Dynamic Scalable State Machine Replication

L. H. Le
University of Lugano
Switzerland

E. Bezerra
University of Lugano
Switzerland

F. Pedone
University of Lugano
Switzerland

***Abstract*—State machine replication (SMR) is a well-known technique for providing a fault-tolerant, high availability service, by having clients commands executed in the same other on all replicas, following the deterministic execution, producing same state of all replica for each client command. The original SMR model was not quite efficient since adding replica to the system doesn't increase the performance or throughput, since all replicas still have to execute same number of commands. Scalable SMR (S-SMR) then was introduced to address that problem by partitioning the application state, that allow client commands go through a combination of replicas, but not necessary all the replicas, while still maintain the consistency of the system. However (S-SMR) .... TODO: complete abstract**

I. Introduction

TODO: add content

II. Model and definitions

A distributed system is considered as a system consisting of an unbounded set of client processes $\mathcal{C} = \{c_1, c_2, ...\}$ and a bounded set of server processes $\mathcal{S} = \{s_1, ..., s_n\}$. Set $\mathcal{S}$ is divided into P disjoint groups of servers, $\mathcal{S}_1, ..., \mathcal{S}_P$. 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 either one-to-one or one-to-many communication, as defined next. The system is asynchronous: there is no bound on message delay or on relative process speed.

We consider a scalable state machine replication system 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 by the primitives *multicast*(γ, m) and *deliver*(m), where γ is a set of server groups. Let relation $\prec$ be defined such that $m \prec m'$ if there is a server that delivers m before m'. Atomic multicast ensures that (i) if a server delivers m, then all correct servers in γ deliver m *(agreement)*; (ii) if a correct process multicasts m to groups in γ, then all correct servers in every group in γ deliver m *(validity)*; and (iii) relation $\prec$ is acyclic *(order)*.[1] The order property implies that if s and r deliver messages m and m', then they deliver them in the same order. Atomic broadcast is a special case of atomic multicast in which there is a single group with all servers.

III. Background and motivation

State-machine replication is a fundamental approach to implement a fault-tolerant service by replicating servers and coordinating the execution of client commands against server replicas [10], [17]. State-machine replication ensures strong consistency (i.e., linearizability [1]) by coordinating the execution of commands in the different replicas: Every replica has a full copy of the service state $\mathcal{V} = \{v_1, ..., v_m\}$ and executes commands submitted by the clients in the same order. A command is a program consisting of a sequence of operations, which can be of three types: *read(v)*, *write(v, val)*, or a deterministic computation.

A. Scaling state-machine replication

By starting in the same initial state and executing the same sequence of deterministic commands, 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 [2], [4].

Despite its simple execution model, classical SMR does not scale: adding resources (e.g., replicas) will not translate into sustainable improvements in throughput. This happens for a few reasons. First, the underlying communication protocol needed to ensure ordered message delivery may not scale itself (i.e., a communication bottleneck). Second, every command must be executed sequentially by each replica (i.e., an execution bottleneck).

Several approaches have been proposed to address SMRs scalability limitations. To cope with communication overhead, some proposals have suggested to spread the load of ordering commands among multiple processes (e.g., [16], [12], [15]), as opposed to dedicating a single process to determine the order of commands (e.g., [3], [11]).

Two directions of research have been suggested to overcome execution bottlenecks. One approach (scaling up) is to take advantage of multiple cores to execute commands concurrently without sacrificing consistency [8], [13], [9], [7]. Another approach (scaling out) is to partition the service's state and replicate each partition (e.g., [6], [14]. In the following section, we review Scalable State-Machine Replication (S-SMR), a proposal in the second category.

B. Scalable State-Machine Replication

TODO: talk about SSMR

SSMR implementation brings into use the concept of *Oracle*, the core of partitioning algorithms, which runs a static deterministic algorithm to return the combination of involved partitions of a command; all clients and partitions have their own version of Oracle, and assumed to be identical. By doing that way, SSMR can ensure Oracle return same results for query from both clients and partitions. However, this implementation leads to some limitations: (i) the Oracles on all parties are not synchronized, thus they need to have the knowledge of all $v \in \mathcal{V}$ during the life-cycle of the system, and (ii) a change in $\mathcal{V}$ on one Oracle will not be recognized by the others. Therefore, SSMR doesn't support creating state variable v_n on the fly, and has to initialize the whole $\mathcal{V}$ on the starting phase.

IV. Dynamic Scalable State Machine Replication

In this section, we introduce Dynamic SSMR, discuss performance optimizations, and argue about D-SSMR's correctness.

A. General idea

S-SMR divides the state variables v into P partitions $\mathcal{P}_1, ..., \mathcal{P}_P$, where for each $\mathcal{P}_i$, $\mathcal{P}_i \subseteq \mathcal{V}$, and each variable v in $\mathcal{V}$ has to be assigned to at least one partition and define $part(v)$ as the partitions that hold v. Each partition $\mathcal{P}_i$ is replicated by servers in group s_i. For brevity, the server s belongs to $\mathcal{P}_i$ with the meaning that $s \in \mathcal{S}_i$, and say that client c multicasts command C to partition $\mathcal{P}_i$ means that c multicasts C to group $\mathcal{S}_i$.

To execute command C, the client multicasts C to all partitions that hold a variable read or updated by C. Consequently, the client must be able to determine the partitions accessed by C, denoted by $part(C)$. If the client cannot accurately estimate which partitions are accessed by C, it must determine a superset of these partitions, in the worst case assuming all partitions. In order for clients to provide a close approximation to the command's actually accessed partitions, there is an

[1] Solving atomic multicast requires additional assumptions [3], [5]. In the following, we simply assume the existence of an atomic multicast oracle.

oracle that tells the client which partitions should receive each command.

Dynamic SSMR improves on SSMR implementation by adding a central Oracle which has the information of all state variables $\mathcal{V}$, as well as provides a mechanism of controlling the access to those variable that ensure linearizability.

We distinguish between three operation types: $update(v)$, an operation that updates the value of a state variable, v, $create(v, P)$, an operation that create a state variable at a specific partition P, and $move(v, \mathcal{P}_n)$, an operation that move v to partition $\mathcal{P}_n$.

Consider the execution depicted in Figure 1 (a), where state variables x is created on partition $\mathcal{P}_1$ in the middle of the execution of SSMR. Command $C_1(x)$ and $C_3(x)$ reads the value of x, $C_2(x, \mathcal{P}_1)$ create x on partition $\mathcal{P}_1$. Client a first multicasts query to the $Oracle$ for location of x, which is not available at that time, hence $Oracle$ return empty result, which tell client a to end the execution. Client b then wants to create variable x on $\mathcal{P}_1$, before sending actual creating command, client b also multicasts query to $Oracle$ to get the involved partition (eg., $\mathcal{P}_1$), then multicasts $C_2(x, \mathcal{P}_1)$ to both $Oracle$ and $\mathcal{P}_1$. Oracle will update information of x, and $\mathcal{P}_1$ will execute create x command. From then, for every read command to x (eg., $C_3(x)$), $Oracle$ could answer with partition $\mathcal{P}_1$ which is the one holds x.

In the following figures (2b, 2c), where read command $C_1(x)$ comes in the middle of the execution of create command $C_2(x, \mathcal{P}_1)$, while the query of $C_1(x)$ comes after multicasts command of $C_2(x, \mathcal{P}_1)$. With the knowledge of x, Oracle response to the query with a positive answer, therefore there are possibilities that the read command either comes during (eg., fig. 2b) or before (eg., fig. 2c) the execution of actual write command. The SSMR model avoid the problem described in figure 2 (b) by ensuring that the execution of every command is atomic (eg., for every server s in partition $\mathcal{P}$ that executes C, there is a server r in every $\mathcal{P}' \in part(C)$ such that $delivery(C, r) < end(C, s)$. Intuitively, this condition guarantees that the execution of C_1 and C_2 at $\mathcal{P}$ overlap in time). Atomic multicast prevents the problem figure 2 (c) from happening as $deliver(C_2) \prec deliver(C_1)$, that leads to situation described in fig. 2b.

Next figure (3 a, 3 b) depicts the scenario of moving object location from a partition to another. Intuitively, the problem with the execution in fig. 3a is that read command $C_3(x)$ executes in between the execution of $C_2(x, \mathcal{P}_2)$ at partitions Px and Py. Before sending C_3 to destination partition, client 1 send query to oracle for possible position of x, which is P_1 at that specific moment, but not correct at the execution time. In D-SSMR, we prevent that from happening by using oracle as the controller for accessing state variable on partitions, by using versioning mechanism on oracle for different type of commands. Whenever a client sends a command that update variable $C(x)$, oracle also increases the version of that variable x in its knowledge.

There are possibilities that a client could fail in the middle of the execution of the command chains, at the moment after requires the lock and before sending the actual C command to partition P. In this case, TODO: which one release lock, either oracle or another client that require lock again.

B. Detailed algorithm

In Algorithm **??**, we show the operation of D-SSMR. To submit a command C, the client first check its cache to get the lastest information $dests$ of the state objects in command. Then it can either uses that info if exists, or sends a query to the Oracle to get set $dests$ (line 9, 10), which is a superset of $part(C)$ used by the client as destination set for C (line 14).

Upon delivering $Q(C)$, oracle O runs function $getPart(C)$ which returns a set $\mathcal{V}$ of involved state variables of command C. If there exists a variable $v_i \in \mathcal{V}$ in oracle memory, which indicates the variable v_i exists on a partition $p_i \in \mathcal{P}$, oracle reply to client with p_i value, or an empty result otherwise. In addition, upon delivering $C(v, \mathcal{P})$, oracle update its memory with new location of v: $\mathcal{V}\_part\{v.id\} \leftarrow \mathcal{P}$

If the $dests$ from Oracle is empty, client stop the command there. (line 12)

Upon delivering C, server P execute C and reply to the client.

C. Performance optimizations

Algorithm 1 can be optimized in many ways. In this section, we briefly mention some of these optimizations and then detail caching.

Client can have a cache copy of the variable location on local. Thus when client issues command C, it does not need to query Oracle, instead it send a message direct to the associated partition based on its knowledge from cache. There are possibilities that client cache is invalid because of move command other client issued, thus the partition just need to return a response indicates invalid location. Then the client can retry the command by starting query from Oracle this time to get the updated location of state variable.

D. Correctness

TODO: add content

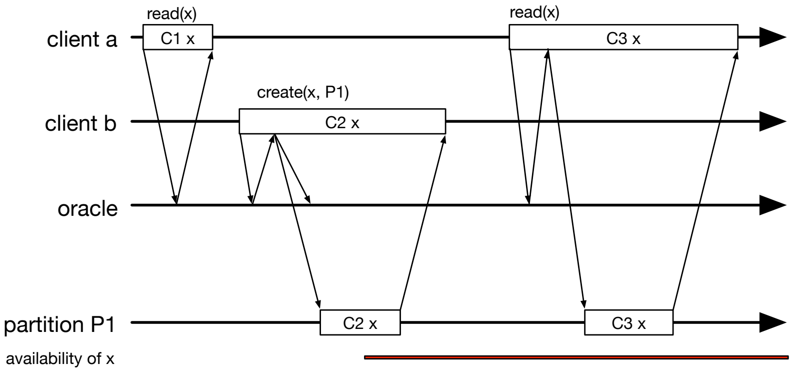
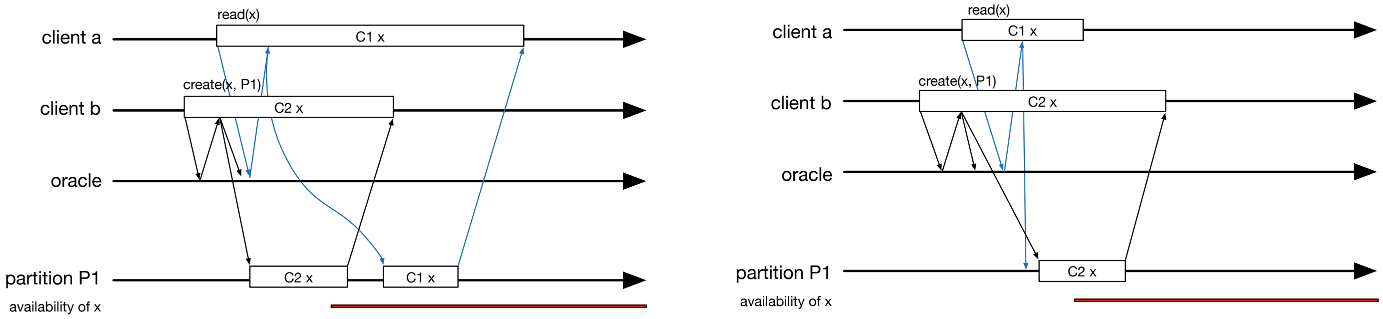


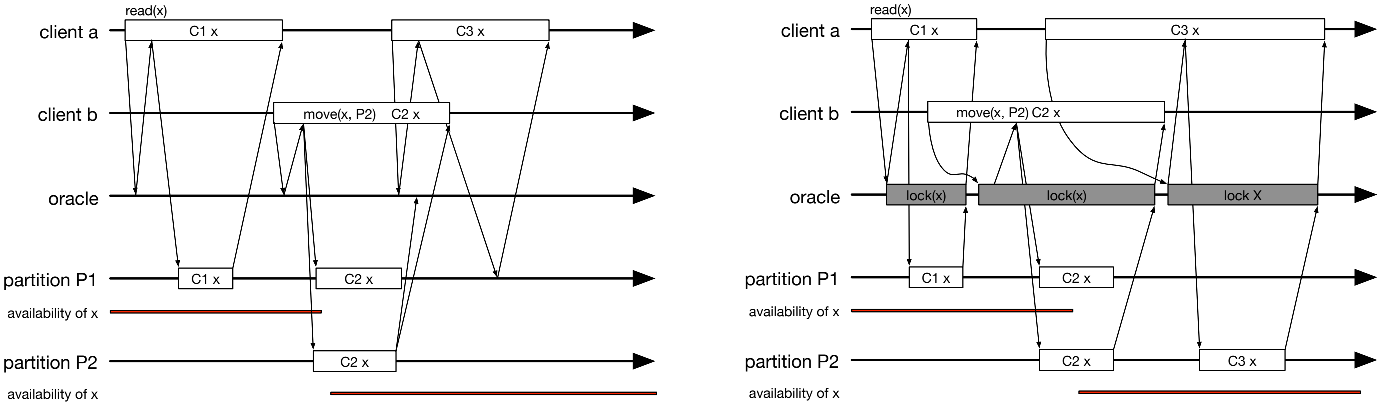
Fig. 1. Execution flow of D-SSMR with *create* command



(b) read command comes in the middle of execution of create command

(c) read command comes before the execution of create command

Fig. 2. Execution flow of D-SSMR with overlapping read-create commands



(a) read command comes in the middle of move command

(b) require lock for object access on oracle

Fig. 3. Execution flow of D-SSMR with overlapping read-update commands

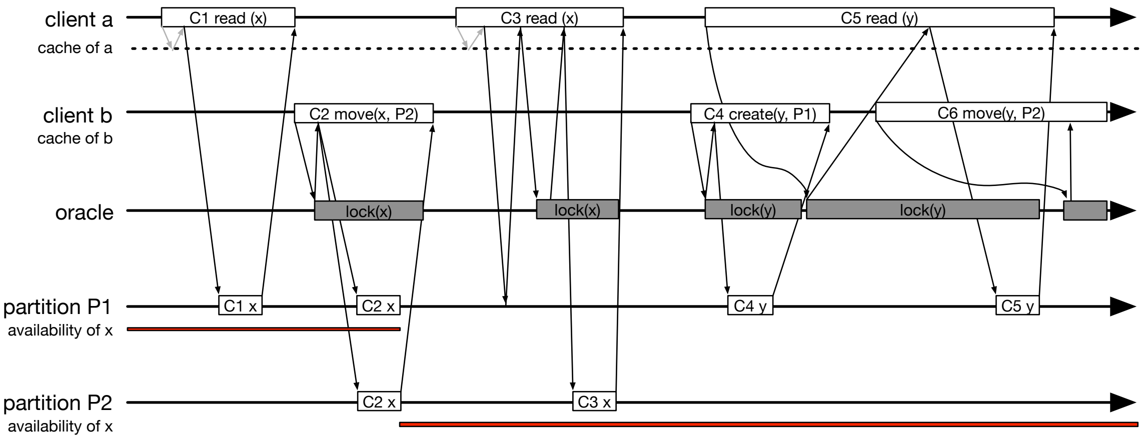


Fig. 4. Execution flow of D-SSMR with oracle caching on client

References

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

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

[3] Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)

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

[5] Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty processor. Journal of the ACM 32(2), 374–382 (1985)

[6] Glendenning, L., Beschastnikh, I., Krishnamurthy, A., Anderson, T.E.: Scalable consistency in Scatter. SOSP pp. 15–28 (2011)

[7] Guo, Z., Hong, C., Yang, M., Zhou, D., Zhou, L., Zhuang, L.: Rex: replication at the speed of multi-core. EuroSys :1-11:14 p. 11 (2014)

[8] Kapritsos, M., 0009, Y.W., Quéma, V., Clement, A., Alvisi, L., Dahlin, M.: All about Eve: Execute-Verify Replication for Multi-Core Servers. OSDI pp. 237–250 (2012)

[9] Kotla, R., Dahlin, M.: High Throughput Byzantine Fault Tolerance. DSN pp. 575–584 (2004)

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

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

[12] Mao, Y., Junqueira, F.P., Marzullo, K.: Mencius: building efficient replicated state machines for wans. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI). pp. 369–384 (2008)

[13] Marandi, P.J., Bezerra, C.E.B., Pedone, F.: Rethinking State-Machine Replication for Parallelism. ICDCS pp. 368–377 (2014)

[14] Marandi, P.J., Primi, M., Pedone, F.: High performance state-machine replication. DSN pp. 454–465 (2011)

[15] Marandi, P.J., Primi, M., Pedone, F.: Multi-Ring Paxos. DSN pp. 1–12 (2012)

[16] Moraru, I., Andersen, D.G., Kaminsky, M.: There is more consensus in Egalitarian parliaments. SOSP pp. 358–372 (2013)

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

Algorithm 1 Dynamic Scalable State-Machine Replication (DSSMR)

```
Initialization:
    oracle: $\mathcal{V}\_part \leftarrow \emptyset$
    client: $cache \leftarrow \emptyset$
    $p \in P$: $variables \leftarrow \emptyset$
Command $C$ is submitted by a client as follows:
    if $getPart(C) \in cache$ and $retryQuery = false$ then
        $dests \leftarrow cache(C)$
    else
        multicast query $Q(oracle, C)$
        $dests \leftarrow O(C)$
    if $dests$ is $\emptyset$ then
        terminate $C$
    else
        multicast($dests, C$)
    receive retry request from server
        $retryQuery \leftarrow true$
Command $C$ is executed by a oracle as follows:
    upon deliver $Q(C)$
        $v \leftarrow getpart(C)$
        if $v$ in $\mathcal{V}\_part$ then
            send reply to client $\mathcal{V}\_part\{v.id\}$
        else
            send reply to client $\emptyset$
    upon deliver $C(v, \mathcal{P})$
        $\mathcal{V}\_part\{v.id\} \leftarrow \mathcal{P}$
Command $C$ is executed by a server in partition $\mathcal{P}$ as follows:
    upon deliver($C$)
        if $\mathcal{V} = getVar(C) \in variables$ and $v.version = v' \in variable$ then
            if C is MOVE command and p.id = C.source then
                $dest \leftarrow C.target$
                $multicastSYNC(dest, \mathcal{V})$
            else
                execute $op$
            send reply to client
        else
            require client to retry
    upon deliver $SYNC(\mathcal{V})$
        $variable \leftarrow \mathcal{V}$
```

Algorithm variables:

$\mathcal{V}\_part$: the set of partitions of all state variables in the system.

$Q(oracle, C)$: multicast command to oracle for querying involved partition of command C

$O(C)$: response of oracle that returns a superset of $part(C)$

$dests$: set of partitions to which C is multicast

$C(v, \mathcal{P})$: command that create variable v in partition $\mathcal{P}$