Non-Blocking Raft for High Throughput IoT Data

1st Tian Jiang
Tsinghua University
jiangtia18@mails.tsinghua.edu.cn

2nd Xiangdong Huang
Tsinghua University
huangxdong@tsinghua.edu.cn

3rd Shaoxu Song
Tsinghua University
sxsong@tsinghua.edu.cn

4th Chen Wang
Tsinghua University
wang_chen@tsinghua.edu.cn

5th Jianmin Wang
Tsinghua University
jimwang@tsinghua.edu.cn

6th Ruibo Li
Alibaba Group
ruibo.lirb@alibaba-inc.com

7th Jincheng Sun
Alibaba Group
jincheng@apache.org

***Abstract*—The Raft consensus protocol naturally fits time series databases, owing to the resemblance between its continuous log and the time series data. While the serialization of appending entries reduces the state space for ease design and implementation, it blocks the subsequent requests and thus limits the parallelism and throughput of Raft. Intuitively, once an entry arrives the follower, we may notice the leader and the client to unblock the subsequent as early, rather than waiting for its appending and committing. In this way, more requests can be processed in parallel, and thus the throughput increases, essential for IoT applications often with vast sensors and fast data ingestion. Of course, with higher parallelism, the risk of persistence for in-processing entries increases. It is a worthwhile trade-off in the IoT scenario since tiny data loss during leader failure is more acceptable than shutting out most data due to a low throughput. Our *Non-Blocking Raft* (NB-Raft) is implemented as the consensus protocol of Apache IoTDB, a commodity time series database management system, supporting various applications in Alibaba Cloud. Extensive evaluation shows that the throughput is improved by about 30% using our NB-Raft compared to the original Raft, a considerable amount of further data saved.**

***Index Terms*—distributed system, IoT, consensus protocol**

I. Introduction

Consensus protocols are essential in efficiently keeping replicas consistent in distributed systems. While there are alternatives, such as the Paxos family [1], [2], [3], [4], [5], [6], [7], we choose to employ Raft [8] for Apache IoTDB [9], a time series database for Internet of Things (IoT). The reason is that Raft models data as a continuous log without holes. It makes the state space smaller than other protocols, and thus easier to understand and maintain. Most importantly, the continuous log naturally resembles time series, where each log entry with a monotonically increasing index corresponds to a series point ordered by timestamp.

Unfortunately, the log serialization scheme of Raft limits its parallelism and consequently the throughput (see the motivation example below). A typical IoT application manages thousands of devices, samples millions of time series, and generates tens of GBs data every second [9], [10]. Directly applying Raft to IoT scenarios pushes the protocol throughput to its limits, and thus potentially shut out data from the system.

Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

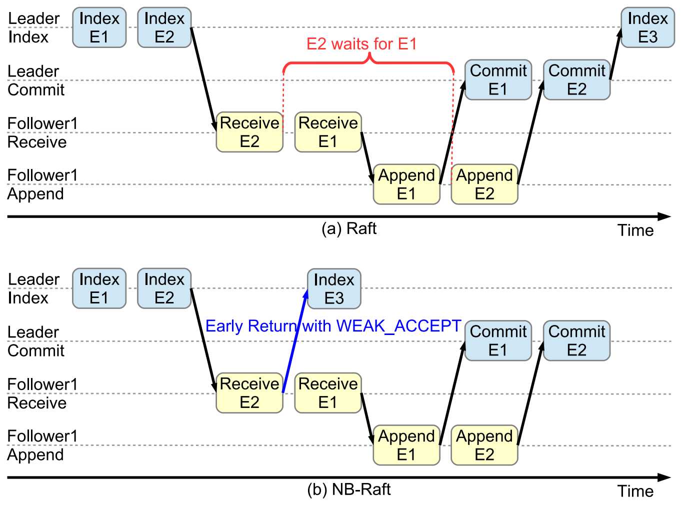

Fig. 1. Raft in (a) blocks the subsequent entry E3 till the commit of the preceding E2 from the same client, while our NB-Raft in (b) introduces an early return to unblock E3 as soon as possible and thus increases throughput.

A. Motivation

Raft introduces serialization by assigning two unique and monotonically increasing numbers, log *index* and *term*, to each command. Followers must append and apply commands in the order of index. A term identifies uniquely the generator of a log entry. Figure 1(a) briefly shows the process of replicating entries from the leader to a follower. The leader first indexes a request as a log entry, e.g., E1, and sends it to the follower. Then, the follower appends the entry locally. After that, the leader commits the entry (assuming only one follower).

When an entry reaches the follower, some previous entries may still be appending or even not received yet. It thus cannot be appended and must wait till the previous entry is appended. For example, there is a long gap between “Receive E2” and “Append E2” in Figure 1(a), waiting for “Append E1”. As a client connection can issue only one request a time, the subsequent requests are also blocked. For instance, E3 in Figure 1(a) cannot be issued until the previous E2 from the same client is committed.

In short, owing to the serialization scheme of Raft log, the requests of a client are blocked and cannot be processed in parallel. The out-of-order arrivals of entries make the blocking time even more serious and severely limit the throughput.

B. Intuition

Note that the availability of commodity distributed systems is high, for example, Alibaba Cloud provides servers with availability over 99.99%. In most cases, the delayed entry will arrive later, and the blocked entry will eventually be appended. Intuitively, once an entry arrives at enough followers (a quorum), rather than waiting for other entries, we may first notify the client so that it can issue the next request. For example, in Figure 1(b), E2 is cached after "Receive E2". Then, a special result (WEAK_ACCEPT as introduced below) is returned and E3 is issued earlier than that in Figure 1(a). The latter phases (append and commit) of E2 are batched with the associated phases of E1. In this way, the parallelism increases, i.e., leading to better throughput, highly demanded in the IoT applications as previously mentioned.

Of course, since more entries are processed in parallel, the risk in persistence increases. For the original Raft in Figure 1(a), if the leader crashes before entering the "Commit E1" phase, then entries E1 and E2 could be lost in the worst case. However, in Figure 1(b), E3 may also be lost since it is already initiated at this point. Nevertheless, it is the price to pay for higher parallelism and throughput, i.e., a worthwhile trade-off. A detailed discussion is given in Section V-G.

C. Contribution

In this paper, to increase parallelism and throughput of Raft, we propose a novel adaption, *Non-Blocking Raft* (NB-Raft). By introducing a new intermediate state WEAK_ACCEPT for entries, the previously mentioned "Early Return" is enabled. Rather than blocking the subsequent entries, multiple entries from the same client can thus be processed in parallel and cached in a *window* of the follower, waiting for appending.

Our major contributions are as follows.

(1) We present a thorough analysis of the log replication process in Raft with the help of Petri Net, a powerful tool for modeling distributed systems with concurrency [11]. It enables us to quantitatively identify the performance bottleneck in the process, due to the previously mentioned log appending serialization. To the best of our knowledge, this is the first study on modeling Raft log replication by Petri Net for efficiency analysis.

(2) We modify the Raft consensus protocol, in client, leader and follower, to enable WEAK_ACCEPT for higher parallelism and throughput. A sliding window is designed in the follower to cache and control the number of entries in the middle state, waiting for log appending. Since multiple requests could be sent by in parallel, clients also need to maintain these in-processing requests, whose number is bounded by the window size.

(3) We show that the original Raft protocol is indeed a special case of our NB-Raft with window size zero. It means no entry having WEAK_ACCEPT in cache waiting for

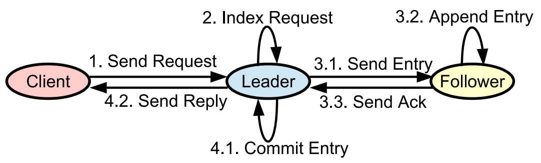

Fig. 2. Interactions among client, leader, and follower in log replication.

Append. We also discuss the trade-off between throughput and persistence.

(4) We conduct extensive experiments in real systems deployed in both local network and cloud. The results illustrate the bottleneck of the Raft throughput. NB-Raft achieves about 30% improvement in throughput compared to the original Raft.

Remarkably, the proposed protocol NB-Raft has been deployed in products and industry. It is integrated into a commodity time series DBMS, Apache IoTDB [12]. The code of NB-Raft is available in the official GitHub repository of Apache IoTDB [13]. The system with NB-Raft is deployed in centralized or geo-distributed clusters in Alibaba Cloud for various applications, from workshop monitoring to central control systems, to satisfy the need of massive time series ingestion. The customers include China Railway Rolling Stock Corporation (CRRC), China Tobacco, China National Nuclear Corporation (CNNC), Tianyuan Technology group, etc.

The remaining of the paper is organized as follows. We model the log replication procedure of Raft using Petri Net and identify its bottleneck in Section II. Section III presents NB-Raft, the proposed consensus protocol. Section IV discuss the trade-off between throughput and persistence. In Section V, evaluation of the original protocol and the modified version is conducted. Section VI discusses related work, and finally Section VII concludes the paper.

II. Log Replication in Raft

Log replication is the key point of consensus efficiency in Raft. Requests are converted to indexed log entries and sent to any majority of replicas. We found the throughput of the original Raft unsatisfying in some of our application cases. Inspired by [14], we use Petri Net [11] to model the consensus protocol and identify its performance bottleneck. Specifically, log replication is modeled as an extended producer-consumer model using Petri Net. It is intended to (1) present the log replication process in a more informative way, (2) characterize and accurately measure the bottleneck, (3) demonstrate how the modified protocol works intuitively.

A. Replication Process

To explain thoroughly the log replication process, we further present the interactions among client, leader and follower in Figure 2. It is an overview of the detailed yet complicated process in Petri Net in Figure 3.

In short, log replication is in four steps: sending requests from a client to the leader in Figure 3(a)-(b), converting requests to indexed log entries in the right part of Figure

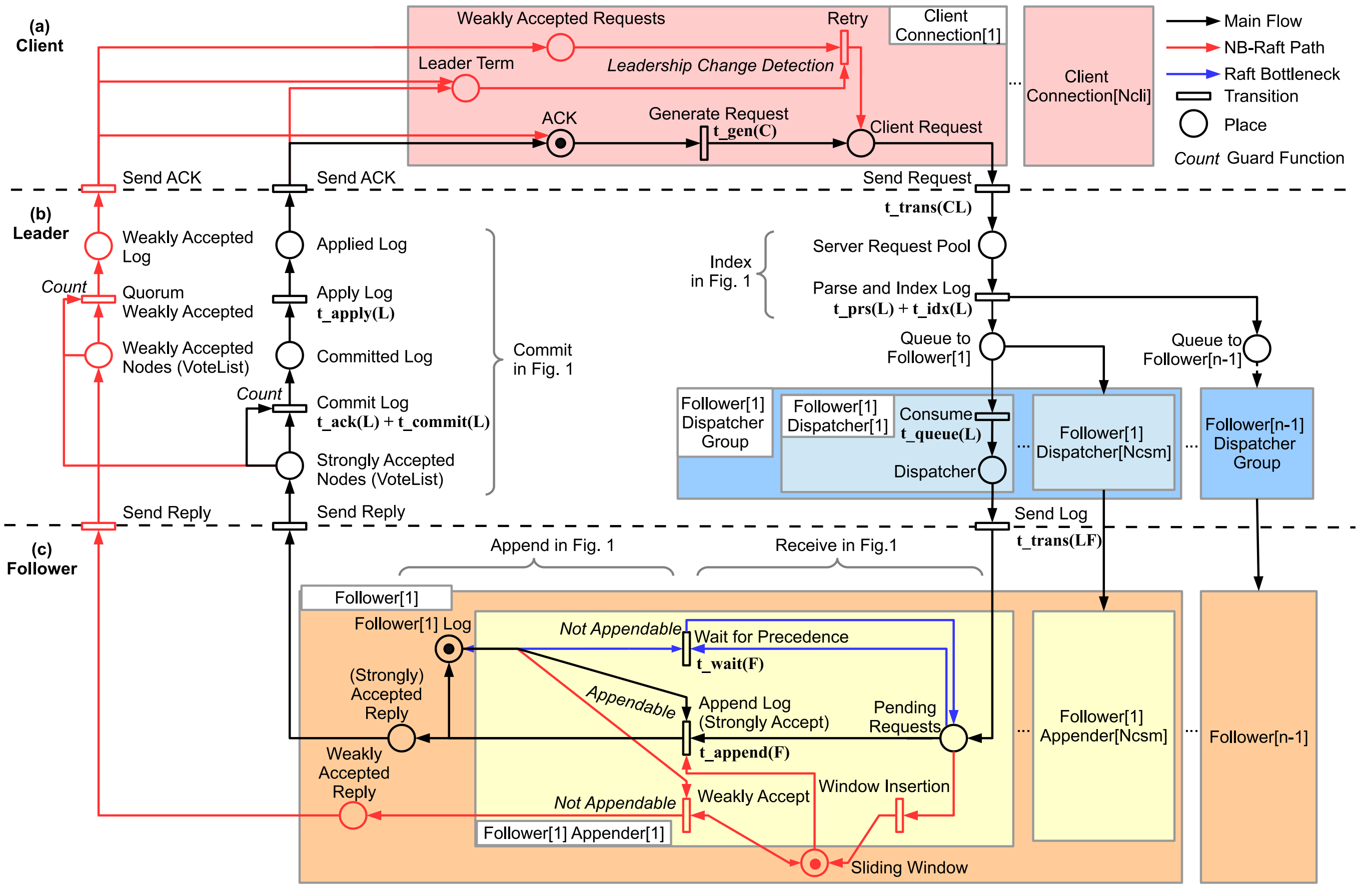

Fig. 3. A detailed log replication process of Figure 2. Black lines denote the original protocol, while red lines present the modifications. The identified bottleneck is marked with blue lines, i.e., a loop enabled when an entry is not appendable and thus waiting. Guard funtions (transition triggering condition) are italic. The parts corresponding to the index, receive, append and commit phases in Figure 1 are also indicated.

3(b), replicating the entries to followers in Figure 3(b)-(c), and committing the log in the left part of Figure 3(b).

To precisely measure the time costs of different steps and identify the bottleneck, we use a three-replica cluster of Apache IoTDB [9] (server specification is given in Section V-A). The cluster is evaluated with TPCx-IoT benchmark [15], using its default settings. By code instrumenting and log profiling, we collected time consumption during log replication, presented in Figure 4.

B. Step 1 Sending Request

As shown in Figure 3(a), there are N_{cli} client connections. Each can generate a request once the leader replies to the previous one, i.e., a token in the place (circle) of *ACK* in Petri Net. The transition *Generate Request*, denoted by rectangle in Petri Net, puts it in the place of *Client Request*. The request is then sent to *Server Request Pool* in the leader via network. This pool is shared among all clients.

The cost of step 1 is mainly in two parts, the time to generate a request in Client $t_{\text{gen}}(C)$ and the time to transmit a request from Client to the Leader $t_{\text{trans}}(CL)$. Given network latency $t_{\text{lat}}(CL)$, request size b, and network bandwidth $w_{\text{net}}(CL)$, the cost $t_{\text{trans}}(CL)$ can be represented as $t_{\text{trans}}(CL) = t_{\text{lat}}(CL) + b/(w_{\text{net}}(CL)/N_{\text{cli}})$, where the bandwidth is shared by all clients. The total cost of step 1 is $t_1 = t_{\text{gen}}(C) + t_{\text{lat}}(CL) + b/(w_{\text{net}}(CL)/N_{\text{cli}})$.

C. Step 2 Indexing Entry

The right part of Figure 3(b) describes the second step, corresponding to "Index" in Figure 1 and step 2 in Figure 2. When the leader receives a request, it is assigned with a unique index and appended locally in the leader. For each follower i, the indexed entry is inserted into *Queue to Follower[i]* to wait for transmission.

Let $t_{\text{prs}}(L)$ and $t_{\text{idx}}(L)$ be the time to parse a request and index an entry. Indexing an entry includes assigning a unique number and appending the entry to the leader's log. The request waits in the queue till it is consumed and sent to a follower. Each queue of entries is consumed by N_{csm} dispatchers, sending entries to the associated follower in parallel. Let $t_{\text{queue}}(L)$ denote the queue time. The total cost of step 2 is $t_2 = t_{\text{prs}}(L) + t_{\text{idx}}(L) + t_{\text{queue}}(L)$.

D. Step 3 Replicating Entry

The step of replicating entry is shown in Figure 3(c). For each dispatcher, there is an appender thread retrieving entries in *Pending Request*. By checking *Follower[i] Log*, an appender decides whether an entry is appendable according to the existence of its precedence. If appendable, it appends the entry to the log and sends an acknowledgment to the leader's *(Strongly) Accepted Nodes*.

The cost of step 3 includes the time of sending an entry to a follower $t_{\text{trans}}(LF)$ via network similar to step 1, waiting for precedent entries $t_{\text{wait}}(F)$, appending an entry $t_{\text{append}}(F)$. As each follower has a different $t_{\text{wait}}(F)$, to reduce the number of variables, $t_{\text{wait}}(F)$ here refers to that of the first follower which receives the entry. The total cost of step 3 is $t_3 = t_{\text{trans}}(LF) + t_{\text{wait}}(F) + t_{\text{append}}(F)$.

Via network, $t_{\text{trans}}(LF)$ is like $t_{\text{trans}}(CL)$. Moreover, $t_{\text{append}}(F)$ is efficient, only taking 0.1% to finish in Figure 4. The protocol-related bottleneck is thus identified as $t_{\text{wait}}(F)$ in step 3, as shown in Figure 3(c). Scheduling and fluctuating delays of connections introduce indetermination, and thus entries can no longer reach a follower in order. The follower must wait for previous entries to keep log continuity. Marked with blue lines in Figure 3(c), there is a loop when entries are not appendable, introducing $t_{\text{wait}}(F)$, the second largest part in Figure 4. $t_{\text{wait}}(F)$ is affected by concurrency, request size, network and scheduling. Notice that from the queuing theory, $avg\_queue\_time \approx avg\_process\_time * avg\_queue\_length$. Therefore, reducing $t_{\text{wait}}(F)$ also reduces $t_{\text{queue}}(L)$.

In the remaining of Figure 3, places are always single-directional, which means a token will not be blocked in places by a loop. Such an anomaly revealed by Petri Net suggests that this part deserves more attention. Moreover, the Petri Net's ability to present concurrency shows the connection between multi-threading and $t_{\text{wait}}(F)$. The loop is controlled by Follower's Log, which is accessed by multiple appenders.

E. Step 4 Committing Entry

As shown in the left part of Figure 3(b), if acks are collected from any quorum, the entry is committed by the leader and inserted into *Committed Log*. Then, it will be moved into *Applied Log* after being applied. When an entry is applied, an ack is sent to the *ACK* of the corresponding client in step 1 to enable subsequent requests.

The cost of step 4 includes collecting enough acks, i.e., from the first follower appending an entry to a quorum appending the entry $t_{\text{ack}}(L)$, committing, and applying an entry, $t_{\text{commit}}(L)$ and $t_{\text{apply}}(L)$. The total cost of step 4 is $t_4 = t_{\text{ack}}(L) + t_{\text{commit}}(L) + t_{\text{apply}}(L)$.

F. Observation in Other System

We observe the measurements in another platform Apache Ratis [16] and its embedding benchmark, in Figure 4. Apache Ratis is a highly customizable Raft protocol implementation in Java [16]. We implement multiple log appenders (dispatchers) in the system for evaluating high throughput IoT data.

As shown in Figure 4, Ratis has a higher $t_{\text{index}}(L)$, since it uses heavier lock for synchronization during indexing than IoTDB. Therefore, its $t_{\text{queue}}(L)$ is partially moved into $t_{\text{index}}(L)$. Unlike Ratis FileStore that triggers an I/O operation

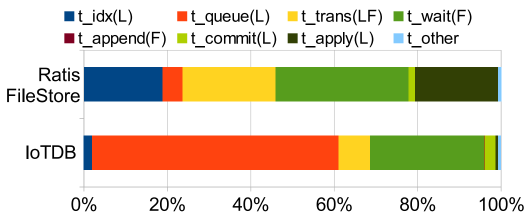

Fig. 4. Proportions of various time during log replication.

TABLE I
NOTATIONS

Notation	Explanation	Bottleneck
$t_{\text{gen}}(C)$	Time to generate a request by a client	IoT device sampling frequency
$t_{\text{trans}}(CL)$	Time to send an entry from the client to the leader	Network bandwidth
$t_{\text{prs}}(L)$	Time to convert a binary string into a meaningful request	Memory allocation
$t_{\text{idx}}(L)$	Time to assign a term and an index to an entry by the leader	CPU for highly concurrent requests
$t_{\text{queue}}(L)$	Time after being indexed and before being sent to a follower	CPU and Network bandwidth
$t_{\text{trans}}(LF)$	Time to send an entry from the leader to a follower	Network bandwidth
$t_{\text{wait}}(F)$	Time from receiving an entry to being appendable in a follower	Out-of-order by high concurrency
$t_{\text{append}}(F)$	Time to append an entry in a follower	CPU
$t_{\text{ack}}(L)$	Time to collect responses for an entry	No clear bottleneck
$t_{\text{commit}}(L)$	Time to mark an entry as committed by the leader	CPU
$t_{\text{apply}}(L)$	Time to execute the command in an entry	Determined by application

with each request, IoTDB batches data in memory and flushes latter, having a smaller $t_{\text{apply}}(L)$.

Nevertheless, both systems incur large $t_{\text{wait}}(F)$, the waiting time among concurrent requests to become appendable. It is introduced by the out-of-order entry arrivals. The consistently observed high cost of $t_{\text{wait}}(F)$ verifies again the necessity of reducing the waiting time, i.e., the focus of this study.

G. Summary of Observations

Table I presents a summary of assumptions and facts, stated in multiple places in this section. For instance, $t_{\text{idx}}(L)$ is the time to assign a term and an index to an entry by the leader, which is limited by the CPU resource for handling the highly concurrent requests and thus difficult to reduce. In particular, $t_{\text{wait}}(F)$ is identified as the time from receiving an entry to being appendable in a follower. It is introduced owing to the out-of-order entry arrivals in high concurrency and is expected to be reduced in this study.

III. Non-Blocking Raft

To avoid blocking, we introduce a new state called WEAK_ACCEPT for entries that are received by not appendable yet. It notifies the leader as early as entries in the state. To distinguish, we call the appended entries STRONG_ACCEPT.

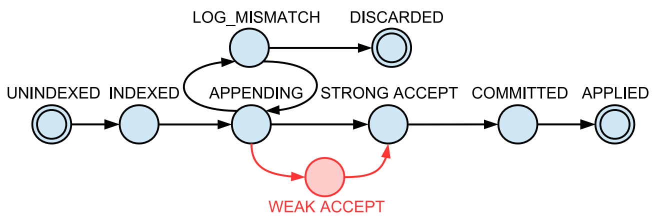

Fig. 5. State transition of a request in Raft.

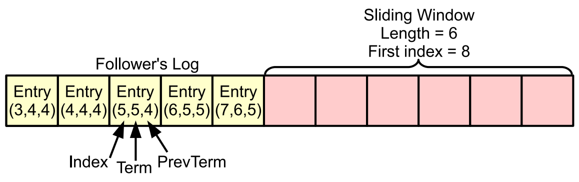

Fig. 6. An empty log window of six positions reserved for caching entries.

The overall states and their transitions are presented in Figure 5. In the following, we describe the specific modifications made to follower, leader, and client to manage the new state WEAK_ACCEPT. The corresponding protocol modifications for unblocking are also shown in Figure 3 in red.

A. Follower Modification

To cache unappended logs, we introduce a sliding window for each follower, analogous to Sliding Window in Figure 3(c). If the index of the follower's last appended entry is i, then window position j should store the entry with index $i + j$. Non-appendable entries will enter the window and return a WEAK_ACCEPT, in red lines in Figure 3(c).

The window size w is configurable according to the parallelism requirement, i.e., the maximum number of entries that can be processed in parallel. Entries exceeding the window size will be blocked till they can be inserted. The larger the window size is, the higher the parallelism will be. In contrast, for a small window size, requests are more likely to be blocked. In this sense, the original Raft can be viewed as a special case where the window size is zero, i.e., always blocking subsequent requests.

Figure 6 shows a follower with five appended entries and an empty log window. The first number in each entry is its index, the second one is its term, and the last one is its previous term (term of the previous entry). The previous term is used to check if the previous entry is generated by the consistent leader.

The length of the window is six, and its first index is 8 (as the last entry's index is 7).

For a new entry arriving at a follower, let *diff* be the index difference between the new entry and the last entry in the follower's log. There are 3 cases to consider according to *diff*.

1) New entry before the window: If *diff* ≤ 0, like the original Raft, the new entry should replace an already appended entry and truncate the following entries. It occurs when a new

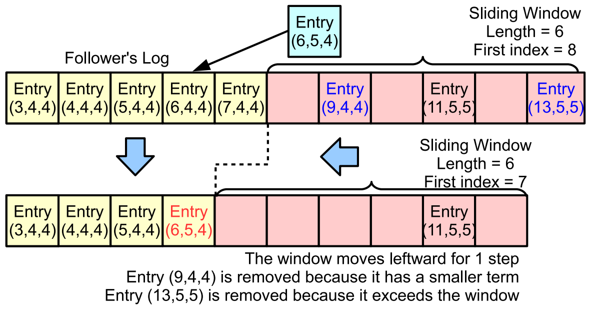

Fig. 7. The window moves leftwards as the appended log is modified, for the case in Section III-A1.

leader generates logs with the same index but a higher term. The uncommitted logs from the old leader should be replaced.

If the new entry finds the previously appended entry, the follower moves the window leftward *diff* positions, and discards any entries that are no longer in the window or have a term less than the new entry. It then reports (STRONG_ACCEPT, new entry index, new entry term) to the leader. As messages could be delayed in an asynchronous system, a follower may receive entries from old leaders, which will be discarded by this step. Otherwise, the entry mismatches. The follower discards it and reports LOG_MISMATCH to the leader as Raft does.

In Figure 7, *diff* is $6 - 7 = -1$ for the new Entry (6,5,4), and thus the appended log is truncated. As the last index in the local log becomes smaller, the sliding window moves leftwards. Entry (9,4,4) is removed for its term smaller than five. Entry (13,5,5) is discarded as it exceeds the window. The follower returns (STRONG_ACCEPT, 6, 5) to the leader, indicating the entry is appended and becomes its last entry.

2) New entry falling into the window: If $0 <$ *diff* $< w$, the new entry (i,j,k) is inserted into position *diff* of the window. According to Raft, the term of the entry with index $i - 1$ is also sent to the follower together with entry (i,j,k), namely k. We call an entry of the follower the *previous entry* of entry (i,j,k), if its index is $i - 1$ and its term is k.

a) Ensure continuity: By checking the predecessor and successor of the inserting entry (i,j,k), continuity is ensured.

If its predecessor at position *diff* $- 1$ is not null, and not the previous entry of the inserting entry (i,j,k), then the predecessor is not continuous with entry (i,j,k). We remove the predecessor at position *diff* $- 1$ to ensure log continuity.

If the successor at position *diff* $+ 1$ is not null, and the new entry (i,j,k) is not its previous entry, then the successor is not continuous with entry (i,j,k). We remove the successor and all entries following it to ensure log continuity.

Figure 8 shows an example of inserting Entry (11,7,6). Entry (10,5,4) is thus removed because it is not the previous entry of Entry (11,7,6). Moreover, Entry (12,5,5) and Entry (13,5,5) are removed, because the *term* of them cannot be 5 (terms are non-decreasing according to Raft). That is, Entry (11,7,6) is not the previous entry of Entry (12,5,5) and Entry (13,5,5).

b) Determine state: Upon the inserted position and the previous entries, the follower determines the state to return,

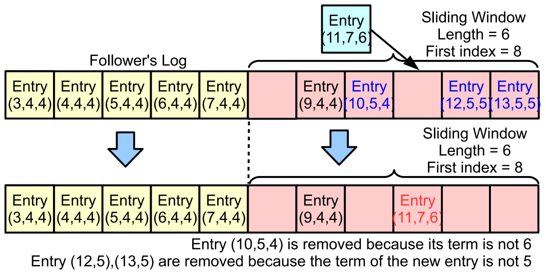
Entry (11,7,6)
Sliding Window Length = 6 First index = 8
Follower's Log
Entry (3,4,4)
Entry (4,4,4)
Entry (5,4,4)
Entry (6,4,4)
Entry (7,4,4)
Entry (9,4,4)
Entry (10,5,4)
Entry (12,5,5)
Entry (13,5,5)
Sliding Window Length = 6 First index = 8
Entry (11,7,6)
Entry (10,5,4) is removed because its term is not 6
Entry (12,5),(13,5) are removed because the term of the new entry is not 5

Fig. 8. After insertion Entry (11,7,6), mismatched entries (in blue) are removed, as introduced in Section III-A2a.

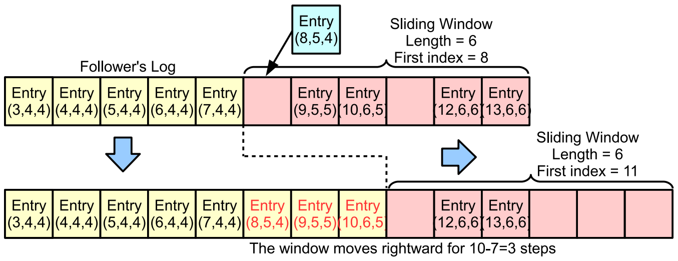

Fig. 9. By inserting in the first position of the window, the non-empty prefix will be appended to the local log, as introduced in Section III-A2b.

STRONG_ACCEPT, LOG_MISMATCH, WEAK_ACCEPT.

If *diff* $= 1$ and the previous entry of the new entry is the last entry in the local log, then the follower moves the non-null prefix of the window to the tail of the local log (namely flush). The reason is that they are now continuous with the local log. Accordingly, the follower moves the window rightward, and reports (STRONG_ACCEPT, last entry index, last entry term) to the leader. The step moves entries from the window to the local log, so that they can be latter committed.

If *diff* $= 1$ and the previous entry of the new entry is not the last entry in the local log, then the follower reports a LOG_MISMATCH to the leader. It informs the leader that entries with smaller indices should be sent.

Otherwise, *diff* $\neq 1$ and the local log is unchanged, it reports a WEAK_ACCEPT to the leader.

As illustrated in Figure 9, when Entry (8,5,4) is inserted to the first position of the sliding window, Entries (8,5,4), (9,5,5), (10,6,5) are moved from the window to the local log. A message (STRONG_ACCEPT, 10, 6) will be returned to the leader as the last entry is (10,6,5), and the leader will know that entries up to index 10 have been appended.

3) New entry after the window: If *diff* $\geq w$, the follower will wait and retry as if the entry is newly arrived. If another entry arrives during the wait and triggers a flush (Section III-A2b), the window will move rightward. This makes room for the waiting entry if enough entries are flushed. It thus controls the length of the window and avoids re-transmission.

B. Leader Modification

In the modification of follower, there are two types of acceptances, STRONG_ACCEPT and WEAK_ACCEPT, denoted

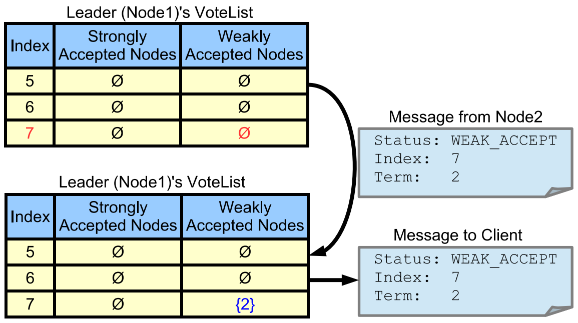

Fig. 10. For Section III-B2, when Node2 returns WEAK_ACCEPT to the leader after it receives an out-of-order Entry 7, the leader records it in the corresponding tuple (having the same index) in *VoteList* and replies WEAK_ACCEPT to the client.

by black and red lines in Figure 3(c). STRONG_ACCEPT equals a vote in the original Raft and can be counted to commit an entry. WEAK_ACCEPT only indicates the reception of entries instead of being appended. The leader modification lies in two aspects, devising a data structure to manage entry states and processing different signals accordingly.

To manage entry states, a leader maintains an ordered list *VoteList* of tuples on (logIndex, Weakly Accepted Nodes, Strongly Accepted Nodes) to track what nodes have appended the entry with logIndex and what nodes have only received it. *VoteList* is used in two places in Figure 3(b), i.e., *Weakly Accepted Nodes* and *Strongly Accepted Nodes*. The original Raft tracks the state of entries similarly, but with only Strongly Accepted Nodes.

During log replication, a leader sends an entry *currLog* to followers in parallel and adds a tuple *tp* (*currLog*'s index, $\emptyset$, $\emptyset$) to *VoteList*. According to responses from followers, the leader processes differently as follows.

1) Case of LOG_MISMATCH: In Section III-A2b, a follower may return LOG_MISMATCH, suggesting missing entries. The leader finds what entries the follower is missing and re-sends them as the original Raft.

2) Case of WEAK_ACCEPT: If a follower f returns a WEAK_ACCEPT, the leader adds f to *tp*'s Weakly Accepted Nodes, where *tp* is the tuple in *VoteList* having the same index as the currently processed entry. Moreover, if *tp*'s Weakly Accepted Nodes together with *tp*'s Strongly Accepted Nodes form a majority, the leader returns (WEAK_ACCEPT, *currLog*'s index, *currLog*'s term) to the client. It indicates that a living quorum has received the entry, which is highly likely to be committed eventually. Thereby, the response can be returned to enable the next request.

Figure 10 gives an example of WEAK_ACCEPT. When the leader Node1 receives (WEAK_ACCEPT, 7, 2) from a follower Node2, it inserts 2 (the follower's node ID) into Weakly Accepted Nodes of entry 7. As the leader itself is a strongly accepted node, the union of Weakly Accepted Nodes and Strongly Accepted Nodes forms a majority (for three replicas). It sends (WEAK_ACCEPT, 7, 2) to the client.

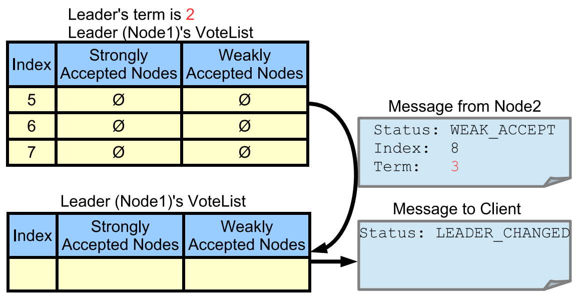

Fig. 11. When Node2 returns STRONG_ACCEPT with a higher term, the leader cleans *VoteList* and replies LEADER_CHANGED to the client. This example is associated with Section III-B3a.

3) Case of STRONG_ACCEPT: If a follower f returns (STRONG_ACCEPT, $lastIndex$, $lastTerm$), in Section III-A2b, there are two cases to consider for the leader.

a) Leader changed: If $lastTerm$ is not the current term, it indicates that a new leader is elected. The leader returns LEADER_CHANGED to the client (for retrying the request in the new leader) and cleans $VoteList$ for garbage collection.

Figure 11 shows the case of a leadership change. When a leader receives a reply with higher term ($3 > 2$), it means that a new leader of term 3 is elected and this leader is invalid. The leader cleans $VoteList$ as only a valid leader uses it. Then a LEADER_CHANGED signal is returned to the client, who can retry with the new leader.

b) Leader valid: Otherwise, $lastTerm$ is the current term, i.e., the leader is still valid. Thereby, for each tuple in $VoteList$, if the tuple's logIndex is smaller than or equals to $lastIndex$, the leader adds f to its Strongly Accepted Nodes. Our window insertion checks in Section III-A2a assures log continuity. If an entry with index i becomes STRONG_ACCEPT, all its previous entries are STRONG_ACCEPT. This is different from WEAK_ACCEPT in Section III-B2, which only modifies one tuple.

If the tuple's Strongly Accepted Nodes form a quorum, the leader commits and removes the tuple since other votes no longer matter. After examining all tuples, if any entry is committed, it returns (STRONG_ACCEPT, last committed entry's index, last committed entry's term) to the client.

Figure 12 illustrates an example case of non-changed leader. When a leader receives (STRONG_ACCEPT, 5, 2) from a follower Node2, it knows entries with index ≤ 5 are also appended by Node2. The follower's node ID 2 is inserted to the Strongly Accepted Nodes of the corresponding entries. Finally, the leader sends (STRONG_ACCEPT, 5, 2) to the client.

C. Client Modification

To track weakly accepted requests, a client maintains a list of tuples on (logIndex, logTerm, request) named $opList$, and a term $listTerm$, which indicates the newest leader known by the client. The $opList$ contains requests replied with WEAK_ACCEPT. Since a new leader may overwrite WEAK_ACCEPT entries of the previous leaders, $listTerm$ is recorded to detect leadership change. In this case, the client

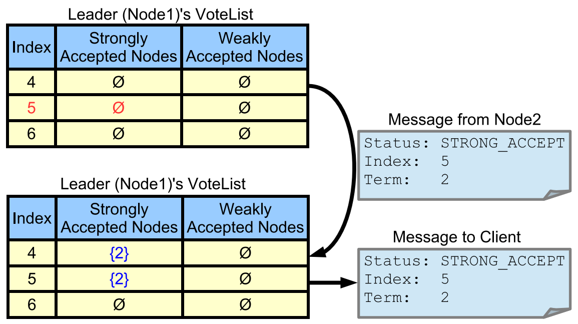

Fig. 12. When Node2 returns STRONG_ACCEPT to the leader after it receives Entry 5, the leader records it in all items in *VoteList* that have an index no larger than 5 and replies STRONG_ACCEPT to the client. This example is associated with Section III-B3b.

will retry all requests on the list to ensure that they are either processed by the old leader or the new leader.

After a client sends a request o to the leader, it may receive two types of responses if the replication succeeds, WEAK_ACCEPT or STRONG_ACCEPT, as in Figure 3(a).

1) Case of WEAK_ACCEPT: When (WEAK_ACCEPT, $index$, $term$) is returned from the leader in Section III-B2, if $listTerm < term$, the client removes and retries all requests in $opList$, i.e., fires transition *Retry* at the top of Figure 3(a). It updates $listTerm$ to $term$, i.e., a new token in place *Leader Term*, as a newer leader emerges. Moreover, it appends ($index$, $term$, o) to $opList$ for future retries, i.e., new tokens in place *Weakly Accepted Requests* in Figure 3(a). A changed term indicates a leader failure and previous WEAK_ACCEPT requests may not be finished, i.e., the client retries.

Otherwise, $listTerm = term$, the client appends ($index$, $term$, o) to $opList$ for future retries, and transition *Retry* is not fired. Since term is monotonically increasing, it is not possible for a client to detect a term less than $listTerm$.

2) Case of STRONG_ACCEPT: If (STRONG_ACCEPT, $index$, $term$) is returned in Section III-B2, and $listTerm < term$, the client removes and retries all requests in $opList$, i.e., firing transition *Retry* in Figure 3(a), and then updates $listTerm$ to $term$. The reason is the same as in Section III-C1.

Otherwise, the client removes all elements in $opList$ that have an index no larger than $index$. Log continuity assures that they are also STRONG_ACCEPT now, i.e., no need to retry (remove tokens from place *Weakly Accepted Requests*). Raft ensures log continuity and commits entries in order. Within the same term, STRONG_ACCEPT means that the current request and all the previous WEAK_ACCEPT requests are committed. Therefore, they can be safely removed from the list.

IV. Persistence

Persistence is satisfied when a request is never lost after entering the system. Raft provides persistence for committed entries by two assumptions: the state machine is durable; the log storage is durable, and each log entry is persisted. NB-Raft preserves the assumptions so that any strongly accepted entries and committed entries are durable.

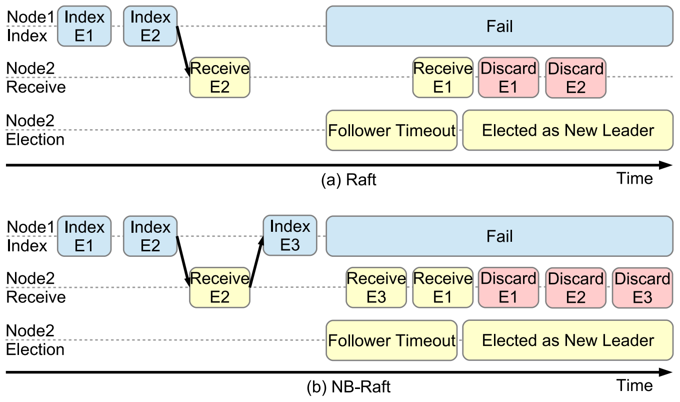

Fig. 13. Entry loss when the old leader fails, and the new leader is elected.

If the leader fails after sending a request from the leader to followers, similar to the original Raft, the persistence of the request is indeterminate. If the new leader receives the request before it is elected and commits the request in its term, the request is persistent. Otherwise, the request will be lost permanently. The loss is thus affected follower timeout.

For example, in Figure 13(a), Node1 (the old leader) fails after sending two entries (E1 and E2). Node2 starts the follower timeout as soon as the old leader fails. During the timeout, Node2 receives E2. It is blocked because E1 does not arrive. When the timeout ends, an election starts and Node2 is elected. Then, Node2 receives E1 but discards it together with E2, since they are from an old leader. If the timeout is longer (covering "Receive E1"), then E1 and E2 can be received and appended before Node2 is elected. As a result, no entry is lost. Likewise, decreasing the request size could also make E1 received before the election, because they take less time to transmit. In this case, entry loss is eliminated.

However, in Figure 13(b), as blocking is reduced by NB-Raft, Node1 sends E3 before it fails. E2 and E3 will be discarded after the election because they are only weakly accepted and E1 is discarded for a smaller term. As a result, one more entry is lost compared with Figure 13(a). Nevertheless, by increasing follower timeout and decreasing the request size, the entry loss of NB-Raft could be reduced as well.

In the worst case, if there are N_{cli} client connections when clients and the leader fail, up to N_{cli} requests will be lost in Raft. This is because each connection is blocked till the current request is committed. As a sliding window of capacity w is introduced, the potential loss is enlarged to $N_{\text{cli}} + w$. Again, when $w = 0$, NB-Raft becomes Raft.

In the IoT applications, such a trade-off between throughput and persistence is worthwhile. (1) For the massive data streaming from sensors, the data loss owing to a low system throughput would be astonishing compared to the previously mentioned data loss introduced by leader failure. According to the experiments in Section V, about 30% more data can be saved by our NB-Raft compared to the original Raft, in contrast to 0.00003% data loss owing to a follower timeout of 0.5s, very unlikely in practice. (2) The data loss in the IoT scenarios is very prevalent. According to our preliminary study [17], the missing rates are as high as 28.2% and 24.9% in the real applications of Turbine and Vehicle, respectively. Again, such high missing rates in the data sources make 0.00003% data loss by NB-Raft and 0.000015% by Raft negligible. (3) The missing data can be accurately imputed. For instance, the accuracy of missing data imputation may reach 0.91 by our proposals in the preliminary study [18], [19].

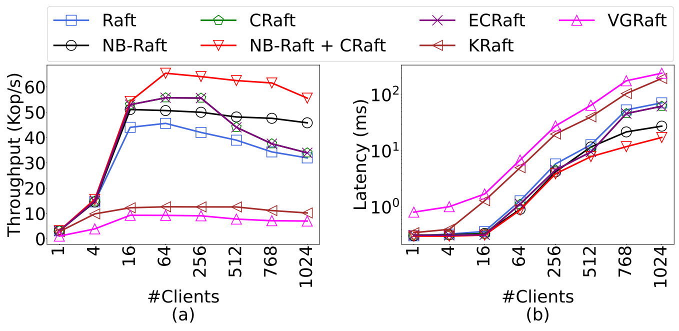

Fig. 14. Varying concurrency with 4KB requests

V. Evaluation

In the experiments, we compare our NB-Raft with the original Raft [8] and show how it cooperates with CRaft [20] for even higher throughput. In short, the results demonstrate that the throughput could be improved by about 30% using our NB-Raft compared to the original Raft. It is a considerable amount of data saved in contrast to 0.00003% data loss owing to a follower timeout of 0.5s, which is unlikely in practice.

A. *Experimental Settings*

We implement the basic Raft protocol and NB-Raft in Java as the consensus module of Apache IoTDB. In addition, we also integrate CRaft [20], which fragments entries to reduce bandwidth usage at the cost of disabling follower read. The reasons why we choose CRaft in comparison are two-fold: (1) it also focuses on improving protocol throughput instead of availability; (2) the method is orthogonal to ours, making them work together for even higher throughput. For NB-Raft, we set the window size to 10000 by default. Indeed, it is never filled up in the experiments.

The system is deployed in 10 servers, each with 4 Intel Xeon Platinum 8260 CPUs, 756 GB memory, CentOS and 10Gb/s network. Note that in practice, the typical replication factor is three, i.e., three replicas in a Raft group. Thereby, in the default configuration, there is a 3-node replication group and another machine as the client. The client sends 4 KB requests with 1024 threads and measures the system throughput. The number of dispatchers is the same as clients to avoid long queues.

In addition to the local network deployment, we also introduce a cloud deployment in Alibaba Cloud in Section V-H.

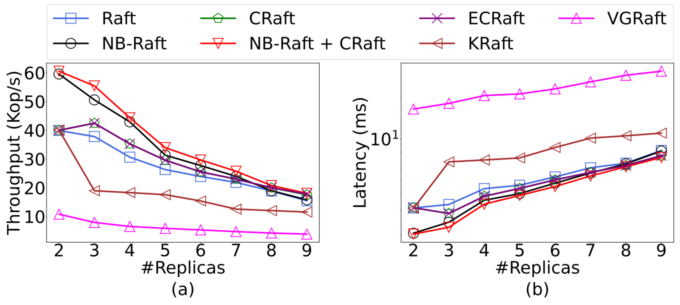

Fig. 15. Varying replication number

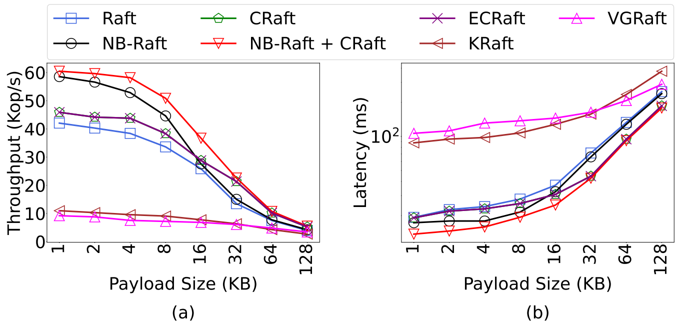

Fig. 16. Varying payload size

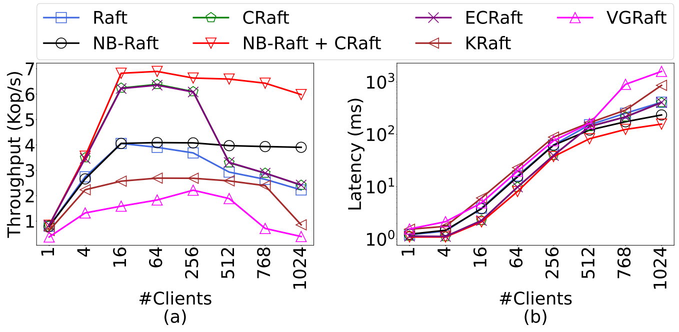

Fig. 17. Varying concurrency with 128KB requests

B. Concurrency

The experiment varies the number of concurrent clients, from 1 to 1024. It determines the maximum degree of disorder and thus impacts the blocking time $t_{\text{wait}}(F)$ in Figure 3.

In Figure 14, when there is only one client, low parallelism limits the throughput. By adding clients, the throughput first increases in Figure 14 (a). Additional clients are served by different CPU cores in parallel, and network contention is insignificant, making entry disorder not problematic. However, when the number of clients grows further, clients contend for resources. The throughput of NB-Raft is improved by about 30% compared to Raft, e.g., at 1024 clients. When integrated with CRaft, the improvement is even more significant.

It is not surprising that the throughput of all the methods decreases by adding even more clients, e.g., greater than 512, owing to resource competition in higher concurrency. That is, the throughput cannot be further improved by adding more clients. NB-Raft drops a bit slower, since it can successfully reduce the waiting time among highly concurrent requests, a favored scenario of the proposal. Thereby, the improvement compared to Raft is consistently observed.

C. Replication Number

Recall that by adding more transitions *Send Log* between Figure 3(b) and (c), more replicas introduce heavier traffic to the network. And two consecutive requests to the same follower may be interleaved with more requests to other followers, causing higher waiting time and lower throughput. When there are two replicas, once the follower is waiting for previous entries, the leader cannot proceed as there are no other followers.

As shown in Figure 15(a), the largest gap between NB-Raft and Raft occurs when #Replica is two. The two-node configuration is slowed by out-of-order entries most significantly. Adding more replicas provides the opportunity to find a follower where entries are in-order. Thereby, the leader is less likely to be blocked by out-of-order entries. However, this also results in more network traffic and a lower throughput. CRaft does not work with only one follower, as entries cannot be fragmented.

D. Payload Size

By increasing the expected time and its variance of transition *Send Log* in Figures 3(b) and (c), we can simulate a larger payload size. A larger payload size will intensify network congestion, and routers may be forced to choose different routing paths for consequent requests to avoid congestion. As a result, there will be more out-of-order arrivals.

Figure 16(a) presents throughput measured by requests per second. In Raft, when the payload size is small (1KB), there will be tremendous requests in the network, Entry disorders become severe, which significantly increases the follower process time. As the payload size increases to 4KB, the transmission time can still be shadowed by round trip time and does not become a bottleneck.

As shown in Figure 16, when the clients send larger size requests, e.g., larger than 32KB, CRaft can split it and thus may show higher throughput than NB-Raft. In contrast, by further splitting small requests, the benefit is limited but increasing the processing overhead, i.e., worse than NB-Raft.

To observe the performance impact of more/less threads in a larger request size, Figure 17 varies concurrency with 128KB requests. When the concurrency is not high, e.g., less than 256 clients, the improvement by CRaft is more significant than Figure 14 of 4KB requests. Nevertheless, NB-Raft still shows clearly better throughput in higher concurrency with more than 512 clients, by reducing the waiting time among concurrent requests. Again, NB-Raft + CRaft has the best performance in all the settings.

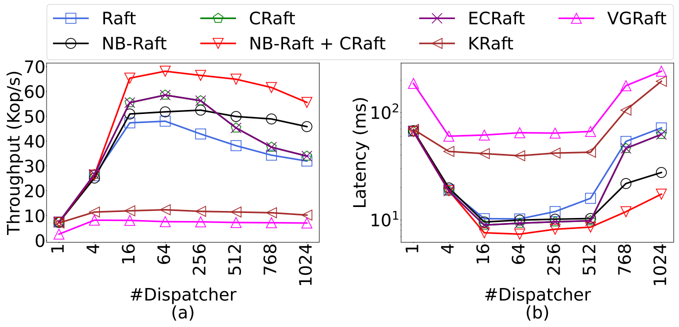

Fig. 18. Varying dispatcher number

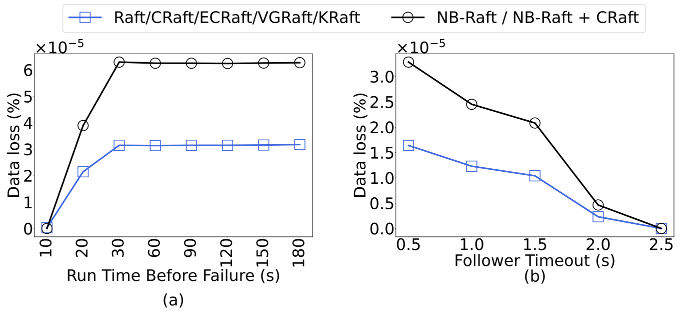

Fig. 19. Data loss in failure under various settings

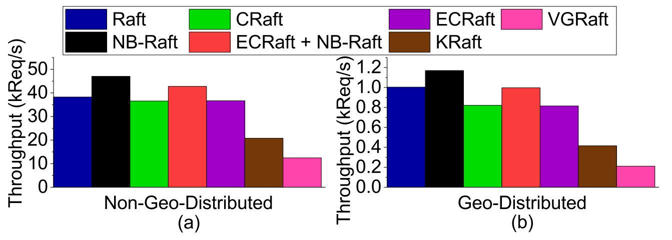

Fig. 20. Non-Geo-Distributed vs. Geo-Distributed in Alibaba Cloud

E. Dispatcher Number

Figure 18 shows the results of varying the number of dispatchers. A small number of dispatchers, e.g., less than 16, indeed causes the requests to queue up, i.e., tokens in Queue To Follower in Figure 3 will accumulate. The corresponding latency (process time) is thus high and the throughput is low in Figure 18. By further increasing the dispatchers, i.e., higher concurrency, similar to Figures 14 and 17, the latency increases again. Since more dispatchers and clients increase the concurrency, their trends of throughput are generally similar, i.e., NB-Raft performs better in higher concurrency with more dispatchers and clients.

F. Latency

We also compared the latency of protocols in Figures 14-18. In general, higher throughput leads to lower latency, when the concurrency is fixed, e.g., in Figures 15 and 16. By varying concurrency in Figures 14, 17 and 18, if the concurrency is small, the system maximum parallelism is not reached. Both throughput and latency are low. With the increase of concurrency, the latency becomes higher. The throughput drops after reaching the maximum, owing to resource contention. Nevertheless, our NB-Raft shows lower latency and higher throughput, by reducing the waiting time of highly concurrent requests, e.g., with 1024 clients or dispatchers.

G. Persistence Loss

As discussed in Section IV, follower timeout affects the persistence loss when a leader fails. To simulate failures, after 30s ingestion, we kill the leader and clients simultaneously. After a new leader is elected, we compare the number of entries of the new leader with the number of requests that clients have issued.

We first perform an experiment on varying the time for terminating the leader and clients to simulate failures, in Figure 19(a). When the system just starts and runs for a very short period, only a small number of requests are issued and processed in concurrently. If a failure occurs, the data loss is not large either. After about 30s, the system becomes stable. The concurrency reaches its maximum, and the data loss does not increase further. Thereby, to simulate failures, we kill the leader and clients simultaneously after 30s ingestion, in the following experiments.

In Figure 19(b), as the follower timeout increases, there is a higher probability that the newly elected leader will receive more entries, and the entry loss is reduced. Even the follower timeout is noticeably short (0.5s), the persistence loss is always kept under 0.00003%. For sensors working around 1Hz, the loss is extremely minor compared with the entire time-series and can be easily imputed [19]. Such a short follower timeout is hardly used in real applications, because it causes frequent elections and reduces the system availability.

H. Cloud Deployment

The system, Apache IoTDB with NB-Raft, has also been deployed in Alibaba Cloud, providing IoT data services. We evaluate the service in a 5-node cluster and change the distribution of nodes to examine how the proposed protocol responds to geo-distribution. Each node is an ecs.s6-c1m2.2xlarge instance. When enabling geo-distribution, the nodes are in Beijing, Guangzhou, Shanghai, Hangzhou, and Chengdu, respectively; otherwise, they are all in Beijing. The experiments use 64 client threads and 1KB size (censored data from real applications) since cloud servers are less powerful.

The result is presented in Figure 20. NB-Raft is at an advantage in both configurations. But CRaft is no longer effective as the cloud servers have limited CPU resources, and computing parity introduces a new bottleneck. Moreover, geo-distribution makes latency more vital than bandwidth, and thus saving bandwidth by CRaft does not benefit much.

I. Comparison with Existing Methods

In this section, we compare with the state-of-the-art methods in the experiments, including (I) KRaft [21], (II) VGRaft [22] and (III) ECRaft [23], e.g., in Figures 14, 15, 16, etc.

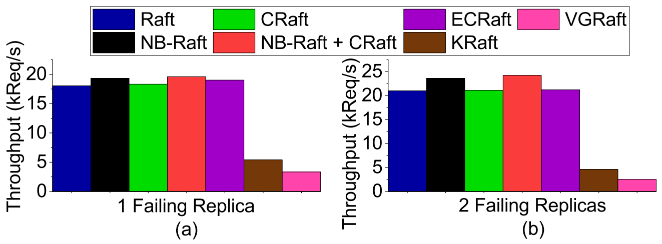

Fig. 21. Varying the number of failing replicas in a 5-replica setting

KRaft [21] chooses a subset of nodes (K-Bucket) for the leader to replicate logs directly and let K-Bucket nodes relay logs to other nodes. In this way, the leader's workload can be reduced by sending less messages directly. However, it is less likely for KRaft to find the fastest quorum, leading to higher latency and lower throughput, e.g., as illustrated in Figures 14-16. Note that when there are only two replicas in Figure 15, KRaft also has only one follower to replicate without further relay, and thus shows similar results as the original Raft.

VGRaft [22] proposes to resist byzantine fault in Raft, by hash and digital signature verification, which is computationally expensive. Also, VGRaft selects a new verification group for each consensus process, introducing heavy overhead. It is not surprising that such heavy overhead results in higher latency and lower throughput. Therefore, VGRaft performs the worst e.g., in Figures 14-16.

ECRaft [23] improves throughput of CRaft after a failure. When no failure occurs, e.g., in Figures 14-16, ECRaft shows almost the same results as CRaft. Thereby, we compare the methods with failing nodes in a 5-replica setting in Figure 21. As shown, ECRaft improves the throughput slightly compared to CRaft. Note that having failing nodes is similar to reducing the replica number in Figure 15, where the throughput of Raft may increase, as also presented in [23]. Consequently, our NB-Raft by further reducing the waiting time of concurrent requests shows better performance than ECRaft.

J. Comparison and Complement to CRaft

It is notable that CRaft is better than NB-Raft in certain cases when compared separately, while the combination of NB-Raft and CRaft is the best. The reason is that NB-Raft targets on reducing the waiting time of concurrent requests, while CRaft splits large requests, improving the performance in two different directions.

Figure 22 summarizes the cases where different methods may perform better. As shown, NB-Raft handles well the highly concurrent requests by reducing the waiting time, while CRaft prefers low concurrency in Figure 14. Moreover, CRaft can split large requests and show better throughput than NB-Raft when the payload size is large in Figure 16. For the same reason, by splitting into more replicas, e.g., 9, CRaft may exceed NB-Raft in Figure 15. Nevertheless, NB-Raft + CRaft achieves the best throughput in various conditions.

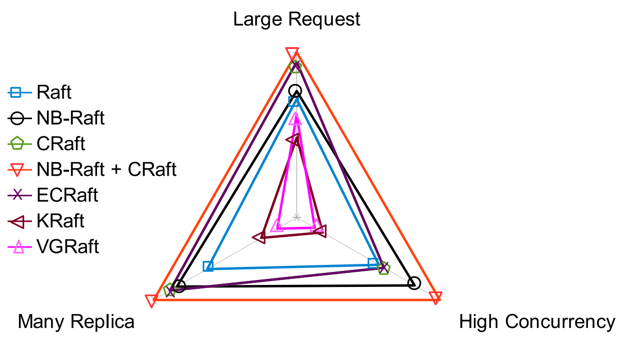

Fig. 22. Summary of throughput in various conditions

K. Preferred Conditions of NB-Raft

Table II summarizes the conditions and IoT deployments where one would expect NB-Raft to perform. In short, NB-Raft shows better throughput in high concurrency, less replicas and small requests, whereas CRaft prefers low concurrency, more replicas and large requests. Moreover, NB-Raft performs with limited CPU resources and supports follower read, while CRaft suffers lower data loss in failure. In this sense, NB-Raft is applicable for higher concurrency with tolerance to a bit more data loss in failure. In addition, NB-Raft + CRaft could be applied for even higher throughput in the IoT deployments with more CPU resources and no need on follower read.

Specifically, the major idea of NB-Raft is to reduce the waiting time of concurrent requests owing to out-of-order arrivals. The higher the concurrency is, the more the out-of-order arrivals as well as the waiting time would be. The improvement by NB-Raft, compared to Raft or CRaft, is thus more significant, e.g., in higher concurrency of more clients in Figures 14 and 17 or more dispatchers in Figure 18. Thereby, as summarized in Table II, NB-Raft favors high concurrency, prevalent in IoT scenarios with a large number of sensors.

NB-Raft chooses to achieve higher throughput by more potential data loss in failure. The trade-off is worthwhile in the IoT scenarios as discussed in Section IV. Moreover, IoT applications like real-time monitoring also incur heavy query loads. Unfortunately, follower read is not supported in CRaft, which seriously limits its query capacity, potential optimization [24], and thus application scope in the IoT scenarios, again summarized in Table II.

NB-Raft + CRaft outperforms NB-Raft as well as CRaft in different settings, since they improve the performance in two different directions. First, recall that NB-Raft outperforms CRaft in high concurrency by reducing the waiting time of concurrent requests. Thereby, NB-Raft + CRaft outperforms CRaft more clearly with more clients in Figures 14 and 17 or more dispatchers in Figure 18. On the other hand, the improvement of NB-Raft + CRaft compared to NB-Raft is more significant in Figure 17 of larger payload size (128KB) than Figure 14 of 4KB requests, since CRaft can split large requests. In this way, NB-Raft + CRaft achieves the best throughput in various conditions.

To verify that CRaft needs heavy computation of parity frag-

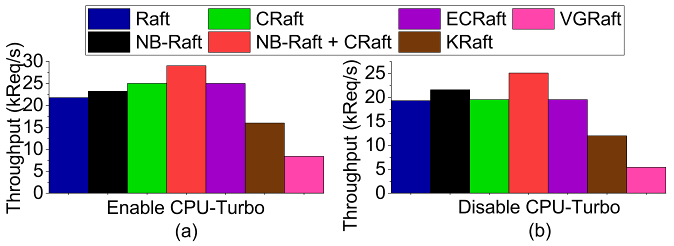

Fig. 23. Throughput under different CPU conditions.

TABLE II
PREFERRED CONDITIONS

Protocol	Concurrency	Replica number	Request size	Persistence	Follower read	CPU usage
Raft	Low	Few	Small	High	Yes	Low
NB-Raft	High	Few	Small	Low	Yes	Low
CRaft	Low	Many	Large	High	No	High
NB-Raft + CRaft	High	Many	Large	Low	No	High
ECRaft	Low	Many	Large	High	No	High
KRaft	Low	Few	Small	High	Yes	Low
VGRaft	Low	Few	Small	High	Yes	High

ments, we conduct an experiment of changing CPU resources via disabling CPU-Turbo. As shown in Figure 23, reducing CPU resources (disabling CPU-Turbo) lowers the throughput of all protocols, while CRaft suffers more from the limited CPU resources. We also summarize this condition in Table II.

L. Summary

In short, NB-Raft shows better throughput in high concurrency, since it reduces the waiting time of concurrent requests. CRaft handles large request size by splitting it, but with higher CPU cost and no follower read. Nevertheless, the combination of NB-Raft and CRaft achieves the best throughput in most experiments, with various request size and concurrency.

VI. Related Work

A. Raft Analysis

In this study, we use Petri Net [11] to demonstrate log replication in Raft and identify the throughput bottleneck. [25] can be used to estimate model parameters. Other methods have been proposed to explore different properties of the protocol.

To study the response time and availability, Raft is modeled with Stochastic Activity Networks [26] in the context of an SDN controller cluster [14]. We concern more about the throughput instead of the single request response time, as for streaming IoT data, ingestion is often batched and concurrent.

Most others focus on leader election and follower timeout. Markov chain is used to model the state transitions during leader election and follower timeouts [27] to investigate how network packet loss increases elections. Leadership uniformity [28] uses common measurements like Std, Var, MAE, and MSE to quantify how leadership distributes in a cluster with different follower timeouts. In addition, network topology has a significant impact on the protocol [29] concerning election time. In this paper, we also focus on the log replication instead of the fail-over and election process.

B. Raft Optimization

While NB-Raft focuses on blocking, other work optimizes Raft from different perspectives and is orthogonal to NB-Raft. In addition to CRaft [20], KRaft [21], VGRaft [22] and ECRaft [23] compared in experiments, Pirogue [30] boosts the performance of Raft by reducing the number of followers needed to commit an entry and introducing a stricter election rule to ensure safety. It reduces liveness, making a cluster more vulnerable to rack or data center failures. Also, less followers to commit means less followers available for read.

C. Other Consensus Protocol

Besides Raft, another important class of consensus protocol is the Paxos family [1], [2], [3], [4], [5], [6], [7], [31], [32]. Our proposed NB-Raft can be viewed as an advantageous position between Paxos and Raft. Paxos allows arbitrary holes in the instance sequence and Raft allows no holes, while NB-Raft allows holes in the suffix using a sliding window. It enables parallel entries processing and thus increases throughput.

To remove heavy synchronization point introduced by indexing, E-Paxos [2] and SD-Paxos [4] build multiple Paxos groups to increase parallelism and determine the order of instances across groups by either a decentralized graph-based conflict resolution or a centralized sequencer. These designs can be applied to Multi-Raft and generate similar variants.

Canopus [33] builds a tree over Raft super-leaves to avoid the single leader bottleneck. Its underlying Raft implementation can be replaced by almost any variant, including NB-Raft.

VII. Conclusion

In this paper, we first notice the bottleneck of log replication in Raft, i.e., waiting for others to append and thus blocking the subsequent requests. In addition to the qualitative analysis, we propose to model the Raft log replication process by Petri Net, which enables a quantitative evaluation of the bottleneck. Upon the analysis, we present Non-Blocking Raft (NB-Raft), a variant of Raft that introduces a WEAK_ACCEPT state to enable fast replies and unblock subsequent requests as early as possible. In this way, the parallelism and throughput increase, essential to the IoT applications with vast sensors constantly generating data. The proposed NB-Raft is implemented as the consensus protocol of Apache IoTDB, a commodity time series database management system, and deployed in Alibaba Cloud for various applications. Extensive experiments demonstrate that the throughput is improved by about 30% using our NB-Raft compared to the original Raft. It is a considerable amount of data saved in contrast to the 0.00003% data loss owing to a follower timeout of 0.5s (very unlikely in practice).

Acknowledgement: This work is supported in part by National Key Research & Development Plan (2021YFB3300500, 2019YFB1705301, 2019YFB1707001), National Natural Science Foundation of China (62072265, 62021002, 62232005), Beijing National Research Center for Information Science and Technology (BNR2022RC01011), and Alibaba Group through Alibaba Innovative Research (AIR) Program. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] L. Lamport, “The part-time parliament,” in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 277–317.

[2] I. Moraru, D. G. Andersen, and M. Kaminsky, “There is more consensus in egalitarian parliaments,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 358–372.

[3] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, “Just say no to paxos overhead: Replacing consensus with network ordering,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 467–483.

[4] H. Zhao, Q. Zhang, Z. Yang, M. Wu, and Y. Dai, “Sdpaxos: Building efficient semi-decentralized geo-replicated state machines,” in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 68–81.

[5] A. Charapko, A. Ailijiang, and M. Demirbas, “Pigpaxos: Devouring the communication bottlenecks in distributed consensus,” in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 235–247.

[6] L. Lamport, “Fast paxos,” *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[7] L. Lamport and M. Massa, “Cheap paxos,” in *International Conference on Dependable Systems and Networks, 2004*. IEEE, 2004, pp. 307–314.

[8] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 305–319.

[9] C. Wang, X. Huang, J. Qiao, T. Jiang, L. Rui, J. Zhang, R. Kang, J. Feinauer, K. A. McGrail, P. Wang *et al.*, “Apache iotdb: time-series database for internet of things,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2901–2904, 2020.

[10] C. Adams, L. Alonso, B. Atkin, J. Banning, S. Bhola, R. Buskens, M. Chen, X. Chen, Y. Chung, Q. Jia *et al.*, “Monarch: Google’s planet-scale in-memory time series database,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3181–3194, 2020.

[11] W. Reisig, *A primer in Petri net design*. Springer Science & Business Media, 2012.

[12] “https://iotdb.apache.org.”

[13] “https://github.com/apache/iotdb/tree/research/nb-raft.”

[14] E. Sakic and W. Kellerer, “Response time and availability study of raft consensus in distributed sdn control plane,” *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 304–318, 2017.

[15] M. Poess, R. Nambiar, K. Kulkarni, C. Narasimhadevara, T. Rabl, and H.-A. Jacobsen, “Analysis of tpcx-iot: The first industry standard benchmark for iot gateway systems,” in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 1519–1530.

[16] “https://ratis.apache.org/.”

[17] C. Fang, S. Song, and Y. Mei, “On repairing timestamps for regular interval time series,” *Proc. VLDB Endow.*, vol. 15, no. 9, 2022.

[18] S. Song and Y. Sun, “Imputing various incomplete attributes via distance likelihood maximization,” in *KDD ’20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, R. Gupta, Y. Liu, J. Tang, and B. A. Prakash, Eds. ACM, 2020, pp. 535–545. [Online]. Available: https://doi.org/10.1145/3394486.3403096

[19] A. Zhang, S. Song, Y. Sun, and J. Wang, “Learning individual models for imputation,” in *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 2019, pp. 160–171. [Online]. Available: https://doi.org/10.1109/ICDE.2019.00023

[20] Z. Wang, T. Li, H. Wang, A. Shao, Y. Bai, S. Cai, Z. Xu, and D. Wang, “Craft: An erasure-coding-supported version of raft for reducing storage cost and network cost,” in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 297–308.

[21] R. Wang, L. Zhang, Q. Xu, and H. Zhou, “K-bucket based raft-like consensus algorithm for permissioned blockchain,” in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2019, pp. 996–999.

[22] S. Zhou and B. Ying, “Vg-raft: An improved byzantine fault tolerant algorithm based on raft algorithm,” in *2021 IEEE 21st International Conference on Communication Technology (ICCT)*. IEEE, 2021, pp. 882–886.

[23] M. Xu, Y. Zhou, Y. Y. Qiao, K. Xu, Y. Wang, and J. Yang, “Ecraft: A raft based consensus protocol for highly available and reliable erasure-coded storage systems,” in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2021, pp. 707–714.

[24] H. Lan, Z. Bao, and Y. Peng, “A survey on advancing the DBMS query optimizer: Cardinality estimation, cost model, and plan enumeration,” *Data Sci. Eng.*, vol. 6, no. 1, pp. 86–101, 2021. [Online]. Available: https://doi.org/10.1007/s41019-020-00149-7

[25] J. Yang, J. Wang, Y. Zhang, W. Cheng, and L. Li, “A heuristic sampling method for maintaining the probability distribution,” *J. Comput. Sci. Technol.*, vol. 36, no. 4, pp. 896–909, 2021. [Online]. Available: https://doi.org/10.1007/s11390-020-0065-6

[26] W. H. Sanders and J. F. Meyer, “Stochastic activity networks: formal definitions and concepts,” in *School organized by the European Educational Forum*. Springer, 2000, pp. 315–343.

[27] D. Huang, X. Ma, and S. Zhang, “Performance analysis of the raft consensus algorithm for private blockchains,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 50, no. 1, pp. 172–181, 2019.

[28] E. Iosif, K. Christodoulou, M. Touloupou, and A. Inglezakis, “Leadership uniformity in raft consensus algorithm,” in *European, Mediterranean, and Middle Eastern Conference on Information Systems*. Springer, 2020, pp. 125–136.

[29] H. Howard, “Arc: analysis of raft consensus,” University of Cambridge, Computer Laboratory, Tech. Rep., 2014.

[30] J.-F. Pâris and D. D. Long, “Pirogue, a lighter dynamic version of the raft distributed consensus algorithm,” in *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2015, pp. 1–8.

[31] L. Lamport *et al.*, “Paxos made simple,” *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[32] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, “Designing distributed systems using approximate synchrony in data center networks,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 43–57.

[33] S. Rizvi, B. Wong, and S. Keshav, “Canopus: A scalable and massively parallel consensus protocol,” in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, 2017, pp. 426–438.