any change to the code-base. Unfortunately, it does not deal with concurrent events, which is, events having the same timestamp value. In particular the ordering component, defined by the *OrderEvents* function, has two conditions that make possible for two processes to deliver two events in a different delivery order, hence breaking total order which should hold deterministically.

The first issue is given in the check done at line 9 where events in the ball are considered. The event is added to the received set in case it has not been delivered yet and its timestamp is greater than or equal to the one corresponding the last delivered event.

By doing so, we argue that it is possible to accept an event with the same timestamp as the one delivered lastly, hence eventually delivering an event that maybe has been already delivered following another order by another process.

Consider a set of three processes where actors 1 and 2, respectively, generate events e_{11}, e_{21} at round $t = 1$, hence both events have the same logical timestamp. Assume, for simplicity that the maximum TTL set is 1, hence both events are delivered as soon as they are received. Suppose the clock drift between the processes is such that events are not distributed within the same round. Therefore, either e_{21} is received before e_{11} or the opposite. We consider the first case for this discussion.

Although e_{11} is received within the round after event e_{21} reception, the generating process for the event e_{11} adds it to the received set in the first round together with event e_{21}. Note that other processes receive only event e_{21} during the first round and only receive e_{11} in the round after. It is easy to see that process p_1 delivers $e_{11} \rightarrow e_{21}$ at t, while other process deliver just e_{21} at t and only later at $t + 1$ deliver e_{11}, breaking total order.

This could have been avoided if processes p_2 and p_3 ignored any message concurrent with the last delivered event, in this case being e_{21} and being e_{11} the event to ignore.

A second criticality is at line 23. Let M be the timestamp value of the event within the received set such that it is a non-deliverable event having the lowest timestamp value. In this phase, events in the deliverable set are examined and those with timestamp strictly lower than M are removed from the deliverable set. Therefore they cannot be delivered in the current round.

With this condition, it is possible to have two events with the same timestamp deemed stable at different rounds, due to different TTL aging steps. Hence, the two events will be delivered in different rounds, making total ordering dependant on the assumption that every process receives an event e at the same round t. It is possible to prove this execution by considering the same scenario described previously, in the case in which the maximum TTL is set to 2.

One of the wrong execution traces we had while we were developing JEpTO is depicted in Figure 1. To address these issues we simply changed the conditions both at line 9 and at line 23 in order to block any event arriving after a concurrent event has already been delivered (line 9) and removing from the deliverable set any event with the timestamp equal to M. These changes (described in Algorithm 1) allow a process to deliver an event only when every concurrent event has been deemed stable by the stability oracle, increasing greatly the delay between the event reception and the corresponding delivery.

IV. Performance

In this section the set of tests we made and the results we obtained are discussed, together with the

Algorithm 1 Ordering component (process p)

```
 5: procedure ORDEREVENTS(ball)
 6:     ...
 8:     for event ∈ ball do
 9:         if event.id ∉ delivered ∧
             event.ts > lastDeliveredTs then
10:             // update ttl if more recent
11:     ...
22:     for event ∈ deliverableEvents do
23:         if event.ts ≥ minQueuedTs then
24:             // remove event from deliverable
25:         else
26:             // remove event from received
```

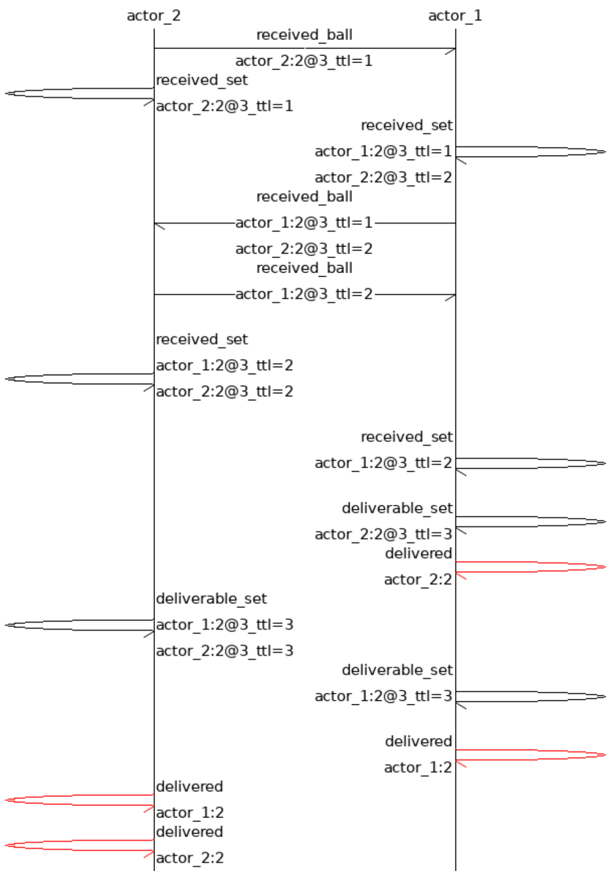


Figure 1. Wrong execution trace depicting a different delivery order. It is possible to notice that actor a_2 delivers $a_{12} \rightarrow a_{22}$ while actor a_1 delivers $a_{22} \rightarrow a_{12}$.

assumptions considered when setting system parameter values. The results presented are produced by JEpTO after applying the modifications described in Algorithm 1.

The main goal of this evaluation is to give a reproducible set of tests both related to the basic functioning of the algorithm and for its scalability in terms of number of processes.

It has been a daunting task to replicate the charts depicted in the original article, due to the fact the authors were using a discrete event simulator to test the behaviour of EpTO and not a real deployable implementation.

Since the protocol makes use of Cyclon for its disseminating component and it is not trivial to take into account also for it into the evaluation, we decided to set the view size of each process to fit the whole set of processes, and to fill the whole view within a short period of time. This, together with the lack of churn possibility allows to evaluate the behaviour of EpTO without any interference from the underlying epidemic algorithm and to have the first round to occur when Cyclon is already in steady state.

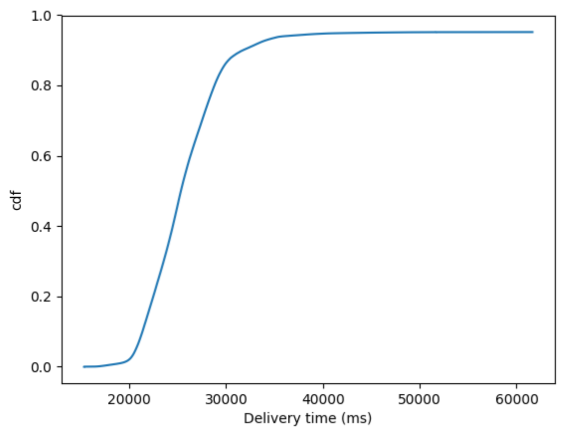
The overall testing behaviour is the same for all tests. Initially a set of n processes is set up. One of the nodes is chosen as tracker and will be made available for any other node to join the network. The initial network topology will be therefore a star topology centred in the tracker.

By exploiting Cyclon, nodes periodically exchange information with the oldest entry from their cache (initially comprising only the tracker) and collect information about other nodes. After a period of time elapsing several Cyclon rounds, EpTO starts.

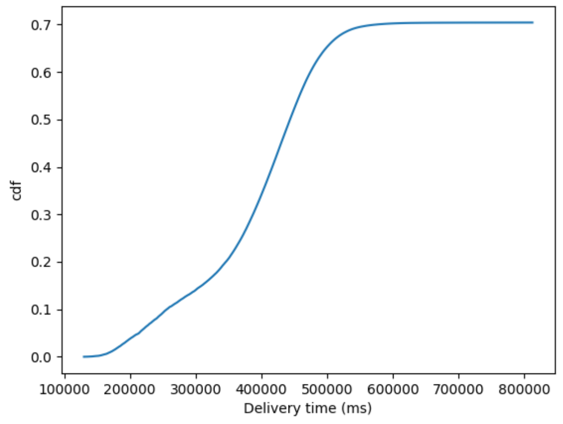
A. Deployment tests

In this part it is investigated the behaviour of the protocol without setting constraints of any kind. The testing environment is such that each process generates a single event at each round and *EptoBroadcast*s it. The protocol continues for twenty minutes after which it is interrupted.

Once finished it is computed the difference between the time a given message has been EptoBroadcast and the time it has been delivered by any process. Similar delta delay values are grouped together and normalised by the number of actors, obtaining values between zero and one. From these values we computed the cumulative distribution function.



(a)



(b)

Figure 2. CDF of the delivery delay of 100(a) and 1000(b) processes running JEpTO for 20 minutes.

It has to be noted that by stopping suddenly the protocol, several events broadcast during the last round and those not already delivered will be taken into account in the performance measures, leading to a drop in the overall delivery probability. In Figure 2a it is depicted the outcome of the test for $n = 100$. The overall delivery probability being around 98%.

In Figure 2b the same test is run for thousand processes. It is possible to notice the poor overall delivery probability. This is to be attributed to the testing environment, which used a single threaded processor employing thousands of concurrent Akka actors, each one generating messages.

From this outcome, it is clear that this estimations can only give a glimpse of the protocol behaviour in a real deploying environment. It is therefore required, to have more meaningful outcomes, to establish more constraints on the number of sending processes.

B. TTL analysis

Here it is analysed how the c constant affects the maximum TTL value after which an event is deemed stable by the stability oracle.

In reference to Equation (2), c is a parameter which increments the maximum TTL value. This leads to an increase in the dwell time of an event, prior to its delivery.

For this evaluation, a single process is allowed to generate a unique event. The test analyses the behaviour of the protocol varying c between two and four, repeating the simulation for a single configuration twenty times. Outcomes related to equivalent c values have been averaged.

The results obtained for n equals to 100 processes and equals to 1000 processes are depicted in Figure 3,respectively.

As expected, as c increases, the delivery time for the packet by other processes grows. Moreover as the number of processes gets higher the time required to achieve a delivery of more than fifty percent is decreased, while it requires more time to spread the event to the whole set of processes. This can be noted by the number of smaller steps in Figure 3 for thousand processes and c equals to three and four, looking for delivery rate higher than eighty percent.

C. Broadcast rate evaluation

In this last performance evaluation, it is measured how the delivery delay is affected by an increase in the number of concurrent event generators.

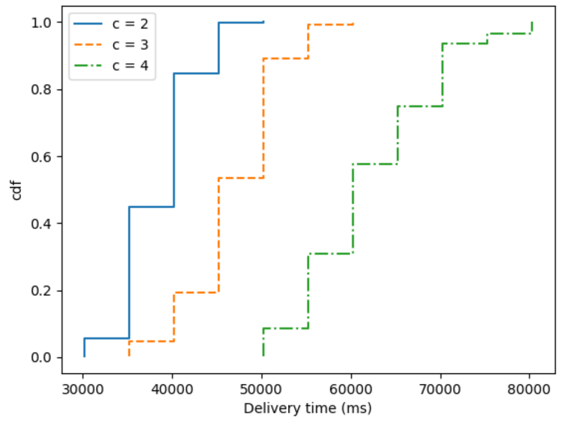
In the test scenarios, s denotes the number of senders, varying between one, five and ten, which in the case of hundred processes gives the protocol behaviour for the broadcast rate of one, five and ten percent, respectively.

In Figure 4 is depicted the measures considering the value of c to be equal to two, three and four, respectively. Within a single graph it is described the variation obtained according to the broadcast rate.

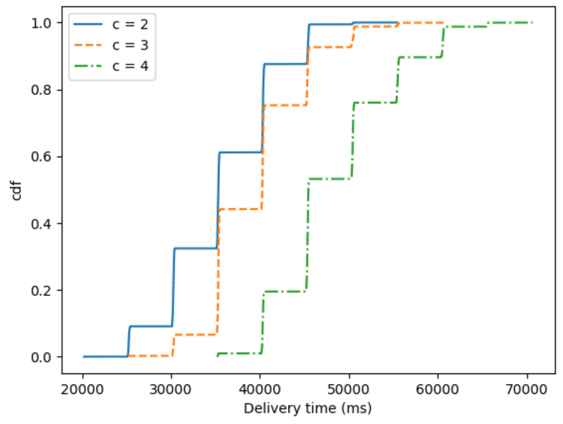
D. Total order

During the tests performed, each simulation has been checked to be compliant to the deterministic properties of EpTO. In particular the simulations have been proved to satisfy total order.

From the point of view of JEpTO, the i-th event e_i produced by process k is represented by a string



(a)



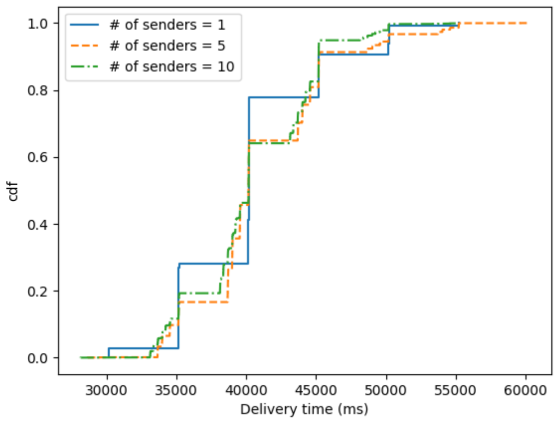
(b)

Figure 3. Delivery delay for a single event when JEpTO is run by 100 and 1000 processes.

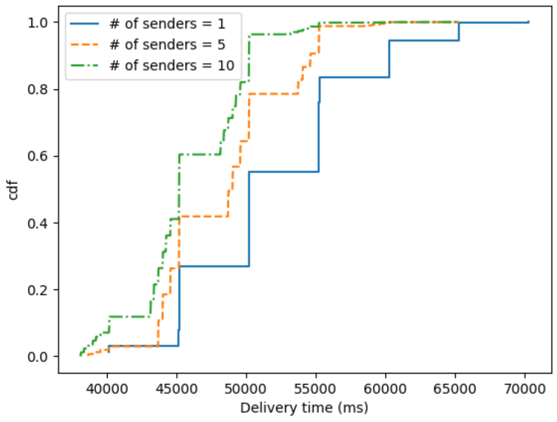
"pk:i". This format uniquely represents an event, such that it is not possible to have two events identified by the same string. From the simulation log, the ordered sequence of delivered events by a process q is extracted and used to verify the total order property to hold.

For the validation, a straightforward algorithm has been designed and implemented. The function performs the check on the delivery sequences of two processes only, hence a pairwise comparison among all processes is required for the property to be checked.

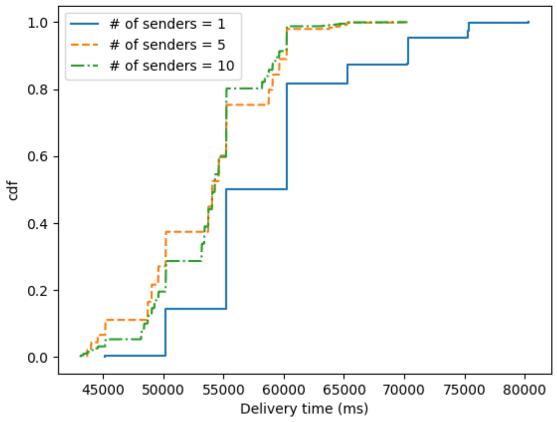
Initially, a structure mapping each element to the corresponding location within the two sequences is built (Algorithm 2). If an element is not present in one sequence, the index is set to -1. Total order holds among the two sequences if, for any pair of successive elements within one of the two sequences, the positions



(a)



(b)



(c)

Figure 4. Delivery delay for broadcast rate of 1, 5, and 10% with c equals to 2, 3, 4.

of the characters in both collections have the same order relationship (Algorithm 3).

Algorithm 2 Save the index of each sequence element

```
1:  function STOREINDEXES(seq1, seq2)
2:      indexMap ← orderedDictionary
3:      for $i \leftarrow 0$, len(seq1) do
4:          charIth ← charAt(seq1, $i$)
5:          // Assume seq2 does not contain
6:          // the element initially
7:          pos $\leftarrow < i, -1 >$
8:          insertAt(indexMap, charIth, pos)
9:      for $i \leftarrow 0$, len(seq2) do
10:         charIth ← charAt(seq2, $i$)
11:         // Assume seq1 does not contain
12:         // the element
13:         pos1 $\leftarrow -1$
14:         if contains(indexMap, charIth) then
15:             // Extract the first element
16:             // from the tuple
17:             pos1 ← get(indexMap, charIth)[0]
18:         pos $\leftarrow < \text{pos1}, i >$
19:         insertAt(indexMap, charIth, pos)
20:     return indexMap
```

V. Conclusions

In this report it is presented JEpTO, an implementation of the EpTO algorithm exploiting the Akka framework. A scenario that breaks the total order property of the original algorithm is presented. A possible solution addressing the issue is proposed and examined. Finally, a set of analysis have been presented to evaluate the performance both in a constrained environment and in a free, deployment-alike setting. JEpTO confirms packet delivery with high probability within the estimated time stated by analytical models. However, we argue that further investigation should be assessed with respect to the packet delivery latency, due to changes applied to the primary algorithm.

Algorithm 3 Check total order property on two sequences

```
function totalOrder(seq1, seq2)
    indexMap ← storeIndexes(seq1, seq2)
    // compare seq1 against seq2
    for $i \leftarrow 0$, len(seq1) $- 1$ do
        // store the position of characters
        // $i$ and $i + 1$ within seq1 and seq2
        $c1 \leftarrow$ charAt(seq1, $i$)
        $c2 \leftarrow$ charAt(seq1, $i + 1$)
        $< s_{11}, s_{21} > \leftarrow$ get(indexMap, $c1$)
        $< s_{12}, s_{22} > \leftarrow$ get(indexMap, $c2$)
        if any of $< s_{11}, s_{21}, s_{12}, s_{22} >= -1$ then
            // missing an element in seq1 or seq2,
            // ignore comparison
            continue
        if $s_{11} < s_{12} \wedge s_{21} > s_{22}$ then
            return false
        if $s_{11} > s_{12} \wedge s_{21} < s_{22}$ then
            return false
    // compare seq2 against seq1
    for $i \leftarrow 0$, len(seq2) $- 1$ do
        // store the position of characters
        // $i$ and $i + 1$ within seq1 and seq2
        $c1 \leftarrow$ charAt(seq2, $i$)
        $c2 \leftarrow$ charAt(seq2, $i + 1$)
        $< s_{11}, s_{21} > \leftarrow$ get(indexMap, $c1$)
        $< s_{12}, s_{22} > \leftarrow$ get(indexMap, $c2$)
        if any of $< s_{11}, s_{21}, s_{12}, s_{22} >= -1$ then
            continue
        if $s_{21} < s_{22} \wedge s_{11} > s_{12}$ then
            return false
        if $s_{21} > s_{22} \wedge s_{11} < s_{12}$ then
            return false
    return true
```

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*, ser. Middleware '15. New York, NY, USA: ACM, 2015, pp. 100–111. [Online]. Available: http://doi.acm.org/10.1145/2814576.2814804

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

[3] 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, jun 2005. [Online]. Available: https://doi.org/10.1007%2Fs10922-005-4441-x