Heron: Scalable State Machine Replication on Shared Memory

Mojtaba Eslahi-Kelorazi
Università della Svizzera italiana
Switzerland
mojtaba.eslahi@alumni.usi.ch

Long Hoang Le
Università della Svizzera italiana
Switzerland
longlh255@gmail.com

Fernando Pedone
Università della Svizzera italiana
Switzerland
fernando.pedone@usi.ch

***Abstract*—The paper introduces Heron, a state machine replication system that delivers scalable throughput and microsecond latency. Heron achieves scalability through partitioning (sharding) and microsecond latency through a careful design that leverages one-sided RDMA primitives. Heron significantly improves the throughput and latency of applications when compared to message passing-based replicated systems. But it really shines when executing multi-partition requests, where objects in multiple partitions are accessed in a request, the Achilles heel of most partitioned systems. We implemented Heron and evaluated its performance extensively. Our experiments show that Heron reduces the latency of coordinating linearizable executions to the level of microseconds and improves the performance of executing complex workloads by one order of magnitude in comparison to state-of-the-art S-SMR systems.**

I. Introduction

State machine replication (SMR) is an established technique for high availability [1], [2]. Servers replicate a service or application and use consensus to agree on the execution order of client requests. Classic SMR increases the availability of a service but does not improve its performance since each replica stores the complete service state (i.e., full replication) and executes all the requests. Some approaches have proposed to partition the service state (sharding) to boost the performance of state machine replication (e.g., [3], [4], [5]). For example, with state partitioning, S-SMR improved the performance of ZooKeeper, a coordination service, scaling throughput linearly with the number of partitions in some cases and outperforming a fully replicated ZooKeeper by up to five times [3]. With state partitioning, requests must be ordered consistently within partitions and across partitions using an *ad hoc* ordering protocol (e.g., [6], [7]) or a communication abstraction like atomic multicast [8].

Efficiently executing multi-partition requests is challenging. This is because providing classic SMR's consistency guarantees [9], [10] without limiting the scope of multi-partition requests (e.g., computation in one partition cannot depend on data stored in another [11]) requires replicas to exchange data during request execution. In DynaStar [4], a state-of-the-art partitioned SMR system, replicas in the partitions involved in the execution of a multi-partition request migrate the data needed to execute the request to the replicas of a single partition, so that these replicas can execute the request. This data exchange during request execution results in substantial overhead: DynaStar clients experience latency of around 1 millisecond for single-partition requests and 10× as much for multi-partition requests.

For years, practical distributed systems have been developed based on message-passing communication. Recent advances in shared memory technology, however, such as RDMA, have enabled systems to benefit from improved communication performance. RDMA provides the potential for high throughput and low latency communication by bypassing the kernel and implementing network stack layers in hardware. With RDMA, servers can access remote memories without involving the host server's CPU. Compared to message passing-based systems, RDMA introduces two additional complications: multiple servers might access a memory region concurrently (race conditions); and a slow server may miss state updates if a faster server modifies the value before the slow server (lagger) has had a chance to read the value.

RDMA has been used by several high-performance replicated systems (e.g., [12], [13], [14], [15], [16]). This paper introduces Heron, the first partitioned state machine replication system on shared memory (see Figure 1). Heron delivers scalable throughput through state partitioning and microsecond latency by careful use of RDMA primitives. Heron relies on an RDMA-based atomic multicast protocol to consistently order requests within and across partitions [17]. It executes multi-partition requests using a combination of different strategies to handle race conditions and laggers.

Replicas coordinate when executing multi-partition requests to ensure that partitions are synchronized. Each replica is responsible for updating its local data only, that is, a replica can issue local and remote reads, but local writes only. As a consequence, replicas do not contend on write operations. Remote reads issued by a replica in one partition that may conflict with local writes issued by a replica in another partition are handled with a dual-versioning technique, where reads are on the most up-to-date version of the object and updates modify the older version. Replica coordination encompasses a majority of replicas in each partition involved in a request. While coordinating with a majority of replicas (instead of all) avoids blocking due to replica failures, it creates the possibility of laggers, slow replicas that do not keep up with the fast majority. Heron uses simple heuristics to reduce the probability of laggers. Finally, when present, laggers resort to an efficient

state synchronization protocol to update their state.

We extensively evaluate Heron by considering its inherent coordination latency and performance in TPCC workloads. We found that Heron adds very low latency of around 3 microseconds for coordinating executions in a workload in which requests involve 4 partitions. Heron is able to execute complex TPCC single-partition requests in 19 microseconds and multi-partition requests in 35 microseconds. The performance evaluation shows more than an order of magnitude performance improvement when compared to DynaStar [5], a state-of-the-art message passing-based partitioned SMR system. We also evaluate Heron's state synchronization protocol and show that lagging replicas can be swiftly brought back to date. Heron is able to recover a replica in a tenth of a second (e.g., about 100 milliseconds for a TPCC warehouse worth of data).

The remainder of the paper is structured as follows. Section II presents the system model, RDMA, and atomic multicast. Section III discusses the challenges involved in Heron's design, describes its algorithm in detail, and argues about its correctness. Section IV presents our prototype, and Section V evaluates its performance. Section VI surveys related work and Section VII concludes the paper.

II. Background

In this section, we introduce the system model and define linearizability, our consistency criterion (Section II-A), present the guarantees of atomic multicast, the communication abstraction used in Heron (Section II-B), and overview Remote Direct Memory Access (RDMA) technology (Section II-C).

A. Preliminaries

We consider a distributed system consisting of a set of client and server processes. Processes are correct, if they never fail, or faulty, otherwise. In either case, processes do not experience arbitrary (i.e., Byzantine) behavior. The system is asynchronous: there is no bound on message delay or on relative process speed. Our protocols ensure safety under both asynchronous and synchronous execution periods. For liveness, we assume the system is partially synchronous [22], that is, it is initially asynchronous and eventually 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 communication and processing delays; after GST, such bounds exist but are unknown.

Linearizability [9], [10] establishes that there should be a way to total order client requests such that (a) it respects the semantics of the objects accessed by the requests, as expressed in their sequential specifications; and (b) it respects the real-time ordering of the requests in the execution. There exists a real-time order among two requests if one request finishes at a client before the other request starts at a client.

B. Atomic multicast abstraction

In this section, we present the guarantees of the atomic multicast protocol used by Heron to order requests [17]. Let Π be the set of server processes in the system and $\Gamma \subset 2^{\Pi}$ the set of process groups, where $|\Gamma| = k$. Groups are disjoint and each group contains $n = 2f + 1$ processes, where f is the maximum number of faulty processes per group.[1] A set of $f + 1$ processes in group g is a *quorum* in g. In Heron, each process group corresponds to a partition of the system.

A (client) process atomically multicasts a message m to groups in $m.dst$ by invoking primitive multicast(m), where $m.dst$ is a special field in m with m's destinations; a (server) process delivers m with primitive deliver(m). We define the relation $\prec$ on the set of messages processes deliver as follows: $m \prec m'$ iff there exists a process that delivers m before m'.

Atomic multicast ensures the following properties:

- *Validity*: if a correct process p multicasts a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, deliver m.
- *Integrity*: for any process p and any message m, p delivers m at most once, and only if $p \in g$, $g \in m.dst$, and m was previously multicast.
- *Uniform agreement*: if a process delivers a message m, eventually all correct processes $q \in m.dst$ deliver m.
- *Uniform prefix order*: for any two messages m and m' and any two processes p and q such that $p \in g$, $q \in h$ and $\{g, h\} \subseteq m.dst \cap m'.dst$, if p delivers m and q delivers m', then either p delivers m' before m or q delivers m before m'.
- *Uniform acyclic order*: the relation $\prec$ is acyclic.

Uniform acyclic order and uniform prefix order ensure that processes deliver messages consistently across the system. For example, any two processes p and q that deliver both messages m and m', where p and q can be in the same group or in different groups, deliver m and m' in the same order. Uniform prefix order prevents the situation in which messages m and m' are multicast to groups that contain p and q, p delivers m and fails before delivering m', and q delivers m' and fails before delivering m.

The atomic multicast protocol used by Heron assigns a unique timestamp, stored in $m.tmp$, to every delivered message m, such that for any two messages m and m', if $m \prec m'$ then $m.tmp < m'.tmp$. Processes in Heron use timestamps to infer the order of delivered messages.

C. Remote Direct Memory Access

In addition to the atomic multicast abstraction presented in the previous section, processes in Heron can communicate through Remote Direct Memory Access (RDMA). RDMA provides one-sided operations (e.g., read, write), two-sided operations (e.g., send, receive), and atomic operations (e.g., compare-and-swap). The two-sided operations rely on memory copies in user space and involve the CPU of the remote host. Thus, when compared to one-sided RDMA verbs, they

[1] The assumption about disjoint groups has little practical implication since it does not prevent collocating processes that are members of different groups on the same machine. Yet, it is important since atomic multicast requires strong synchronous assumptions when groups intersect [23].

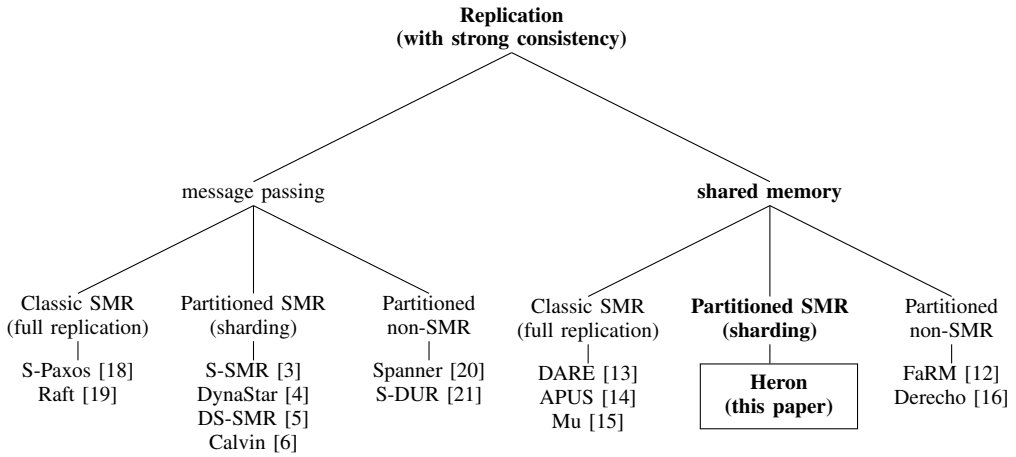

Fig. 1: Message-passing versus shared memory (RDMA) replicated systems (see Section VI for details).

introduce additional overhead [12]. Previous studies have established guidelines to use RDMA operations efficiently [24], [25], [26]. In Heron, processes communicate using remote read and write operations only. We refrain from using remote write operations in the execution of requests since it leads to simpler logic. We resort to remote writes when coordinating the execution and handling state transfer.

RDMA offers three transport modes: Unreliable Datagram (UD), Unreliable Connection (UC), and Reliable Connection (RC). UD supports both one-to-one and one-to-many transmission without establishing connections, whereas UC and RC are connection-oriented and only support one-to-one data transmission. RC guarantees that the data transmission is reliable and correct in the network layer, while UC does not have such a guarantee. Heron relies on RC to provide in-order and reliable delivery. The RDMA-enabled network card on each remote host creates a logical RDMA endpoint known as a Queue Pair, which includes a send queue and a receive queue for storing data transfer requests, to establish a connection between two remote hosts.

III. Scalable state machine replication on shared memory

In this section, we discuss the challenges involved in Heron's design (Section III-A), present Heron in detail (Section III-B), argue about its correctness (Section III-C), and consider a few extensions to the current design (Section III-D).

A. Design overview

In Heron, application state is partitioned (or sharded), for performance, and each partition is replicated, for high availability [3], [5], [11], [20], [27]. Clients use atomic multicast to propagate requests to the partitions involved in the request. A partition is involved in a request if the request reads or writes an object in the partition. This scheme assumes that the objects read and written by a request are estimated before the request is executed. This assumption is common in partitioned SMR systems (e.g., [3], [5], [27], [3]) and in some transactional systems (e.g., one-shot transactions [6]). Moreover, Heron assumes that the execution of a request has a reading phase, during which a replica reads local and remote objects without updating any objects, and a writing phase, during which the replica updates local objects. Once the replica starts the writing phase, it does not read any objects. This assumption is not fundamental and could be relaxed, at the cost of additional complexity in how Heron executes requests.

Single-partition requests are handled as in classic state machine replication: replicas of a partition execute requests deterministically and sequentially in the order induced by atomic multicast. Since requests involve a single partition, all data read and written as part of the execution of a request are local to the replicas of the partition involved.

Multi-partition requests require coordination between partitions and remote operations (Figure 2). After a replica r_i delivers a multi-partition request R (Phase 1) and before r_i executes R, r_i coordinates with replicas in other partitions involved in R (Phase 2) to ensure that these partitions have also delivered R and their state reflects all requests that precede R. Although the coordination used by Heron is analogous to barriers, instead of waiting for every replica of each partition involved in a request, a replica waits for a majority of replicas in the other partitions. This ensures that no replica remains blocked in case of replica failures, as each partition has a majority of correct servers. Coordinating with a majority of replicas only, however, may leave a replica behind other replicas in its partition, a lagger. We describe later in the section how to reduce the likelihood of laggers and how to cope with them when they happen.

In Heron, all partitions involved in a multi-partition request execute the request, reading local and remote objects, and updating local objects only (Phase 3). A replica does not update objects in other partitions, as these objects will be updated by the replicas that host the objects (i.e., their local objects)—in Section III-D we discuss an alternative approach. The execution of a request has a reading step, when the replica reads local and remote objects without updating any objects, and a writing step when the replica updates local objects. In

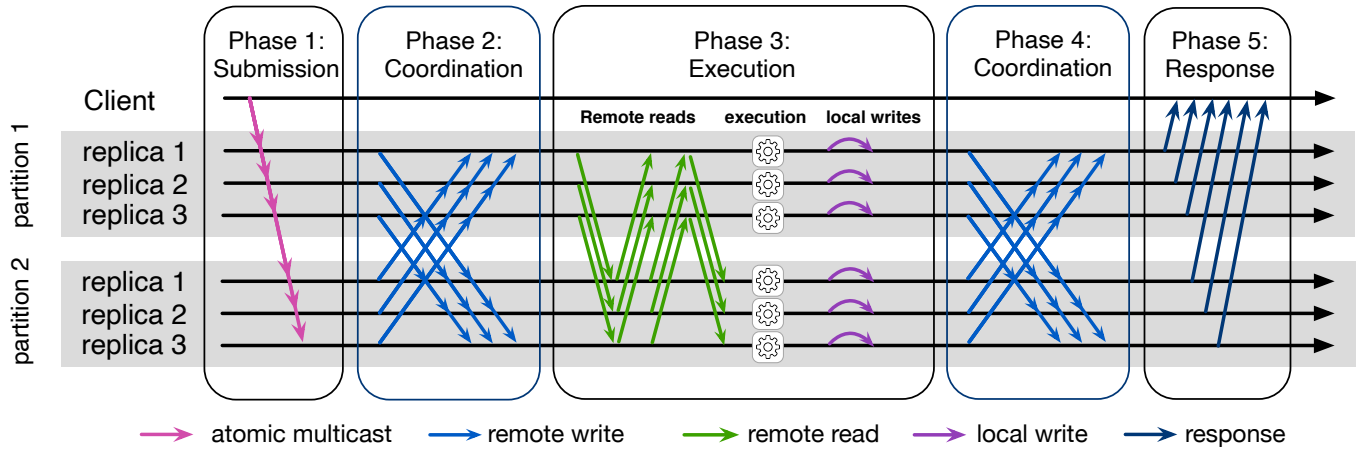

Fig. 2: The lifespan of multi-partition requests in Heron.

order for remote reads issued by replica r_i against replica r_j to be consistent, we need to ensure that (a) r_j has executed every request that precedes R, (b) r_i and r_j do not run into a race condition while executing R (i.e., r_i reads an object while r_j is updating the object), and (c) r_j has not started executing a request that succeeds R.

To ensure case (a), r_i only issues remote reads to r_j if r_j has coordinated with r_i in Phase 2. To avoid race conditions during the execution of a request (case (b)), Heron adopts a dual-versioning technique that keeps two versions of every object. When executing a request, replicas read the most recent version of the object and update the older version. Each version is tagged with the timestamp of the request that creates the version, provided by atomic multicast (Section II-B). To determine the most recent version of an object, a replica compares timestamps and chooses the version with the largest timestamp. To handle case (c), after a replica has executed request R, the replica coordinates with replicas in other partitions involved in R to ensure that remote reads issued by these replicas will be consistent, that is, they do not reflect requests that come after R (Phase 4). Essentially, a replica only moves to the next request after the current request has been executed at every involved partition. After coordinating with replicas in other partitions, a replica replies to the client (Phase 5).

We illustrate the need for Phases 2 and 4 with two counterexamples (Figure 3). In the execution on the left, after delivering request R, replica r_k remotely reads objects x and y, stored at r_i and r_j, respectively. Although requests are delivered consistently across replicas (i.e., $R' \prec R$), this order does not ensure coordinated execution, and r_k reads the value of x from r_i that succeeds R' and the value of y from r_j that precedes R'. This execution is not linearizable as R cannot be both before and after R'. In the execution on the right, thanks to Phase 2 coordination, r_k only reads from the other replicas after they both have reached request R, and therefore completed the execution of R'. This is linearizable since the values read by r_k both reflect R'. Without Phase 4, however, it is possible that r_k completes the execution of R and moves to the next request, R'', a single-partition request that updates object z. This creates the situation in which r_i reads z from r_k before R'' and r_j reads z from r_k after R''. This results in a non-linearizable execution as R cannot be both before and after R''. Phase 4 avoids this problem since r_k can only execute R'' after replicas r_i and r_j have finished R.

Since Phases 2 and 4 only require a majority of replicas in each involved partition, a replica may be left behind other replicas in its partition, a lagger. In this case, the lagger may not be able to execute a multi-partition request because it may not be able to consistently read the value of remote objects. This happens because the replicas that store the object needed have already moved to a later request and updated the object. In Heron, a lagger needs to transfer a consistent state from other replicas in its partition, as described below.

When a replica realizes that it lags behind other replicas in its partition, the replica requests a state transfer to the other replicas in the partition. A replica finds out that it is lagging behind when it reads remote objects with timestamps higher than the timestamp of the request the replica is currently executing. A lagger needs to update its state from the state of other replicas in its partition. To communicate state transfer requests, Heron replicas maintain the State Transfer Memory, an array of RDMA-registered buffers of size equal to the number of replicas in the partition. Each array entry stores two values: *req_tmp* and *status*. *req_tmp* is the timestamp of the request that the replica failed to execute. *status* is the stage of state transfer protocol: 0 when there is no state transfer in execution at the replica and 1 when the replica has requested a state transfer. In addition, replicas maintain a log record of updated values while executing requests in normal execution. This log is used during state transfer to reduce the objects that must be synchronized.

To reduce the probability that a replica r_i lags behind, after coordinating with a majority of replicas in another partition, replicas wait an additional small delay to allow r_i to catch up, should r_i be slower than a majority of replicas in its partition.

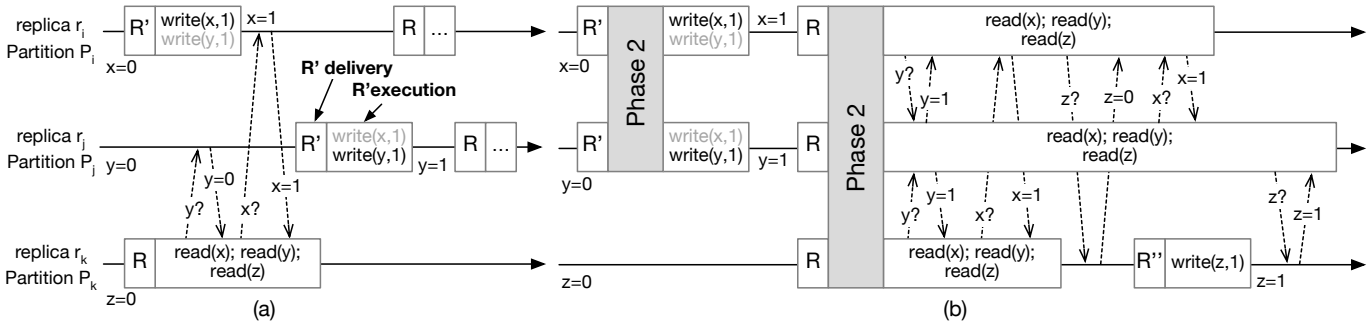

Fig. 3: The need for Phases 2 and 4 (three partitions, one replica per partition). Neither execution (a) nor (b) is linearizable: (a) without Phase 2, R at P_k reads values of x and y that succeed and precede R', respectively; (b) without Phase 4, R at P_i reads a value of z that precedes R'' and R at P_j reads a value of z that succeeds R''. Phases 2 and 4 ensure that the execution is consistent with the order established by atomic multicast: $R' \prec R \prec R''$.

We show experimentally that waiting for a small fraction of the time needed to execute a multi-partition request is enough to practically avoid laggers.

B. Detailed algorithm

Algorithm 1 shows the coordination logic of Heron. Clients submit a request by atomically multicasting it to the destination partitions. Upon delivery of a request, a server process p first checks if the request must be skipped, which is the case when a client has received state updates through state transfer after lagging behind (lines 3–4). In case the request is single partition, process p skips coordinations and executes the request right away (lines 5–7). Otherwise, p executes the coordination phase by writing coordination messages on processes involved in the request and waits for coordination messages from a majority of processes in each involved partition (lines 8–10). Next, the request is executed (lines 11–13). This includes reading states locally and remotely and writing new values locally using *read_objects* and *write_objects* procedures. In Phase 4, p goes through another round of coordination, similar to Phase 2 (lines 14–16). Finally, p responds to the client (line 17).

Algorithm 2 presents the procedures to perform object reads and writes. For each object in *read_set*, process p finds out the object's partition by querying an application-defined partitioning method (lines 2–3). If the object is local, p reads the object value and moves to the next object (lines 4–7). For the remote objects that are read for the first time, p queries the object's address and waits to hear from at least a majority of processes (lines 8–13). This guarantees that p knows the memory address of the object in at least one correct process.

Having the memory address of remote objects, p is able to read object values. For remote reads, p randomly chooses a remote process in partition h (line 15). To ensure consistency, the selected process must be among the ones that p has heard from in Phase 2. If the object address in the selected process is unknown, p chooses another process (lines 16–18). Otherwise, p reads the object value. If the remote process is failed, p finds out about the failure through RDMA exceptions for the read operation and chooses another process (lines 19–21).

While reading object values, Heron's dual-versioning technique ensures that p reads the most recent value. The most recent value is the one with the smaller timestamp than the current request's and is the maximum among the two (line 22). If such a value is not found, it implies that p is lagging and it must initiate the state transfer protocol (lines 23-25). After executing a request, a new object version is created and overwrites the older value of the object (lines 29–31).

Algorithm 3 presents Heron's state transfer protocol. A replica initiates state transfer by remotely writing in a pre-assigned entry in the memory of all replicas in the partition (lines 2–4). Upon reading a state transfer request, a replica is deterministically selected for performing state transfer (line 10). Then, the states to be synchronized are specified (line 12) and the replica synchronizes the states (lines 13–15). At the end of the synchronization, the replica informs the other replicas in the partition about the completion of state transfer by updating the *request id* and *status* values in their memory (lines 16–17). *request id* specifies the last request that its state modifications are synchronized. Finally, p updates its $last\_req$ field to prevent executing earlier requests (line 6). In case the selected replica is suspected (timeout passed), another one is selected for state transfer.

The atomicity and coherence of timestamps are essential for the correctness of remote reads. Timestamps are implemented as integers, whose access is ensured to be atomic by RDMA [28]. There is no ambiguity while deciding the appropriate timestamp while reading remote values because timestamps are ever-increasing consistent integers across involved partitions thanks to atomic multicast. In order to ensure consistency, Heron prevents reading stale values from laggers. This is done by reading values from remote replicas involved in the coordination in Phase 2, which guarantees that the replica has already delivered and executed all the previous requests.

C. Correctness

In this section, we argue that Heron produces linearizable executions: For any execution σ of Heron, there is a total

Algorithm 1 Coordination

```
1:  Process p in group g to execute request r
2:    upon delivery of request r do
3:      if r.tmp ≤ last_req then return
4:      else last_req ← r.tmp
5:      if r.dest.size = 1 then
6:        response ← exec_callback(r)
7:        return response to client
8:      for all h ∈ r.dest, for each q ∈ h do
9:        write_coord(q, p, ⟨r.tmp, 1⟩)
10:     wait til ∀h ∈ r.dest, ∃ majority of q ∈ h :
              coord_mem[h][q].tmp = r.tmp
11:     response ← exec_callback(r){
12:     read_objects(r)...
13:     write_objects(r)}
14:     for all h ∈ r.dest, for each q ∈ h do
15:       write_coord(q, p, ⟨r.tmp, 2⟩)
16:     wait til ∀h ∈ r.dest, ∃ majority of q ∈ h :
              (coord_mem[h][q].tmp = r.tmp &
              coord_mem[h][q].state = 2) |
              coord_mem[h][q].tmp > r.tmp
17:     return response to client
```

Variables:

$r.tmp$: request timestamp

$r.dest$: destination partitions

$r.read\_set$: set of the objects read in the execution of r

$r.write\_set$: set of the local objects modified in the execution of r

$last\_req$: tmp of the last request, initially 0

$object\_list$: set of local objects; there are two versions of objects each tagged with a timestamp (tmp):
- get() returns the value with the higher tmp
- set() overwrites the value with the lower tmp; updates the tmp

$object\_map$: map of $\langle oid, q\rangle$ to the address of object oid in process q

$statesync\_mem[q]$: state sync memory entry for process q

log: log of objects updated while executing requests

$timeout$: time processes wait for state transfer to complete

$coord\_mem[h][q]$: coordination memory entry for proc q in part h

Methods:

$multicast(r, r.dest)$: atomically multicasts request r to $r.dest$

$write\_coord(q, p, v)$: rdma write v to process p's entry in the $coord\_mem$ of process q.

$exec\_callback(r)$: application's execute callback method

$select\_proc(h, r)$: returns a process id from h which coordinated in phase 2 for request r (processes with tmp in $coord\_mem$)

$query\_mapping(oid)$: query the partition that stores object oid

$query\_obj\_addr(q, oid)$: query address of oid in the memory of process q

$write\_state\_transfer(q, \langle r, s\rangle)$: write state transfer msg on proc q for req r with status s

$log.get\_objects(r1.tmp, r2.tmp)$: returns objects updated from req $r1$ to req $r2$ (included)

$rdma\_read(q, addr)$: rdma read memory address $addr$ on process q

$rdma\_write(q, addr, v)$: rdma write value v to memory address $addr$ on process q

Algorithm 2 Execution

```
1:  Procedure read_objects(r):
2:    for oid in r.read_set do
3:      h ← query_mapping(oid)
4:      if h = g then
5:        val ← object_list.get(oid)
6:        r.set_value(oid, val)
7:        continue
8:      if ∀q ∈ h, (⟨oid, q⟩) ∉ object_map then
9:        for all q in h do
10:         query_obj_addr(q, oid)
11:       while not heard from majority in h do
12:         q, addr ← wait()
13:         object_map.set(⟨oid, q⟩, addr)
14:     while true do
15:       q ← select_proc(h, r)
16:       addr ← object_map.get(⟨oid, q⟩)
17:       if addr is null then
18:         continue
19:       res, val1, val2 ← rdma_read(q, addr)
20:       if res is RDMA_EXCEPTION then
21:         continue
22:       val ← value with higher tmp smaller than r.tmp in {val1,
          val2}; null otherwise
23:       if val is null then
24:         invoke state transfer protocol
25:         return
26:       else
27:         r.set_value(oid, val)
28:         break
29: Procedure write_objects(r):
30:   for ⟨oid, val⟩ in r.write_set do
31:     object_list.set(oid, val, r.tmp)
```

Algorithm 3 State transfer

```
1:  Process p in group g to initiate recovery
2:    upon state transfer invoke on request r do
3:      for all q in g do
4:        write_state_transfer(q, ⟨r, 1⟩)
5:      wait on statesync_mem[p].st to be 0
6:      last_req ← statesync_mem[p].rid
7:  Process p in group g to handle recovery
8:    upon state transfer for req r, proc q  do
9:      while true do
10:       proc ← choose a process from g
11:       if proc = p then
12:         objs ← log.get_objects(r.tmp, last_req)
13:         for obj in objs do
14:           addr ← object_map.get(q, obj.id)
15:           rdma_write(q, addr, obj)
16:         for all proc in g do
17:           write_state_transfer(proc, ⟨last_req, 0⟩)
18:         return
19:       else
20:         while timeout not passed do
21:           if statesync_mem[q].st = 0 then
22:             return
```

order π on client requests that (i) respects the semantics of the requests, as defined in their sequential specifications, and (ii) respects the real-time precedence of requests [9], [10].

Let π be a total order of requests in σ that respects $\prec$, the order atomic multicast induces on requests. To argue that π respects the semantics of requests, let C_i be the i-th request in π and p a process in partition x that executes C_i. We claim that when p executes C_i, all read operations issued by p as part of C_i result in values that reflect all requests that precede C_i and no value created by a request that succeeds C_i. We prove the claim by induction on i. For the base step, request C_0, the claim trivially holds for local reads, as objects are initialized correctly. Assume that p successfully reads an object from process q in partition y. Since p only accepts the remote read if the timestamp of the value read is smaller than the timestamp of C_0, p knows that q has not executed any later request that modifies the object read.

For the inductive step, assume the claim holds for $C_0, ..., C_{i-1}$. If p reads a local object, then the claim holds from the inductive hypothesis. Assume that p reads a remote object from process q in partition y. There are two cases to consider. When p reads the object, (a) q has already executed every request that precedes C_i, and (b) q has not executed any requests that succeed C_i. For (a), from the algorithm, p only issues a remote read operation for an object stored on q if q coordinated with p in phase 2. For (b), as in the base step, p only accepts the remote read if the timestamp of the value read is smaller than the timestamp of C_i. Thus, q did not execute any later request that modifies the object read by p when p reads the object from y.

We now argue that π respects the real-time precedence of requests in σ. Assume that C_i ends at a client before C_j starts at a client. We must show that either $C_i \prec C_j$; or neither $C_i \prec C_j$ nor $C_j \prec C_i$. For a contradiction, assume that $C_j \prec C_i$. And let C_k and C_l be two consecutive requests in $C_j \prec ... \prec C_i$, where $C_k \prec C_l$. Thus, there is some partition x involved in C_k and C_l such that servers in x deliver first C_k and then C_l. Since servers execute one request at a time in the order they are delivered, it follows that C_k is executed before C_l by servers in x, and it cannot be that C_l ends before C_k starts. From a simple induction, it cannot be that $C_j \prec C_i$, and so, either $C_i \prec C_j$; or neither $C_i \prec C_j$ nor $C_j \prec C_i$.

D. Extensions

Heron adopts a relatively simple design in that replicas execute one request at a time and multi-partition requests are executed by all partitions involved in a request. We now comment on how to relax these requirements. We note that these are not part of our prototype and are left as future work.

1) Multi-threaded execution: Heron scales performance by partitioning the application state and allowing parallel execution of requests that do not involve partitions in common. Nevertheless, execution within a replica is single-threaded. Several approaches have been proposed to integrate multi-threaded execution of requests in state machine replication (e.g., [29]). A common strategy is to identify requests that do not contain conflicting operations (i.e., requests that do not access common objects or only read objects in common) and assign such requests to different working threads within a replica. Since concurrent requests are non-conflicting, there is no need to synchronize their execution. Heron could directly benefit from this technique to introduce multi-threaded execution of single-partition requests. Multi-threaded execution of multi-partition requests would probably require a redesign of the system.

2) On the execution of multi-partition requests: In general, there are two solutions to the problem of executing a multi-partition request: (a) all involved partitions execute the request (i.e., Heron's approach), and (b) one partition, among the partitions involved in the request, executes the request. In the second solution, to execute a request, the *active partition* reads local and remote objects, and updates its local objects and the remote objects stored in the other partitions involved in the request, the *passive partitions*. This solution saves computing resources, as requests are executed by the replicas of the active partition only. But it complicates the design as replicas in the active partition compete with each other to update remote objects in the passive partitions. Moreover, replicas in the active partition may fail while updating remote objects. Heron avoids these issues by having each replica update its local objects as part of the execution of multi-partition requests.

IV. Implementation

We implemented a prototype of Heron in Java. We use an open-source user-level library developed by IBM for RDMA communication [30] called jVerbs (DiSNI library v2.1).[2] jVerbs offers low latency overhead to applications running Java by exposing RDMA network hardware resources directly to the Java Virtual Machine. Heron relies on RamCast [17],[3] a shared-memory atomic multicast primitive for ordered delivery of requests. RamCast is a state-of-the-art atomic multicast primitive that leverages RDMA writes to reduce the latency of message delivery. Heron's source code is publicly available.[4]

A. TPCC benchmark

We implemented a Java version of TPCC that runs on top of Heron. TPCC is an established standard for evaluating the performance of storage and database systems. TPCC defines a transactional workload for a database system in a wholesale supplier company. The company has a possibly variable number of distributed warehouses (Warehouse table). Each warehouse has 10 districts (District table) and each district services 3,000 customers (Customer table). Warehouses maintain a stock of 100,000 items (Item and Stock tables). The customer orders (Order and New-Order tables) are also stored per order item (Order-Line table), and a history of customers orders are maintained (History table). There are five transaction types that simulate a warehouse-centric order processing application: New-Order (45% of transactions in the

[2]https://github.com/zrlio/disni

[3]https://github.com/longle255/libRamcastV3

[4]https://github.com/meslahik/heron

workload), Payment (43%), Delivery (4%), Order-Status (4%) and Stock-Level (4%).

Each Heron partition stores one TPCC warehouse. The Warehouse and Item tables are replicated in all partitions, since they are not updated in the benchmark. Other tables are warehouse-specific and replicated in one partition. As shown in Figure 2, there is no remote writes in the execution phase. This allows our TPCC implementation to partially execute transactions in some partitions.

Each row in TPCC tables is an object in Heron. To allow processes to access remote objects, these objects must be stored in memory regions, registered with RDMA. Currently, Java does not support Value Types [31]. This prevents us from using Java List to store remotely accessible arrays of objects. One workaround is to store the data in Java's ByteBuffer. The serialized data can then be stored in RDMA-registered memory for remote access. Accessing serialized tables, locally or remotely, involves deserializing the data to retrieve values and serializing again in the case of data modification. The data in two tables, Stock and Customer, are stored serialized. These tables are accessed by remote processes while executing TPCC requests. Other tables are stored in memory using Java HashMap since they are not accessed remotely.

V. Evaluation

In this section, we motivate our experimental study (Section V-A), describe the experiment's environment (Section V-B), and discuss the results of our evaluation (Sections V-C–V-E).

A. Roadmap

We seek to answer the following questions through three sets of experiments:

1) Performance (Section V-C): What is the overall performance and scalability of Heron while running complex transactions (i.e., TPCC)? How does Heron's shared memory model compare to message passing-based scalable SMR systems?
2) Latency (Section V-D): What is the latency of Heron's coordination? What is the latency of running TPCC transactions on Heron?
3) State transfer (Section V-E): How long does it take for Heron to recover a replica? How to determine the efficient cut-off time for coordination?

B. Environment and configuration

We conducted all experiments in CloudLab [32] in XL170 nodes. Each node is equipped with one ten-core Intel E5-2640v4 processor running at 2.4GHz, 64 GB of main memory, and a Mellanox ConnectX-4 NIC. A 25-Gbps network link with around 0.1ms round-trip time connects all nodes running Ubuntu Linux 18.04 with kernel 4.15 and Oracle Java SE Runtime Environment 11. In all experiments, clients and servers are independent processes with in-memory storage. Clients submit requests in a closed-loop, that is, a client submits a request to servers and waits for a response before submitting the next request. Unless stated otherwise, each partition has 3 replicas. For performance experiments, we spawn enough clients to saturate the servers. For latency experiments, we spawn one client to show the inherent latency of the protocol execution. The CDF graphs show how tail latencies are different from the average. Clients measure latency as the interval between submitting a request and the response received from one server in each partition addressed by the request.

C. Performance

1) The performance of Heron: Figure 4 shows the maximum throughput of 4 sets of TPCC experiments as we increase the number of warehouses from 1 to 16. In the first three sets, the ratio of single- and multi-partition requests is given by TPCC. In the last set of bars (Local Tpcc), all requests are local. For 1WH experiments, Heron skips coordination since there is only one partition in the system.

The first set of bars shows the performance of RamCast, without coordination and execution. Ramcast sports a close-to-linear scalability as we increase the number of warehouses. This is a promising result that sets the stage for fast coordination and execution. The second set of bars represents the performance of Heron with null requests. This helps understand the cost of coordination in Heron, without the overhead of request execution. From 1WH to 2WH, performance does not increase due to the overheard of coordination needed in 2WH. Performance increases by factors of 1.57x, 2.98x, and 4.80x thereafter. The third set of bars shows the performance of TPCC on Heron. As before, the performance of TPCC is the same for 1WH and 2WH. Performance for 4WH, 8WH, and 16WH increases by the factors of 1.52x, 2.65x, 3.98x, respectively.

In the above experiments, the performance improvement from 8 to 16 partitions is less pronounced than from 4 to 8 partitions. We attribute this to the network infrastructure of our testbed. According to Cloudlab documentation [33], XL170 nodes are connected via an experimental link to Mellanox switches in groups of 40 servers. Each of the groups' experimental switches are then connected to another Mellanox switch at 5x100Gbps. This means that above 40 nodes, there are always requests that go beyond the Top-Of-Rack switch to reach destinations, with no bandwidth guarantees.

Finally, as a sanity check, we consider a workload with local TPCC transactions only. We modify the TPCC client code so that requests access objects reside in one partition only. In this case, we expect linear scalability since there in no cross-partition requests. The forth set of bars confirms this expectation while executing local TPCC workload.

2) Heron vs. DynaStar: We now compare Heron to DynaStar, a message-passing partitioned SMR system [4] (see Figure 1). We choose DynaStar because it matches Heron's SMR execution model, it supports both single- and multi-partition requests, it has been shown to outperform other related systems, it is available as open source, and it is also implemented in Java. Figure 5 shows peak performance and latency of both systems when executing TPCC as we increase the number of warehouses. In the 16WH configuration, we

ran out of machines for DynaStar to deploy enough clients to saturate the system, which resulted in lower throughput and latency than expected. The performance results show that Heron outperforms DynaStar by an order of magnitude in all configurations considered. Heron improves performance by 17x in the 1WH experiment, up to 27x in the 16WH experiment. The latency results show that Heron has substantially lower latency than DynaStar which has 43.9x, 68.3x, 69.7x, and 72.0x higher latency than Heron for 1WH to 8WH, respectively.

There are three reasons for Heron's impressive performance. First, Heron directly benefits from efficient RDMA verbs, avoiding expensive message-passing primitives (i.e., no overhead with context switches and communication protocol stacks). This impacts both the coordination and the execution of application requests. Second, in Heron, multi-partition requests read remote objects through RDMA verbs, while in DynaStar, the execution of a multi-partition request involves rounds of message exchanges to move objects from one partition to another. Third, Heron benefits from a carefully designed execution path. Optimizations include a manually (de)serialization of objects rather than using a serializer library, and storing strings as byte buffers as (de)serialization of Java Strings is quite expensive.

D. Latency

1) Latency without contention: Figure 6 (bottom bar) shows the breakdown of the average latency when one client submits TPCC New Order requests in a closed loop. We consider a workload with a single client to avoid queuing effects due to contention. The breakdown shows the latency footprint of three stages of running a request on Heron. In this workload, Heron's coordination constitutes only about 2 microseconds of the whole latency of 35.4 microseconds, while ordering and execution take 18 and 16 microseconds, respectively.

We further study the latency of Heron for requests that target a fixed number of partitions (four top bars in Figure 6). For that, we modify TPCC NewOrder transactions so that they access objects in the specified number of partitions. In the 1WH workload, there is no cross-partition requests: all requests are local and there is no coordination. In the 4WH workload, requests always target 4 partitions, accessing at least one object in each of these partitions.

From 1WH to 4WH, all stages of running a request become more expensive. For the ordering, the slight increase in latency is due to the higher number of partitions in the destination of the request. For the execution, the additional latency comes from the fact that more remote objects must be read per request. Coordination latency never goes above 3 microseconds in all workloads.

The CDF graph in Figure 6 reveals more insights about the latency of request execution. For 1WH, all requests are local so latency experiences little variation, with some outliers that constitute about 8% of the requests. In TPCC, about 10% of requests are multi-partition. This results in similar latencies as in 1WH workload for about 82% of the requests. Then, the outliers of single-partition requests show up until about 90% of latency values. Multi-partition requests show even higher latencies. A similar interpretation applies to latencies for other workloads.

2) The latency of TPCC transactions: Figure 7 shows the average latency of various TPCC transaction types. For each transaction type, one client submits that transaction type in a closed loop. The bars differentiate between latencies for single- and multi-partition transactions that expand to multiple partitions (New Order and Payment transactions). The blue bars show the average latency of single-partition transactions. The green bars show the additional latency added by multi-partition requests.

New Order and Payment transactions are heavy transactions. OrderStatus and Delivery transactions are local, light-weight transactions and their latencies are as low as 16.5 and 17.6 microseconds, respectively. StockLevel is a heavy local transaction that accesses items in the last 20 orders. StockLevel transactions are expensive because they access many items in a serialized table (i.e., Stock table), and the data must be deserialized, modified, and stored back serialized.

E. State transfer

1) The impact on latency of "waiting for all": We first measure the impact of tentatively waiting for all replicas in a partition when coordinating. Table I shows the percentage of delayed transactions and the average delay in microseconds in four different configurations: 2 and 4 partitions, and 3 and 5 replicas per partition. A transaction is delayed at a replica if when the replica checks for a majority of coordination messages in its data structures, it does not already have messages from all replicas. The average delay is the amount of time the replica needs to wait to have coordination messages from all replicas, if the transaction is delayed. An important observation from the results is that very few transactions need to be delayed, in the worst case 8%, and the delay per transaction is a fraction of the average latency of a transaction. Since clients wait for a reply from each partition involved in a request, the perceived increase in latency by the client is given by the maximum delay among the partitions involved in the request. Moreover, only the second coordination phase needs to use this additional delay in order to keep replicas in sync.

In all configurations, the percentage of delayed transactions increases with the partition id, while the average delay decreases. This happens because of the order in which a replica updates the coordination data structure in the other replicas involved in a request. In particular, a replica starts with the smallest partition id and then proceeds to the next partition id and so on. Within a partition, the replica updates the other replica in order of their id.

As a result, a replica in the first partition id (among those involved in the request) has higher chances of finding all coordination messages when it checks its data structure than replicas in partitions with higher id. However, the average delay decreases in replicas with larger id because it takes longer for these replicas to have all coordination messages, and

Fig. 4: Performance of RamCast, Heron, TPCC, and TPCC local with increasing number of partitions.

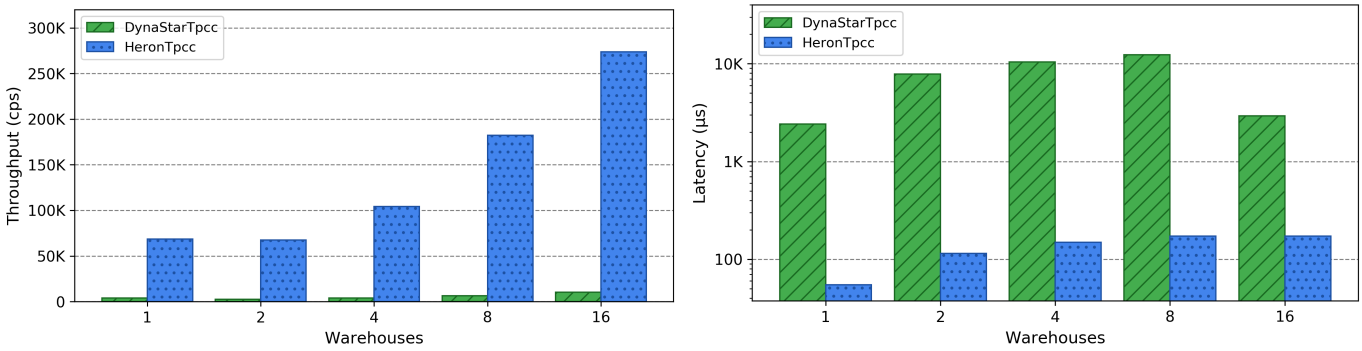

Fig. 5: Performance and latency of Heron vs. DynaStar.

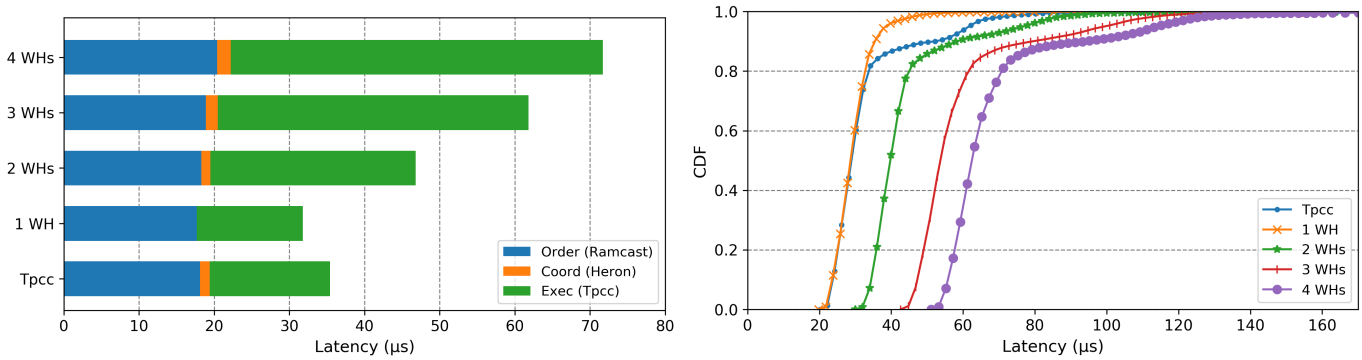

Fig. 6: Heron's latency for single- and multi-partition requests with 1 client: breakdown of average latency (left) and cumulative distribution function (CDF) (right).

so, the increase in latency is not so substantial as in replicas in partitions with smaller id. Also, the delay is so small that the experiments show no meaningful difference between delays in 3- and 5-partition configurations.

2) State transfer latency: Figure 8 shows the latency of the state transfer for TPCC tables in logarithmic scale. For each transfer size, we show average latency (bars) and the standard deviation (whiskers). The standard deviation in all cases shows minimal deviation from the average latency except for the "Protocol" experiment. For state transfer, the data is transferred through RDMA writes with payloads of 32KBs, which has better performance than smaller payload sizes for the same amount of data [17].

The "Protocol" bar shows the latency of state transfer for a null application, when no data is transferred. This represents the overhead of Heron's state transfer protocol without data exchange, and it amounts to two RDMA writes (i.e., one by the replica that requests the state transfer and the other by the replica that responds to this request). The next bars show the state transfer with various data sizes for two scenarios. The two scenarios differentiate between state transfer of serialized and non-serialized data. We chose 64KB as a representative small

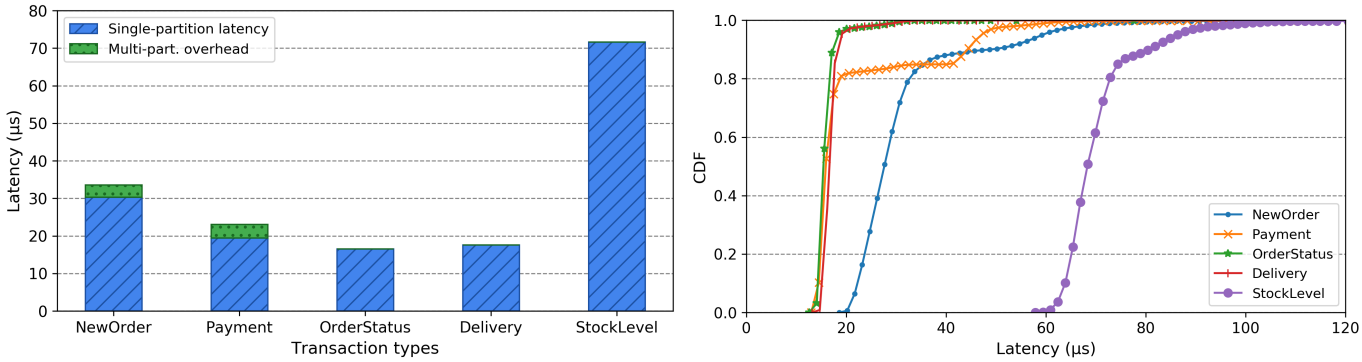

Fig. 7: Latency of TPCC transactions: average latency of single- and multi-partition transactions (left) and cumulative distribution function (CDF) (right).

TABLE I: Transaction delay when waiting for all and a majority of replicas during coordination.

2 Partitions				
	3 replicas per partition		5 replicas per partition	
	max throughput: 53,340 tps average latency: 35.7 μs		max throughput: 42,658 tps average latency: 45 μs	
partition id	delayed transactions	average delay	delayed transactions	average delay
#1	1%	5.3 μs	2%	18.6 μs
#2	8%	4 μs	4%	9.3 μs
4 Partitions				
	3 replicas per partition		5 replicas per partition	
	max throughput: 92,808 tps average latency: 41.3 μs		max throughput: 73,724 tps average latency: 52.2 μs	
partition id	delayed transactions	average delay	delayed transactions	average delay
#1	1%	29.6 μs	3%	16 μs
#2	3%	11.8 μs	3%	11.1 μs
#3	3%	6.9 μs	3%	5.4 μs
#4	4%	2.1 μs	4%	8.8 μs

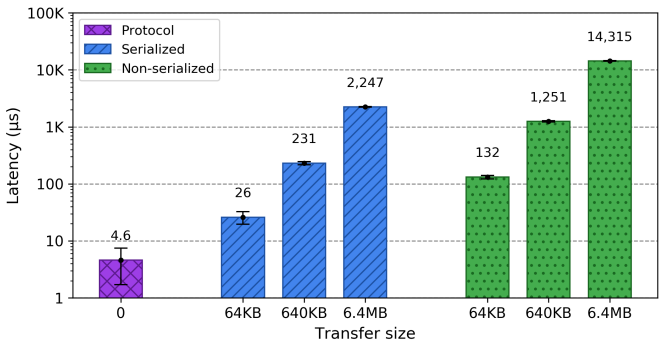

Fig. 8: Latency of state transfer. Protocol shows latency of state transfer protocol without transferring any data. Other bars show state transfer for various data sizes.

data size to be transferred during state sync, while 640KB and 6.4MB show state sync when 1% and 10% of a default TPCC table (i.e., Stock table) is transferred.

In the first scenario, only serialized data (e.g., TPCC Stock table) is transferred. In this case, state transfer includes writing the missing data to recipient's memory where the outdated data resides. Figure 8 shows that for 64KB of data, it takes 26 microseconds for Heron to perform the state synchronization. Latency increases proportionally to data size (640KB and 6.4MB) as expected. In the second scenario, non-serialized data is transferred (e.g., TPCC Item table). In this case, state transfer includes serializing the data and remotely writing the data in a part of the receiver's memory. The receiver then deserializes the data and updates the application states accordingly. The results show that (de)serialization has a considerable degrading effect on the latency.

The time needed to recover depends on how much the replica lags behind. If the replica misses a single request, it may recover in tens of microseconds. In the worst case, upon recovering from a failure, a replica needs to transfer the complete state from another replica. In our prototype, a warehouse stores 137.69 MB worth of data, 105.3MB serialized and 32.39MB non-serialized.[5] This amounts to a transfer time of 109.4ms (36.9ms serialized, 72.5ms non-serialized).

VI. Related Work

In the scope of strong consistency (e.g., linearizability, serializability), one can categorize replicated systems according to

[5]This represents some point during the execution, as some tables in TPCC increase constantly. The changes in size are minimum though and do not impact the state sync time significantly.

three aspects (see Figure 1): (a) how processes communicate, either using message passing (e.g., [4], [5], [18], [19], [20], [21]) or shared memory (e.g., [12], [13], [14], [16]); (b) full replication (e.g., [13], [14], [15], [18], [19]) versus partial replication, that is, sharding combined with replication (e.g., [4], [6], [12], [16], [20]); and (c) systems that embrace SMR's programming model (e.g., [4], [13], [15], [19]) versus systems with a different programming model, notably transactions (e.g., [6], [12], [20], [21]). In this context, Heron is the first partitioned SMR system to rely on shared memory.

A. Message-passing versus shared-memory protocols

Although messaging passing is the prevalent communication paradigm, some replicated systems have explored the potential of shared memory and RDMA. By bypassing the kernel, implementing network stack layers in hardware, and accessing another server's memory without involving the host server's CPU, RDMA promises high throughput and low latency. To benefit from these advantages, however, system designers must address RDMA's challenges, such as race conditions and laggers. We have experimentally compared Heron to DynaStar, a state-of-the-art message-passing replicated system that, as Heron, implements partitioned SMR. The results have shown that shared memory fulfills its promise.

B. Full replication versus partial replication

Classic SMR assumes full replication, that is, each replica stores the whole application state. There have been several proposals for classic SMR, both for message passing [18], [19] and shared memory [13], [14], [15], [34]. DARE [13] is a crash-tolerant replication protocol in which the consensus leader responds to read requests and replicates requests to its followers through RDMA writes. APUS [14] is another leader-based consensus protocol that intercepts inbound socket calls on the leader host and turns these calls into consensus requests. The leader executes the requests and replicates the log entry on followers using RDMA writes. Mu [15] implements Protected Memory Paxos [35], a consensus algorithm that, in normal execution, uses one RDMA write to replicate a consensus request. Mu colocates the client and the leader roles of Paxos for optimizing latency and makes use of memory protection semantics of RDMA for leader change. Velos [34] extends Mu and proposes a leader-based consensus algorithm that relies solely on one-sided RDMA verbs.

Since in classic SMR every replica executes all requests, throughput is determined by how many requests replicas can execute per time unit (or how many requests can be ordered per time unit). Partial replication (i.e., sharding combined with replication) addresses the performance limitation of full replication. Partially replicated SMR systems have been proposed for message passing (e.g., [3], [4], [5], [11]). Marandi et al. [11] introduce a variant of SMR in which data items are partitioned but requests have to be totally ordered and with the limitation that a partition cannot access objects in other partitions. S-SMR [3] and DS-SMR [5] allow partially ordered requests in a statically and dynamically partitioned application, respectively. DynaStar [4] improves on DS-SMR by employing a graph partitioning technique to group frequently used data. Multi-partition requests are executed by a single partition only, after the partition receives all the data needed.

Partial replication has been also explored by shared memory systems that do not comply with SMR's execution model (e.g., [12], [16]). In FaRM [12], applications use transactions to interact with a key/value store that uses RDMA reads for GETs and RDMA writes for PUTs. Derecho [16] introduces a library that allows structuring applications into shards and replicating them. Even though Derecho organizes processes into subgroups and shards, it does not offer any abstraction that provides total order for operations involving multiple shards.

C. SMR versus non-SMR protocols

One distinguishing aspect of SMR is that requests are first ordered and then deterministically executed according to the established order. In partitioned SMR, in order to be effective, the partitions involved in a request have to be identified before the request is executed so that the request can be propagated to and ordered by the involved partitions. An alternative approach (i.e., non-SMR) is to define the order of requests as the execution evolves using locks or optimistic concurrency control (e.g., [12], [20]). While there is no need to identify the partitions involved in a request a priori, requests may need to be undone if they reach a situation in which they cannot be ordered (e.g., after reading an invalid value). Defining ordering during request execution is particularly suitable to transactional systems, as undoing the effects of a request can be implemented with a transaction abort.

Various attempts have been made to increase the performance of state machine replication, targeting both the ordering of requests (e.g., [18], [36]) and the execution of requests (e.g., [37], [38], [39]). These techniques are orthogonal to Heron (but see discussion in Section III-D).

VII. Conclusion

Microsecond latency applications are becoming the de facto standard for latency-critical services. This paper contributes to such systems by introducing Heron, the first scalable state machine replication system that targets microsecond latency applications. Heron's contribution include a novel shared-memory algorithm for coordinating linearizable execution of requests and a state synchronization protocol that recovers lagging replicas very quickly. We have implemented Heron and extensively evaluated its performance. The results show that Heron provides microsecond latency for coordinating strongly consistent executions and achieves more than ten-fold improvement in the throughput of TPCC workloads in comparison to its competitors.

Acknowledgments

We wish to thank Ken Birman and the anonymous reviewers for the constructive feedback. This work was partially supported by the Swiss National Science Foundation (project number 175717).

REFERENCES

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

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

[3] C. E. Bezerra, F. Pedone, and R. van Renesse, "Scalable state machine replication," in *DSN*, pp. 331–342, 2014.

[4] L. Hoang Le, E. Fynn, M. Eslahi-Kelorazi, R. Soulé, and F. Pedone, "Dynastar: Optimized dynamic partitioning for scalable state machine replication," in *ICDCS*, 2019.

[5] L. H. Le, C. E. Bezerra, and F. Pedone, "Dynamic scalable state machine replication," in *DSN*, pp. 13–24, 2016.

[6] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 1–12, 2012.

[7] J. Cowling and B. Liskov, "Granola: Low-overhead distributed transaction coordination," in *Proceedings of the 2012 USENIX Annual Technical Conference*, (Boston, MA, USA), USENIX, June 2012.

[8] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," tech. rep., Cornell University, USA, 1994.

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

[10] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[11] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *DSN*, pp. 454–465, 2011.

[12] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *NSDI*, 2014.

[13] M. Poke and T. Hoefler, "Dare: High-performance state machine replication on rdma networks," in *HPDC*, 2015.

[14] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "Apus: Fast and scalable paxos on rdma," in *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 94–107, 2017.

[15] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. J. Marathe, A. Xygkis, and I. Zablotchi, "Microsecond consensus for microsecond applications," in *OSDI*, 2020.

[16] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. V. Renesse, S. Zink, and K. P. Birman, "Derecho: Fast state machine replication for cloud services," *ACM Transactions on Computer Systems (TOCS)*, vol. 36, no. 2, pp. 1–49, 2019.

[17] L. H. Le, M. Eslahi-Kelorazi, P. Coelho, and F. Pedone, "Ramcast: Rdma-based atomic multicast," in *Proceedings of the 22nd International Middleware Conference*, pp. 172–184, 2021.

[18] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-Paxos: Offloading the leader for high throughput state machine replication," in *SRDS*, pp. 111–120, 2012.

[19] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 305–319, USENIX Association, June 2014.

[20] J. C. Corbett, J. Dean, and M. e. a. Epstein, "Spanner: Google's globally distributed database," in *OSDI*, 2012.

[21] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *DSN*, pp. 1–12, 2012.

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

[23] R. Guerraoui and A. Schiper, "Genuine atomic multicast in asynchronous distributed systems," *Theor. Comput. Sci.*, vol. 254, no. 1-2, pp. 297–316, 2001.

[24] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, cpu-efficient key-value store," in *USENIX-ATC*, 2013.

[25] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *USENIX-ATC*, 2016.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *SIGCOMM*, 2014.

[27] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in scatter," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 15–28, 2011.

[28] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "Apus: Fast and scalable paxos on rdma," in *SoCC*, 2017.

[29] A. Burgos, E. Alchieri, F. Dotti, and F. Pedone, "Exploiting concurrency in sharded parallel state machine replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 9, pp. 2133–2147, 2022.

[30] P. Stuedi, B. Metzler, and A. Trivedi, "jverbs: ultra-low latency for data center applications," in *SoCC*, 2013.

[31] J. Rose, B. Goetz, and G. Steele, "Java value types." http://cr.openjdk.java.net/ jrose/values/values-0.html, 2014.

[32] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 1–14, July 2019.

[33] Cloudlab, "Hardware specification." http://docs.cloudlab.us/hardware.html, 2022.

[34] R. Guerraoui, A. Murat, and A. Xygkis, "Velos: One-sided paxos for rdma applications," *arXiv preprint arXiv:2106.08676*, 2021.

[35] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. Marathe, and I. Zablotchi, "The impact of rdma on agreement," in *PODC*, 2019.

[36] M. Kapritsos and F. P. Junqueira, "Scalable agreement: Toward ordering as a service," in *Sixth Workshop on Hot Topics in System Dependability (HotDep 10)*, 2010.

[37] N. Santos and A. Schiper, "Achieving high-throughput state machine replication in multi-core systems," in *ICDCS*, pp. 266–275, 2013.

[38] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *DSN*, pp. 575–584, 2004.

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