Università della Svizzera italiana

Faculty of Informatics

USI Technical Report Series in Informatics

DynaStar: Optimized Dynamic Partitioning for Scalable State Machine Replication

Long Hoang Le[1], Enrique Fynn[1], Mojtaba Eslahi-Kelorazi[1], Robert Soulé[1], Fernando Pedone[1]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland

Abstract

Sharding and replication are the mechanisms of choice of most scalable and fault-tolerant distributed systems. The performance of a sharded system, however, heavily depends on the partitioning of the data: in order to scale, most commands must involve a single shard and load across shards must be balanced. Estimating a good partitioning of the application state is challenging since it requires a priori information about the workload. Moreover, even if such information is available, access patterns may change during system execution (e.g., a good data partitioning for uniform access patterns may lead to poor performance under skewed access patterns). The paper introduces DynaStar, a scalable and fault-tolerant system that supports dynamic state partitioning. DynaStar does not require a priori knowledge about the workload and can adapt to workload variations. The paper describes DynaStar's design and implementation, and presents a detailed performance evaluation using two benchmarks, a social network based on real data and TPC-C.

Report Info

Published
December 2018

Number
USI-INF-TR-2018-1

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Modern distributed systems typically scale performance by sharding application data and tolerate failures by replicating each shard. Clients submit commands for execution to one or more shards. Within a shard, replicas coordinate by means of a consensus protocol (e.g., Paxos [1]). To coordinate the execution of multi-shard commands, replicated shards rely on some distributed coordination protocol (e.g., two-phase locking [2], optimistic concurrency control [3], atomic multicast [4]).

Even though different techniques have been proposed to handle commands that access state in multiple shards, inherently multi-shard commands are more expensive than single-shard commands. This happens due to overhead from ordering and coordinating commands across shards to ensure strong consistency. Moreover, if data is not distributed carefully, then load imbalances can nullify the benefits of sharding. Consequently, the achilles heel of sharded systems 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 [5, 6]. 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 rely on state partitioning delegate the task of partitioning the service state to the application designer (e.g., [7, 4, 2, 8]).

In this paper, we introduce DynaStar, a new approach to scalable replication. DynaStar differs from previous solutions in that it supports *dynamic state partitioning*. This means that data can be relocated from one shard (or partition) to another to optimize performance. Moreover, data relocation is done on-the-fly, with minimal service disruption. As a result, DynaStar does not require any a priori information about the workload 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 (METIS). The location oracle maintains two data structures: (i) a mapping of application state variables to shards, 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 shards on which state variables are stored. If the command accesses variables in multiple shards, the client chooses one shard to execute the command and instructs the other involved shards to temporarily relocate the needed variables to the chosen shard. Of course, when relocating a variable, the client is faced with a choice of which shard to use as a destination. The client estimates the partition that minimizes 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. In most workloads of interest, DynaStar largely outperforms existing dynamic schemes. We also show that the location oracle never becomes a bottleneck in our experiments and can handle workload graphs with millions of vertices.

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.

The rest of the paper is structured as follows. Section 2 presents the system model. Section 3 reviews existing scalable state machine replication approaches. Section 4 introduces DynaStar, describes the technique in detail and argues about its correctness. Section 5 overviews our prototype. Section 6 reports on the results of our experiments. Section 7 surveys related work and Section 8 concludes the paper.

2 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.

2.1 Processes and communication

We consider a distributed system consisting of an unbounded set of client processes $\mathscr{C} = \{c_1, c_2, ...\}$ and a bounded set of server processes (replicas) $\mathscr{S} = \{s_1, ..., s_n\}$. Set $\mathscr{S}$ is divided into disjoint groups of servers $\mathscr{S}_0, ..., \mathscr{S}_k$. 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).

Processes communicate by message passing, using 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 atomic multicast, defined in §2.2.

The system is *partially synchronous* [9]: it is initially asynchronous and eventually becomes synchronous. When the system is asynchronous, there are no bounds on the time it takes for messages to be transmitted and actions to be executed; when the system is synchronous, such bounds exist but are unknown to the processes.

2.2 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.

2.3 Correctness criterion

DynaStar ensures linearizable executions. An execution is *linearizable* if there is a way to reorder the client commands 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 [10].

3 Background

Distributed systems typically rely on sharding and replication to scale performance and tolerate failures. 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 classical state machine replication [11, 12]: replicas of the concerned partition use a consensus protocol to agree on the execution order of commands and each replica executes the commands independently. Different approaches to multi-partition commands have been proposed in the literature.

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 [13, 14]. 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 executed by each involved partition, possibly after it is ordered across the involved partitions using a distributed protocol (e.g., atomic multicast, two-phase commit). Some protocols assume the workload can be sharded such that multi-partition commands are executed locally by each of the involved partitions [15], 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 z allows these variables to be placed in different partitions.

More general solutions do not impose restrictions on state partitioning, but must incur additional overhead in the execution of multi-partition commands [4, 2]. More precisely, upon delivering a multi-partition command, replicas may need to exchange state, since some partitions may not have all the data 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. If the system does not support dynamic partitioning of state, then partitions can first create temporary copies of the variables they do not have so that both partitions execute the command [4]. With dynamic partitioning of state, either x can be moved to the partition that contains y or vice-versa, and the command is executed at a single partition [13].

DynaStar supports dynamic partitioning of state. It does not require the workloads to be perfectly partitioned and does not impose restrictions on state partitioning. Moreover, DynaStar implements multi-partition commands differently from previous proposals. In DynaStar, one chosen partition first "borrows" the needed variables from the other partitions and then executes the command locally.

4 DynaStar

4.1 Overview

DynaStar defines a dynamic mapping of application variables to partitions.[1] Such a mapping is managed by a partitioning oracle, which is handled as a replicated partition. To simplify the discussion, we initially assume that every command involves the oracle. In §4.3, we explain how clients can use a cache to avoid the oracle in the execution of most commands.

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.

4.2 The detailed protocol

Algorithms 1, 2, and 3 describe the client, oracle, and server processes, respectively. For brevity, we omit the delete command since the coordination involved in the create and delete commands are analogous. A proof of correctness for the protocol is available as additional material.[2]

4.2.1 The client process

To execute a command, the client atomically multicasts the command to the oracle (Algorithm 1). The oracle replies with a prophecy, which may already tell the client that the command cannot be executed (e.g., it needs a variable that does not exist, it tries to create a variable that already exists). If the command can be executed,

[1]Applications may also define other granularity of data when mapping application state to partitions. For example, in our social network application (§5.4), each user (together with the information associated with the user) is mapped to a partition; in our TPC-C implementation (§5.3), every district in a warehouse is mapped to a partition.

[2]https://github.com/dynastar2018/DynaStar/blob/master/appendix-correctness.pdf

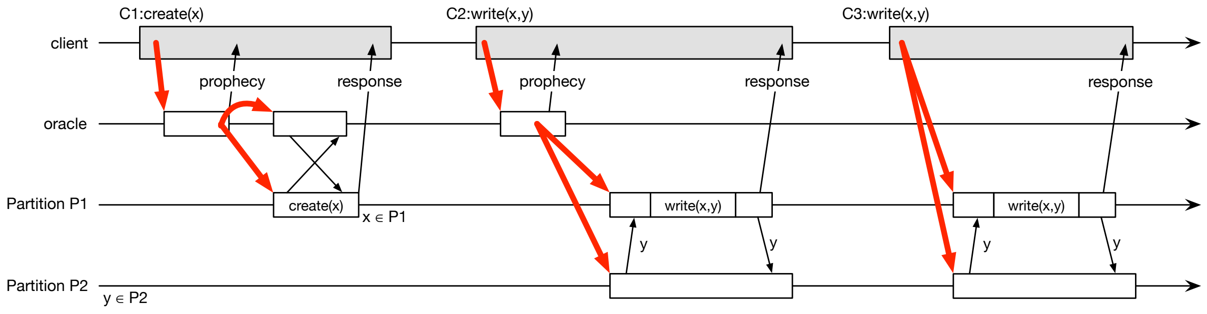

Figure 1: The execution of a create (C1) and a write without client cache (C2) and with client cache (C3) in DynaStar.

the client receives a prophecy containing the partition where the command will be executed. The client then waits for the result of the execution of the command.

4.2.2 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) [4]. The oracle also keeps track of the 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 determine the destination partition of a set of variables, as part of a move, the oracle uses the last computed partitioning.

Algorithm 2 Oracle

```
1:  when a-deliver($exec(C)$)                                   {Task 1}
2:    case $C$ is a $create(v)$ command:
3:      if $partition(\{v\}) \neq \perp$ then                    {if $v$ already exists...}
4:        $prophecy \leftarrow nok$                              {...notify client}
5:      else                                                     {if $v$ doesn't exist...}
6:        $\mathscr{P} \leftarrow$ choose $v$'s partition        {...determine $v$'s partition}
7:        $prophecy \leftarrow \mathscr{P}$                      {prepare client's response}
8:        $alldest \leftarrow \{oracle\} \cup \{\mathscr{P}\}$
9:        a-mcast($alldest$,($\mathscr{P}, create(v)$))
10: case $C$ is any command, but $create(v)$:
11:     $\omega \leftarrow vars(C)$                              {variables accessed by $C$}
12:     if $\exists v \in \omega : partition(\{v\}) = \perp$ then  {if $v$ ¬exists:}
13:       $prophecy \leftarrow nok$                              {tell the client}
14:     else                                                     {if all vars in $\omega$ exist}
15:       $dests \leftarrow partition(\omega)$                   {get all partition involved}
16:       $\mathscr{P}_d \leftarrow target(dests, C)$            {$\mathscr{P}_d$ will excecute $C$}
17:       a-mcast($dests, C$)
18:       $prophecy \leftarrow \mathscr{P}_d$
19:   send $prophecy$ to the client
20: when a-deliver($\mathscr{P}_v, create(v)$)                  {Task 2}
21:   $\forall x \in \mathscr{P}_v$: send ($\langle signal, C\rangle$) to $x$   {exchange signal...}
22:   wait until $\langle signal, C\rangle \in rcvd\_msgs$      {...to coordinate}
23:   $\mathscr{P}_v \leftarrow \mathscr{P}_v \cup \{v\}$
24: when receive ($\langle val, C\rangle$)                        {Task 3}
25:   $rcvd\_msgs \leftarrow rcvd\_msgs \cup \{\langle val, C\rangle\}$
26: function partition($vars$)
27:   $dests \leftarrow \{\mathscr{P} : \exists v \in vars \cap \mathscr{P}\}$
28:   return $dests$
29: when a-deliver($hint(V_h, E_h)$)                            {Task 4}
30:   update $G_W$ with $(V_h, E_h)$
31:   $inc(changes)$
32:   if $changes \geq threshold$ then
33:     $partitioning \leftarrow$ compute $\mathscr{I}_1, ..., \mathscr{I}_m$ from $G_W$
34:     $alldest \leftarrow \{oracle\} \cup allpartitions$
35:     a-mcast($alldest$,($partitioning$))
36: when a-deliver($partitioning$)                               {Task 5}
37:   apply $partitioning$
38: function target($\mathscr{P}, C$)
39:   $\mathscr{P}_d \leftarrow$ deterministicly compute partition to execute
          command $C$ from $\forall \mathscr{P}_i \in \mathscr{P}$
40:   return $\mathscr{P}_d$
```

Algorithm variables:

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

4.2.3 The server process

When a server delivers a command C, it first checks if it has all variables needed by the command. If the server has all such variables, it executes the command and sends the response back to the client (Tasks 1a and 2 in Algorithm 3). If not all the variables needed by the command are in that partition, the server runs a deterministic function to determine the destination partition to execute the command (Task 1b). The function uses as input the variables needed by the command and the command itself. In this case, each server that is in the multicast group of the command C but is not the destination partition sends all the needed

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

```
1:  when a-deliver($C$)                                                    {Task 1}
2:    $\omega \leftarrow vars(C)$                                          {variables accessed by $C$}
3:    if $\forall v \in \omega : v \in \mathscr{P}$ then                   {Task 1a}
4:      execute command $C$
5:      send response to the client
6:    else                                                                 {Task 1b}
7:      $dests \leftarrow partition(\omega)$                               {get all involved partition}
8:      $\mathscr{P}_d \leftarrow target(dest, C)$                         {$\mathscr{P}_d \in dest$ will execute $C$}
9:      if $\mathscr{P} = \mathscr{P}_d$ then                              {$\mathscr{P}$ is the target partition:}
10:       wait until $\forall v \in \omega \setminus \mathscr{P} : \exists \langle vars, C \rangle \in rcvd\_msgs : v \in vars$   {wait for needed variables}
11:       execute command $C$
12:       send response to the client
13:       for each $\mathscr{Q} \in \mathscr{P}_s$ do $\forall x \in \mathscr{Q}$:
                                  send $(\langle v : v \in \mathscr{Q} \rangle, C)$ to $x$   {return the variables}
14:     else                                                               {if $\mathscr{P}$ is not the target partition:}
15:       $vars \leftarrow \omega \cap \mathscr{P}$                        {all needed variables in $\mathscr{P}$}
16:       r-mcast($\mathscr{P}_d, \langle vars, C \rangle$)                 {send variables to destination}
17:       wait until $\forall v \in vars \in rcvd\_msgs : v \in vars$

18: when a-deliver($\mathscr{P}, create(v)$)                                 {Task 2}
19:   $\forall x \in oracle$: send $(\langle signal, C \rangle)$ to $x$     {exchange signal to...}
20:   wait until $\langle signal, C \rangle \in rcvd\_msgs$                 {...coordinate}
21:   $\mathscr{P} \leftarrow \mathscr{P} \cup \{v\}$
22:   send $ok$ to the client
23: when a-deliver($partitioning$)                                         {Task 3}
24:   for each $\mathscr{Q} \in \mathscr{P}_v \in partitioning$ do
25:   if $\mathscr{P}$ is $\mathscr{Q}$ then
26:     $vars \leftarrow partitioning(\mathscr{Q}) \setminus v : v \in \mathscr{P}$
27:     if $\forall v \in var, \mathscr{P}_i \in partitioning$ :
                         $\exists \langle vars, \mathscr{P}_i \rangle \in rcvd\_msgs : v \in vars$ then
28:       apply $partitioning$
29:   else
30:     $vars \leftarrow partitioning(\mathscr{Q}) \cap v : v \in \mathscr{P}$
31:     $\forall x \in \mathscr{Q}$: send($\langle vars, \mathscr{P} \rangle$) to $x$   {send objs that are in $\mathscr{Q}$}
32: when receive ($\langle val, C \rangle$)                                   {Task 4}
33:   $rcvd\_msgs \leftarrow rcvd\_msgs \cup \{\langle val, C \rangle\}$
34: function target($\mathscr{P}, C$)
35:   $\mathscr{P}_d \leftarrow$ deterministicly compute partition to execute
          command $C$ from $\forall \mathscr{P}_i \in \mathscr{P}$
36:   return $\mathscr{P}_d$
  Algorithm variables:
  $partitioning$: the partition configuration from the $oracle$
```

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 command 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.

4.3 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.

5 Implementation

5.1 Atomic multicast

Our DynaStar prototype uses the BaseCast atomic multicast protocol described in [16] and available as open source.[3] Each group of servers in BaseCast executes an instance of Multi-Paxos.[4] Groups coordinate to ensure that commands multicast to multiple groups are consistently ordered (as defined by the atomic multicast properties §2.2). BaseCast is a genuine atomic multicast in that only the sender and destination replicas of a multicast message communicate to order the multicast message.

5.2 DynaStar

Our DynaStar prototype is written as a Java 8 library. The code is publicly available as open-source. Application designers who use DynaStar to implement a replicated service must extend three key classes:

– *PRObject*: provides a common interface for replicated data items.

– *PartitionStateMachine*: encapsulates the logic of the server proxy. Note that 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. Our default implementation uses METIS[5] 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 METIS to allow a 20% unbalance among partitions.

We note one important implementation detail. The oracle is multi-threaded, and can service 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.

5.3 TPC-C benchmark

The TPC-C benchmark is an industry standard for evaluating the performance of OLTP systems. It has 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 the TPC-C benchmark 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 or 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 district, only those objects will be moved on demand, rather than the whole district.

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

5.4 Chirper social network service

For the purposes of micro benchmark, we have developed a Twitter-like social network service using the DynaStar library. 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, then 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.

5.5 Alternative system

Throughout our evaluation, we compare DynaStar to an optimized version of S-SMR [4], available in a public repository.[6] S-SMR has been shown to scale 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*.

6 Performance evaluation

In this section, we evaluate DynaStar according to various metrics, under a variety of different operational parameters. We ran a number of experiments using two benchmarks: the TPC-C benchmark 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.

6.1 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 acceptors (in total five nodes per partition).

6.2 Methodology and goals

The experiments seek to answer the following questions:

- *What is the impact of repartitioning on a real dataset?*
- *How does partitioning affect performance when the workload grows with the number of partitions and when the workload has constant size?*
- *How does DynaStar performance compare to other approaches?*
- *How does DynaStar perform under dynamic workloads?*
- *What is the performance of the oracle?*

[6]https://bitbucket.org/kdubezerra/eyrie

Performance metrics. The latency was measured as the end-to-end time between issuing the command, and receiving the response. Throughput was measured as the number of posts/second or transactions/second that the clients were able to send.

6.3 TPC-C benchmark

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

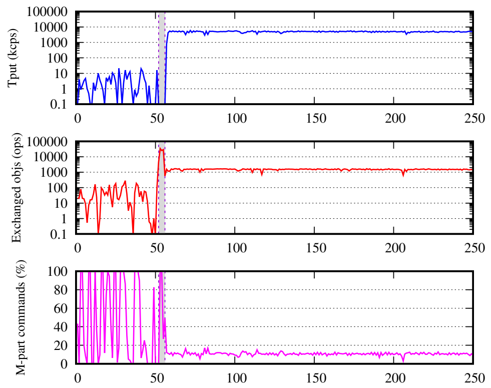

Figure 2: Repartitioning on DynaStar; throughput (top), objects exchanged between partitions (middle), and percentage of multi-partition commands (bottom).

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 4 warehouses and 4 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., 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. However, during the repartitioning servers continue to execute transactions. 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.

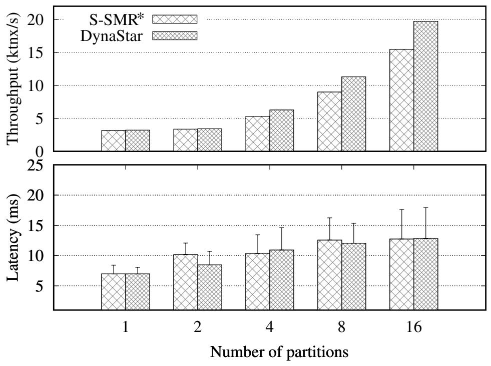

Figure 3: Performance scalability with TPC-C.

Scalability. In order to show how DynaStar scales out, we ran experiments with 1, 2, 4, 8 and 16 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.

6.4 Social network

We used Higgs Twitter Dataset [17] as the social graph used 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 S-SMR* with the dataset mentioned above. 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 ($\rho = 0.95$). 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.

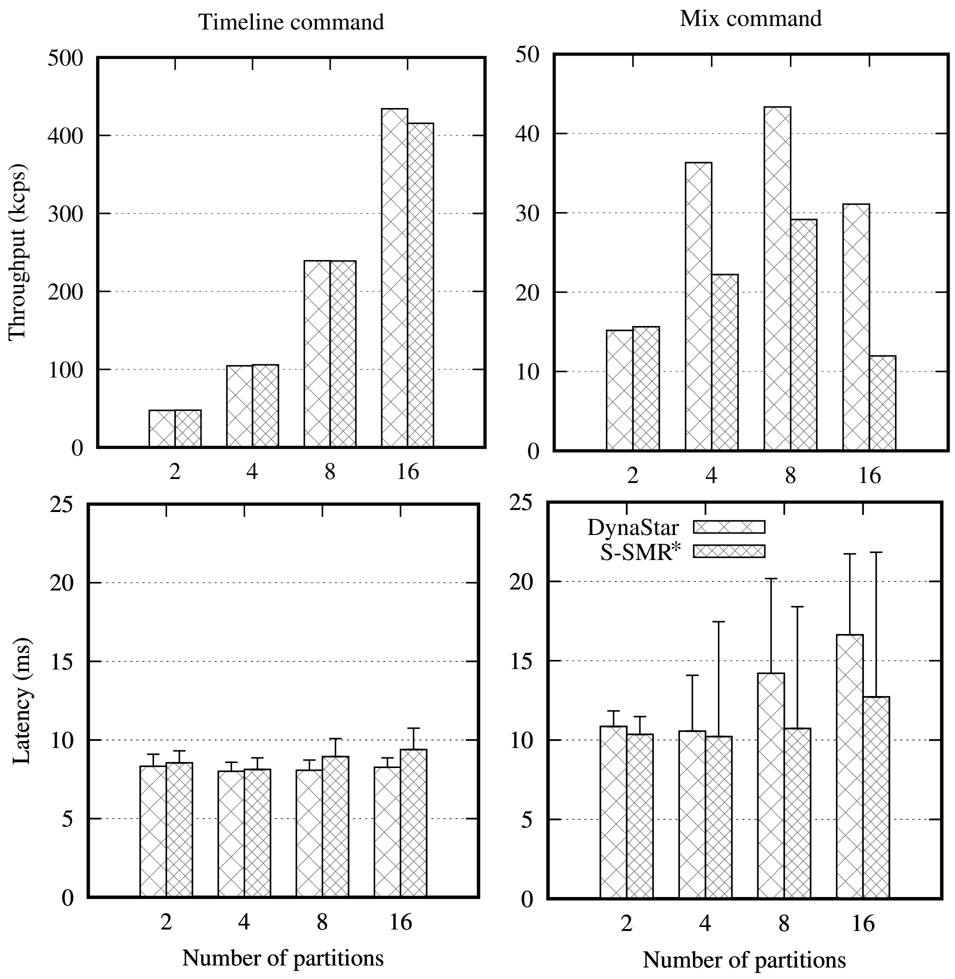

Figure 4: Throughput and latency for different partition numbers. Throughput is in thousands of commands per second (kcps). Latency for ≈75% peak throughput is in milliseconds (bars show average, whiskers show 95-th percentile).

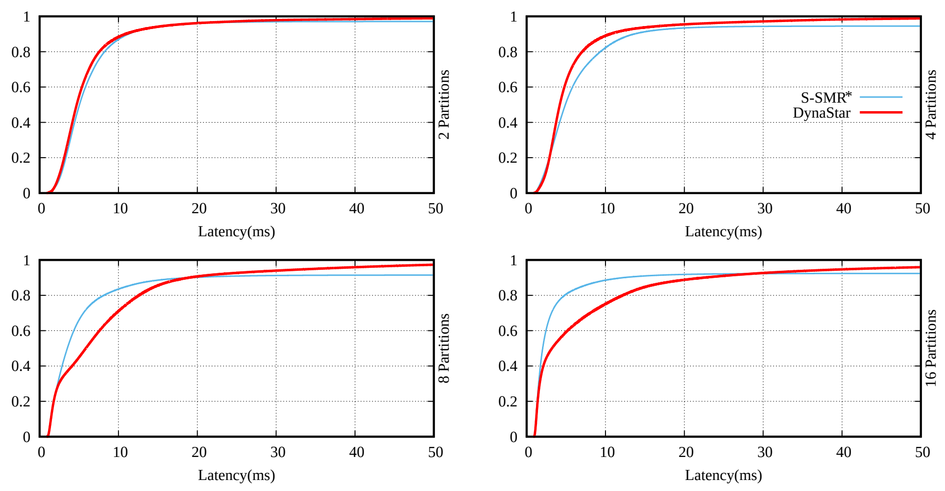

Figure 5: Cumulative distribution function (CDF) of latency for mix workloads on different partitioning configurations.

DynaStar vs. alternative technique. Figure 4 shows the peak throughput and latency for approximately 75% of peak throughput (average and 95-th percentile) of the evaluated techniques on different workloads, as we vary the number of partitions of the fixed graph for the social networks.

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

In the experiment with mix workload, we see that the throughput still scales with the number of partitions in experiments with up to 8 partitions. While increasing the number of partitions to 16 with that fixed graph starts introducing a tradeoff. On the one hand, additional partitions should improve performance as there are more resources to execute the 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.

Figure 5 shows the cumulative distribution functions (CDFs) of latency for the mix workload of DynaStar and S-SMR* on different configurations. The results suggest that S-SMR* achieves lower latency than DynaStar for 80% of the load. This is expected, as for multi-partition commands, partitions in DynaStar have to send additional data to return objects to their original location after command execution .

Performance under dynamic workloads. Figure 6 depicts performance of DynaStar and S-SMR* of 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 got better throughput than 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.

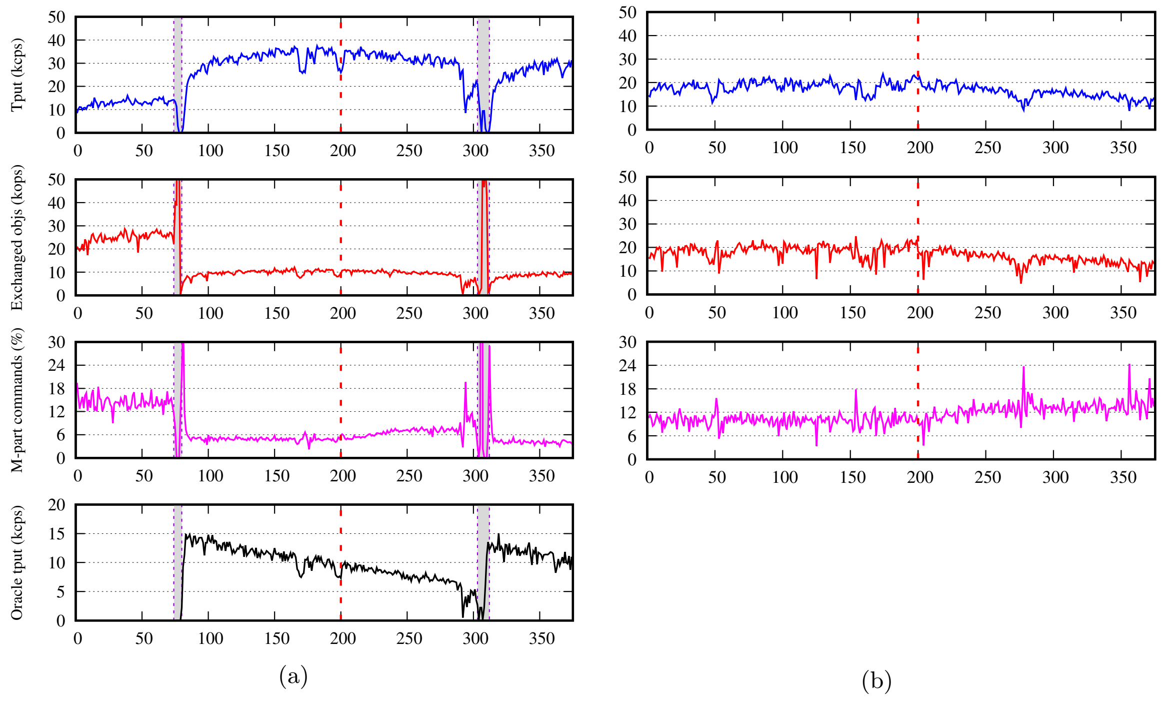

Figure 6: The impact of repartitioning of a dynamic workload in DynaStar (a) and S-SMR without repartitioning (b).

Table 1 shows the throughput of each partition when the system reached the maximum throughput at the second of 180. Although the objects were evenly distributed among partitions, there was still a skew in the load of the system, e.g., partition 1 and 2 served more commands than the other partitions. This happened because of the skew in the access pattern: some users were more active and posted more than the others, thus the requests that sent to their servers are more than to other servers.

Table 1: Average load at partitions at peak throughput.

Partition	Tput	M-part commands per sec	Exchanged objects per sec
1	12766	887	3907
2	11790	643	3036
3	6775	440	1503
4	6458	400	1490

6.5 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 an overally 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. We conducted two experiments to evaluate if the DynaStar oracle is a bottleneck to system performance. The results show that the load on the oracle is low, suggesting that DynaStar scales well.

The first experiment assesses the scalability of the METIS algorithm only. We measured the time to compute the partitioning solution, and the memory usage of the algorithms for increasingly large graphs. The results, depicted in Figure 7, show that METIS scales linearly in both memory and computation time on graphs of up to 10 million vertices.

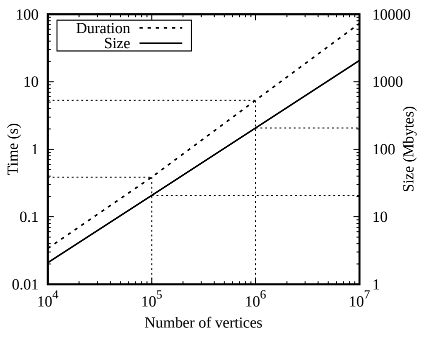

Figure 7: METIS processor and memory usage.

The second experiment evaluates the oracle in terms of the number of queries sent to the oracle over time, for varying numbers of partitions. The experiment results was measured when the system was running stable, and the clients had cache all requests. The results are shown in Figure 8. The number of queries processed to the oracle is zero at the beginning of the experiment, as the clients has cached the location of all objects. After 80 second, the repartitioning was triggers, made all the cache 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 to zero. This is because access to the oracle is necessary only when clients have an invalid cache or when a repartition happens. These experiments suggest that the oracle would not become a bottleneck.

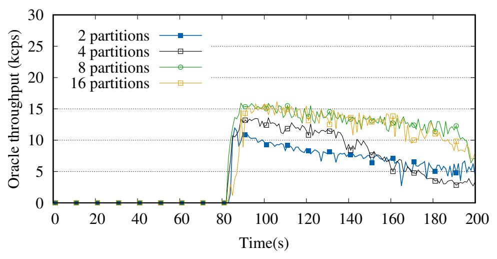

Figure 8: Throughput at the oracle (social network).

7 Related work

State machine replication [18, 19, 11, 20, 12] 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, [21] proposes 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 [22], Paxos [23] 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 [20], the authors propose to parallelize the receipt and dispatching of commands, while executing commands sequentially. In [19], application semantics is used to determine which commands can be executed concurrently and still produce a deterministic outcome (e.g., read-only commands). In [18], 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 [24, 25, 26, 27], 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 [28] or snapshot isolation [29], 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 [4, 2, 30, 13, 31]. In [30], the authors propose a scalable key-value store based on DHTs, ensuring linearizability, but only for requests that access the same key. In [31], variant of SMR is proposed in which data items are partitioned but commands have to be totally ordered. Spanner [2] 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. S-SMR [4] ensures consistency across partitions without any assumption about clock synchronization, but relies on a static partitioning of the state. DS-SMR [13] 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 [32, 33, 34]. In this work, we do not introduce a new graph partitioning solution, but instead we use a well-known one (METIS [32]) to partition the state of a service implemented with state machine replication. Similarly to DynaStar, Schism [5] and Clay [35] 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. Sword [36] 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 [6] 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., [37, 38, 39]). 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.

8 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 by Le et al. [13]. 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.

References

[1] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, May 1998.

[2] J. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems*, vol. 31, no. 3, pp. 8:1–8:22, 2013.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, pp. 4:1–4:26, June 2008.

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

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

[6] 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.

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

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

[9] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

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

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[12] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

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

[14] A. Nogueira, A. Casimiro, and A. Bessani, "Elastic state machine replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 2486–2499, Sept 2017.

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

[16] P. Coelho, N. Schiper, and F. Pedone, "Fast atomic multicast," in *DSN*, 2017.

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

[18] 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.

[19] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *DSN*, 2004.

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

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

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

[23] L. Lamport, "The Part-Time Parliament," *ACM Trans. Comput. Syst. ()*, vol. 16, no. 2, pp. 133–169, 1998.

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

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

[26] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *DSN*, 2012.

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

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

[29] 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*, vol. 34, no. 2, pp. 11:1–11:49, 2009.

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

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

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

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

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

[35] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker, “Clay: Fine-grained adaptive partitioning for general database schemas,” *PVLDB*, vol. 10, no. 4, pp. 445–456, 2016.

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

[37] K. Birman, D. Malkhi, and R. van Renesse, “Virtually synchronous methodology for dynamic service replication,” Tech. Rep. MSR-TR-2010-151, Microsoft Research, 2010.

[38] 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*, vol. 1, no. 1, pp. 19–33, 2007.

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