Exon: An Oblivious Exactly-Once Messaging Protocol

1st Ziad Kassam
HASLab, INESC TEC & Minho University
Braga, Portugal
ziad.a.kassam@inesctec.pt

2nd Paulo Sérgio Almeida
HASLab, INESC TEC & Minho University
Braga, Portugal
psa@di.uminho.pt

3rd Ali Shoker
RC3, CEMSE - KAUST
KSA
ali.shoker@kaust.edu.sa

***Abstract*—TCP is typically the default transport protocol of choice for its supposed reliability, even for message-oriented middleware (e.g., ZeroMQ) or inter-actor communication (e.g., distributed Erlang). However, under network issues, TCP connections can fail, which requires ensuring both at-least-once and at-most-once delivery at the upper middleware layer. Moreover, the use of TCP at scale, in highly concurrent systems, can lead to drastic performance loss due to the need for TCP connection multiplexing and the resulting head-of-line blocking. This paper introduces *Exon*, an oblivious exactly-once messaging protocol, and a corresponding lightweight library implementation. Exon uses a novel strategy of a per-message four-way protocol to ensure oblivious exactly-once messaging, with on-demand protocol-level "soft half-connections" that are established when needed and safely discarded. This achieves correctness, obliviousness, and performance, through merging and pipelining basic protocol messages. The empirical evaluation of Exon demonstrates significant improvements in throughput and latency under packet loss, while maintaining a negligible overhead over TCP in healthy networks.**

***Index Terms*—Exactly-Once EO; message delivery; reliability; fault tolerance; oblivious**

I. Introduction

As modern applications are becoming more networked and distributed, there is an increasing interest in reliable messaging protocols to reduce the application complexity and the burden on developers. At the core of reliable messaging, exactly-once message delivery is a crucial property to guarantee both at-least-once and at-most-once delivery [19]. The TCP protocol [27] is often the transport protocol of choice as it is known to ensure exactly-once within a connection (in addition to its good performance).

However, TCP poses two problems. The first is when the connection fails, exactly-once is no longer guaranteed and must, therefore, be delegated to the upper layers [9], [5], [4], [23]. This promoted the trend of using some message queue (MQ) middlewares on top of the transport layer to ensure exactly-once, and simplify building distributed applications. However, these MQs are heavyweight, going beyond tolerating network problems, aiming for tolerance to node failures, by using persistent storage and replication (e.g., Apache Kafka [23], RabbitMQ [29], ActiveMQ[34]). If the purpose is simply tolerating network failures while scaling to highly concurrent settings (e.g., distributed Erlang [38]), the quest for a general purpose lightweight exactly-once messaging protocol is not over.

The second problem of TCP is the head-of-line (HOL) blocking while using TCP multiplexing, where a single connection is shared by several application entities [9]. TCP multiplexing pervades large-scale distributed systems (e.g., CORBA ORBs [15], distributed Erlang [38]) to reduce used resources (e.g., ports and buffers) and latency of TCP connection establishment. Nevertheless, multiplexing TCP connections can cause needless latency in delivering messages from concurrent entities, which could be delivered in any order, but will be constrained by the FIFO order in the multiplexed TCP stream. This problem, observed in HTTP/2 due to multiplexing TCP connections lead to abandoning TCP in HTTP/3 [7]. Moreover, this problem is aggravated by message loss. As our evaluation shows that even a small message loss rate degrades the performance significantly.

This paper presents *Exon*, a new general purpose oblivious exactly-once messaging protocol, and corresponding library. The protocol guarantees exactly-once by design: 1) at-most-once, as no message payload, carried in a *token*, can be delivered without previously "booking" a unique receptor *slot*, which is consumed upon delivery; 2) at-least-once because no message is discarded without being sure that the corresponding token has been consumed.

Moreover, even though the protocol is conceptually 4-way per-message, to ensure correctness, it achieves efficiency by using what we call soft half-connections: on-demand half-duplex connections transparent to the API, which remains message-based. This allows achieving both: 1) latency similar to TCP, that requires three one-way trips to deliver the first of a sequence of messages and then just one RTT, 2) obliviousness, being able to discard all per connection state without depending on any timing assumption for correctness (i.e., no TIME_WAIT or similar concept).

Being connectionless, Exon allows long-lived messaging that survives IP changes, such as in mobility scenarios, using logical node identifiers. Being oblivious, Exon is useful for systems having constrained devices, such as in IoT, and when each node communicates with many others over time. This is possible by avoiding memory buildup, as it requires only a single sequence number as a persistent state. Moreover, the experimental evaluation shows Exon performance to be

significantly better than TCP under packet loss, and to have negligible overhead otherwise.

In general, Exon is a useful abstraction that can serve many different roles: replacing TCP in inter node communication used by distributed algorithms on a large highly concurrent system; replacing a too-heavy messaging middleware used by a higher-level application; replacing TCP in a generic messaging middleware.

The contributions in this paper are summarized in the following: (1) A novel general purpose protocol, Exon, for oblivious exactly-once messaging. (2) A proof (sketch) of safety and liveness properties to achieve exactly-once delivery. (3) An open-source library, Exon-Lib, implementing Exon over UDP, that exposes a generic API to be used as a building block for distributed applications. (4) An empirical evaluation in different network settings.

The rest of the paper is organized as follows. We start with an overview of related work in Section II to show how our work positions itself. We then present the protocol in Section III, followed by the proof sketches in Section IV. We finally present a performance evaluation in Section V, and we conclude with Section VI.

II. Related Work

The works on reliable communications, to provide both exactly-once (EO) delivery and good performance, date back to the early days of the ARPA network [10]. In particular, the research on the transport layer [37], [11] lead to the foundation of the Internet Protocol (IP) that results in two mainstream transport protocols: TCP and UDP. UDP [28] is a basic datagram protocol that provides no reliability guarantees, but stands as a communication primitive to support building other protocols as needed. TCP [27] is a stream- and connection-based protocol that provides reliability guarantees like exactly-once, FIFO ordering, and performance (e.g., congestion and flow control).

Nevertheless, EO guarantees in TCP only hold within a connection/session; when the connection fails (likely to occur in current WAN environments and long-lived communications), it either allows for message loss or duplication, as Belsnes [6] shows for any single-message communication. Attiya et al. [2] proved that when state information is not saved between incarnations, the problem is solvable if and only if the network is FIFO—which is not the case for most networks. Therefore, to ensure EO, it is necessary to retain inter-connection information, e.g., at an upper layer. One possibility is using a fail-over protocol [40], [33], [12], [35].

Among the TCP fail-over protocols, Zandy et al. [40] used the idea of *Persistent Connections* to preserve the endpoint of a failed connection in a suspended state for an arbitrary period of time. Then, the protocol automatically reconnects using session fail-over to another process transparently. However, no guarantees are made beyond the defined time. Similarly, Snoeren [33] used *Connection Migration* to migrate a connection session to another endpoint. Both Robust TCP (RTCP) and Exactly-Once Middleware used a similar technique, called *Connection Persistence*, when a TCP connection breaks [12], [20]. They used an out-of-band UDP connection recovery for EO and FIFO. RTCP retains unique connection identifier (CID) in order to distinguish between different incarnations; this makes it non-oblivious, unlike Exon. FT-TCP [1] used a wrapper that saves the states in a *logger* (another process) to mask and recover the TCP connection, even under node failures.

Since TCP is optimized for the general use, it exhibits limitations in scenarios like bulk transfer, multicast, concurrent systems, computational grids, fast networks, etc. [36], [16], [13], [25], [24], [21], [17]. This motivated the design of reliable transport-level alternatives on top of UDP. Nevertheless, although these protocols managed to solve the ordering and performance reliability (congestion control) issues of TCP, they only partially solved the EO. The reason is that they embraced the connection-based approach, which eventually led to the same TCP issues discussed above.

In particular, Reliable UDP (RUDP) uses redundant connections over UDP [8]. A connection failure is solved by signaling a timed state transfer to an Upper Layer Protocol. If the latter does not transfer the state before the timer expires (after one second), the connection state is lost, and its buffers are freed—thus not ensuring EO. The RTP [31] protocol has the same issues as it provides UDP connections without EO guarantees; but it relies on an RTP Control Protocol (RTCP) to maintain reliability through storing a lot of meta-data sessions with timeouts. The same holds for RBUDP [18] that must keep a tally of the packets to determine which packets must be retransmitted at the end of a bulk transmission under failure. Like Exon, other protocols like SCTP, UDT, and ENet improved reliability without ordering guarantees [36], [16], [41], [30], but again using connection timeouts.

A. Efficiency, exactly-once, obliviousness and no timing assumptions, pick four

The novelty of Exon is simultaneously achieving EO delivery, obliviousness, efficiency and no dependence on timing assumptions for correctness. For efficiency, some notion of connection is needed. Our starting point, a 4-way message protocol would need 3 one-way trips to deliver each message. Less than 3 one-way trips is possible: but ensuring optimal two-way handshake in [32] is achieved by forgoing obliviousness, as it assumes the existence of a cache that holds meta-data about nodes.

Indeed, Attiya et al. [3] proved that if the nodes have unbounded memory, a three-way handshake oblivious protocol exists, whereas a two-way handshake oblivious protocol does not exist. The same paper proved also that if a bound on maximum packet lifetime (MPL) exists and is known, then a two-way handshake oblivious protocol is possible, but at least MPL must elapse between the time two consecutive incarnations are established—which is impractical in most networked applications. Therefore, most reliable and efficient protocols are three-way handshake based, but they expose

full-duplex connections in the API, and depend on timing assumptions for obliviousness.

The protocol most related to Exon is Attiya's three-way handshake oblivious unbounded memory protocol [3], in also only requiring a single integer as node state between incarnations. But Attiya's protocol is classically connection based, with API-visible full-duplex connections, which prevents safely ensuring exactly-once message delivery, as the receiving side can concurrently close the connection, preventing delivery.

Our approach starts from a message-based four-way protocol, inefficient but ensuring exactly-once, obliviousness, and no dependence on timing assumptions. It is then augmented with on-demand half-duplex soft-connections, hidden from the API, to achieve performance, while retaining the other three properties. This novel combination of components results in an efficient, oblivious, exactly-once messaging protocol with no timing assumptions for correctness.

III. Exon Protocol

Exon is a host-to-host message-based protocol that is optimized to guarantee the exactly-once (EO) delivery of these messages. This is possible through the concept of reserving slots at the destination host before sending any payload. When a slot is first consumed at the destination host, it is deleted and, therefore, duplication will not occur no matter how many retransmissions are done, e.g., given possible network issues.

In a nutshell, Exon has a combination of components which allows ensuring EO while being network and memory efficient, namely:

Soft- half-connections: connections are useful to group identifiers like sequence numbers and achieve performance. We have what we call soft- half-connections (s-connection), that group messages from the same sender-receiver pair, created on-demand if messages are requested to be sent. For performance, the s-connection can be discarded if there are no pending unacknowledged messages, after some non-short timeout. EO correctness is ensured without timeouts though (e.g., no TIME_WAIT as in TCP).

Oblivious: Exon achieves EO correctness without the need to keep s-connection-related information forever, keeping only a single integer per node as permanent state, when no s-connections are present.

Order-less: to be more generic, Exon is deprived from unnecessary ordering restrictions of messages. Message ordering (e.g., FIFO) can be implemented on top of Exon if required.

In the following, we emphasize Exon in more detail.

A. System Model

We consider a networked or distributed system of any number of nodes. A node can have wide or constrained capacities, but a local memory is required. Nodes can be long-lived sticky members of some service or transient ones (e.g., as in vehicular networks). Nodes can crash but will eventually recover with the content of the last state prior to the crash. In this case, a stable persistent storage is assumed.

On the network side, any node can communicate with any other node via a network (e.g., WAN, WSN, LAN). The network is asynchronous, with no global clock, no bound neither on the time it takes for a message to arrive, nor on the processing speeds. The network is unreliable: messages can be lost, duplicated or reordered (but are not corrupted). The network may have long partitions, but these will eventually heal. Exon assumes the existence of an underlying transport-level communication channel—or any equivalent non-IP abstraction—to send messages in any form (bytes, datagrams, etc.); although our current Exon-lib is implemented on top of UDP.

B. Overview

The communication is done between two nodes, a sender (i) and a receiver (j), as follows: (1) Node i starts the communication where it has a message to send to j, it sends a REQSLOTS message to j, (2) j creates a *slot* for i and sends it back. Then, (3) after node i receives the slot, it generates a message payload *token* and sends it to j. After that, (4) j receives the token, removes the slot, delivers the "payload" to the specific upper layer application, and sends an acknowledgment *ack* to i, where i can safely removes the token.

In order to optimize the slot-at-a-time request overhead, the sender can request a window of slots in advance using empty message place-holders, we call *envelopes*. The sender can associate message tokens to these ready envelopes with reserved slots when needed.

C. The Algorithm

We now describe the algorithm details of Exon. To simplify the presentation, we refer to the corresponding lines in Algorithm 1. We also exemplify the algorithm with a simple instance, which we walk through to clarify how the state is changing at each step.

Algorithm 1 is the protocol for a generic node i. It conveys the node state, the types used in defining it, the atomic actions (both for when a send is requested locally and when messages arrive), and the called auxiliary procedures. The algorithm also presents a procedure that runs periodically, to show in a minimal way how to cope with message loss (an actual implementation could have, e.g., timeouts per half-connection to trigger these sends).

1) **Notations and Definitions:** Each node i has an Exon state. The state is a node-wide clock (ck_i) keeping a monotonically increasing integer, and a pair of maps: S_i keeping sender-side half-connection **records** of type $\mathbb{S}$ and R_i keeping receiver-side half-connection records of type $\mathbb{R}$. We use an i subscript to denote node state variables or actions and unsubscripted names for local temporary variables; we use $:=$ for assignment, typically to a state variable, and $=$ for a let binding which binds a name to a value.

While in abstract, and for correctness, we use the concepts of slots, envelopes and tokens as globally unique entities, which are kept in the node state grouped in the sender-side

```
1  types
2    𝕀, node identifiers
3    𝕄, message payloads
4    𝕊 : record {
5      sck : ℕ, sender clock
6      rck : ℕ, receiver clock
7      msg : 𝕄*, messages queued to send
8      env : ℕ*, list of available envelopes
9      tok : ℕ ↪ 𝕄, tokens with messages
10   }, sender-side connection record
11   ℝ : record {
12     sck : ℕ, sender clock
13     rck : ℕ, receiver clock
14     slt : 𝒫(ℕ), set of available slots
15   }, receiver-side connection record
16 parameters
17   N : ℕ, number of slots requested in advance
18 state:
19   ck_i : ℕ = 0, node clock
20   S_i : 𝕀 ↪ 𝕊 = ∅, map of sender-side records
21   R_i : 𝕀 ↪ ℝ = ∅, map of receiver-side records
22 on EOsend_i(j, m)
23   if j ∉ dom(S_i) then
24     S_i[j] := 𝕊{sck : ck_i, rck : 0, msg : [m], env :
         [], tok : ∅}
25     requestSlots_i(j)
26   else
27     c = S_i[j]
28     if c.env ≠ [] then
29       e = c.env.dequeue()
30       if |c.env| = N − 1 then
31         requestSlots_i(j)
32       c.tok[e] := m
33       send_{i,j}(TOKEN, e, c.rck, m)
34     else
35       c.msg.enqueue(m)

36 proc requestSlots_i(j)
37   c = S_i[j]
38   n = N + |c.msg| − |c.env|
39   if n > 0 then
40     l = if c.tok ≠ ∅ then min(dom(c.tok))
41       else if c.env ≠ [] then c.env[0]
42       else c.sck
43     send_{i,j}(REQSLOTS, c.sck, n, l)
44   else if c.tok = ∅ and c.msg = [] then
45     send_{i,j}(REQSLOTS, c.sck, 0, c.sck)
46     ck_i := max(ck_i, c.sck)
47     S_i.remove(j)

48 on receive_{j,i}(REQSLOTS, s, n, l)
49   if j ∉ dom(R_i) then
50     R_i[j] := ℝ{sck : s, rck : ck_i, slt : ∅}
51     ck_i := ck_i + 1
52   c = R_i[j]
53   c.slt := {m ∈ c.slt | m ≥ l}
54   if n > 0 then
55     if s + n > c.sck then
56       c.slt.union({c.sck, . . . , s + n − 1})
57       c.sck := s + n
58     send_{i,j}(SLOTS, s, c.rck, n)
59   if c.slt = ∅ then
60     R_i.remove(j)

61 on receive_{j,i}(SLOTS, s, r, n)
62   if j ∉ dom(S_i) then
63     send_{i,j}(REQSLOTS, ck_i, 0, ck_i)
64   else if s = S_i[j].sck then
65     c = S_i[j]
66     c.rck := r
67     c.env.append([s, . . . , s + n − 1])
68     c.sck := s + n
69     while c.env ≠ [] and c.msg ≠ [] do
70       e = c.env.dequeue()
71       m = c.msg.dequeue()
72       c.tok[e] := m
73       send_{i,j}(TOKEN, e, c.rck, m)
74     requestSlots_i(j)

75 on receive_{j,i}(TOKEN, s, r, m)
76   if j ∈ dom(R_i) then
77     c = R_i[j]
78     if r = c.rck and s ∈ c.slt then
79       c.slt.remove(s)
80       deliver_i(m)
81   send_{i,j}(ACK, s, r)

82 on receive_{j,i}(ACK, s, r)
83   if j ∈ dom(S_i) then
84     c = S_i[j]
85     if r = c.rck and s ∈ dom(c.tok) then
86       c.tok.remove(s)

87 periodically
88   for (j, c) in S_i do
89     for (s, m) in c.tok do
90       send_{i,j}(TOKEN, s, c.rck, m)
91     requestSlots_i(j)
92   for (j, c) in R_i do
93     send_{i,j}(SLOTS, c.sck, c.rck, 0)
```

ALGORITHM 1: Exon algorithm for a generic node i

or receiver-side half-connection records. We now define these concepts and describe how they are stored in nodes.

Definition 1 (Slot)**.** A slot with id (j, i, s, r) represents the obligation of node i to deliver a message from j tagged by this id, or for node j to explicitly waive this obligation.

A (j, i, s, r) slot is kept at node i as a j entry in the R_i map, having $\mathsf{rck} = r$ and slt containing s.

Definition 2 (Envelope)**.** An envelope with id (i, j, s, r) represents the option (but not the obligation) of node i to generate a token with this same id (consuming the envelope) to which a user message is associated

An (i, j, s, r) envelope is kept at node i as a j entry in the S_i map, having $\mathsf{rck} = r$ and env containing s. For uniformity and compactness of presentation, this env field is a list of integers, when an actual implementation would need only a pair of integers, as this list always contains a contiguous sequence.

Definition 3 (Token)**.** A token with id (i, j, s, r) associated with user message m, created from an envelope of this same id, represents the obligation of node i to request j to deliver message m until acknowledged.

An (i, j, s, r) token associated with user message m is kept at node i as a j entry in the S_i map, having $\mathsf{rck} = r$ and tok mapping s to m.

2) **Messaging Steps:** We now present the four main messaging steps of Exon by referring to Algorithm 1. For simplicity, we assume the communication is occurring between a sender node "A" and a receiver node "B". To ease tracking the steps of Algorithm 1, we added Figure 1.

Step 1. Requesting Slots: Sender node A does not hold a previous S-record for receiver node B. Consequently, the former creates an S-record for B and requests "n" slots from B via REQSLOTS.

EOsend$_a$(b,m) called first at node A in order to send a payload "m" to node B. Node A instantiates an S-record (line 24) for node B since it is sending for it the first time, puts the payload "m" in a message queue (msg), and then calls the function requestSlots (line 25). Consequently, node A sends a REQSLOTS message (line 43) to node B in order to request a number of slots "n" (line 38) based on N (number of standby slots for node A at node B to satisfy N subsequent EOSends), number of messages queued, and envelopes (discussed later). The REQSLOTS message holds the variables sck (sender clock), n (number of slots), and l (the "Garbage Collection" frontier to tell node B to safely remove the old slots (line 53) which (node A) has no tokens for them).

Step 2. Sending Slots: Node B creates the requested slots and sends them back to A. If B has a prior record for A with outdated or consumed slots, it checks if garbage collection can be done.

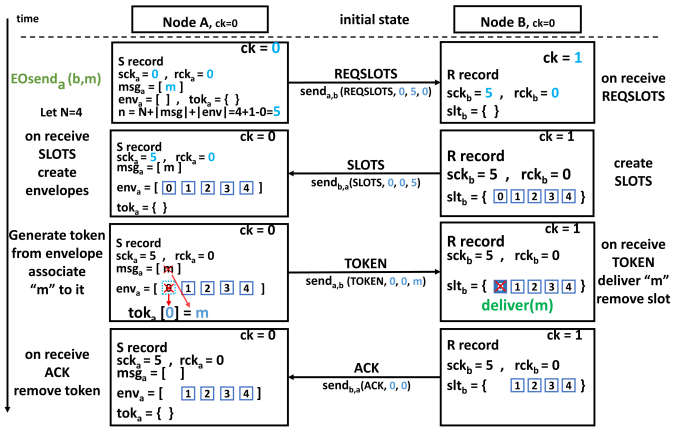


Figure 1: Exon step-by-step example for node 'a' communicating with node 'b'; window $n = 5$, $n = N+ \mid msg \mid - \mid env \mid$.

Node B then receives the REQSLOTS message (line 48), it checks if there is any receiving record (R-record) for node A from a past communication. If not, node B then instantiates the R-record of node A (line 50). In the R-record at B, sck is assigned the received clock "s" from A in order to synchronize the clocks (sck) between them. Also, rck is assigned the global clock ck as an incarnation number, and then ck is incremented.

Node B then checks if the summation of the received sck and the number of slots requested $(s + n)$ is greater than the local sck to know if the received message is an old duplicate or for garbage collection purpose, then it creates the slots (line 56) and sets the lower limit for the next range of slots to be created (line 57).

Therefore, after creating the slots, node B can send them (line 58) by sending 1) the sender clock s received from node A, where node A can assure that the requested slots received are based on its sck, 2) the incarnation number rck (node B receiver clock), and 3) the variable n that refers to the number of created slots.

Step 3. Sending Token (Holding a payload m): Node A creates a token from an envelope, associates a message m to it, and sends it.

Node A then receives the slots (line 61), actually it receives "s" (sck: the base sender clock), the rck receiver clock "r", and the number of created slots "n". Then it verifies if the received slots are based on the sender clock (sck) (line 64), in a way to avoid old slots (duplicates). Then, node A assigns the received rck "r" to the local rck in order to synchronize the incarnation clocks between the two nodes, creates the envelopes (line 67), and advances the sck by "n". After that, node A checks if there is any message in the msg queue and if there is an available envelope (line 69), in order to create a token that holds the message m and sends it to B (line 73).

When the sender runs out of envelopes and there are messages still in the queue, it requests more slots. Also, if the messages in the message queue become empty, then the "requestSlots" function will be called (line 74) in order to request slots in advance and keep N available envelopes at the sender for future EOsends.

Step 4. Payload delivery and sending ACK: Node B receives a token from node A, delivers its payload if there is a corresponding slot, and deletes the slot. Node B then sends an ACK to A.

Node B then receives the token (line 75), it receives "s" (token number), "r" (rck of node A), and the payload "m". It checks if the received token is not from an old incarnation and has the same incarnation number as the rck, and also checks if the token has a slot for it in slt. Therefore the slot can be removed from node B and the message "m" could be delivered to its destination layer successfully. After that, an "ACK" message is sent to node A (line 81) telling it to remove the token safely (line 86).

IV. Correctness Proofs

In this section, we provide a proof sketch of Exon exactly-once correctness, referring to Algorithm 1. In particular, we prove at-most-once which represents the *safety* property, and at-least-once, representing the *liveness* property. We analyze these properties under potential message failures, i.e., loss and duplication, demonstrating how Exon guarantees exactly-once despite such situations.

A. At-most-once

Exon is optimized for the at-most-once case by design. It achieves this through booking a slot for each message token to be sent. As long as this association is unique, the message will be delivered at-most-once. Consequently, the following should be proven correct: (1) Slots and tokens are unique, which requires an increasing clock across soft-connection (a.k.a., s-connection or incarnation). This is proven in Lemma 1. (2) Within an incarnation, the combination slot-envelope is unique. This means a slot is created at most once at the receiver, and the corresponding envelope at the sender is created at most once. These are proven in Lemmas 2 and 3. (3) A message payload is associated to at most one token and one envelope (Lemma 4); and a slot is consumed once by the receiver (Prop. 1).

Lemma 1. *For each node i, each receiver-side s-connection identifier* rck *is instantiated at most once, whether for the same or different senders.*

Proof. When a receiver-side's s-connection record is created (upon a first message receipt), the s-connection's identifier rck gets assigned the node's clock ck_i, that is then incremented (lines 50, 51). Since ck_i is always incremented across s-connections (line 46), then rck can be instantiated at most once. Under duplication, any subsequent (duplicate or new) message will belong to this incarnation (notice that no incarnation request exist). If the incarnation is deleted (e.g., after a long silence period), a new rck$' >$ rck is assigned to the new incarnation. □

Lemma 2. *Each slot (j, i, s, r) is created at most once.*

Proof. For each node i, from Lemma 1, slots created at i for different incarnations will have different rck values. For

each incarnation with a given r, for some sender j, each slot (j, i, s, r) is created in a range with s starting from the sck in the connection record, which is incremented to become one past the last value in the range created (lines 56, 57). This makes the subsequent creations in the same incarnation to have non-overlapping ranges and no duplicate slots are created under retransmissions. □

Lemma 3. *Each envelope (i, j, s, r) is created at most once.*

Proof. For each node i, for each incarnation with a given r, for some receiver j, each envelope (i, j, s, r) is created in a range with s starting from the sck in the connection record, which is incremented to become one past the last value in the range created (lines 67, 68). This makes the subsequent creations in the same incarnation to have non-overlapping ranges. When an incarnation terminates (e.g., not being used for long time, since Exon has no notion of closing connections), the node's clock ck_i is made to be at least as large as the sck in the incarnation being discarded (line 46), making subsequent incarnations have larger starting sck value, non-overlapping with previous incarnations to the same receiver. Therefore, no duplicate envelopes are created under retransmissions. □

Remark: different co-existing sending-side half-connection incarnations can have at some point overlapping values of sck, but as they are to different receivers, the uniqueness of each envelope created is maintained.

Lemma 4. *A message payload m to be sent to j is associated into some token (i, j, s, r) at most once.*

Proof. A message payload m is associated either immediately (lines 29-33) or later (lines 70-73) after being queued for some time to a uniquely created envelope that is immediately consumed. In addition, a token, associated to a single envelope and a message payload m, is deleted after message delivery (line 86), which makes it impossible to associate any message to that token (including the duplicated message itself). □

Proposition 1. *A sent message payload m to receiver node j is delivered at j at most once.*

Proof. From Lemmas 3 and 4, a message is associated to at most one token before it gets deleted. On the other hand, Lemmas 2 and 3 prove that at most one slot is created for an envelope (associated to a token). The proposition holds since a message is only delivered by consuming the slot corresponding to the token holding the message (lines 79, 80). A duplicate token message will not have a corresponding slot and will results in discarding it or sending an ACK if retained (line 81). □

Remark: in principle, deleting a token and delivering the message in lines 79 and 80 should be atomic. Since we do not assume node crashes, we exclude this from the algorithm.

B. At-least-once

To guarantee at-least-once message delivery, we need to prove the following: (1) A sender having a buffered message m will not give up until receiving a corresponding slot (Lemma 5). (2) A sender having received a slot will not give up retransmitting the corresponding token until the messages is delivered and ACKed (Lemma 6).

Lemma 5. *A sender node of message payload m will eventually receive a* SLOTS *message, to send a corresponding token.*

Proof. The sender having a message to send will send a REQSLOTS message to receive a SLOT. It however does not know if a REQSLOTS message is lost or delayed. Therefore, it may send the same or a new REQSLOTS message based on different cases: either some slots may have been received, or some EOSends are invoked (lines 87-93). In either case, a different slot range is calculated (line 38). Notice that the protocol will not block even if some tokens have been lost since a receiver node can still create new slots without limits. □

Lemma 6. *A receiver node of message payload m will eventually receive a* TOKEN *message delivering m.*

Proof. The sender having a message token to send will send a TOKEN message until it receives an ACK. The sender does not know whether a TOKEN or ACK message is lost. Therefore it keeps retransmitting the TOKEN message (lines 89,90) until an ACK is received. A duplicate TOKEN will have no effect because a slot has been already deleted at the receiver; however, the latter will resend the ACK (lines 78,81). A duplicate ACK will not have effect on the sender because the first ACK deletes the corresponding TOKEN at the sender and the duplicated ACKs are just discarded. □

Therefore, Lemmas 5 and 6 ensure at-least-once message delivery under network loss and duplication.

We make two final notes on correctness. The first is that Exon is not blocking if a node is communicating with many nodes since each node has its specific independent record either for sending (S record) or in receiving (R record). The second is that the algorithm does not keep any garbage metadata throughout the different phases. In particular, sending the ACK messages in the end is important not only to stop the senders retransmissions of TOKEN messages, but also to garbage collect these delivered message tokens.

V. Evaluation

A. Implementation

We implemented the Exon-lib as a Java library [22], in order to test and evaluate the algorithm under different settings and scenarios. The architecture of the Exon-lib is multi-threaded; one thread runs all the algorithm code and another thread reads from the network. Communication occurs through two main ArrayBlockingQueues: AQMsg (algoQueue), read by the main thread, for all client EOsend requests and protocol messages (received from the network and parsed and forwarded by the network reader thread); DQMsg (deliveryQueue) for delivering messages to the local client. A semaphore is used for flow

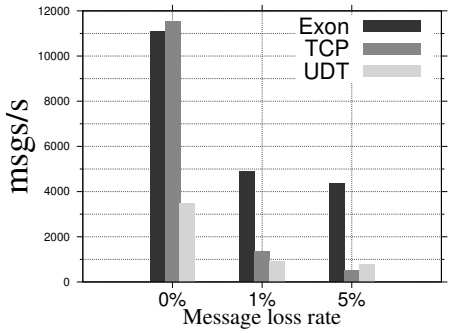


Figure 2: One-way throughput under packet loss (Setting: RTT=10ms, Bandwidth=100 Mbps)

control, blocking a client thread doing a send if many messages were previously requested to be sent but are not yet acknowledged. We currently use UDP for the entire message transport, leaving message fragmentation and merging for future work.

B. Experimental Setup

To evaluate the performance of our protocol we prepared experiments that reflect a real-world environment. For this purpose, we used Emulab [39], an online network testbed that has a wide range of environments in which researchers can develop, debug, and evaluate their systems. We used in the experiments two machines, each with a 2.4 GHz 64-bit 8-Core Xeon E5-2630v3 processors, 64 GB DDR4 RAM and 20 MB cache, and running Linux 16.04 64-bit Ubuntu OS, We configure the network as two hosts connected to a router in between. The router is configured to induce delays and message loss, mimicking a real network.

C. Evaluation Methodology

We examine two common messaging patterns: a no-wait unidirectional pattern **one-way** without replies, and **Remote Procedure Call (RPC)**, a request/reply pattern where subsequent request is sent only after the ACK of the preceding request is received.

The evaluation focuses on throughput and latency, being the most relevant metrics in messaging protocols. We send a sufficient number of messages (between 10K and 1M) per burst to stabilize the network (in addition to warming up). Finally, we do not measure the latency in the one-way message because it is less important in this pattern and it is hard to measure in the lack of perfect synchronized clocks.

We compared Exon with TCP and UDT. We chose TCP as the most widely used reliable and efficient transport protocol—discarding the inter-connection EO issues; and UDP-based Data Transfer (UDT) [16] as a reliable messaging protocol on top of UDP.

To compare the three protocols with large scale or highly concurrent systems, where each node has many objects/ processes/actors of unpredictable lifetimes, we perform RPC tests where a given number of actors perform independent (concurrent) RPCs to a server. Such systems that use TCP (e.g., distributed Erlang) normally use multiplexed TCP connections shared by many local actors. We run tests for different numbers of actors, until we saturate the bandwidth. With respect to message size, we opted for 1 KB messages, to have relatively "full" datagrams, allowing good use of bandwidth, without risking UDP fragmentation.

We run the experiments under two environments: **fault-free environment**, to test the performance of the protocols under normal circumstances, under different network latencies and bandwidths; **message loss environment**, to evaluate how well the protocols tolerate network faults. Thus, we tested the protocols under three variables: bandwidth, network latency, and message loss rate. We tried to set the parameter's values to common network cases. For bandwidth tolerance, protocols were tested under a fixed latency (RTT= 10ms) over different bandwidths: 1, 10, and 100 Mbps. For network latency tolerance, bandwidth was set to 10 Mbps, and RTT taking the values 6, 10, and 100 ms. For message loss tolerance, bandwidth and latency were set to 100 Mbps and RTT=10 ms respectively, while using message loss rates of 0%, 1%, and 5%. In each of these experiments, we tested the protocols using the two patterns described: one-way messaging and RPC. In the one-way messaging pattern, we measured the throughput (msgs/s) while in the RPC messaging pattern, we measured the throughput (requests/s) and response latency (ms).

D. Tolerance to Packet Losses

In this experiment, we aim to compare the throughput and latency of Exon, TCP, and UDT under packet loss. In all cases, no messages were lost or delivered in duplicate, despite packet loss and retransmissions. We configured Emulab to induce packet loss, dropping packets at 1% and 5% rate. To assess packet loss overhead, we also provide a baseline experiment with 0% loss. We used for network parameters RTT=10 ms, and bandwidth of 100 Mbps.

*1) **One-way messaging**:* Figure 2 shows the throughput results of the protocols under 0%, 1%, and 5% message loss. In these scenarios the sender loops sending one million 1KB messages, as fast as possible (throttled only by flow control). The throughput of Exon and TCP are very close in the 0% loss case, delivering more than 11K messages/sec (i.e., around 88Mbps), which is close to the maximum bandwidth capacity. This means that even though Exon is based on a four-way per-message exchange, it can pro-actively requests a batch of N slots in advance (where N is a parameter based on the BANDWIDTH*DELAY product), by the occasional REQ-SLOTS message, which causes negligible overhead over TCP. However, the UDT throughput is much worser than Exon and TCP. It is believed that being equipped with DAIMD has resulted in degrading the throughput [26].

Nevertheless, with a 1% and 5% packet drop rate, Figure 2 shows a significant throughput drop, i.e., around 50% in Exon, 80% in UDT, and more than 90% in TCP. This is not surprising due to the delays and network congestion caused by retransmissions of failed packets in the three protocols. The drop was however sharper in the TCP and UDT cases because of the congestion control they use. Indeed, following

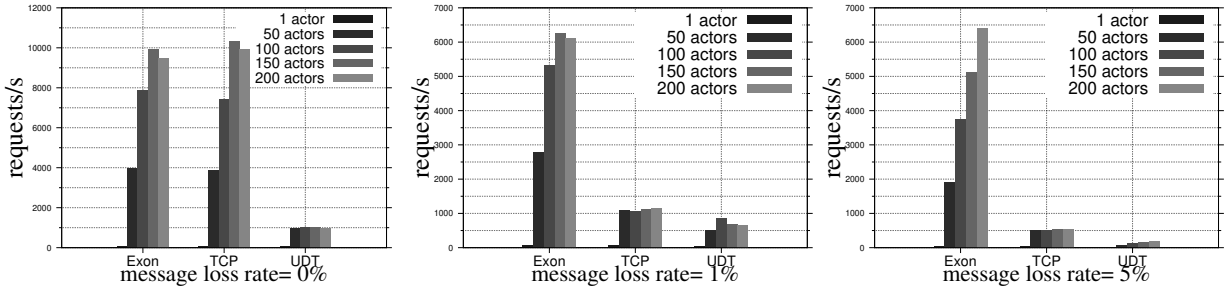


Figure 3: RPC latency under packet loss (Setting: RTT=10ms, Bandwidth=100 Mbps)

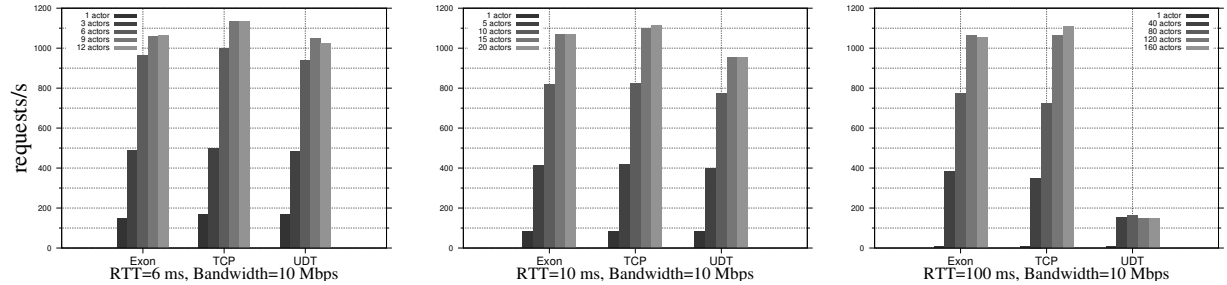


Figure 4: RPC throughput under packet loss (Setting: RTT=10ms, Bandwidth=100 Mbps)

the retransmission timeout (RTO) degradation scheme [14] causes a faster drop in throughput than Exon as shown in the 5% packet loss case. To the contrary, the retransmission timeout in Exon is less pessimistic, i.e., proportional to the network RTT, which is possible given the protocol's core resilience to dropping and duplication. Also we can see that the dropping of UDT is lighter than that of TCP since that the DAIMD in UDT does not overreact to packet losses as the AIMD in TCP. In the 5% packet loss rate, Exon's throughput is 8 times higher than TCP and UDT, which demonstrates its high resilience to hostile networks with packet losses.

*2) **RPC messaging**:* Here we test the scenario in which a host (client) contains a given number of independent actors, each actor performing RPCs to the other host (server) in a sequential loop. For TCP, a single connection is shared by all actors, as is common in real general purpose distributed actor middleware, like Erlang. We aim to see how many requests per second (in aggregate) can be achieved and RPC latency, increasing the number of actors until we saturate the bandwidth.

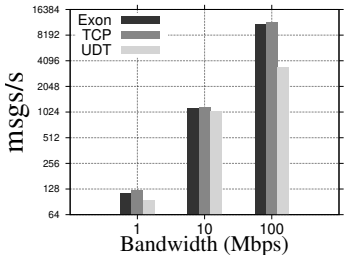
Figures 3 and 4 convey the latency and throughput results for 0%, 1%, and 5% packet loss rate experiments, in the same WAN setting as before (RTT=10ms, Bandwidth=100 Mbps). In general, the conclusions are similar to the one-way, demonstrating the high resilience of Exon to packet losses, in terms of performance, compared to TCP and UDT.

For the **fault-free** test, UDT has the worst performance compared with Exon and TCP where they have similar performance, a little worse or better depending on the number of actors. It can be seen that as we increase the number of actors, the RPC latency increases, first slowly, and then abruptly when the number of client actors (200) saturates the available bandwidth; unlike the UDT case, where the RPC latency increases rapidly as the number of actors increase. As in the one-way case, UDT has the worst performance, where that of Exon and TCP increases roughly linearly with the number of actors, until reaching network saturation, when it even decreases slightly.

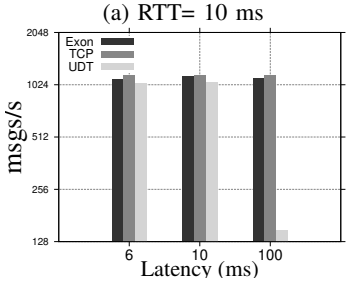
For scenarios with **packet loss**, even 1%, the performance of TCP and UDT drops drastically, compared with Exon, even more than for the one-way tests. The reason is that, while for Exon each message is delivered independently, not delaying other messages in case of packet loss, for TCP packet loss will delay the whole stream, which must be delivered in order. This means that, for TCP, for a single packet loss, all other concurrent actors will have their requests or responses delayed, increasing RPC latency, as can be seen in Figure 3, and delaying the issue of their next RPC. It can be seen in Figure 4, for both 1% and 5% packet loss cases that, while for Exon throughput still scales linearly with the number of actors (before saturating the network), for TCP and UDT throughput stops scaling much sooner. For 5% packet loss, TCP throughput stops at around 500 requests/s, UDT at around 200 requests/s, while Exon reaches around 6400 requests/s. This shows the serious impact of packet loss, due to HOL blocking, when using multiplexed TCP connections in general purpose middleware, something which unfortunately is common.

E. Overhead under Normal Conditions

In this experiment, we aim to compare the throughput and latency of Exon, TCP, and UDT under different bandwidths and latencies, in a fault-free scenario, for the two messaging patterns (one-way and RPC).

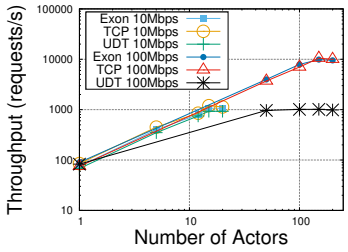


(a) RTT= 10 ms

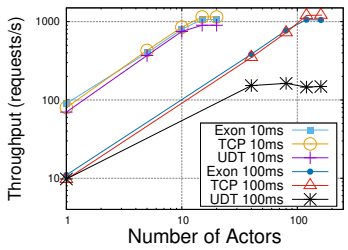


(b) Bandwidth=10Mbps

Figure 5: One-way throughput as bandwidth and RTT varies



(a) RTT=10ms



(b) Bandwidth=10Mbps

Figure 6: RPC throughput as bandwidth and RTT varies-log. scale

*1) **One-way messaging**:* Figure 5a shows the throughput of Exon, TCP, and UDT for RTT=10ms, and bandwidths of 1, 10, and 100Mbps. In the three cases, the three protocols make almost full bandwidth utilization except UDT under the 100Mbps case. This is expected since the network is saturated with the successive one-way 1KB messages, where the RTT effect is negligible. Nevertheless, the overhead in Exon is 8% in the worst case, and this is referred to the overhead of REQSLOT messages that asks for a window of N slots, as mentioned above. This small overhead justifies the use of Exon in systems that require EO guarantees and exhibit some packet loss.

Regarding UDT, its performance increases as bandwidth increases, however it does not reach the maximum bandwidth utilization with 100 Mbps, and it shows bad performance comparing with Exon and TCP.

Figure 5b shows that the full bandwidth utilization remain the same with a fixed bandwidth and different RTT (6, 10 and 100 ms) for the same reasons. However, UDT shows bad performance with high latency.

*2) **RPC messaging**:* We aim to see how many requests per second (in aggregate) can be achieved, using several concurrent actors, each performing RPCs in a loop, increasing the number of actors until we saturate the bandwidth.

Figure 6a conveys the throughput with **varying bandwidth** of 10 and 100 Mbps (with RTT=10ms). We observe a max of 7% throughput overhead of Exon over TCP in the worst case. Both protocols hit the bandwidth limits as the number of actors increase (which is higher for higher bandwidth) unlike with UDT where as the number of actors increase, the performance got worse, and this is consistent with the one-way case in sending burst of messages. The low throughput in the three protocols in the 1Mbps may look surprising at a glance; but this looks reasonable after a deeper observation since this bandwidth is not fast enough to compensate the RTT delay, which causes a buffering bottleneck quickly.

On the other hand, the throughput is negatively affected when **RTT varies** (between 10 and 100ms) as shown in Figure 6b (with fixed bandwidth=10Mbps).

In the RPC pattern, the RTT is major factor since a successor message depends on the round-trip delay of the previous one. However, as more actors are used, the channel gets more utilized and the protocols hit the bandwidth limits (10Mbps). Also, UDT has the worst performance as the number of actors increase, and this conforms with the result of the throughput experiment one-way which has a similar scenario (sending one-way burst messages).

A nice observation (in Figure 6b) is that as the RTT increases, the difference between Exon and TCP protocols almost fades away (i.e., from 10% to 2%). The same is observed in Figure 6a where the overhead degrades from 7% to 1% as the bandwidth increases.

This can be explained by the relative overhead of the REQSLOTS and SLOTS messages in these scenarios, as the algorithm sends a new REQSLOTS message upon receiving a SLOT (as long as some TOKEN messages where sent). This problem is easily fixed either by piggybacking REQSLOTS messages in TOKEN, and SLOTS in ACK, or by having a low-high watermark system for envelopes in reserve, so as to send REQSLOTS less frequently. We leave this improvement for further work.

VI. Conclusion

We presented Exon, a message-based lightweight transport-level exactly-once oblivious protocol. Exon ensures at-most-once delivery by design, by reserving placeholders at the destination before (re)transmitting the payload. Exon is oblivious as it can forget all information about other nodes, retaining

only one integer as node state when communication stops, without violating safety, not depending on timeouts for correctness. Performance is obtained by on-demand "soft-half-connections", hidden from the API, created and freed transparently, while ensuring exactly-once by preventing concurrent-close as in standard full-duplex connections.

Exon is order-less, making it more flexible as a building block for concurrent systems, avoiding the HOL blocking that occurs when using TCP, thus leaving ordering for upper layers. This makes Exon robust and performant even under packet loss. Even the current prototype already performs similarly to TCP under normal fault-free conditions and much better under even small packet loss rates. We identified a slight performance loss under low bandwidth and low network latency, for which we have already devised a solution, to be implemented for the next library version.

ACKNOWLEDGMENT

Partially funded by project AIDA – Adaptive, Intelligent and Distributed Assurance Platform (POCI-01-0247-FEDER-045907) co-financed by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalisation (COMPETE 2020) and by the Portuguese Foundation for Science and Technology (FCT) under CMU Portugal.

REFERENCES

[1] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping server-side tcp to mask connection failures. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 1, pages 329–337. IEEE, 2001.

[2] H. Attiya, S. Dolev, and J. L. Welch. Connection management without retaining information. *Information and Computation*, 123(2):155–171, 1995.

[3] H. Attiya and R. Rappoport. The level of handshake required for managing a connection. *Distributed Computing*, 11(1):41–57, 1997.

[4] B. Aziz. A formal model and analysis of the mq telemetry transport protocol. In *2014 Ninth International Conference on Availability, Reliability and Security*, pages 59–68. IEEE, 2014.

[5] G. Banavar, T. Chandra, R. Strom, and D. Sturman. A case for message oriented middleware. In *International Symposium on Distributed Computing*, pages 1–17. Springer, 1999.

[6] D. Belsnes. Single-message communication. *IEEE Transactions on Communications*, 24(2):190–194, 1976.

[7] M. Bishop. Hypertext Transfer Protocol Version 3 (HTTP/3). Internet-Draft draft-ietf-quic-http-34, Internet Engineering Task Force, Feb. 2021. Work in Progress.

[8] T. Bova and T. Krivoruchka. Reliable UDP Protocol. Internet-draft, Internet Engineering Task Force, Feb. 1999.

[9] G. Camarillo, R. Kantola, and H. Schulzrinne. Evaluation of transport protocols for the session initiation protocol. *IEEE network*, 17(5):40–46, 2003.

[10] C. Carr, S. Crocker, and V. Cerf. Host-host communication protocol in the arpa network. In *AFIPS '70 (Spring)*, 1970.

[11] V. Cerf and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on communications*, 22(5):637–648, 1974.

[12] R. Ekwall, P. Urbán, and A. Schiper. Robust tcp connections for fault tolerant computing. In *Ninth International Conference on Parallel and Distributed Systems, 2002. Proceedings.*, pages 501–508. IEEE, 2002.

[13] W.-c. Feng and P. Tinnakornsrisuphap. The failure of tcp in high-performance computational grids. In *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages 37–37. IEEE, 2000.

[14] B. Ganguly, B. Holzbauer, K. Kar, and K. Battle. Loss-tolerant tcp (lt-tcp): Implementation and experimental evaluation. In *MILCOM 2012 - 2012 IEEE Military Communications Conference*, pages 1–6, 2012.

[15] A. Gokhale and D. C. Schmidt. Principles for optimizing corba internet inter-orb protocol performance. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 7, pages 376–385. IEEE, 1998.

[16] Y. Gu and R. L. Grossman. Udt: Udp-based data transfer for high-speed wide area networks. *Computer Networks*, 51(7):1777–1799, 2007.

[17] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[18] E. He, J. Leigh, O. Yu, and T. A. DeFanti. Reliable blast udp: Predictable high performance bulk data transfer. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 317–324. IEEE, 2002.

[19] P. Helland. Life beyond distributed transactions: an apostate's opinion. *Queue*, 14(5):69–98, 2016.

[20] N. Ivaki, F. Araujo, and R. Barbosa. A middleware for exactly-once semantics in request-response interactions. In *2012 IEEE 18th Pacific Rim International Symposium on Dependable Computing*, pages 31–40. IEEE, 2012.

[21] C. Jin, D. X. Wei, and S. H. Low. Fast tcp: motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, volume 4, pages 2490–2501. IEEE, 2004.

[22] Z. Kassam, P. S. Almeida, and A. Shoker. Exon Exactly-Once Oblivious Messaging Library. https://github.com/ziadkassam/Exon. Accessed: 2021-08-09.

[23] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[24] I. Lightbend. Apache akka: Message delivery reliability. https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html. Accessed: 2021-08-09.

[25] A. Lindberg, S. Merle, and P. Stritzinger. Scaling erlang distribution: going beyond the fully connected mesh. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang*, pages 48–55, 2019.

[26] M. Masirap, M. H. Amaran, Y. M. Yussoff, R. Ab Rahman, and H. Hashim. Evaluation of reliable udp-based transport protocols for internet of things (iot). In *2016 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*, pages 200–205. IEEE, 2016.

[27] J. Postel. Rfc0793: Transmission control protocol, 1981.

[28] J. Postel et al. User datagram protocol, 1980.

[29] RabbitMQ. https://www.rabbitmq.com/.

[30] L. Salzman. ENet v1.3.17: Reliable UDP networking library. http://enet.bespin.org/Features.html. Accessed: 2021-08-09.

[31] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, et al. Rtp: A transport protocol for real-time applications, 1996.

[32] A. U. Shankar and D. Lee. Modulo-n incarnation numbers for cache-based transport protocols. In *1993 International Conference on Network Protocols*, pages 46–54. IEEE, 1993.

[33] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *USITS*, volume 1, pages 19–19, 2001.

[34] B. Snyder, D. Bosanac, and R. Davies. Introduction to apache activemq. *Active MQ in action*, pages 6–16, 2017.

[35] R. Stewart and C. Metz. Sctp: new transport protocol for tcp/ip. *IEEE Internet Computing*, 5(6):64–69, 2001.

[36] R. R. Stewart. Stream Control Transmission Protocol. RFC 4960, Sept. 2007.

[37] C. Sunshine and Y. K. Dalal. Connection management in transport protocols. *Comput. Networks*, 2:454–473, 1978.

[38] R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG*. Prentice Hall International (UK) Ltd., 1996.

[39] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270, 2002.

[40] V. C. Zandy and B. P. Miller. Reliable network connections. In *Proceedings of the 8th annual International Conference on Mobile Computing and Networking*, pages 95–106, 2002.

[41] M. Zhang, B. Karp, S. Floyd, and L. Peterson. Rr-tcp: a reordering-robust tcp with dsack. In *11th IEEE International Conference on Network Protocols, 2003. Proceedings.*, pages 95–106. IEEE, 2003.