DynaStar: Optimized Dynamic Partitioning for Scalable State Machine Replication

Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé and Fernando Pedone
Università della Svizzera italiana (USI)
Lugano, Switzerland

***Abstract*—Classic state machine replication (SMR) does not scale well, since each replica must execute every command. To address this problem, several systems have investigated the use of state partitioning in the context of SMR, allowing client commands to be executed on a subset of replicas. Prior approaches range from completely static schemes, which do not adapt as workloads change, to dynamic schemes, which move data on-demand.**

This paper presents DynaStar, a new dynamic partitioning scheme for scaling state machine replication. In contrast to prior dynamic schemes, DynaStar uses a replicated location oracle to maintain a global view of the workload and inform heuristics about data placement. Using this oracle, DynaStar can adapt to workload changes over time, while also minimizing the number of state moves. The result is a practical technique that achieves excellent performance.

***Keywords*-state machine replication; multi partition command; state partitioning**

I. Introduction

State machine replication (SMR) is a fundamental technique for building fault-tolerant services. With SMR, state is replicated on a set of servers and each replica deterministically executes the same sequence of client commands to maintain consistency [25], [37]. SMR provides configurable fault tolerance (i.e., by setting the number of replicas) but limited performance since all replicas must execute all commands. In order to provide both configurable fault tolerance and scalable performance, some recent proposals have extended state machine replication with sharding (e.g., [2], [5], [11], [17], [29]). Essentially, these approaches partition (shard) the service state and replicate each partition. Commands that access state in a single partition are handled as in classic state machine replication. Different techniques have been proposed to handle commands that access state in multiple partitions, but inherently multi-partition commands are more expensive than single-partition commands. This happens due to overhead from ordering and coordinating commands across partitions to ensure strong consistency. Moreover, if data is not distributed carefully, then load imbalances can nullify the benefits of partitioning. Consequently, the Achilles heel of scalable state machine replication lies on the partitioning of the service state. An ideal partitioning scheme is one that would both (i) allow commands to be executed at a single partition only, and (ii) evenly distribute data so that load is balanced among partitions. Under such conditions, performance will scale with the number of partitions.

Partitioning the service state in order to achieve scalable performance, however, is challenging. First, it depends on a good understanding of the workload, which may not be available. Second, even if enough information is available, finding a good partitioning is a complex optimization problem [12], [41]. Third, workloads may vary over time. In social networks, for example, some users may experience a surge increase in their number of followers (e.g., new “celebrities”); workload demand may shift along the hours of the day and the days of the week; and unexpected (e.g., a video that goes viral) or planned events (e.g., a new company starts trading in the stock exchange) may lead to exceptional periods when requests increase significantly higher than in normal periods. These challenges perhaps explain why most approaches that extend SMR with state partitioning delegate the task of partitioning the service state to the application designer (e.g., [2], [5], [11], [29]).

In this paper, we introduce DynaStar, a new approach to scalable state machine replication. DynaStar differs from previous solutions in that it supports *dynamic* state partitioning. This means that data can be relocated from one partition to another to optimize performance. Moreover, data relocation is done on-the-fly, with minimal service disruption. DynaStar does not require any a priori information about the frequency of commands and can adapt to workload variations. To achieve this design, DynaStar maintains a location oracle with a global view of the application state. The oracle minimizes the number of state relocations by monitoring the workload, and re-computing an optimized partitioning on demand using a partitioning algorithm. The location oracle maintains two data structures: (i) a mapping of application state variables to partitions, and (ii) a *workload graph* with state variables as vertices and edges as commands that access the variables. Before a client submits a command, it contacts the location oracle to discover the partitions on which state variables are stored. If the command accesses variables in multiple partitions, the oracle chooses one partition to execute the command and instructs the other involved partitions to temporarily relocate the needed variables to the chosen partition. Of course, when relocating a variable, the oracle is faced with a choice of which partition to use as a destination. DynaStar chooses the partition for relocation by partitioning the workload graph using the METIS [1] partitioner and selecting the partition that would minimize the number of state relocations.

To tolerate failures, DynaStar implements the oracle as a regular partition, replicated in a set of nodes. To ensure that the oracle does not become a performance bottleneck, clients cache location information. Therefore, clients only query the oracle when they access a variable for the first time or when cached entries become invalid (i.e., because a variable changed location). DynaStar copes with commands addressed to wrong partitions by telling clients to retry a command.

We implemented DynaStar and compared its performance to alternative schemes, including an idealized approach that knows the workload ahead of time. Although this scheme is not achievable in practice, it provides an interesting baseline to compare against. DynaStar's performance rivals that of the idealized scheme, while having no a priori knowledge of the workload. We show that DynaStar largely outperforms existing dynamic schemes under two representative workloads, the TPC-C benchmark and a social network service. We also show that the location oracle never becomes a bottleneck.

In summary, the paper makes the following contributions:

- It introduces DynaStar and discusses its implementation.
- It evaluates different partitioning schemes for state machine replication under a variety of conditions.
- It presents a detailed experimental evaluation of DynaStar using the TPC-C benchmark and a social network service populated with a real social network graph that contains half a million users and 14 million edges.

The rest of the paper is structured as follows. Section II presents the system model. Section III reviews existing scalable state machine replication approaches. Section IV introduces DynaStar and describes the technique in detail. Section V overviews our prototype. Section VI reports on the results of our experiments. Section VII surveys related work and Section VIII concludes the paper.

II. System model and definitions

DynaStar relies on multicast abstractions to handle complex coordination among partitions without violating correctness. In this section, we detail the system model, define multicast, and state our correctness criterion.

A. Processes and communication

We consider a message-passing distributed system consisting of an unbounded set of client processes $\mathcal{C} = \{c_1, c_2, ...\}$ and a bounded set of server processes (replicas) $\mathcal{S} = \{s_1, ..., s_n\}$. Set $\mathcal{S}$ is divided into disjoint groups of servers $\mathcal{S}_0, ..., \mathcal{S}_k$. Processes are either *correct*, if they never fail, or *faulty*, otherwise. In either case, processes do not experience arbitrary behavior (i.e., no Byzantine failures).

The system is partially synchronous [15], that is, it is initially asynchronous and eventually it becomes synchronous. The time when the system becomes synchronous is called the Global Stabilization Time (GST) and it is unknown to the processes. Before GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After GST, such bounds exist but are unknown.

Processes communicate using either one-to-one or one-to-many communication. One-to-one communication uses primitives $send(p, m)$ and $receive(m)$, where m is a message and p is the process m is addressed to. If sender and receiver are correct, then every message sent is eventually received. One-to-many communication relies on reliable multicast and atomic multicast, defined in §II-B and §II-C, respectively.

B. Reliable multicast

To reliably multicast a message m to a set of groups γ, processes use primitive r-mcast(γ, m). Message m is delivered at the destinations with r-deliver(m). Reliable multicast has the following properties:

– If a correct process r-mcasts m, then every correct process in γ r-delivers m *(validity)*.
– If a correct process r-delivers m, then every correct process in γ r-delivers m *(agreement)*.
– For any message m, every process p in γ r-delivers m at most once, and only if some process has r-mcast m to γ previously *(integrity)*.

C. Atomic multicast

To atomically multicast a message m to a set of groups γ, processes use primitive a-mcast(γ, m). Message m is delivered at the destinations with a-deliver(m). We define delivery order $<$ as follows: $m < m'$ iff there exists a process that delivers m before m'.

Atomic multicast ensures the following properties:

– If a correct process a-mcasts m, every correct process in a group in γ a-delivers m *(validity)*.
– If a process a-delivers m, then every correct process in a group in γ a-delivers m *(uniform agreement)*.
– For any message m, every process a-delivers m at most once, and only if some process has a-mcast m previously *(integrity)*.
– If a process a-mcasts m and then m', then no process a-delivers m' before m *(fifo order)*.
– The delivery order is acyclic *(atomic order)*.
– For any messages m and m' and any processes p and q such that $p \in g$, $q \in h$ and $\{g, h\} \subseteq \gamma$, if p delivers m and q delivers m', then either p delivers m' before m or q delivers m before m' *(prefix order)*.

Atomic broadcast is a special case of atomic multicast in which there is a single group of processes.

D. Correctness criterion

Our consistency criterion is linearizability. A system is *linearizable* if client commands can be reordered in a sequence that (i) respects the semantics of the commands, as defined in their sequential specifications, and (ii) respects the real-time precedence of commands [3].

III. BACKGROUND

DynaStar builds on prior work on classic state machine replication and scalable state machine replication. In this section, we briefly review these techniques.

A. State machine replication

State machine replication is a fundamental approach to fault tolerance. It replicates servers and coordinates the execution of client commands at replicas [25], [37]. Every replica has a full copy of the service state, identified by a set of state variables $\mathcal{V} = \{v_1, ..., v_m\}$. A command is a sequence of deterministic operations that can read and modify the state. By starting in the same initial state and executing the same sequence of deterministic commands in the same order, servers make the same state changes and produce the same reply for each command. To guarantee that servers deliver the same sequence of commands, SMR can be implemented with atomic broadcast: commands are atomically broadcast to all servers, and all correct servers deliver and execute the same sequence of commands [8], [13].

B. Scaling State Machine Replication

Despite its simple execution model, classic SMR does not scale; adding resources (e.g., replicas) will not translate into sustainable improvements in throughput. Several approaches have been proposed to address SMR's scalability limitations. In all proposed protocols, the idea is to shard the service's state and replicate each shard. More precisely, the service state $\mathcal{V}$ is composed of k partitions, $\mathcal{P}_1, ..., \mathcal{P}_k$, where each partition $\mathcal{P}_i$ is assigned to server group $\mathcal{S}_i$. (For brevity, we say that server s belongs to $\mathcal{P}_i$ meaning that $s \in \mathcal{S}_i$.) Commands that access a single partition are executed as in classic SMR. Existing protocols differ in (a) how to handle multi-partition commands and (b) whether state partitioning is static or dynamic.

There are three approaches to handling multi-partition commands. Some protocols assume a workload that can be "perfectly partitioned," that is, there is a way to shard the service state that avoids multi-partition commands [27], [33]. Since commands are essentially single-partition, if the load is balanced across partitions then performance scales with the number of partitions. With protocols that support commands that span multiple partitions, the multi-partition command is first ordered across the involved partitions (e.g., using an atomic multicast) and then executed by each involved partition. Some protocols assume the workload can be sharded such that multi-partition commands are executed locally by each of the involved partitions [32], that is, the execution of a multi-partition command in one partition does not need data stored in a different partition. For example, to support the assignment command "$x := y$", variables x and y must be placed in the same partition. However, a command that increments x and increments y allows these variables to be placed in different partitions.

More general solutions do not impose restrictions on state partitioning, but incur additional overhead in the execution of multi-partition commands [5], [27]. Upon delivering a multi-partition command, replicas may need to exchange state, since some partitions may not have all the variables needed in the command. For example, consider again the assignment command "$x := y$" in a system where x and y are stored in different partitions. With S-SMR [5], the partitions first exchange x and y and then both execute the command. DS-SMR [27] supports dynamic partitioning of state; thus, either x is moved to the partition that contains y or vice-versa, and the command is executed at a single partition.

None of the existing approaches discuss how to partition the state of a general service in order to minimize multi-partition commands and keep the load across partitions balanced. DS-SMR [27] achieves these goals for services that can be perfectly partitioned. In the next section, we introduce DynaStar, a protocol that minimizes multi-partition commands and balances partition load without these restrictions.

IV. DynaStar

DynaStar defines a dynamic mapping of application variables to partitions. Application programmers may also define other granularity of data when mapping application state to partitions. For example, in our social network application (§V-D), each user (together with the information associated with the user) is mapped to a partition; in our TPC-C implementation (§V-C), every district in a warehouse is mapped to a partition. Such a mapping is managed by a partitioning oracle, which is handled as a replicated partition. The oracle allows the mapping of variables to partitions to be retrieved or changed during execution. To simplify the discussion, in §IV-A–IV-B we initially assume that every command involves the oracle. In §IV-C, we explain how clients can use a cache to avoid the oracle in the execution of most commands.

A. Overview

Clients submit commands to the oracle and wait for the reply. DynaStar supports three types of commands: $create(v)$ creates a new variable v and initially maps it to a partition defined by the oracle; $access(\omega)$ is an application command that reads and modifies variables in set $\omega \subseteq \mathcal{V}$; and $delete(v)$ removes v from the service state. The reply from the oracle is called a $prophecy$, and usually consists of a set of tuples $\langle v, \mathcal{P} \rangle$, meaning $v \in \mathcal{P}$, and a target partition $\mathcal{P}_d$ on which the command will be executed. The $prophecy$ could also tell the clients if a command cannot be executed (e.g., it accesses variables that do not exist). If the command can be executed, the client waits for the reply from the target partition.

If a command C accesses variables in ω on a single partition, the oracle multicasts C to that partition for execution. If the command accesses variables on multiple partitions, the oracle multicasts a $global(\omega, \mathcal{P}_d, C)$ command to the involved partitions to gather all variables in ω to the target partition $\mathcal{P}_d$. After having all required variables, the target partition executes command C, sends the reply to the client, and returns the variables to their source.

The oracle also collects hints from clients and partitions to build up a workload graph and monitors the changes in the graph. In the workload graph, vertices represent state variables and edges dependencies between variables. An edge connects two variables in the graph if a command accesses both of them. Periodically, the oracle computes a new optimized partitioning and sends the partitioning plan to all partitions. Upon delivering the new partitioning, the partitions exchange variables and update their state accordingly. DynaStar relocates variables without blocking the execution of commands.

B. The detailed protocol

Algorithms 1, 2, and 3 describe the client, oracle, and server processes, respectively. We omit the delete command since the coordination involved in the create and delete commands are analogous.

1) The client process: To execute a command C, the client atomically multicasts C to the oracle (Algorithm 1). The oracle replies with a prophecy, which may already tell the client that C cannot be executed (e.g., it needs a variable that does not exist, it tries to create a variable that already exists). If C can be executed, the client receives a prophecy containing the partition where C will be executed. The client then waits for the result of the execution of C.

2) The oracle: When the oracle delivers a request, it distinguishes between two cases (Task 1 in Algorithm 2).

- If the command is to create a variable v, and v does not already exist, the oracle chooses a random partition for v, multicasts the create command to the partition and itself, and returns the partition to the client as a prophecy (Figure 1).
- If the command reads and writes existing variables, the oracle first checks that all such variables exist. If the variables exist and they are all in a single partition, the oracle multicasts the command to that partition for execution. If the variables are distributed in multiple partitions, the oracle deterministically determines the destination partition, and atomically multicasts a command to the involved partitions so that all variables are gathered at the destination partition. The oracle chooses as the destination partition the partition that contains most of the variables needed by the command. (In case of a tie, one partition is chosen deterministically, among those that contain most variables.) Once the destination partition has received all variables needed by the command, it executes the command and returns the variables to their source partition.

Algorithm 1 Client

```
1: To issue a command $C$, the client does:
2:    a-mcast(oracle, $exec(C)$)
3:    wait for $prophecy$
4:    if $prophecy = nok$ then                  {if receive nok then...}
5:        $reply \leftarrow prophecy$            {...there's nothing to execute}
6:    else                       {in this case, prophecy is ($dest$)}
7:        wait for $reponse$ from a server in $prophecy$
8:        $reply \leftarrow reponse$
9:    return $reply$ to the application
```

Upon delivering a create (Task 2), the oracle updates its partition information. As part of a create command, the oracle coordinates with the partition to ensure correctness (Task 3) [5]. The oracle also keeps track of the

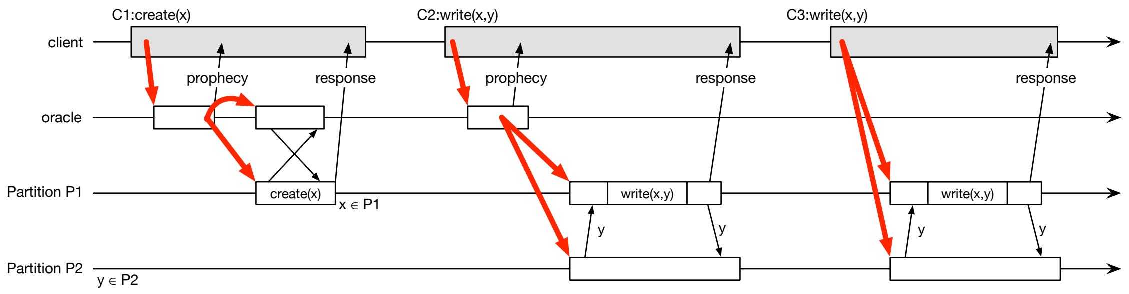


Figure 1: Execution of a create (C1) and a write without client cache (C2) and with client cache (C3) in DynaStar.

Algorithm 2 Oracle

```
when a-deliver($exec(C)$)                                        {Task 1}
  case $C$ is a $create(v)$ command:
    if $partition(\{v\}) \neq \bot$ then                   {if $v$ already exists...}
      $prophecy \leftarrow nok$                                 {...notify client}
    else                                                  {if $v$ doesn't exist...}
      $\mathcal{P} \leftarrow$ choose $v$'s partition
      $prophecy \leftarrow \mathcal{P}$                  {prepare client's response}
      a-mcast($\{oracle, \mathcal{P}\}, (\mathcal{P}, create(v))$)
  case $C$ is any command, but $create(v)$:
    $\omega \leftarrow vars(C)$                        {variables accessed by $C$}
    if $\exists v \in \omega : partition(\{v\}) = \bot$ then     {if $v$ ¬exists:}
      $prophecy \leftarrow nok$                                   {tell the client}
    else                                            {if all vars in $\omega$ exist}
      $dests \leftarrow partition(\omega)$          {get all partition involved}
      $\mathcal{P}_d \leftarrow target(dests, C)$         {$\mathcal{P}_d$ will excecute $C$}
      a-mcast($dests, C$)
      $prophecy \leftarrow \mathcal{P}_d$
  send $prophecy$ to the client
when a-deliver($\langle \mathcal{P}_v, create(v) \rangle$)                   {Task 2}
  $\forall x \in \mathcal{P}_v$: r-mcast($x, \langle signal, create(v) \rangle$)        {exchange signal...}
  wait until $\langle signal, create(v) \rangle \in rcvd\_msgs$          {...to coordinate}
  $\mathcal{P}_v \leftarrow \mathcal{P}_v \cup \{v\}$
when r-deliver ($\langle val, C \rangle$)                               {Task 3}
  $rcvd\_msgs \leftarrow rcvd\_msgs \cup \{\langle val, C \rangle\}$
when a-deliver($hint(V_h, E_h)$)                                 {Task 4}
  update $G_W$ with $(V_h, E_h)$
  $inc(changes)$
  if $changes \geq threshold$ then
    $partitioning \leftarrow$ compute $\mathcal{I}_1, ..., \mathcal{I}_m$ from $G_W$
    a-mcast($\{oracle, all\_partitions\}, (partitioning)$)
when a-deliver($partitioning$)                                   {Task 5}
  apply $partitioning$
```

Algorithm variables:

G_W: the set of all variable and their locations
$partitioning$: the partition configuration of G_W

Algorithm 3 Server in partition $\mathcal{P}$

```
when a-deliver($C$)                                              {Task 1}
  $\omega \leftarrow vars(C)$                          {variables accessed by $C$}
  if $\omega \subseteq \mathcal{P}$ then                                 {Task 1a}
    execute command $C$
    send response to the client
  else                                                             {Task 1b}
    $dests \leftarrow partition(\omega)$             {get all involved partition}
    $\mathcal{P}_d \leftarrow target(dests, C)$      {$\mathcal{P}_d \in dest$ will execute $C$}
    $\mathcal{P}_s \leftarrow dests \setminus \mathcal{P}_d$     {$\mathcal{P}_s \in dests$ will send variables}
    if $\mathcal{P} = \mathcal{P}_d$ then               {$\mathcal{P}$ is the target partition:}
      wait until $\forall v \in \omega \setminus \mathcal{P} : \exists \langle vars, C \rangle \in rcvd\_msgs$ :
        $v \in vars$                               {wait for needed variables}
      execute command $C$
      send response to the client
      r-mcast($\mathcal{P}_i \in \mathcal{P}_s, \langle vars, C \rangle$)      {return the variables}
    else                                   {if $\mathcal{P}$ is not the target partition:}
      $vars \leftarrow \omega \cap \mathcal{P}$                {all needed variables in $\mathcal{P}$}
      r-mcast($\mathcal{P}_d, \langle vars, C \rangle$)                     {send variables}
      wait until $\langle vars, C \rangle \in rcvd\_msgs$
when a-deliver($\langle \mathcal{P}, create(v) \rangle$)                     {Task 2}
  $\forall x \in oracle$: send ($\langle signal, create(v) \rangle$) to $x$    {exchange signal}
  wait until $\langle signal, create(v) \rangle \in rcvd\_msgs$ {...coordinate}
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{v\}$
  send $ok$ to the client
when a-deliver($partitioning$)                                   {Task 3}
  for each $\mathcal{Q} \in \mathcal{P}_v \in partitioning$ do
  if $\mathcal{P}$ is $\mathcal{Q}$ then
    $vars \leftarrow partitioning(\mathcal{Q}) \setminus v : v \in \mathcal{P}$
    if $\forall v \in var, \mathcal{P}_i \in partitioning$ :
              $\exists \langle vars, \mathcal{P}_i \rangle \in rcvd\_msgs : v \in vars$ then
      apply $partitioning$
  else
    $vars \leftarrow partitioning(\mathcal{Q}) \cap v : v \in \mathcal{P}$
    $\forall x \in \mathcal{Q}$: send($\langle vars, \mathcal{P} \rangle$) to $x$         {send objs in $\mathcal{Q}$}
when r-deliver ($\langle val, C \rangle$)                               {Task 4}
  $rcvd\_msgs \leftarrow rcvd\_msgs \cup \{\langle val, C \rangle\}$
```

Algorithm variables:

$partitioning$: the partition configuration from the $oracle$

workload graph by receiving hints with variables (i.e., vertices in the graph) and executing commands (i.e., edges in the graph). These hints can be submitted by the clients or by the partitions, which collect data upon executing commands and periodically inform the oracle (Task 4). The oracle computes a partitioning plan of the graph and multicasts it to all servers and to itself. Upon delivering new partition plan, the oracle updates its location map accordingly (Task 5).

To compute an optimized partitioning, the oracle uses a graph partitioner. A new partitioning can be requested by the application, by a partition, or by the oracle itself (e.g., upon delivering a certain number of hints). To

Algorithm 4 Shared functions of the oracle and servers $\mathcal{P}$

```
1: function target($\mathcal{P}, C$)
2:     $\mathcal{P}_d \leftarrow$ deterministicly compute partition to execute
           command $C$ from $\forall \mathcal{P}_i \in \mathcal{P}$
3:     return $\mathcal{P}_d$
4: function partition($vars$)
5:     $dests \leftarrow \{\mathcal{P} : \exists v \in vars \cap \mathcal{P}\}$ {get all involved partitions}
6:     return $dests$
```

determine the destination partition of a set of variables, as part of a move, the oracle uses the last computed partitioning.

3) The server process: When a server delivers a command C, it first checks if it has all variables needed by C. If the server has all such variables, it executes C and sends the response back to the client (Tasks 1a and 2 in Algorithm 3). If not all the variables needed by C are in that partition, the server runs a deterministic function to determine the destination partition to execute C (Task 1b). The function uses as input the variables needed by C and C itself. In this case, each server that is in the multicast group of C but is not the destination partition sends all the needed variables stored locally to the destination partition and waits to receive them back. The destination partition waits for a message from other partitions. Once all variables needed are available, the destination partition executes the C, sends the response back to the client, and returns the variables to their source. Periodically, the servers deliver a new partitioning plan from the oracle (Task 3). Each server will send the variables to the designated partition, as in the plan, and wait for variables from other partitions. Once a server receives all variables, it updates its location map accordingly. To determine the destination partition for a command, the servers uses the last computed partitioning.

C. Performance optimization

In the algorithm presented in the previous section, clients always need to involve the oracle, and the oracle dispatches every command to the partitions for execution. Obviously, if every command involves the oracle, the system is unlikely to scale, as the oracle will likely become a bottleneck. To address this issue, clients are equipped with a location cache. Before submitting a command to the oracle, the client checks its location cache. If the cache contains the partition of the variables needed by the command, the client can atomically multicast the command to the involved partition and thereby avoid contacting the oracle.

The client still needs to contact the oracle in one of these situations: (a) the cache contains outdated information; or (b) the command is a create, in which case it must involve the oracle, as explained before. If the cache contains outdated information, it may address a partition that does not have the information of all the variables accessed by the command. In this case, the addressed partition tells the client to retry the command. The client then contacts the oracle and updates its cache with the oracle's response. Although outdated cache information results in execution overhead, it is expected to happen rarely since repartitioning is not frequent.

V. Implementation

A. Atomic multicast

Our DynaStar prototype uses the BaseCast atomic multicast protocol [10], available as open source.[1] Each group of servers in BaseCast executes an instance of Multi-Paxos.[2] Groups coordinate to ensure that commands multicast to multiple groups are consistently ordered (as defined by the atomic multicast properties §II-C). BaseCast is a genuine atomic multicast in that only the sender and destination replicas of a multicast message communicate to order the multicast message.

B. DynaStar

Our DynaStar prototype is written as a Java 8 library. Application designers who use DynaStar to implement a replicated service must extend three key classes:

– *PRObject*: provides a common interface for replicated data items.
– *ProxyClient*: provides the communication between application client and server, while encapsulating all complex logic of handling caches or retried requests. The proxy client also allows sending multiple asynchronous requests, and delivering corresponding response to each request.
– *PartitionStateMachine*: encapsulates the logic of the server proxy. The server logic is written without knowledge of the actual partitioning scheme. The DynaStar library handles all communication between partitions and the oracle transparently.
– *OracleStateMachine*: computes the mapping of objects to partitions. The oracle can be configured to trigger repartitioning in different ways: based on the number of changes to the graph, or based on some interval of time. Our default implementation uses METIS[3] to provide a partitioning based on the workload graph. While partitioning a graph, METIS aims to reduce the number of multi-partition commands (edge-cuts) while trying to keep the various partitions balanced. We configured

[1]https://bitbucket.org/paulo_coelho/libmcast
[2]http://libpaxos.sourceforge.net/paxos_projects.php#libpaxos3
[3]http://glaros.dtc.umn.edu/gkhome/views/metis

METIS to allow a 20% unbalance among partitions. METIS repartitions a graph without considering previous partitions. Consequently, DynaStar may need to relocate a large number of objects upon a repartitioning. Other partitioning techniques could be used to introduce incremental graph partitioning [39].

We note one important implementation detail. The oracle is multi-threaded: it can serve requests while computing a new partitioning concurrently. To ensure that all replicas start using the new partitioning consistently, the oracle identifies each partitioning with a unique id. When an oracle replica finishes a repartitioning, it atomically multicasts the id of the new partitioning to all replicas of the oracle. The first delivered id message defines the order of the new partitioning with respect to other oracle operations.

C. TPC-C benchmark

TPC-C is an industry standard for evaluating the performance of OLTP systems. TPC-C implements a wholesale supplier company. The company has of a number of distributed sales districts and associated warehouses. Each warehouse has 10 districts. Each district services 3,000 customers. Each warehouse maintains a stock of 100,000 items. The TPC-C database consists of 9 tables and five transaction types that simulate a warehouse-centric order processing application: *New-Order* (45% of transactions in the workload), *Payment* (43%), *Delivery* (4%), *Order-Status* (4%) and *Stock-Level* (4%).

We implemented a Java version of TPC-C that runs on top of DynaStar. Each row in TPC-C tables is an object in DynaStar. The oracle models the workload at the granularity of districts, thus each district and warehouse is a node in the graph. If a transaction accesses a district and a warehouse, the oracle will create an edge between that district and the warehouse. The objects (e.g., customers, orders) that belong to a district are considered part of district. However, if a transaction requires objects from multiple districts, only those objects will be moved on demand, rather than the whole district. The ITEM table is replicated in all servers, since it is not updated in the benchmark.

D. Chirper social network service

Using DynaStar, we have also developed a Twitter-like social network service, named Chirper. In our social network, users can follow, unfollow, post, or read other users' tweets according to whom the user is following. Like Twitter, users are constrained to posting 140-character messages.

Each user in the social network corresponds to a node in a graph. If one user follows another, a directed edge is created from the follower to the followee. Each user has an associated *timeline*, which is a sequence of post messages from the people that the user follows. Thus, when a user issues a post command, it results in writing the message to the timeline of all the user's followers. In contrast, when users read their own timeline, they only need to access the state associated with their own node.

Since DynaStare guarantees linearizable executions, any causal dependencies between posts in Chirper will be seen in the correct order. More precisely, if user B posts a message after receiving a message posted by user A, no user who follows A and B will see B's message before seeing A's message.

Overall, in Chirper, post, follow or unfollow commands can lead to object moves. Follow and unfollow commands can involve at most two partitions, while posts may require object moves from many partitions.

E. Alternative system

We compare DynaStar to an optimized version of S-SMR [5] and DS-SMR [27], publicly available.[4] S-SMR scales performance with the number of partitions under a variety of workloads. It differs from DynaStar in two important aspects: multi-partition commands are executed by all involved partitions, after the partitions exchange needed state for the execution of the command, and S-SMR supports static state partitioning. In our experiments, we manually optimize S-SMR's partitioning with knowledge about the workload. In the experiments, we refer to this system and configuration as S-SMR*.

VI. Performance evaluation

In this section, we report results from two benchmarks: TPC-C and Chirper social networking service described in the previous section. Our experiments show that DynaStar is able to rapidly adapt to changing workloads, while achieving throughputs and latencies far better than the existing state-of-the-art approaches to state machine replication partitioning.

A. Experimental environment

We conducted all experiments on Amazon EC2 T2 large instances (nodes). Each node has 8 GB of RAM, two virtual cores and is equipped with an Amazon EBS standard SSD with a maximal bandwidth 10000 IOPS. All nodes ran Ubuntu Server 16.04 LTS 64 and had the OpenJDK Runtime Environment 8 with the 64-Bit Server VM (build 25.45-b02). In all experiments, the oracle had the same resources as every other partition: 2 replicas and 3 Paxos acceptors (in total five nodes per partition).

[4]https://bitbucket.org/kdubezerra/eyrie
https://bitbucket.org/usi-dslab/ds-smr

B. *TPC-C benchmark*

In the experiments in this section, we deploy as many partitions as the number of warehouses.

1) The impact of graph repartitioning: In order to assess the impact of state partitioning on performance, we ran the TPC-C benchmark on an un-partitioned database. Figure 2 shows the performance of DynaStar with 8 warehouses and 8 partitions. At the first part of the experiment, all the variables are randomly distributed across all partitions. As a result, almost every transaction accesses all partitions. Thus every transaction required coordination between partitions, and objects were constantly moving back and forth. This can be observed in the first 50 seconds of the experiment depicted in Figure 2: low throughput (i.e., a few transactions executed per second), high latency (i.e., up to several seconds), and a high percentage of cross-partition transactions.

After 50 seconds, the oracle computed a new partitioning based on previously executed transactions and instructed the partitions to apply the new partitioning. When the partitions delivered the partitioning request, they exchanged objects to achieve the new partitioning. It takes about 10 seconds for partitions to reach the new partitioning. During the repartitioning, transactions which access objects that are not being relocated will continue to process. After the state is relocated, most objects involved in a transaction can be found in a local partition, which considerably increases performance and reduces latency.

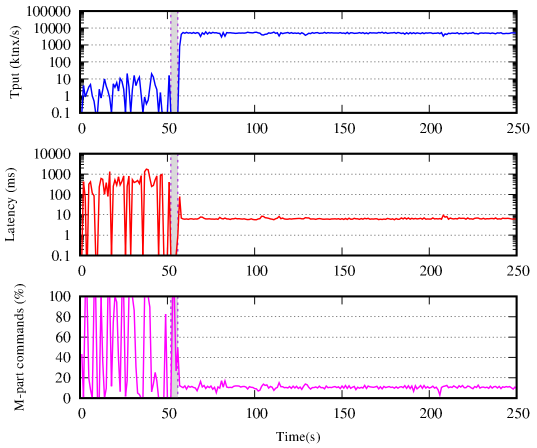


Figure 2: Repartitioning in DynaStar; throughput (top), objects exchanged between partitions (middle), and percentage of multi-partition commands (bottom).

2) Scalability: In order to show how DynaStar scales out, we varied the number of partitions from 1 to 128 partitions. We used sufficient clients to saturate the throughput of the system in each experiment. Figure 3 shows the peak throughput of DynaStar and S-SMR* as we vary the number of partitions. Notice that we increase the state size as we add partitions (i.e., there is one warehouse per partition). The result shows that DynaStar is capable of applying a partitioning scheme that leads to scalable performance.

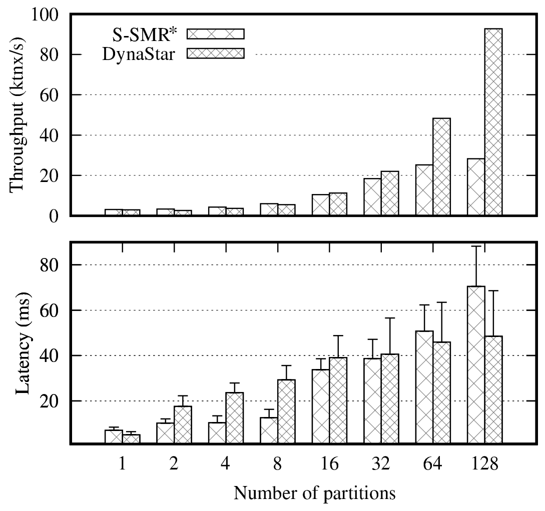


Figure 3: Performance scalability with TPC-C. Throughput (in thousands of transactions per second, ktps) and latency for ≈75% peak throughput in milliseconds (bars show average, whiskers show 95-th percentile).

C. *Social network*

We used the Higgs Twitter dataset [28] as the social graph in the experiments. The graph is a subset of the Twitter network that was built based on the monitoring of the spreading of news on Twitter after the discovery of a new particle with the features of the elusive Higgs boson on 4th July 2012. The dataset consists of 456631 nodes and more than 14 million edges.

We evaluate the performance of DynaStar and other techniques. With S-SMR*, we used METIS to partition the data in advance. Thus, S-SMR* started with an optimized partitioning. DynaStar started with random location of the objects. Each client issues a sequence of commands. For each command, the client selects a random node as the active user with Zipfian access pattern (ρ = 0.95). We focused on two types of workloads: timeline only commands and mix commands (85% timeline and 15% post). Each client operates in a closed loop, that is, the client issues a command and then waits from the response to submit the next command.

1) DynaStar vs. other techniques: Figure 4 shows the peak throughput and latency for approximately 75% of peak throughput (average and 95-th percentile) of the

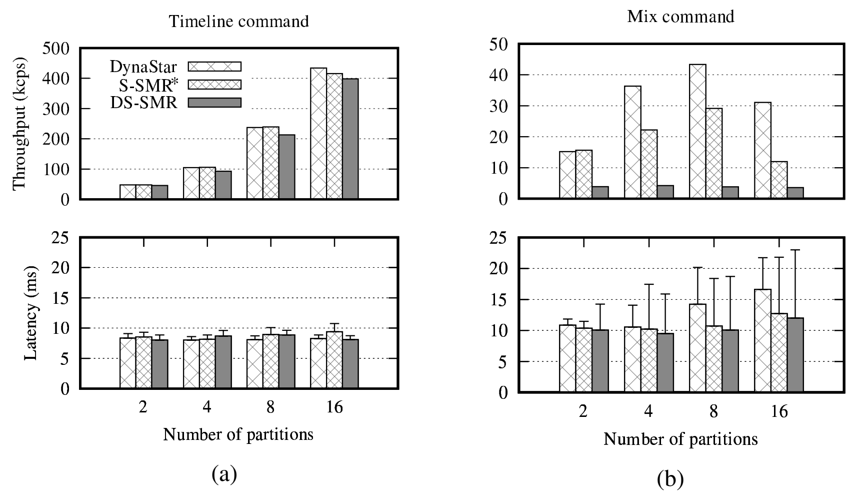


Figure 4: Performance of social network service. Throughput (in thousands of commands per second, kcps) and latency for different partitions. Latency for ≈75% peak throughput in milliseconds (bars show average, whiskers show 95-th percentile).

evaluated techniques as we vary the number of partitions of the fixed graph for the social networks.

In the experiment with timeline commands, all three techniques perform similarly. This happens because no moves occur in DynaStar or DS-SMR, and no synchronization among partitions is necessary for S-SMR* in this case. Consequently, all three schemes scale remarkably well, and the difference in throughput between each technique is due to the implementation of each one.

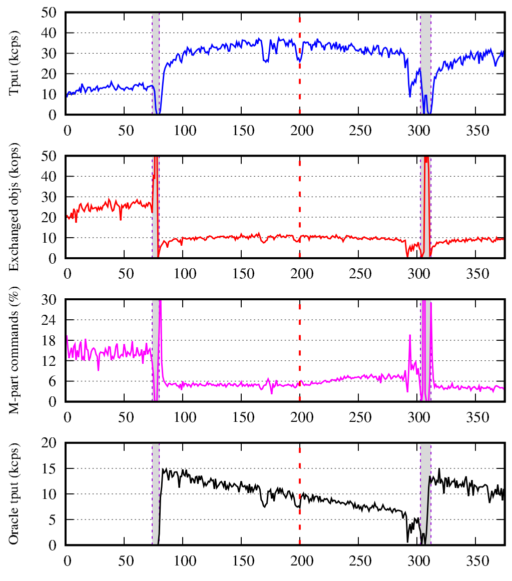
In the experiment with the mix workload, we see that DS-SMR performance decreases significantly. This happens because objects in DS-SMR will only converge if there is a perfect way to partition the data, that is, data items can be grouped such that eventually no command accesses objects across partitions. In the mix workload experiments, objects in DS-SMR are constantly moving back and forth between partition without converging to a stable configuration.

In contrast, for DynaStar and S-SMR*, the throughput scales with the number of partitions in experiments with up to 8 partitions. Increasing the number of partitions to 16 with the fixed graph reveals a tradeoff: On the one hand, additional partitions should improve performance as there are more resources to execute commands. On the other hand, the number of edge cuts increases with the number of partitions, which hurts performance as there are additional operations involving multiple partitions.

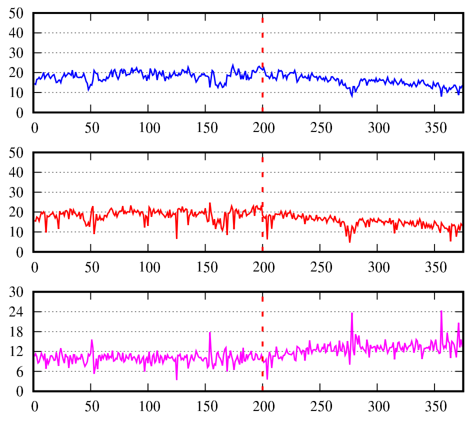
Notice that only post operations are subject to this tradeoff since they may involve multiple partitions. The most common operation in social networks is the request to read a user timeline. This is a single-partition command in our application and as a consequence it scales linearly with the number of partitions.

2) Performance under dynamic workloads: Figure 5 depicts the performance of DynaStar and S-SMR* with an evolving social network. We started the system with the original network from Higg dataset. After 200 seconds, we introduced a new celebrity user in the workload. The celebrity user posted more frequently, and other users started following the celebrity.

At the beginning of the experiment, DynaStar performance was not as good as S-SMR* (i.e., lower throughput, higher number of percentage of multi-partition commands, and higher number of exchanged objects), because S-SMR* started with an optimized partitioning, while DynaStar started with a random partitioning. After 50 seconds, DynaStar triggered the repartitioning process, which led to an optimized location of data. Repartitioning helped reduce the percentage of multi-partition commands to 10%, and thus increased the throughput. After the repartitioning, DynaStar outperforms S-SMR* with the optimized partitioning. After 200 seconds, the network started to change its structure, as many users started to follow a celebrity, and created more edges between nodes in the graph. Both DynaStar and S-SMR suffered from the change, as the rate of multi-partition command increased, and the throughput decreased. However, when the repartitioning takes place in DynaStar, around 300 seconds into the execution, the previously user mapping got a better location from the oracle, which adapted the changes. After the repartitioning, the objects are moved to a better partition, with a resulting increase in throughput.



(a)



(b)

Figure 5: Repartitioning a dynamic workload with DynaStar (a) and S-SMR* without repartitioning (b).

D. *The performance of the oracle*

DynaStar uses an oracle that maintains a global view of the workload graph. The oracle allows DynaStar to make better choices about data movement, resulting in overall better throughput and lower latency. However, introducing a centralized component in a distributed system is always a cause for some skepticism, in case the component becomes a bottleneck, or a single point of failure. To cope with failures, the oracle is implemented as a replicated partition.

The oracle keeps a mapping of objects to partitions and the relations between objects. The size of the mapping depends on the complexity and the granularity of the graph. In the social network dataset, where each user is modeled as an object in the workload graph, the graph uses 1.5 GB of the oracle's memory. In the TPC-C experiments, only district and warehouse objects are in the workload graph; thus, the oracle only needs 1 MB of memory to store the graph for each warehouse.

The results in Figure 5 suggest that the oracle would not become a bottleneck. The number of queries processed to the oracle is zero at the beginning of the experiment, as the clients have cached the location of all objects. After 80 seconds, the repartitioning was triggered, making all cached data on clients invalid. Thus the throughput of queries at the oracle increases, when clients started asking for new location of variables. However, the load diminishes rapidly and gradually reduce. This is because access to the oracle is necessary only when clients have an invalid cache or when a repartition happens

VII. Related work

State machine replication [21], [24], [25], [36], [37] provides strong consistency guarantees, which come from total order and deterministic execution of commands. Since consistent ordering is fundamental for SMR, some authors proposed to optimize the ordering and propagation of commands. For instance, Kapritsos and Junqueira [20] propose to divide the ordering of commands between different clusters: each cluster orders only some requests, and then forwards the partial order to every server replica, which then merges the partial orders deterministically into a single total order that is consistent across the system. In S-Paxos [6], Paxos [26] is used to order commands, but it is implemented in a way that avoids overloading the leader process, which would turn it into a bottleneck.

Multi-threaded execution is a potential source of non-determinism, depending on how threads are scheduled to be executed in the operating system. Some works attempted to circumvent this problems and come up with a multi-threaded, yet deterministic implementation of SMR. In [36], Santos and Schiper propose to parallelize the receipt and dispatching of commands, while executing commands sequentially. In CBASE [24], application semantics is used to determine which commands can be executed concurrently and still produce a deterministic outcome (e.g., read-only commands). In Eve [21], commands are tentatively executed in parallel. After the parallel execution, replicas verify whether they reached a consistent state; if not, commands are rolled back and re-executed sequentially.

Many database replication schemes aim at achieving high throughput by relaxing consistency, that is, they do not ensure linearizability. In deferred-update replication [9], [23], [38], [40], replicas commit read-only transactions immediately, not always synchronizing with each other. Although this indeed improves performance, it allows non-linearizable executions. Database systems usually ensure serializability [4] or snapshot isolation [30], which do not take into account real-time precedence of different commands among different clients. For some applications, these consistency levels may be enough, allowing the system to scale better, but services that require linearizability cannot be implemented with such techniques.

Efforts to make linearizable systems scalable have been made in the past [5], [11], [16], [27], [31]. In Scatter [16], the authors propose a scalable key-value store based on DHTs, ensuring linearizability, but only for requests that access the same key. In the work of Marandi et al [31], variant of SMR is proposed in which data items are partitioned but commands have to be totally ordered. Spanner [11] uses a separate Paxos group per partition and, to ensure strong consistency across partitions, clocks are assumed to be synchronized. Although the authors say that Spanner works well with GPS and atomic clocks, if clocks become out of synch beyond tolerated bounds, correctness is not guaranteed. M^2Paxos [34] proposes a scheme where leases are used instead of partitions owning objects, but assumes full state replication. S-SMR [5] ensures consistency across partitions without any assumption about clock synchronization, but relies on a static partitioning of the state. DS-SMR [27] extends S-SMR by allowing state variables to migrate across partitions in order to reduce multi-partition commands. However, DS-SMR implements repartitioning in a very simple way that does not perform very well in scenarios where the state cannot be perfectly partitioned. DynaStar improves on DS-SMR by employing well-known graph partitioning techniques to decide where each variable should be. Moreover, DynaStar dillutes the cost of repartitioning by moving variables on-demand, that is, only when they are accessed by some command.

Graph partitioning is an interesting problem with many proposed solutions [1], [19], [22], [42]. In this work, we do not introduce a new graph partitioning solution, but instead we use a well-known one (METIS [1]) to partition the state of a service implemented with state machine replication. Similarly to DynaStar, Schism [12] and Clay [39] also use graph-based partitioning to decide where to place data items in a transactional database. In either case, not much detail is given about how to handle repartitioning dynamically without violating consistency. Turcu et al. [42] proposed a technique that reduces the amount of cross-partition commands and implements an advanced transaction routing. Sword [35] is another graph-based dynamic repartitioning technique. It uses a hypergraph partitioning algorithm to distribute rows of tables in a relational database across database shards. Sword does not ensure linearizability and it is not clear how it implements repartitions without violating consistency. E-Store [41] is yet another repartitioning proposal for transactional databases. It repartitions data according to access patterns from the workload. It strives to minimize the number of multi-partition accesses and is able to redistribute data items among partitions during execution. E-Store assumes that all non-replicated tables form a tree-schema based on foreign key relationships. This has the drawback of ruling out graph-structured schemas and m-n relationships. DynaStar is a more general approach that works with any kind of relationship between data items, while also ensuring linearizability.

Some replication schemes are "dynamic" in that they allow the membership to be reconfigured during execution (e.g., [7], [14], [18]). For instance, a multicast layer based on Paxos can be reconfigured by adding or removing acceptors. These systems are dynamic in a way that is orthogonal to what DynaStar proposes.

VIII. Conclusion

In this paper we present DynaStar, a partitioning strategy for scalable state machine replication. DynaStar is inspired by DS-SMR, a decentralized dynamic scheme proposed in [27]. Differently from DS-SMR, however, DynaStar performs well in all workloads evaluated. When the state can be perfectly partitioned, DynaStar converges more quickly than DS-SMR; when partitioning cannot avoid cross-partition commands, it largely outperforms DS-SMR. The key insights of DynaStar are to build a workload graph on-the-fly and use an optimized partitioning of the workload graph, computed with an online graph partitioner, to decide how to efficiently move state variables. The paper describes how one can turn this conceptually simple idea into a system that sports performance close to an optimized (but impractical) scalable system.

Acknowledgements

We wish to thank the reviewers for the insightful comments. This work was supported in part by SNF grant numbers 175717 and 166132.

References

[1] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS'06*, 2006.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[3] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2004.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] C. E. Bezerra, F. Pedone, and R. V. Renesse. Scalable state-machine replication. In *DSN*, 2014.

[6] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-Paxos: Offloading the leader for high throughput state machine replication. In *SRDS*, 2012.

[7] K. Birman, D. Malkhi, and R. van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research, 2010.

[8] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, feb 1987.

[9] P. Chundi, D. Rosenkrantz, and S. Ravi. Deferred updates and data placement in distributed databases. In *ICDE*, 1996.

[10] P. Coelho, N. Schiper, and F. Pedone. Fast atomic multicast. In *DSN*, 2017.

[11] J. Corbett et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8:1–8:22, 2013.

[12] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 2010.

[13] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[14] S. Dustdar and L. Juszczyk. Dynamic replication and synchronization of web services for high availability in mobile ad-hoc networks. *Service Oriented Computing and Applications*, 1(1):19–33, 2007.

[15] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[16] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *SOSP*, 2011.

[17] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. E. Anderson. Scalable consistency in Scatter. In *SOSP*, 2011.

[18] Z. Guessoum, J.-P. Briot, O. Marin, A. Hamel, and P. Sens. Dynamic and adaptive replication for large-scale reliable multi-agent systems. In A. Garcia, C. Lucena, F. Zambonelli, A. Omicini, and J. Castro, editors, *Software Engineering for Large-Scale Multi-Agent Systems*, volume 2603 of *Lecture Notes in Computer Science*, pages 182–198. Springer Berlin Heidelberg, 2003.

[19] B. Hendrickson and T. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 2000.

[20] M. Kapritsos and F. Junqueira. Scalable agreement: Toward ordering as a service. In *HotDep*, 2010.

[21] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *OSDI*, 2012.

[22] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 1970.

[23] T. Kobus, M. Kokocinski, and P. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *ICDCS*, 2013.

[24] R. Kotla and M. Dahlin. High throughput byzantine fault tolerance. In *DSN*, 2004.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[26] L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst. ()*, 16(2):133–169, 1998.

[27] L. H. Le, C. E. Bezerra, and F. Pedone. Dynamic scalable state machine replication. In *DSN*, 2016.

[28] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[29] B. Li, Wenbo, M. Z. Abid, T. Distler, and R. Kapitza. Sarek: Optimistic parallel ordering in byzantine fault tolerance. In *EDCC*, 2016.

[30] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and J. Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Transactions on Database Systems*, 34(2):11:1–11:49, 2009.

[31] P. J. Marandi, M. Primi, and F. Pedone. High performance state-machine replication. In *DSN*, 2011.

[32] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *OSDI*, 2016.

[33] A. Nogueira, A. Casimiro, and A. Bessani. Elastic state machine replication. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2486–2499, Sept 2017.

[34] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran. Making fast consensus generally faster. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 156–167, June 2016.

[35] A. Quamar, K. A. Kumar, and A. Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *EDBT*, 2013.

[36] N. Santos and A. Schiper. Achieving high-throughput state machine replication in multi-core systems. In *ICDCS*, 2013.

[37] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[38] D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. In *DSN*, 2012.

[39] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *PVLDB*, 10(4):445–456, 2016.

[40] A. Sousa, R. Oliveira, F. Moura, and F. Pedone. Partial replication in the database state machine. In *NCA*, 2001.

[41] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-Store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 2014.

[42] A. Turcu, R. Palmieri, B. Ravindran, and S. Hirve. Automated data partitioning for highly scalable and strongly consistent transactions. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):106–118, Jan 2016.

Appendix: Correctness proof

To prove that DynaStar ensures linearizability, we must show that for any execution σ of the system, there is a total order π on client commands that (i) respects the semantics of the commands, as defined in their sequential specifications, and (ii) respects the real-time precedence of commands (§II-D). Let π be a total order of operations in σ that respects $<$, the order atomic multicast induces on commands.

To argue that π respects the semantics of commands, let C_i be the i-th command in π and p a process in partition $\mathcal{P}_p$ that executes C_i. We claim that when p executes C_i, it has updated values of variables in $vars(C_i)$, the variables accessed by C_i. We prove the claim by induction on i. The base step trivially holds from the fact that variables are initialized correctly. Let $v \in vars(C_i)$, C_v be the last client command before C_i in π that accesses v, and q a process in $\mathcal{P}_q$ that executes C_v. From the inductive hypothesis, q has an updated value of v when it executes C_v. There are two cases to consider: (a) $p = q$. In this case, p obviously has an updated value of v when it executes C_i since no other command accesses v between C_v and C_i. (b) $p \neq q$. Since processes in the same partition execute the same commands, it must be that $\mathcal{P}_p \neq \mathcal{P}_q$. From the algorithm, when q executes C_v, $v \in \mathcal{P}_q$ and when p executes C_i, $v \in \mathcal{P}_p$. Thus, q executed a command to move v to another partition after executing C_v and p executed a command to move v to $\mathcal{P}_p$ before executing C_i. Since there is no command that accesses v between C_v and C_i in π, q has an updated v when it executes C_v (from inductive hypothesis), and p receives the value of v at q, it follows that p has an updated v when it executes C_i.

We now argue that there is a total order π that respects the real-time precedence of commands in σ. Assume C_i ends before C_j starts, or more precisely, the time C_i ends at a client is smaller than the time C_j starts at a client, $t^{cli}_{end}(C_i) < t^{cli}_{start}(C_j)$. Since the time C_i ends at the server from which the client receives the response for C_i is smaller than the time C_i ends at the client, $t^{srv}_{end}(C_i) < t^{cli}_{end}(C_i)$, and the time C_j starts at the client is smaller than the time C_j starts at the first server, $t^{cli}_{start}(C_j) < t^{srv}_{start}(C_j)$, we conclude that $t^{srv}_{end}(C_i) < t^{srv}_{start}(C_j)$.

We must show that either $C_i < C_j$; or neither $C_i < C_j$ nor $C_j < C_i$. For a contradiction, assume that $C_j < C_i$ and let C_j be executed by partition $\mathcal{P}_j$.

There are two cases:

(a) C_i is a client command executed by $\mathcal{P}_j$. In this case, since C_i only starts after C_j at a server, it follows that $t^{srv}_{end}(C_j) < t^{srv}_{start}(C_i)$, a contradiction.

(b) C_i is a client command executed by $\mathcal{P}_i$ that first involves a move of variables $vars$ from $\mathcal{P}_j$ to $\mathcal{P}_i$. At $\mathcal{P}_j$, $t^{srv}_{end}(C_j) < t^{srv}_{start}(global(vars, \mathcal{P}_j, \mathcal{P}_i))$ since the move is only executed after C_j ends. Since the move only finishes after variables in $vars$ are in $\mathcal{P}_i$ and C_i can be executed, it must be that $t^{srv}_{end}(global(vars, \mathcal{P}_j, \mathcal{P}_i)) < t^{srv}_{start}(C_i)$. We conclude that $t^{srv}_{end}(C_j) < t^{srv}_{start}(C_i)$, a contradiction.

Therefore, either $C_i < C_j$ and from the definition of π, C_i precedes C_j or neither $C_i < C_j$ nor $C_j < C_i$, and there is a total order in which C_i precedes C_j.

For termination, we argue that every correct client eventually receives a response for every command C that it issues. This assumes that every partition (including the oracle partition) is always operational, despite the failure of some servers in the partition. For a contradiction, assume that some correct client submits a command C that is not executed. Atomic multicast ensures that C is delivered by the involved partition. Therefore, C is delivered at a partition that does not contain all the variables needed by C. As a consequence, the client retries with the oracle, which moves all variables to a single partition and requests the destination partition to execute C, a contradiction that concludes our argument.