SAREK: Optimistic Parallel Ordering in Byzantine Fault Tolerance

Bijun Li, TU Braunschweig, bli@ibr.cs.tu-bs.de
Wenbo Xu, TU Braunschweig, wxu@ibr.cs.tu-bs.de
Muhammad Zeeshan Abid, KTH Stockholm, mzabid@kth.se
Tobias Distler, FAU Erlangen–Nürnberg, distler@cs.fau.de
Rüdiger Kapitza, TU Braunschweig, rrkapitz@ibr.cs.tu-bs.de

***Abstract*—Recently proposed Byzantine fault-tolerant (BFT) systems achieve high throughput by processing requests in parallel. However, as their agreement protocols rely on a single leader and make big efforts to establish a global total order on all requests before execution, the performance and scalability of such approaches is limited. To address this problem we present SAREK, a parallel ordering framework that partitions the service state to exploit parallelism during both agreement as well as execution. SAREK utilizes request dependency which is abstracted from application-specific knowledge for the service state partitioning. Instead of having one leader at a time for the entire system, it uses one leader per partition and only establishes an order on requests accessing the same partition. SAREK supports operations that span multiple partitions and provides a deterministic mechanism to atomically process them. To address use cases in which there is not enough application-specific knowledge to always determine a priori which partition(s) a request will operate on, SAREK provides mechanisms to even handle mis-predictions without requiring rollbacks. Our evaluation of a key-value store shows an increase in throughput performance by a factor of 2 at half the latency compared to a single-leader implementation.**

***Index Terms*—Byzantine Failures; Multi-leader; Parallel Agreement and Execution; Generalized Consensus; State Partition**

I. INTRODUCTION

State machine replication is a general method for providing fault tolerance in distributed systems [1]: Requests issued by clients are first ordered using an agreement protocol and then sequentially executed on multiple server replicas. Depending on the used protocol, systems are able to tolerate fail-stop [2], [3] or Byzantine failures [4] of replicas. In the context of Byzantine fault tolerance (BFT), in order to overcome the limitations of sequential execution and to exploit the potential of multi-core hardware, systems have been proposed to allow non-conflicting requests to be processed in parallel [5], [6], [7]. In this paper, we refer to such requests as being *independent* which, for example, applies to requests that operate on different parts of the service state. In contrast, *dependent* requests must still be executed sequentially as otherwise non-faulty replicas could become inconsistent.

Parallel execution offers great benefits for use case scenarios where the actual request processing time is not negligible. However, it does not lower the risk of having a system bottleneck, referring to the fact that existing BFT systems use a single replica as the leader (either for totally ordering requests [5] or for execution preparation [7]), which actually slows down application processing speed. To a certain extent, this problem can be mitigated by rotating the leader role among replicas [8], [9]. Nevertheless, as these approaches are based on a totally-ordered sequence of requests, and only split the responsibility of establishing it, the effects are limited.

In this paper we present SAREK, a parallel ordering framework that instantiates multiple single-leader-based BFT protocols independently. Besides enabling concurrent request executions, SAREK exploits parallelism during request ordering: It partitions the service state and only linearly orders the requests accessing the same partition(s), by creating a partition-specific *schedule*. In this way, agreement for independent requests can be performed concurrently. Furthermore, by selecting different replicas as leaders of different partitions, the system is able to balance the load induced by the leader role across all replicas. After agreement is complete, a dedicated execution instance is responsible for processing requests in each partition according to the local schedule.

While handling an operation that accesses only a single partition is straightforward in such an environment, additional efforts have to be taken to support requests operating on multiple partitions ("cross-border requests"). For example, despite being ordered in several partitions, a cross-border request must not be executed more than once. In addition, the processing of a cross-border request must be consistent with the individual schedules that are determined for all affected partitions. SAREK satisfies these requirements by using a mechanism that is based on a combination of prioritizing partitions and safe reordering of requests: Only the execution instance of the partition with the highest priority actually processes a cross-border request while the instances of other involved partitions are put on hold in the meantime.

SAREK relies on application-specific knowledge to define service-state partitions as well as to *predict* which partitions a request will operate on. For this purpose, each replica holds a deterministic PREDICT() function which identifies the state objects to be read or written during processing of a particular request. As the implementation of a precise PREDICT() function may not be feasible (or considered too costly) for some applications, for example, due to the set of accessed objects being dependent on internal service state,

This research was partially supported by Siemens Rail Automation Graduate School (iRAGS) and the German Research Council (DFG) under the grants no. KA 3171/1-2 and DI 2097/1-2.

SAREK offers support to deal with imprecise knowledge up to the point where mis-predictions are handled: During execution, the system monitors accesses to state objects and consequently detects if a request tries to operate on a partition not included in the output of PREDICT(). If this is the case, SAREK initiates a re-prediction of the request and then safely updates the schedules of the partitions affected, thereby preventing any form of rollback.

In this work, we show that it is possible to realize SAREK based on an existing BFT implementation without requiring modifications to the most complex part: the agreement protocol. Instead, the agreement stage can be treated as a black box that is instantiated multiple times, once for each partition. This makes most existing BFT agreement protocols compatible with SAREK. We evaluate our prototype via microbenchmarks and the YCSB (Yahoo! Cloud Serving Benchmark) [10] benchmark, which apart from providing operations to access single objects, also allow clients to issue requests that access multiple objects atomically, thereby relying on SAREK's support for cross-border request.

In particular, this paper makes the following contributions:

- It presents SAREK's approach to partitioning service state and making use of multiple leaders to exploit parallelism in a BFT system.
- It details SAREK's mechanism to support operations spanning multiple partitions, which is based on a combination of per-operation partition priorities and dynamic request reordering.
- It explains how SAREK is able to deal with imprecise predictions of the partitions a request will operate on.
- It evaluates SAREK in comparison to a BFT system that comprises only a single leader.

The remainder of this paper is organized as follows: Section II summarizes related works. Section III discusses the system model and additional assumptions. Section IV presents the design of SAREK. Section V focuses on our prototype implementation and presents the evaluation. Finally, Section VI concludes the paper and discusses future work.

II. Related Work

SAREK provides parallelism at both the agreement stage where requests are ordered, and the execution stage where they are processed. In the following, we discuss related works aiming to improve the scalability and throughput of both stages [11] as well as approaches that feature distributed transactional coordination.

a) Execution Stage: Kotla et al. [5] introduced a parallelizer module between agreement stage and execution stage that allows a BFT system to concurrently execute requests that do not interfere with each other. Similar to SAREK, such requests are identified based on application-specific knowledge. However, in contrast to SAREK there is no parallelism at the agreement stage and a total order is established across all requests, not only the dependent ones. The same applies to ODRC [6], which achieves parallelism at the execution stage by processing each request on a subset of replicas instead of all. As a result, the resources freed on each replica can be used to execute additional requests.

In EVE [7], replicas do not agree on requests before processing them. Instead, replicas first execute requests concurrently and then try to agree on the corresponding state changes. If this attempt fails, replicas perform a rollback and repeat the execution, this time in sequential order. EVE assumes that the majority of requests do not share data or that there is limited contention, making conflicts a seldom event. SAREK assumes there will be sets of requests that share data and accordingly require ordering but that these sets usually do not interfere with each other. In the latter case, this is detected and addressed while completely avoiding rollbacks.

b) Agreement Stage: Aardvark [12] minimizes the negative impact a malicious leader can have on system performance by enabling the other replicas to monitor its performance. If the leader fails to propose new requests within a certain period of time, which is gradually decreased, another replica becomes the new leader. In Spinning [8], the leader changes even more frequently than in Aardvark, that is, automatically after each request. This approach has the benefit of distributing the additional load associated with the leader role across all replicas. Nevertheless, the single-leader bottleneck remains because the system still totally orders all requests.

RBFT [13] relies on multiple concurrent BFT agreement instances for robustness: All requests are totally ordered by all agreement instances but only executed by a dedicated master instance. That is, RBFT introduces parallelism at the agreement stage by fully replicating the entire agreement process, thereby also suffering from the single-leader bottleneck. SAREK, on the other hand, partitions the agreement stage and uses multiple leaders to improve performance.

Farsite [14] is a large-scale distributed file system that is resilient against Byzantine faults. The system achieves scalability by partitioning the service state and rarely coordinates when more than one partition is involved (i.e., for rename operations). In contrast to SAREK, Farsite relies on multiple dedicated BFT clusters that each executes an independent BFT agreement protocol. Compared to SAREK it does not address parallelization of request processing at agreement and execution stage for a wide range of applications or offers support for dealing with imprecise knowledge about the application.

Generalized Paxos [15] relaxes the consensus from agreeing on a single request to agreeing on a partially ordered set of requests. It is applied to handle concurrently issued but non-interfering requests, where the execution order does not matter, to ensure that they can always be executed in two message delays. EPaxos [16] is a crash-tolerant protocol that allows replicas to agree on requests without requiring a designated leader. EPaxos orders only those requests that interfere with each other. Compared to SAREK, Generalized Paxos and EPaxos do not address BFT. EPaxos offers the best performance in case of low contention, as a replica must delay the execution of a command until it receives commit confirmations for the command's dependencies.

P-SMR [17] achieves parallelism by using different multi-

cast groups for requests that can be executed concurrently. The mapping of requests to multicast groups is based on service-specific semantics. In a follow-up work [18], Marandi et al. investigated the impact of performing this mapping optimistically and proposed an approach that triggers rollbacks when inaccurate assumptions lead to inconsistencies. In contrast to P-SMR, SAREK is not limited to fail-stop failures. Furthermore, SAREK detects mis-predictions and is consequently able to initiate re-predictions before replicas become inconsistent, thereby avoiding rollbacks.

S-SMR [19] achieves scalable throughput by partitioning the application state and using caches to reduce synchronization across partitions. It relies on an atomic multicast to order commands, and implements execution atomicity to guarantee linearizability. This approach needs to be adapted to tolerate Byzantine faults, eventually resulting in a Farsite-like system.

COP [20] achieves high scalability in BFT systems by executing consecutive consensus instances in parallel with independent pipelines. However it does not involve state partitioning therefore the execution stage is not paralleled.

c) Distributed Transactional Systems: Granola [21] presents a coordination infrastructure for distributed transactions in a fail-stop model, which provides strong consistency while reducing coordination overhead. This is guaranteed by using a timestamp-based coordination mechanism to achieve serializability of transactions executing on a single storage node or across a set of nodes but requiring no agreement. Having similar functionality to Granola, Calvin [22] provides high availability and full ACID transactions in partitioned database systems. It accomplishes this by implementing a sequencing layer above the storage system to handle data replication, which runs a global agreement protocol upon read and write transactions across all replicas. According to the produced deterministic locking order, it deploys a transaction scheduling layer to serve as concurrency control. Compared to both systems, SAREK is developed for a different failure model. Furthermore, it neither enforces a total order upon all requests nor requires transactional semantics to handle conflicts.

III. System Model and Assumptions

SAREK and all its components are based on a common BFT system model [4], [23], [24], [14], [11], [5] where a minimum of $3f + 1$ replicas are required to tolerate up to f Byzantine failures. A faulty replica may behave in an arbitrary and malicious way, possibly trying to prevent non-faulty replicas from providing their services. While SAREK is able to handle at most f faulty replicas, there is no upper bound on the number of faulty clients.

Messages are authenticated by clients and replicas. We assume that an adversary is not able to break cryptographic techniques and consequently cannot send messages on behalf of a non-faulty client or replica without being detected. The network possibly fails to deliver messages, corrupt them, or deliver them out of order. To simplify presentation, we assume that such problems, excluding network partitions, are handled by lower network layers (e.g., TCP).

SAREK assumes a fully-replicated system. As in most state machine replication systems, the applications executed on top of SAREK are stateful and expose interfaces that allow clients to read and update service state by issuing requests. Here, service state is defined as a set of disjoint objects [24], [5] which can be monitored and accessed by SAREK at runtime [24], [5], [25]. We also assume all objects can be split into a set of non-overlapping partitions. Application instances must resemble a deterministic state machine which requires that after executing the same sequence of inputs, all non-faulty replicas must produce the same sequence of outputs. With state partitions, this can be relaxed regarding non-interfering requests that can be processed concurrently.

IV. SAREK Design

The aim of SAREK, is to distribute the extra workload bound to the leader role in single-leader agreement protocols ([4], [23], [24], [12]) over all replicas, and enable parallelism at the agreement as well as the execution stage. Most existing single-leader BFT systems conservatively assume a total order of all requests, however, this is typically far too pessimistic. For various applications such as key-value stores and many web applications, only fractions of requests interfere with one another, therefore only those dependent requests accessing shared state should be ordered with respect to each other. Consequently, SAREK features the idea to run multiple BFT agreement instances in parallel, where each one manages a fraction of the application state and maintains a partial order upon requests accessing it. Thereby, SAREK is lightly inspired by *Generalized Consensus* [15] which showed under the crash-stop failure model that consensus can be relaxed from agreeing on a single request to agreeing on an partially ordered set of requests.

A. Parallelizing Agreement and Execution

Parallelism in SAREK relies on state partitioning. A replica hosts multiple BFT instances. Each instance manages a partition of the service state. In the fault-free case the leader role for the individual instances is distributed (see Figure 1).

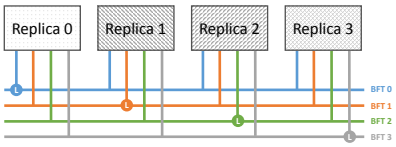

Fig. 1: In the fault-free case each replica leads one of the BFT agreement protocol instances.

For partitioning, application-specific knowledge is essential to determine to which partition a state object belongs to. For example, in a key-value store application (see Section V) where state objects are key-value pairs, the system state can be split into partitions by dividing the key space. This way, a BFT agreement instance takes only responsibility of ordering the

requests that access its associated state partition (e.g, keys that belong to the assigned part of the key space). Thus, SAREK orders requests per partition instead of "blindly" establishing a total order upon all requests.

Figure 2 depicts the system architecture of SAREK. The agreement is the same as in a single-leader BFT protocol in order to make SAREK compatible with common BFT protocols that feature a separation of the agreement and execution stage.

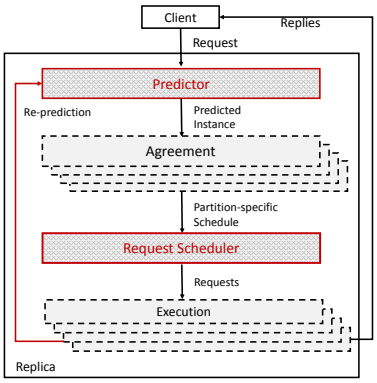

Fig. 2: System architecture of SAREK.

To assign client requests to the responsible BFT instances, SAREK introduces a *predictor* component that hosts an application-specific PREDICT() function. Each non-faulty replica has its own local predictor, which is deterministic and consistent with others. The PREDICT() function analyzes each request by its type, payload and runtime state to compute the state objects accessed during execution, and eventually indicates the partitions those objects belonging to. Similar ideas regarding a request analyzing component can be found in literature [5], [6], [24]. In a key-value store (see Section V), for example, keys are a central part of almost every request, thus they can be used for prediction.

The predictor component executes prior to the agreement enabling a coordinated request distribution (see Algorithm 1). At system startup, each replica i is promoted to be the leader of one BFT agreement instance. A client broadcasts a request m to all replicas for prediction, which causes the PREDICT() function of each replica to return a set of partitions that m will access. The reason why multiple partitions can be predicted is detailed in the following section. Next, each replica determines if it is leading one or more BFT instances of the predicted partitions (line 7 - 9). If so, it will start the agreement stage to order the request. From now on, we do not distinguish between instance and partition as they are one-to-one correspondent.

Once the agreement is complete, the request will be delivered to the execution stage as part of the partition-specific schedule (see Figure 2), which represents the order of the requests accessing the same partition. A new component placed between agreement and execution, namely the *request scheduler*, enforces the executions to follow the correct order determined by agreement. In case of *simple requests*, which access only a single partition, agreement and execution processes are rather straightforward: The one and only replica that

Algorithm 1 Predict and order requests

```
1  initialize:
2      bft_id as a BFT agreement instance
3      P_mi as a set of predicted partitions
4  upon receiving ⟨REQUEST, m⟩ at replica i do
5      P_mi := PREDICT(m) // predict partitions
6      for each par in P_mi
7          bft_id := instanceOf(par)
8          // get responsible instance
9          if i equals leaderOf(bft_id)
10             start agreement stage of ⟨REQUEST, m⟩
11         end if
12     end for
13 end
```

is the leader of the single partition initiates the ordering. Once the agreement of the ordering finishes, all replicas deliver the request to the execution stage. Partition access is monitored by the request schedulers to guarantee consistency among non-faulty replicas.

B. Handling Cross-Border Request

If a request accesses multiple partitions, namely a *cross-border request*, additional measures must be taken to ensure that 1) this request is ordered by all predicted BFT instances while 2) it is executed exactly once.

1) Cross-border Request Agreement: To fulfill these two requirements, a cross-border request splits into multiple *sub-requests* (each sub-request contains the original request) to be ordered by the corresponding instances. As a result, the order of any pair of requests accessing the same partition is uniquely defined across all replicas, based on the agreement and the corresponding partition schedules.

2) Cross-border Request Execution: After agreement, all non-faulty replicas hold identical schedules upon individual partitions. In SAREK the key to consistent execution is that, among all sub-requests of a cross-border request, only one should be executed, while others act as placeholders that only define the sequences of those sub-requests in their partition schedules. This requires to deterministically pick one instance as an *execution instance*, for example, by prioritizing which instance should execute the sub-request. The non-executing instances put their sub-requests on hold and remove them only after the execution instance finishes. In our implementation, the execution instance is chosen as the one with the smallest ID number among all involved instances.

SAREK assigns each request/sub-request a specific type, before it delivers them to the execution stage. In case of a simple request accessing one partition, it is marked as EXECUTE. Otherwise a sub-request of a cross-border request is further differentiated as 1) the CROSS-BORDER-EXEC type that should be executed by a deterministically chosen execution instance, or 2) the CROSS-BORDER-SYNC type that acts a placeholder. Requests are delivered and queued for execution after being classified.

Figure 3 shows an example, where the circle $R2$ stands for the CROSS-BORDER-EXEC type to be executed by $T0$, while the square $R2$ at $T1$ is of type CROSS-BORDER-SYNC that

Algorithm 2 Instance at execution stage

```
1  initialize:
2      // array only used for resolving request cycle
3      blocked_by := Array[no_of_partitions]
4  while executing at instance bft_id do
5      req := PartitionSchedules[bft_id].peek()
6      if req.type is EXECUTE
7          execute(req)
8          PartitionSchedules[bft_id].dequeue()
9      else if req.type is CROSS-BORDER-EXEC
10         // check sync partitions notifies
11         if req.ready()
12             executeCrossBorderRequest(req)
13             PartitionSchedules[bft_id].dequeue()
14             // remove req's corresponding sync partitions
15             for all sync_partition in req's sync partitions
16                 PartitionSchedules[sync_partition].dequeue()
17                 blocked_by[sync_partition] := NULL
18                 notify instance sync_partition
19             end for
20         // detect request cycle
21         else if detect_and_resolve_cycle(bft_id, req)
22                 goto 11
23             else
24                 blocked_by[bft_id] := req
25                 wait for corresponding CROSS-BORDER-SYNC
26                 continue
27             end if
28         end if
29     else if req.type is CROSS-BORDER-SYNC
30         exec_partition := the partition of corresponding
                 CROSS-BORDER-EXEC
31             blocked_by[exec_partition] := NULL
32             notify instance exec_partition
33         blocked_by[bft_id] := req
34         wait for corresponding CROSS-BORDER-EXEC to finish req
35     end if
36 end while
```

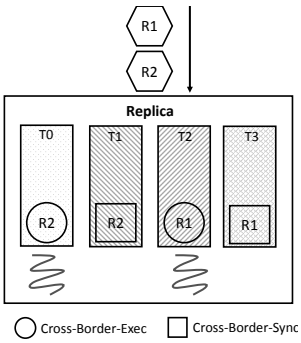

Fig. 3: Cross-border requests distribution in SAREK.

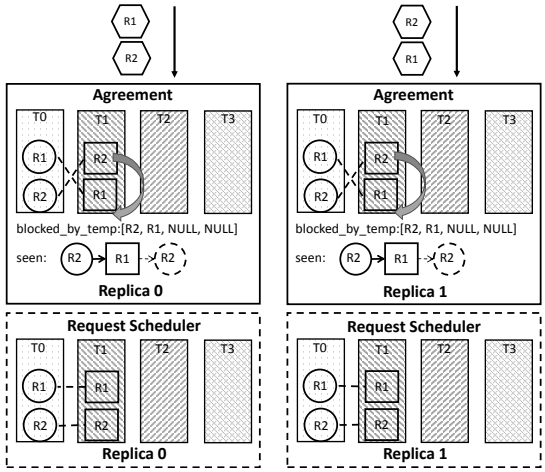

Fig. 4: Cause of request cycle and solution.

only synchronizes with the execution of $T0$. The procedure can be summarized as detailed by Algorithm 2.

Execution attempt is triggered when a request reaches the head of the local partition schedule (basically resembling a queue), and depending on its request type: 1) Those of EXECUTE type can be executed immediately since the execution is independent and non-interfering to other partition schedules (line 7). 2) In case of a CROSS-BORDER-EXEC type, it is first checked whether all other CROSS-BORDER-SYNC sub-requests are also at the head or their individual partition schedules (line 11). If true, this instance executes and afterwards removes all related sub-requests (line 12 - 19). Otherwise, it tries to detect and handle request cycles if there are any (line 21 - 28), which will be discussed later. 3) For the CROSS-BORDER-SYNC type, the instance notifies the corresponding CROSS-BORDER-EXEC type sub-request and waits for execution to complete (line 29 - 35). This way, system consistency holds as one request is guaranteed to execute only once.

As aforementioned, partition schedules at different instances might diverge, because of out-of-order request arrivals at the absence of a deterministic delivery. Figure 4 shows an example where requests $R1$ and $R2$ arrive at $Replica0$ and $Replica1$ in different orders. Assume these two replicas are leading instances $T0$ and $T1$, respectively, eventually the sub-requests of $R1$ and $R2$ would sequence differently in two instances' partition schedules. According to Algorithm 2, $R2$'s CROSS-BORDER-EXEC should wait for the notification from its CROSS-BORDER-SYNC, nevertheless it blocks $R1$'s CROSS-BORDER-EXEC. At the meantime, $R1$'s CROSS-BORDER-SYNC is waiting for its CROSS-BORDER-EXEC while blocking $R2$'s CROSS-BORDER-SYNC. As a result, a *request cycle* occurs as both instances have been blocked and waiting for each other. Similar scenarios can occur when more than two partitions are involved.

SAREK uses the request scheduler to detect and resolve such cycles as shown in Algorithm 3. It is invoked when an instance is holding a CROSS-BORDER-EXEC and waiting for notifications from corresponding CROSS-BORDER-SYNCs. The procedure is protected by a mutex lock, that is, only one instance can access it at a time. The first step is to find out which instances are genuinely blocked and have no chance to resume due to being part of a request cycle. The detection relies on the array $blocked\_by$ that holds the currently waiting instances. At the beginning of Algorithm 3, a snapshot of $blocked\_by$ is taken. However, this array is changing during runtime, so we recursively gather the free instances that are not blocked in the snapshot or still have the chance to be completed (line 16 - 23). This way, we might overestimate the

free instances, but more importantly do not miss any blocked instances, according to the following lemma. (All proofs can be found in the appendix of the paper).

Lemma 1. *For any partition $bft\_id \in \mathcal{B}$ determined by line 24 in Algorithm 3, the $blocked\_by[bft\_id]$ will keep the same status as in the snapshot and not change unless performing the cycle resolving.*

After knowing all blocked instances, the algorithm starts to detect cycles. It uses the linked list $seen$ to store requests blocking their instances. For a given element in $seen$, it has to wait until its next element finishes, before it can be executed. We have the following lemma to ensure that the linked list will encounter a loop in the end:

Lemma 2. *The while-loop starting at line 32 in Algorithm 3 will terminate.*

We also show that the next element is deterministically found:

Lemma 3. *In the cycle detection of Algorithm 3, line 36 will always deterministically find the next node for any node in $seen$. The next node is only determined by the ordering result of each partition, but irrelevant to race conditions at runtime.*

The last step is to resolve the cycle. We deterministically choose the request appearing at head of the instance with the smallest ID (line 44), and let it execute first. Because both cycle detection and resolution are deterministic and irrelevant to race conditions, we can conclude the following theorem:

Theorem 1. *Consistency between any non-faulty replicas can be guaranteed in Algorithm 3.*

C. Fault Handling

Now we discuss how to handle faults in SAREK. In general, we rely on two things: 1) The common fault handling mechanisms provided by the BFT protocol itself, and 2) additional measures for dealing with faulty behaviors of the predictor.

1) Fault Detection: In SAREK, besides the common faults addressed in any BFT system, replicas also have to face attacks on the predictor, which may cause the PREDICT() function to generate arbitrary results. SAREK hence makes additional efforts to efficiently prevent wrong prediction results from harming system safety. The detection scenarios can be briefly summarized as follows: 1) During agreement, after a replica calls the PREDICT() function, it sets a timer while waiting for the responsible replicas to start ordering the request. If a timeout triggers while the replica is still missing one or more sub-requests, a fault has likely been detected. 2) During execution, if a faulty replica intentionally fabricates a false "missing" partition access and attempts to cause a re-prediction, it is detected as a fault. Table I classifies the scenarios with various causes of faults as well as their consequences.

2) Fault Correction: Faulty replicas can deny to order a request, or concoct the ordering of an irrelevant request by manipulating a perfect (always accurate) PREDICT() function.

Algorithm 3 Detect and resolve cycle

```
1  initialize:
2      blocked_by_temp as a snapshot of the current blocked_by array
3      Node as a data structure defined as ⟨request, instance id⟩
4      seen := NULL // empty linked list indicating each Node seen so
           far is blocked by its successor
5      F := NULL // empty set to store free instances
6      B := NULL // empty set to store blocked instances
7  detect_and_resolve_cycle(self_id, self_req) do
8      //mark the partition itself also blocked
9      blocked_by_temp[self_id] := self_req
10     //determine blocked partitions
11     for bft_id from 0 to no_of_partitions − 1
12         if blocked_by_temp[bft_id] equals NULL
13             F.add(bft_id)
14         end if
15     end for
16     do
17         for all bft_id not in F
18             req := PartitionSchedules[bft_id].peek()
19             if req.partitions ∩ F not equals ∅
20                 F.add(bft_id)
21             end if
22         end for
23     while F is changed
24     B := {bft_id|bft_id ∉ F}
25     if B equals ∅
26         return FALSE
27     end if
28     //detect cycle
29     bft_id := min{bft_id|bft_id ∈ B}
30     current_node := Node(blocked_by_temp[bft_id], bft_id)
31     seen.append(current_node)
32     while TRUE do
33         current_req := current_node.req
34         next_id := min{bft_id|bft_id ∈ current_req.partitions
35             ∧blocked_by_temp[bft_id] not equals current_req}
36         next_node := Node(blocked_by_temp[next_id], next_id)
37         if seen.contains(next_node)
38             loop := the truncated part of seen starting at next_node
39             break
40         seen.append(next_node)
41         current_node := next_node
42     end while
43     //resolve cycle
44     req_to_move := arg min_req{bft_id|(req, bft_id) ∈ loop}
45     for all bft_id in req_to_move.partitions
46         move req_to_move to head in PartitionSchedules[bft_id]
47     end for
48     return TRUE
49 end
```

However, these behaviors do not affect non-faulty replicas in SAREK, as they all rely on their local PREDICT() function. Moreover, they will suspect the faulty leaders upon the denial and eventually enforce a view-change. Faked ordering proposals will be ignored since the non-faulty replicas stick to their own PREDICT() results.

If during the execution of a request a faulty replica fabricates a re-prediction to involve irrelevant instances to handle it, SAREK is able to detect this. Although a re-prediction is initiated locally, a false re-prediction proposal will never be approved as this step would require confirmations by $2f + 1$ replicas. Consequently, a false re-prediction will eventually be ignored by non-faulty replicas. This prevents a faulty replica from harming the system, and leads itself to being suspected. Details are explained in Section IV-D based on a generic

TABLE I: Fault causes and consequences.

Faults in SAREK		
Scenario	**Cause**	**Consequence**
During Agreement	1) **Missing sub-requests.** Slow or faulty replica fails to initiate ordering the sub-request in its leader instance(s).	Time-out triggers a view change.
	2) **Extra sub-requests.** Faulty replica manipulates prediction and fabricates a sub-request.	Faulty replica's behavior does not affect correct replicas and will eventually be suspected.
During Execution	3) **False re-prediction.** Faulty replica can intentionally cause a false re-prediction process.	A re-prediction process is processed locally by faulty replica but it cannot affect other replicas and will eventually be suspected.

approach to handling imprecise predictions. Once a faulty replica is under suspicion, non-faulty replicas start a view-change to derive its leader role upon a state partition. This requires modifications to the view-change operation as well as to checkpoint support. Details are explained in the following subsection.

3) Checkpoints: In single-leader BFT systems, checkpoints are generated locally without having an agreement in advance. This is because single-threaded processing and total order can guarantee that the same amount of requests correspond to the same set in the same sequence. Therefore, the creation of a checkpoint is triggered by a local counter and a distributed checkpoint certificate is collected to make the checkpoint stable ([4], [23], [24]). However, this is not applicable to SAREK, because 1) after executing the same amount of requests, replicas might be in different states and 2) checkpoints should be made at replica level rather than partition level because of cross-border requests. Hence a synchronization before collecting checkpoints is necessary.

SAREK performs this synchronization by introducing a special *create-checkpoint* request, which resembles a cross-border request that accesses all partitions. Each create-checkpoint request has a dedicated sequence number and is triggered by $2f + 1$ *pre-checkpoint* messages. Pre-checkpoint messages are sent by each BFT instance independently as shown in Algorithm 4. Every instance maintains a counter indicating how many (sub-)requests (i.e., EXECUTEs, CROSS-BORDER-EXECs, and CROSS-BORDER-SYNCs) it has processed since the last stable checkpoint. After a predefined threshold is reached, it broadcasts a pre-checkpoint message with the replica ID and the expected next sequence number. Algorithm 5 describes how the replica handles the pre-checkpoint message. The replica records the sequence number of the last stable checkpoint. If the received message has a stale sequence number, or has already been provided by the same replica (potentially by another BFT instance), it is discarded immediately (line 5). Otherwise the message is buffered. Once the replica has obtained $2f + 1$ pre-checkpoint messages with the same sequence number, it generates the create-checkpoint request and begins to order it in the same way as handling a client request accessing all partitions. The execution stage of the create-checkpoint request keeps unchanged as in Algorithm 2, and the actual execution of this request (line 12 of Algorithm 2) is as follows:

1) Create the checkpoint.
2) Every BFT instance resets the counter c to 0, and updates cp_next to $\max(cp\_next, s + 1)$.
3) The replica updates cp_stable to s.

This way, a stable checkpoint is consistently collected across all replicas. In order to update its state, a slow or recovering replica needs to obtain a stable checkpoint from another replica as well as f matching checkpoint hashes from additional replicas proving that the checkpoint is valid.

4) View Changes: The leader role of a faulty replica for a partition is revoked and granted to another replica through a view change. In existing BFT systems, the new leader is usually selected deterministically relying on a round-robin strategy. The challenge in SAREK is that after several view changes in different partitions, one replica might have to lead several partitions at a time, compromising load balancing and parallelism; in the worst case, if a single replica assumed the leader roles for all partitions, the system would degrade

Algorithm 4 Send pre-checkpoint at each instance

```
1  initialize:
2      interval as checkpoint interval
3      cp_next := 1
4      c := 0
5  while executing at instance bft_id of replica i do
6      if PartitionSchedules[bft_id].dequeue() is called in
           Algorithm 2
7          c := c + 1
8          if c mod interval equals 0
9              broadcast ⟨PRE-CHECKPOINT, i, cp_next⟩
10             cp_next := cp_next + 1
11         end if
12     end if
13 end while
```

Algorithm 5 Order create-checkpoint at each replica

```
1  initialize:
2      cp_stable := 0
3  upon receiving ⟨PRE-CHECKPOINT, rep_id, s⟩ at replica i do
4      if s ≤ cp_stable or already received the same message before
5          discard the message
6      else if received PRE-CHECKPOINT from 2f + 1 replicas with the
               same s
7          for each bft_id in all BFT instances
8              if i equals leaderOf(bft_id)
9                  start agreement stage of ⟨CREATE-CHECKPOINT, s⟩
10             end if
11         end for
12     end if
13 end
```

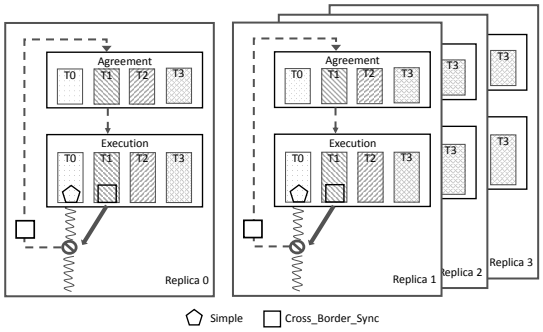

Fig. 5: Handle imprecise prediction with re-prediction.

to a single-leader BFT system. To avoid such scenarios and to keep the leader roles distributed, we define a *preferred leader* for each partition. This means that after a view change has occurred, SAREK executes the agreement protocol for a predefined number of requests using the new leader and then returns to the preferred leader. If the fault that has caused the previous view change is transient, the system sticks with the preferred leader. Otherwise, another view change is triggered. To prevent too frequent view changes, we add a switching penalty to the preferred leader after every view change, which increases the number of requests SAREK waits until returning to the preferred leader.

D. Handling Imprecise Predictions

SAREK can use imprecise application knowledge to make predictions in case implementing a perfect PREDICT() function is not feasible. In particular, the system is able to address *mis-predictions* caused by unforeseen data access patterns or internal state changes.

1) Re-predictions: In order to ensure consistency, SAREK monitors the state access of requests during execution. Once the execution instance observes a request attempting to access a partition that is not included in the prediction for this request, which indicates a mis-prediction, it immediately suspends the execution and prepares for a re-prediction. For this purpose, the execution instance adds the affected partition to the original request and sends the request as a *re-prediction proposal* to the responsible agreement instance at the local replica. Next, the agreement instance broadcasts the proposal to all other replicas to validate it. Having received $2f + 1$ matching re-prediction proposals, replicas are able to prove the correctness of re-prediction, and the leader replica starts ordering the proposal, as shown in Figure 5. The execution instance $T0$ sends a re-prediction proposal to the local responsible instance $T1$ after detecting a mis-prediction. Each $T1$ then broadcasts the proposal to all other replicas. With $2f+1$ matching proposals, the leader replica of $T1$ (assuming $Replica1$) can safely order the proposal.

Ordering the re-prediction proposal ensures that data access to the newly-predicted partition is consistent across all replicas. After the proposal is committed, it is classified as a CROSS-BORDER-SYNC and forwarded to the corresponding instance, which in turn notifies its execution instance at which the mis-prediction has been detected. Upon receiving such a notification, the execution instance resumes processing the request and accesses the affected partition, which is now safe due to the partition being blocked by the CROSS-BORDER-SYNC.

SAREK's mechanism for handling re-predictions guarantees that faulty replicas cannot trigger unnecessary re-predictions. Although a faulty replica is able to create a false re-prediction proposal, in the presence of at most f faulty replicas, it will not succeed in finding $2f$ other replicas that issue the same proposal. Consequently, the faulty proposal will never be committed by the agreement protocol.

2) Concurrent Re-predictions: As discussed in the previous section, when detecting a mis-prediction, SAREK suspends execution for the affected partition until the corresponding re-prediction proposal is committed. While handling a single re-prediction at a time is straightforward with this approach, dealing with concurrent re-predictions requires special attention, because without additional measures, re-predictions occurring in different partitions could introduce cyclic waiting (e.g., if a request $R1$ executing at partition P unpredictably wants to access partition P' and concurrently a request $R2$ executing at partition P' unpredictably wants to access partition P).

To address this issue, the partitions are constructed in the way that no such cyclic dependencies can happen. According to the distributed deadlock prevention principle [26], we utilize the prioritizing mechanism (the same as in handling cross-border requests) together with the application-specific knowledge to fulfill the following requirement: A re-prediction can only access partitions $P_i > P_{max}$ with P_{max} being the highest partition included in any previous (re-)prediction of the same request. As a result, there can no longer be cycles between concurrent re-prediction proposals (e.g., if $P < P'$ in the example above, the first prediction for both requests must be P, which would automatically lead to the requests being serialized by SAREK.)

Note that the constraint discussed above only applies to use cases in which mis-predictions can actually occur due to the PREDICT() function not being completely accurate. A practical approach to fulfill the requirement in such cases without having to conduct an extensive application analysis is to implement the PREDICT() function conservatively, that is, to include additional partitions in the initial prediction.

V. IMPLEMENTATION AND EVALUATION

In this section, we evaluate the performance of SAREK. A comparison of a single-leader approach with its SAREK-adapted multi-leader version is made to show the advantage of state partitioning and parallel agreement.

A. System Setup

We use a cluster of four machines to host the replicas and one dedicated machine to simulate clients. Each physical machine is equipped with an Intel i7-4770 (quad-core with hyper-threading) CPU and 16 GB of main memory. All machines are

running Ubuntu Linux Server 14.04 64-bit and are connected via switched 1 Gb/s Ethernet.

We built our SAREK prototype based on a PBFT implementation provided by a BFT library [27] written in Java, which follows the traditional single-leader design. The parallelism of BFT instances in SAREK is done by instantiating the PBFT protocol engine multiple times. This way, the same code base can be used for all experiments, and the switching between the original system and SAREK is only a matter of configuration. SAREK itself introduces only about 900 lines of code, including the implementations of predictor, multi-leader agreement handler with executors and re-prediction scheme. Altogether they increase the code base by only 1/7.

To evaluate and compare the performance, we have utilized two benchmarks: 1) A microbenchmark deploying a key-value store application and 2) YCSB (Yahoo! Cloud Serving Benchmark) [10] with a customized database server. In the following, the single-leader approach is referred to as "baseline (BL)".

B. Microbenchmark Setup

For the microbenchmark experiments, we use a hash-map-based data store to evaluate the performance (i.e., latency and throughput) and resource demand (i.e., CPU usage) of SAREK for different workloads. The data store implements the following functionalities: 1) It accepts each client request that contains specific keys accessing one or multiple objects and 2) generates and returns a reply to the request.

The sizes of request and reply messages vary depending on the operations performed. For the $get()$ operation, which returns the data to a specified key, requests are usually smaller than replies. In contrast, storing data with the $put()$ operation in general involves requests that are larger than their corresponding replies. Additionally, the data store provides a $putall()$ operation, which combines multiple updates in one request and enables to test the handling of cross-border requests in SAREK.

For SAREK, we divide the state into four partitions of approximately equal sizes. The actual partitioning is done with a PREDICT() function that performs a $modulo$ operation on each key to determine the ID of the responsible BFT agreement instance. For all experiments we deploy up to 200 clients, thereby saturating the system, and calculate the average of multiple independent runs for results.

C. Microbenchmark Results

For an initial throughput evaluation, we use three combinations of different request/reply sizes: 50 bytes/500 bytes, 500 bytes/50 bytes and 500 bytes/500 bytes. Accordingly, these experiments provide insights about the performance of read-heavy ($get()$), write-heavy ($put()$), and mixed workloads, respectively. This scenario only includes simple requests, cross-border requests are further investigated later. Figure 6 shows that the throughput maximum of the baseline system is less than 20,000 requests/s. In contrast, the maximum throughput achieved by SAREK is about twice as high. It also

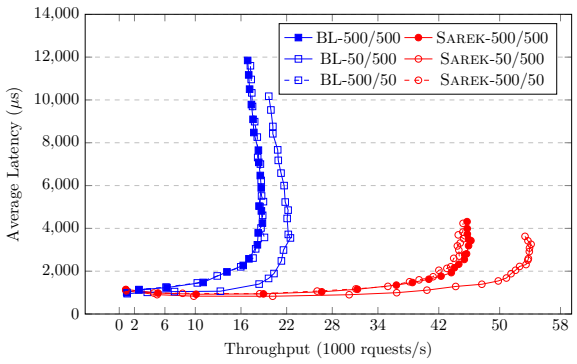

Fig. 6: Throughput and latency of simple requests.

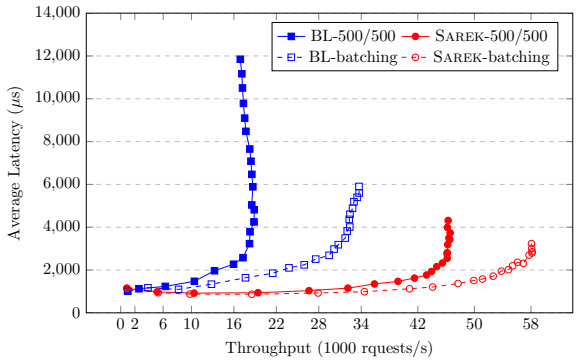

Fig. 7: Throughput and latency with batching.

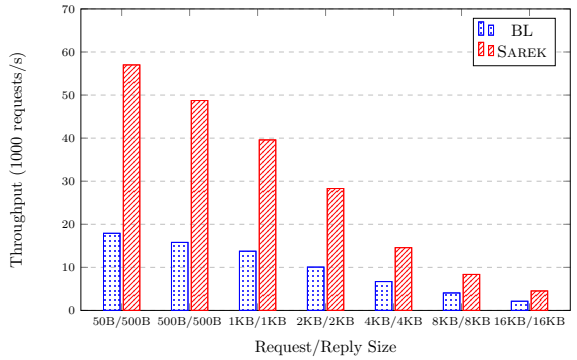

Fig. 8: Throughput with increased request/reply sizes.

shows that in both systems, smaller request size leads to higher throughput whereas reply size has little effect.

In our second experiment, we evaluate the effects of request batching, a common optimization used in BFT systems to minimize the agreement overhead by ordering request batches instead of individual requests [4]. The maximum batch size is set to 100 and the experiment is performed with a request/reply size of 500 bytes/500 bytes. Figure 7 indicates that the use of request batching results in a significant throughput increase in both systems, while the leading position of SAREK remains the same by nearly a factor of two. Therefore, we conclude that in

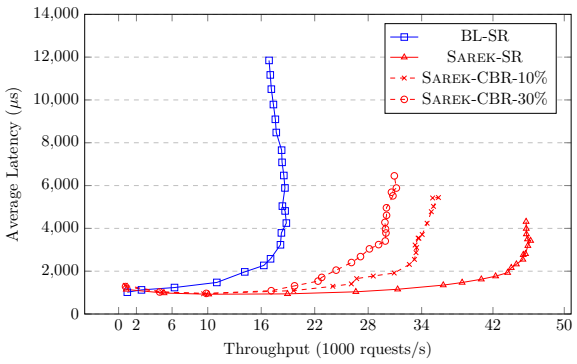

Fig. 9: Throughput and latency with cross-border requests.

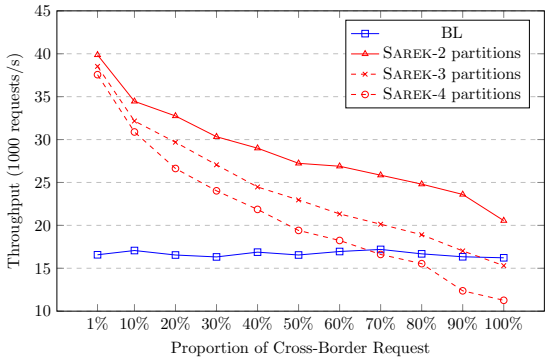

Fig. 10: Throughput in the presence of cross-border requests.

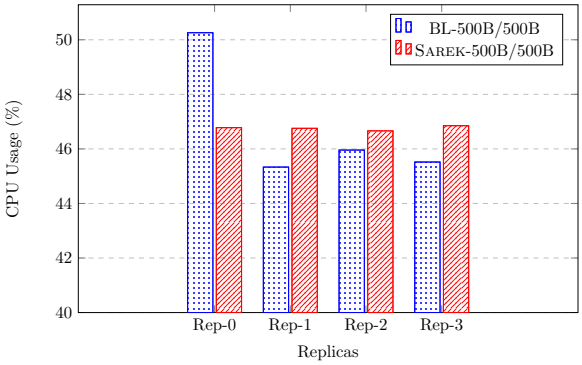

Fig. 11: CPU usage.

essence batching can be considered an orthogonal technique to be used in combination with SAREK.

Next, we measure the impact of increased request/reply sizes on system throughput. Figure 8 shows that the throughput of the baseline system is limited by the restricted processing capability of the leader, whereas SAREK can increase throughput to almost three times until the bandwidth limit is reached.

As an essential feature of SAREK, performance of handling cross-border requests is also of key interest. Figure 9 reflects the impact of different amounts of cross-border requests on system throughput. Although for 10% and 30% cross-border requests the throughput of SAREK decreases by 20% and 30%, respectively, it still performs better than the baseline system. Moreover, even with 30% cross-border requests, SAREK is still able to provide low latency.

In the next experiment, we expose SAREK to an extreme condition by increasing the proportion of cross-border request to 100% in three settings: a request accessing two, three, and four partitions. Figure 10 shows that, having 100% cross-border requests, SAREK still wins for requests accessing two partitions. When accessing three and four partitions, turning points are reached at about 90% and 70% cross-border requests, accordingly. This is due to the overhead of cross-border requests being ordered multiple times during agreement. Such harsh condition indicates that either the majority of requests are highly depending on each other, thus parallelism can hardly be increased, or the implemented partitioning scheme needs to be revised.

Figure 11 details the CPU usage at the peak throughput of the baseline system and proves the load balancing feature of SAREK. While a SAREK replica utilizes 1.5% more CPU than a follower replica in the baseline system, the load in SAREK is distributed equally across all replicas when reaching the same throughput as the baseline system.

D. YCSB Benchmark Setup

The Yahoo! Cloud Serving Benchmark (YCSB) [10] is a program suite for measuring the performance of NoSQL database systems, which we use to further evaluate SAREK. For this purpose, we created a new key-value store server by extending the `DB` class of the YCSB library in order to utilize its benchmarking APIs at the client side. This server provides the following operations: $insert()$ creates a new record, $read()$ retrieves an existing record, $update()$ modifies an existing record, and $scan()$ executes a range scan over a specified number of records. For each experiment, we initialize the data store by inserting 4,000 records and afterwards run one of the predefined YCSB workloads: $read/update$ or $scan/update$. At the client side, 60 client instances are generated.

The state partitioning mechanism of this application remains simple as $insert()$, $read()$, and $update()$ all access single records. In contrast, $scan()$ typically accesses multiple records and consequently can cause cross-border requests. Therefore, we partition the entire key space of the data store into four continuous segments so that each segment contains a set of consecutive keys. Depending on the density of key distribution in each partition and the defined scan range, a $scan()$ operation may lead to handling cross-border requests accessing partitions in a monotonic order, as described in Section IV-D.

1) Read/Update Workload: For this workload, a read/update ratio of 50/50 is used to prove the capability of handling an update-heavy workload, where each client instance issues 200,000 operations.

2) Scan/Update Workload: For the scan/update workload, the proportion of $scan()$ operation ranges from 5% to 25%, and the traffic is generated by 100,000 operations per client

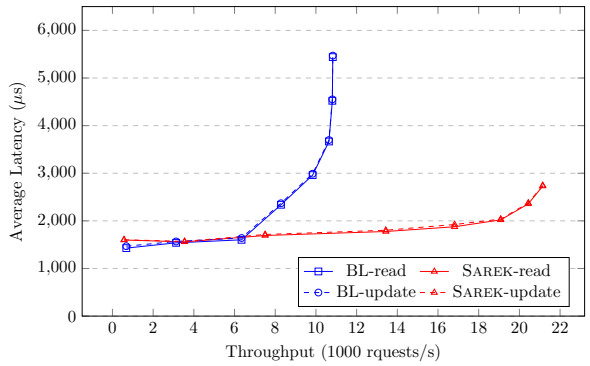

Fig. 12: Throughput and latency of read/update workload.

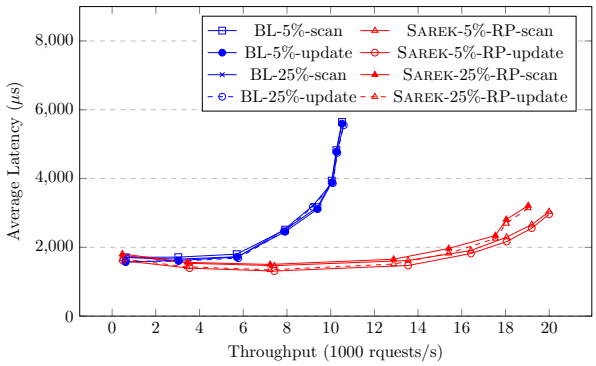

Fig. 13: Throughput and latency of scan/update workload.

instance. The maximum scan range is set to 10 keys due to the fact that the key distribution in the entire key space is rather sparse with 4,000 initial records. The $scan()$ operation particularly provides a range query starting from a given key associated with a scan range. Note that since only the start key is visible to the PREDICT() function, depending on where the start key locates in one partition, it is highly possible that a $scan()$ operation accesses multiple partitions. Once this happens, our re-prediction mechanism is activated to handle the mis-prediction case.

E. YCSB Benchmark Results

The results of the YCSB benchmark indicate good performance for SAREK regarding throughput and latency. The measurement results are analyzed separately for each workload.

1) Read/Update Workload: Similar to the microbenchmark, Figure 12 indicates that SAREK performs better than the baseline system, providing twice the maximum throughput at significantly lower latency. Note that as we did not activate any read optimizations in SAREK, the average latency of $read()$ is not significantly smaller than the latency of $update()$, as the agreement protocol is also executed for reads. Unlike the baseline system where the request latency dramatically increases for throughputs higher than about 6,000 requests/s, SAREK keeps its low latency until almost 20,000 requests/s.

2) Scan/Update Workload: The scan/update workload of the YCSB benchmark leads to larger reply sizes and is challenging for SAREK due to including cross-border requests. We set the $scan()$ operation proportions to be 5%, 15%, and 25%, as shown in Figure 13. The baseline system reaches its peak throughput at around 10,000 requests/s, while SAREK achieves a maximum throughput that is twice as high. Latencies in SAREK for both $scan()$ and $update()$ are much smaller than the respective latencies provided by the baseline system for these operations. As the results for the experiment with 25% $scan()$ operations show, the overhead for handling mis-predictions in SAREK increases latency only slightly.

VI. CONCLUSION

We presented SAREK, a parallel ordering framework for BFT systems that relies on multiple leaders to improve performance. In contrast to single-leader approaches, SAREK distinguishes requests by their data access patterns and establishes a partial order on dependent requests. This is achieved by splitting the service state into partitions and managing each of them in an independent BFT agreement instance. To distribute the agreement load, under fault-free conditions the leader roles of all BFT agreement instances are evenly balanced over all replicas. Furthermore, SAREK provides means to effectively deal with requests that need to access multiple partitions and situations where data access patterns are hard to predict or might even be mis-predicted. The conducted microbenchmarks as well as the YCSB benchmark show a throughput increase of up to a factor of two. Even in the case of cross-border requests accessing multiple partitions, which requires coordination among the involved BFT agreement instances, a moderate throughput increase can be achieved.

REFERENCES

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, 1990.

[2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[3] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proc. of the 7th annual ACM Symp. on Principles of distributed computing*. ACM, 1988, pp. 8–17.

[4] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation (OSDI '99)*, 1999, pp. 173–186.

[5] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *Proc. of the 2004 Int'l Conf. on Dependable Systems and Networks (DSN '04)*. IEEE, 2004, pp. 575–584.

[6] T. Distler and R. Kapitza, "Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency," in *Proc. of the 6th ACM European Conf. on Computer Systems (EuroSys '11)*. ACM, 2011, pp. 91–105.

[7] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin *et al.*, "All about eve: Execute-verify replication for multi-core servers." in *OSDI*, 2012, pp. 237–250.

[8] G. S. Veronese, M. Correia, A. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *Proc. of the 28th IEEE Int'l Symp. on Reliable Distributed Systems (SRDS '09)*. IEEE, 2009, pp. 135–144.

[9] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "EBAWA: Efficient Byzantine agreement for wide-area networks," in *Proc. of the 12th Symp. on High-Assurance Systems Engineering (HASE '10)*. IEEE, 2010, pp. 10–19.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. of the 1st ACM Symp. on Cloud Computing*. ACM, 2010, pp. 143–154.
[11] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *ACM SIGOPS Operating Systems Review*. ACM, 2003, pp. 253–267.
[12] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults." in *NSDI*, 2009, pp. 153–168.
[13] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "Rbft: Redundant byzantine fault tolerance," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd Int'l Conf. on*. IEEE, 2013, pp. 297–306.
[14] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 1–14, 2002.
[15] L. Lamport, "Generalized consensus and paxos," Technical Report MSR-TR-2005-33, Microsoft Research, Tech. Rep., 2005.
[16] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. of the 24th ACM Symp. on Operating Systems Principles*. ACM, 2013, pp. 358–372.
[17] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *Proc. of the 34th Int'l Conf. on Distributed Computing Systems*. IEEE, 2014, pp. 368–377.
[18] P. J. Marandi and F. Pedone, "Optimistic parallel state-machine replication," in *Proc. of the 33rd Int'l Symp. on Reliable Distributed Systems*, 2014, pp. 57–66.
[19] C. E. Bezerra, F. Pedone, and R. Van Renesse, "Scalable state-machine replication," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP Int'l Conf. on*. IEEE, 2014, pp. 331–342.
[20] J. Behl, T. Distler, and R. Kapitza, "Consensus-oriented parallelization: How to earn your first million," in *Proc. of the 16th Middleware Conference (Middleware '15)*. ACM, 2015, pp. 173–184.
[21] J. Cowling and B. Liskov, "Granola: low-overhead distributed transaction coordination," in *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 223–235.
[22] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data*. ACM, 2012, pp. 1–12.
[23] M. Castro and B. Liskov, "Proactive recovery in a byzantine-fault-tolerant system," in *Proc. of the 4th Conf. on Symp. on Operating System Design & Implementation-Volume 4*. USENIX Association, 2000, pp. 19–19.
[24] R. Rodrigues, M. Castro, and B. Liskov, "Base: Using abstraction to improve fault tolerance," in *ACM SIGOPS Operating Systems Review*. ACM, 2001, pp. 15–28.
[25] R. Kapitza, M. Schunter, C. Cachin, K. Stengel, and T. Distler, "Storyboard: Optimistic deterministic multithreading," in *Proc. of the 6th Workshop on Hot Topics in System Dependability (HotDep '10)*. USENIX, 2010, pp. 1–8.
[26] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice hall Englewood Cliffs, 2002, vol. 2.
[27] T. Distler, C. Cachin, and R. Kapitza, "Resource-efficient Byzantine fault tolerance," *IEEE Transactions on Computers*, 2015.

APPENDIX

A. Proof of Lemma 1

Proof. If an item in array $blocked\_by$ is changed from $NULL$ to some request, the corresponding instance will wait for the notification. The item can only be set back to $NULL$ by another non-blocked partition (line 17 and 31), and then the corresponding instance is notified. If line 24 has determined any $bft\_id \in \mathcal{B}$, it must be waiting at the time of snapshot. All other instances which have the potential to change and notify $blocked\_by[bft\_id]$ are also waiting at that time. Thus $blocked\_by[bft\_id]$ will not be changed. And the instance bft_id will wait until a cycle resolving is performed. □

B. Proof of Lemma 2

Proof. Each node in $seen$ can be uniquely identified by its partition bft_id. So the number of nodes in $seen$ is no more than the number of partitions (in fact, no more than $|\mathcal{B}|$). That means the while-loop cannot add infinite nodes into $seen$, and will eventually terminate. □

C. Proof of Lemma 3

Proof. Whether a request can be successfully executed without any cycles, is irrelevant to the runtime race conditions but only depending on the ordering result. Only nodes that will eventually get into a cycle can be added into $seen$. Assume a node (req, bft_id) is added into $seen$. In all partitions in $req.partitions$, either req is blocking at the head, or another request is blocking at the head. In the former case, every request in front of req in the partition can be successfully executed without any cycles. In the latter case, the blocking request is the first request getting into a cycle in that partition. These only depend on the partition schedule and are irrelevant to the race conditions. And in at least one partition req is not at the head, otherwise req can be executed immediately, which violates Lemma 1. So line 34 - 36 will find the next node deterministically. □

D. Proof of Theorem 1

Firstly we have the following corollary directly from lemma 2 and lemma 3:

Corollary 1. *Assume two replicas have the identical request orders in all partitions at a certain time and there is no cycle resolving since then. If both have added the same node into $seen$ during Algorithm 3, then in the end they will have the same truncated linked list $loop$ (they might start from different nodes, but if we link the last node to the first node in $loop$, they will be the same "loop").*

Then we can make sure if two replicas have detected the same request cycle, they will move the same request to the head (line 43) to resolve the cycle. So we have the following lemma:

Lemma 4. *Assume $Replica0$ and $Replica1$ have the identical request orders in all partitions at a certain time and there is no cycle resolving since then. If a request in $Replica0$ occurs in a cycle and moves to head by Algorithm 3 and gets executed, then it will be moved in the same way in $Replica1$ before it can be executed.*

Proof. Because the requests are ordered in the same way in all partitions of both replicas, if one request is stuck in a cycle in one replica, so will it in the other replica. According to Corollary 1, if one replica detects the cycle and resolves it, the other replica will do it in the same way. □

Finally we can prove that if a sequence of cycle resolving procedures affecting the same partition happens in one replica, the same sequence also happens in the other replica. As a result, the relation between any two requests stays the same in different replicas, so the system consistency is guaranteed.