Running Distributed Systems Like Clockwork

Karan Newatia
University of Pennsylvania, USA

Robert Gifford
University of Pennsylvania, USA

Qingjie Lu
University of Pennsylvania, USA

Andreas Haeberlen
University of Pennsylvania, USA

Linh Thi Xuan Phan
University of Pennsylvania, USA

Abstract

Distributed Systems are commonly built using a set of standard assumptions: we assume that message delays are unbounded, that any packet can be lost in the network, and that clocks cannot be closely synchronized. On the one hand, these conservative assumptions result in robust systems that can operate reliably in a wide variety of conditions. On the other hand, they also force the system to do a lot of complex ad-hoc coordination and thus limit the performance it can achieve.

In this paper, we take a look at what lies beyond this standard model. We observe that, on modern hardware in a single-tenant data center, distributed systems are able to closely coordinate and essentially "run like clockwork" with very little effort. If we are willing to additionally rule out some worst-case failure scenarios, this results in a large performance improvement, both in practice and even in theory. We demonstrate this effect using state-machine replication (SMR) as a case study: our SMR protocol, Watchmaker, exceeds the throughput of state-of-the-art algorithms by two orders of magnitude, and it requires only half as many replicas to tolerate the same number of faults.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases State-machine replication, distributed systems, data centers, clock synchronization, fault tolerance, synchrony

Digital Object Identifier 10.4230/OASIcs.NINeS.2026.26

Funding This work was supported in part by NSF grants CNS-1955670, CNS-2111688, and CCF-2326606.

Acknowledgements We thank our shepherd, Scott Shenker, and the anonymous reviewers for their thoughtful comments and suggestions.

1 Introduction

Distributed systems are typically built using a standard set of assumptions: 1) the network is unreliable and can lose, or arbitrarily delay, any packet; and 2) nodes can run at any speed relative to each other and do not have access to synchronized clocks. This *asynchronous model* results in robust systems that can work correctly in a variety of conditions.

However, this robustness comes at a price: without a reliable means of communication or a common notion of time, coordination across nodes is difficult. Seemingly simple tasks, such as ordering events, require elaborate and costly protocols; others, such as distinguishing faulty ones from merely slow ones, are provably impossible. This price is widely accepted

and rarely questioned today – it is seen as simply the cost of doing business in distributed systems. But the price is and remains high, so it seems reasonable to examine whether distributed systems are still getting their money's worth in return.

Do distributed systems need the robustness that the asynchronous model affords? There are indeed many that do! For instance, the Domain Name System shares its links with other traffic that might spike at any time, so it is perfectly reasonable to assume that any packet could potentially be lost; its servers might have other workloads too, so it makes a lot of sense to assume that they could run at any speed; and their clocks could be of low quality or be set inaccurately, so it does seem unwise to rely on them for correctness. But would our answers be the same, say, in a modern single-tenant data center? Here, we can control what traffic is being sent and when [87]; we can know, and control, the software and workload that runs on each node; and we can use high-quality OCXO-based clocks. Even running some extra wires from each node to a shared oscillator, for frequent clock synchronization, does not seem out of the question.

One way for systems to do better in such a controlled environment is to optimize for a "well-behaved" common case, in which the clocks are mostly in sync, packets are rarely lost, and there is little interference from other workloads. Such systems have in fact been built [88], and they do offer better performance. However, *these systems are still fundamentally asynchronous* and thus remain subject to the all the lower bounds, impossibilities, and complexities of the asynchronous model. For instance, SMR still requires $2f + 1$ replicas because we may still need to rely on a quorum if and when the system is not so well-behaved.

Our key observation is that such controlled settings are *almost* synchronous and thus sit right at the edge of an enormous performance cliff: if they really were synchronous, certain impossibilities would no longer apply (no FLP [37]!), lower bounds would improve (agreement with $f + 1$ replicas!), and some complex problems would have trivial solutions. And the synchronous zone seems almost within our reach! Imagine for a moment a hypothetical data center in which everything we can control has been controlled: the machines are in the same rack, they are running a single workload whose timing we thoroughly understand, they are connected to a dedicated switch and closely coordinate their transmissions, and we have strung extra wires to connect them all to a shared oscillator, so they can closely synchronize their clocks. This rules out almost all of the complications that the asynchronous model was meant to handle. The only one that is left is occasional physical-layer packet loss.

What would it take to bridge the gap? At first glance, it seems to require an assumption that packets are *never* lost, which is unrealistic. But in fact, occasional packet losses are not a problem – just like a limited number of node failures is not a problem for fault-tolerance techniques. All it would take is a limit on the *worst-case failure pattern*: if we send the same packet again and again over an otherwise working link, are we assuming that it could be lost every single time – or are we willing to assume that at most k such packets can be lost in a row, much like fault-tolerance techniques assume that at most f nodes can fail? If so, this is enough to unlock most of the benefits of the synchronous model.

In this paper, we use state-machine replication (SMR) as a case study to explore what happens when we add such an assumption. We introduce a *quasi-synchronous* model that is meant to describe a modern single-tenant data center, with the additional assumption that at most k consecutive packets can be lost on an otherwise working link. We then describe an initial SMR protocol, Watchmaker, for this model. Watchmaker uses the additional assumption to closely coordinate between the replicas, so they can indeed "run like clockwork". Thus, Watchmaker achieves a throughput that is two orders of magnitude higher, and tail latencies that are two orders of magnitude lower, than comparable SMR protocols. It also

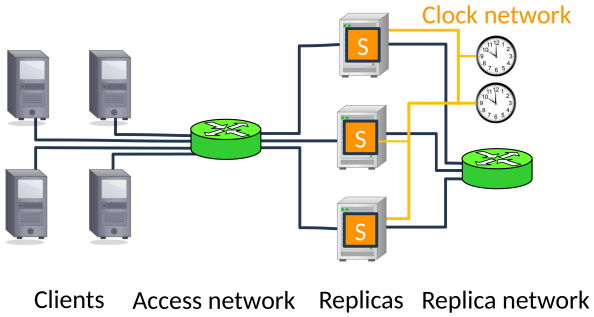

Figure 1 Scenario.

requires only half as many replicas.

To us, Watchmaker is part of a larger point: the standard (asynchronous) model for distributed systems is not the only possible choice, and it is not necessarily the best fit for all possible deployments. We think the quasi-synchronous model is an interesting alternative for modern single-tenant data centers, but more than that, it is a reminder that *there are alternatives* – and Watchmaker illustrates just how much such alternatives can influence the properties of the system. Exploring other models, and their consequences for other types of algorithms, could be an interesting avenue for future work. In summary, our contributions are:

- the quasi-synchronous model (Section 2);
- the Watchmaker algorithm (Section 3);
- a prototype implementation of Watchmaker (Section 4); and
- an experimental evaluation of Watchmaker (Section 5).

We discuss related work in Section 6 and present our conclusions in Section 7.

2 Overview

For concreteness, we assume that an SMR system, when deployed in such a carefully-controlled setting, would look something like Figure 1. A critical service S is implemented as a deterministic state machine; there is a set of *clients*, who can generate requests for the state machine to execute, and a set of N *replicas* that each run a copy of S. The clients are connected to the replicas by an *access network*, about which we make no special assumptions. The replicas are connected to a *replica network* to which they have exclusive access, and additionally to a special *clock network* that connects them to a set of C clocks, which provide periodic pulses that the replicas use to synchronize their local clocks. As usual in SMR, our goals are that 1) the clients should see consistent responses that could have been sent by a single instance of S, and that 2) for some $1 \leq f < N$ and $1 \leq c < C$, the service as a whole should remain available even when up to f of the replicas and up to c of the clocks fail by crashing.

Following Fred Schneider's definition [92], a valid SMR protocol has to provide the following two properties:

- **Agreement:** Every non-faulty state machine replica receives every request.
- **Order:** Every non-faulty state machine replica processes the requests it receives in the same order.

If the state machine S is deterministic and all replicas start in the same state, this is sufficient to ensure that the responses from the non-faulty replica will be identical.

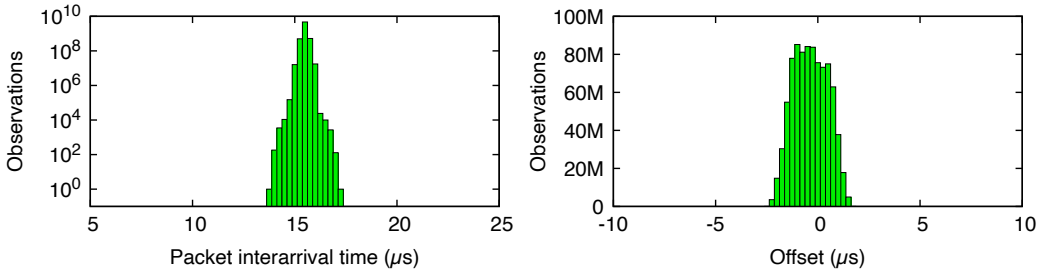

Figure 2 Stability of packet timing (left) and clock synchronization accuracy (right).

2.1 Physical-layer loss is rare

Is it even plausible to assume a bound on the number of consecutive packet losses on an otherwise working link? To approach this question, we did a little experiment: we connected two Dell PowerEdge R440 servers with Intel X710 NICs to an optical 10 Gbps Ethernet switch (a Juniper EX4550-32F-AFI). One was configured to send a steady stream of 1,500-byte packets, each with a sequence number and a predictable bit pattern, at a rate of 66K packets per second, or about 800 Mbps; the machine was configured to receive the packets, check the bit pattern and the sequence number, and report any omissions or signs of corruption. To rule out kernel issues, we wrote a custom NIC driver and ran our code directly on the hardware. During a 24-hour run, 6 billion packets were sent. Figure 2 (left) shows that the packet timing was very stable: inter-arrival times were consistently between 13 μs and 17 μs. The number of omissions or corruptions we saw during this time was *exactly zero*.

This is not surprising: data-center hardware is amazingly reliable these days. When a link is working, packet loss is almost always due to queue overflow, which can be avoided, e.g., with Infiniband or through careful coordination [6, 87]. Of course, link and switch failures can and do occur, and individual packets are still lost occasionally: in a small data center with 100 fully-utilized 10 Gbps links and the 10^{-8} packet loss rate from the Ethernet standard, one would expect roughly one loss per second in the entire data center.

2.2 Clock can be closely synchronized

In our setup, is it possible to synchronize clocks well enough that we can rely on them for correctness? Historically, both clock chips and clock synchronization protocols have had a reputation of being unreliable, but both have come a long way: modern oven-controlled crystal oscillators (OCXOs) can achieve a frequency stability of ± 1 ppb or less [95], which corresponds to a worst-case drift of 86μs in an entire *day*, and modern synchronization protocols can achieve amazing accuracy: Sundial [70] offers a bound of $\pm 0.1\mu$s, Huygens [40] gets to tens of nanoseconds, while Firefly [78] and Synchronous Ethernet [48] reach sub-10 nanoseconds.

In our example setting from Figure 1, we have made the problem even easier: the machines have access to a shared pulse (say, 10kHz) and can use it to periodically synchronize their clocks; thus, their internal clocks need only be precise enough to cover the 100μs between pulses. In practice, this is a little harder than it sounds, because commodity servers do not have a "clock port" that such a pulse can be fed into. However, we did a second experiment to check whether an existing pin – in our case, the CTS pin on a server's RS-232 port – can be used instead. We used a Siglent SDG1032X function generator to generate a

Table 1 Assumptions A1–A10 in the quasi-synchronous model (left) and idealized equivalents A1'–A7' (right) we use for ease of exposition. A11 + A12 are standard for SMR.

Assumption	Quasi-synchronous		Fully sync.	
Channel	A1	Links+switches	Broadcast	A1'
Prop. delays	A2	Any, but known	Uniform	A2'
Link capacity	A3	Finite+known	Infinite	A3'
Corrupt. loss	A4	$\leq k$ in a row	No	A4'
Link failures	A5	Yes	No	A5'
Partitions	A6	Yes	No	A6'
Clocks	A7	Diff. $\leq \Phi$	Identical	A7'
Queue lengths	A8	Finite but known		
Topology	A9	Known		
Cross traffic	A10	Absent		
Node faults	A11	Crash		
Node IDs	A12	Available		

10kHz pulse, connected it to the CTS pin, and measured the number of ticks on the CPU's cycle counter between consecutive pulses. Figure 2 (right) shows our results: all of the values were within $\pm 3\,\mu$s. In absolute terms, this precision is not great – the CTS pin has to be read with our Xeon CPU's `inb` instruction, which is not optimized and can take a few microseconds. But the key point is that this bound is *rock-solid* and can be relied upon for correctness. Regular clock synchronization protocols use regular network packets, so there is always a question mark behind their precision: what if spiky cross traffic throws off their estimates? But with dedicated wires, all the machines truly see the same pulse at the same time.

2.3 The quasi-synchronous model

What assumptions can we make about such a closely controlled system? For the purposes of this paper, we choose ten assumptions A1–A10 (see Table 1); we will refer to these collectively as the *quasi-synchronous model*. At a high level, these assumptions imply the existence of bounds Δ and Φ on message latencies and clock discrepancies, respectively; the difference is that we do not simply assume these bounds but rather that the system has the *means* to guarantee them. If clocks are synchronized (A7) and propagation delays (A2), link capacities (A3), queue lengths (A8), and the network topology (A9) are all known, the nodes can carefully coordinate their transmissions to prevent queue drops and bound queueing delays. If the network will not lose more than k consecutive packets from the same sender (A4), an algorithm could theoretically ensure reliable delivery simply by sending each packet $k+1$ times. (Of course Watchmaker does not do this – it would be extremely wasteful.) Assumptions A5 and A6 account for the possibility that the network might partition and that links or switches might fail.

Notice that, despite the similarity to the synchronous model, proofs for the synchronous model will not necessarily hold in the quasi-synchronous model. One obvious reason is that most liveness guarantees can only hold in the absence of partitions, but there is also a more subtle reason: the synchronous model does not limit *how many* messages can be sent, so its bound Δ can only hold for networks with infinite capacity. Since we assume a more realistic network with finite capacity, most proofs would need to additionally assume that the capacity is sufficiently high.

```
 1: replicas : set of nodes
 2: cur : node = replicas.rand()
 3: active : set of (int,request)
 4: procedure SENDREQUEST(req)
 5:     now = getTime()
 6:     setTimeout(now+T, now)
 7:     pending.add(now,req);
 8:     send(cur,(now,req))
 9: procedure RECEIVE(orig,msg)
10:     (ts,resp) = msg
11:     if active.contains(ts,_) then
12:         deliver(resp)
13:         pending.remove(ts,_)
14: procedure TIMEOUT(x)
15:     cur = replicas.random()
16:     setTimeout(getTime()+T, x)
17:     send(cur,pending.get(x)))
```

```
1: procedure RECEIVE(src,msg)
2:     if isClient(src) then
3:         (ts,req) = msg
4:         broadcast((src,ts,req))
5:     else
6:         (orig,ts,req) = msg
7:         resp = execute(req)
8:         send(orig,(ts,resp))
```

Figure 3 Client side (left) and server side (right) of a fully synchronous SMR algorithm (using assumptions A1'–A7').

3 Watchmaker

How do these different assumptions affect algorithm design, and the guarantees that can be given? To illustrate this, we now present Watchmaker, an SMR protocol for the quasi-synchronous model. Since our goal is to explain the design approach and to highlight the differences between the quasi-synchronous model and the synchronous one, we do not just present the completed algorithm but rather derive it in several steps, starting from a fully-synchronous strawman.

3.1 Fully synchronous protocol

We begin by temporarily replacing the quasi-synchronous assumptions A1–A7 in Table 1 with assumptions A1'–A7'– that is, we assume a reliable, totally-ordered broadcast channel, clocks are exactly synchronized, and link failures and partitions cannot occur. This is one of several ways to describe the fully-synchronous model; it is sometimes also described in terms of synchronous "rounds", where all messages sent in one round are also received in that round [33, §3], but these formulations are roughly equivalent.

In this model, SMR is almost trivial: when a replica receives a request from a client, it can simply broadcast it. The replicas can then execute the requests in the order they are received, and respond to the client. For concreteness, we show pseudocode in Figure 3; this will be our baseline for modifications. Agreement and ordering hold simply because of the broadcast channel, and the state machine will remain available as long as at most $f \leq N-1$ of the replicas fail.

3.2 Finite capacities

What happens if we make the more realistic assumption that the capacity of the broadcast channel is finite but known (A3)? In this case, the replicas cannot simply broadcast requests whenever they arrive, since they might fill up queues in the network and cause packet loss. However, the replicas can use the synchronized clocks to coordinate their transmissions – e.g., using a form of TDMA, where each replica is given a fixed set of "transmission slots" to use.

```
 1: sendWait : fifoQueue                ▷ Client requests waiting to be broadcast
 2: execWait : priorityQueue                ▷ Requests waiting to be executed
 3: procedure INIT
 4:     timeout(0)
 5: procedure RECEIVE(src,msg)
 6:     (src,ts,req) = msg
 7:     if isClient(src) then
 8:         sendWait.append(src,ts,req);
 9:     else execWait.insert(ts,msg)
10: procedure TIMEOUT(x)
11:     if sender[x mod L]=myID then
12:         broadcast(sendWait.empty() ? sendWait.pop() : ⊥)
13:     tSafe = getTime() - maxDelay        ▷ maxDelay is 0 in 3.2, Δ in 3.3
14:     while execWait.minPrio() ≤ tSafe do
15:         (orig,ts,req) = execWait.pop()
16:         resp = execute(req)
17:         send(orig,(ts,resp))
18:     setTimeout(getTime()+σ, x+1)
```

Figure 4 Server-side algorithm for finite network capacity.

Figure 4 shows a modified server-side algorithm for this scenario. (The client side from Figure 3 (left) will continue to work). Instead of broadcasting each request as it arrives, the replicas now append the requests to the `sendWait` queue, and broadcast as many as possible when the next transmission slot arrives. For simplicity, we assume here that there is a fixed transmission schedule that consists of L packet-sized slots, with σ being the transmission time for a single packet on the channel, and slot i being assigned to node `sender[i]`. The concrete schedule does not matter, as long as each replica gets at least one slot. When transmissions arrive, they are put into the `execWait` set and are executed when the next slot timer hits. This set, and the `maxDelay` constant, will become important later; for now, we set `maxDelay`=0.

Agreement continues to hold because, although queue drops are now possible theoretically, they cannot occur in practice because transmissions are coordinated and there is no cross traffic (A10) that could fill up the queues. Ordering holds because the packets are executed in the order of the timestamps they contain, perhaps using the node IDs (A12) as a tie breaker.

3.3 Point-to-point links

Next, we replace the broadcast channel with ordinary links and switches (A1) and allow arbitrary propagation delays (A2). This creates a new complication: although we are still sending broadcast packets, the network will send these packets along a spanning tree and create copies at each internal node. Thus, if two nodes each broadcast a packet, these packets may arrive on different nodes in a different order, and their copies may be queued in front of each other on some paths, if they happen to be in the same queue at roughly the same time. To maintain agreement, we must now choose the transmission schedules carefully, so that queue overflows cannot occur; to maintain ordering, we can no longer execute each request immediately upon arrival; rather, we need to wait until we can be sure that no packet with an earlier timestamp could still arrive.

We can find a suitable set of transmission schedules by essentially simulating the network: we can inject simulated packets according to each node's schedule, and we can then simulate what happens in each queue and on each link. Figure 5 illustrates this process with a simple example with three nodes and the non-uniform latencies shown on the left. The candidate

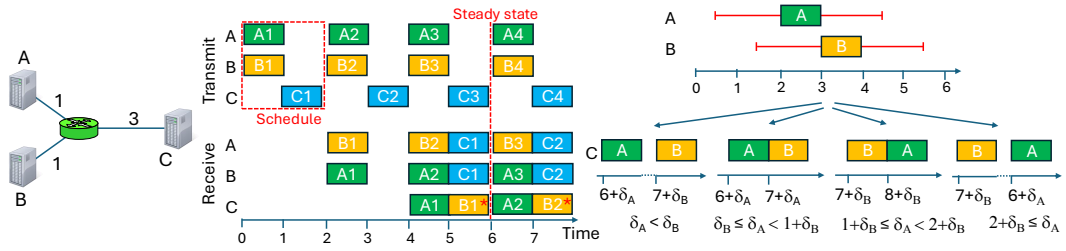

Figure 5 Example topology (left); transmission schedule for 3 replicas with non-uniform latencies (middle); scenario with $\Phi = \pm 1.5$ of clock drift (right).

schedule in the middle is two time units long and is repeated over and over; the network reaches steady state at $t = 6$. A's and B's packets reach the switch at the same time and one of them (here, B's packets) must wait in C's downstream queue, so, for this schedule to work, the queues at the switch must be able to hold at least two packets. Only packets from B to C experience queueing delay (one time unit, marked with asterisks in the figure), so the effective one-way delay is 2 from A to B and from B to A; 4 from A to C; and 5 from B to C. With this schedule, the algorithm in Figure 4 continues to be safe if we set `maxDelay`= 5. If a packet drop had occurred during the simulation, we could have repeated the process with a different schedule. This step is needed only once: once a suitable schedule has been found for a given replica network, it can be used without further simulation.

The schedule does not necessarily have to give the same number of slots to each node – different nodes can be given slots at different rates, as long as all the nodes use the same, consistent schedule. The schedule can even be changed at runtime, since the nodes can use the synchronized clocks to switch to a new schedule at a consistent point in time. In our prototype, we use this feature to adjust to imbalanced and/or changing workloads.

3.4 Imperfect clock synchronization

Next, we change our assumption about clocks and require only that they are synchronized up to some known upper bound Φ (A7). At first glance, this seems dangerous because we relied on the clocks for both ordering and coordination. However, recall that for ordering, it is sufficient if, once we execute a request, no other request with a lower timestamp can arrive. For this, we can simply wait a little longer.

How much longer? We can find the answer by adjusting our network simulation. We can 1) introduce a variable δ_i for the offset of node i's clock, relative to some hypothetical global clock, we can 2) express the transmission times of the packets as global timestamps plus some linear combination of these variables, and we can 3) maintain multiple "possible worlds", each with particular constraints on the values of the δ_i, roughly analogous to symbolic execution [55].

For instance, suppose $\Phi = \pm 1.5$, and assume that, in the topology from Figure 5, A and B are supposed to broadcast packets at global times $t = 2$ and $t = 3$, respectively. Clearly, copies of both packets need to pass through C's downstream queue. In the simpler simulation from Section 3.3, the switch would simply have sent A's packet at $t = 3$ and B's packet at $t = 4$, but with $\Phi = \pm 1.5$, the packets could arrive at the switch in any order, and they may or may not be in the queue at the same time. Thus, there are four possible cases: 1) the switch finishes sending A's packet before B's packet arrives; 2) B's packet arrives while A's packet is being sent; 3) A's packet arrives while B's packet is being sent; or 4) the switch finishes sending B's packet before A's packet arrives. Figure 5 shows the

```
 1: sendWait : fifoQueue                                   ▷ Client requests waiting to be broadcast
 2: ctrlWait : fifoQueue                                   ▷ Control messages waiting to be broadcast
 3: execWait : set                                         ▷ Requests waiting to be executed
 4: highWater : map                                        ▷ Each other replica's high water mark
 5: termSeq : map                                          ▷ Terminal sequence number
 6: numOmit : map                                          ▷ #consecutive missing packets by sender
 7: deadDecl : set
 8: alive : set = {1, 2, ..., N}                           ▷ Who the local replica thinks is alive
 9: myHighWater: int = -1
10: execUpTo : int = -1
11:
12: procedure TIMEOUT(x)                                   ▷ Invoked once per slot
13:     RECEIVEFROMCLIENTS()
14:     RECEIVEFROMREPLICAS(x)
15:     DOBOOKKEEPING()
16:     if sender[x mod L]=myID then SEND(x)
17:     EXECUTE()
18:     setTimeout(getTime()+S, x+1)
19: procedure SEND(x)                                      ▷ If it is our turn to send, make a transmission
20:     if !ctrlWait.isEmpty() then
21:         (origSeq,msg) = ctrlWait.popFirst()
22:     else if !sendWait.isEmpty() then
23:         (origSeq,msg) = (x,sendWait.popFirst())
24:     else
25:         (origSeq,msg) = (x,⊥)
26:     broadcast(x,origSeq,myHighWater,msg)
27:     execWait.add(x,origSeq,msg)
28: procedure EXECUTE                                      ▷ Execute requests when it is safe to do so
29:     safe = min_{i∈alive} highWater.get(i)
30:     while execUpTo<safe andxex(_,execUpTo+1,req) ∈ execWait do
31:         execWait.remove(_, ++execUpTo, _)
32:         if req = REQ(orig, id, req) then
33:             resp = execute(req)
34:             sendToClient(orig,(id,resp))
35: procedure RECEIVEFROMCLIENTS
36:     while (orig,id,req) = messageFromClient() do
37:         sendWait.append(REQ(orig,id,req))
38: procedure RECEIVEFROMREPLICAS(x)
39:     dt = min_y ( OneWayDelay(myID,sender[(x-y) mod L]) = y)
40:     expSlot = (L + x - dt) mod L
41:     expSrc = sender[expSlot]
42:     if (src,(seq,origSeq,srcHWM,msg)) = msgFromReplica() then
43:         numOmit.put(src, 0)
44:         if origSeq ≠ seq then execWait.add(seq,origSeq,msg)
45:         execWait.add(seq,seq,msg)
46:         highWater[src].put(srcHWM)
47:         if srcHWM< expSlot and (srcHWM+1, _, req)∈execWait then
48:             ctrlWait.add(srcHTM+1,req)
49:         if msg = DEAD(who,lastSeq) then
50:             deadDecl.add(src, who, lastSeq)
51:     else // Message from expSrc is missing
52:         missingSoFar = numOmit.get(expSrc) + 1
53:         numOmit.put(expSrc, missingSoFar)
54:         if missingSoFar = k + 1 then
55:             lastReceived = max_s ((expSrc, s, _) ∈ execWait)
56:             for i = 1 to k + 1 do
57:                 ctrlWait.add(DEAD(expSrc,lastReceived)
58:             deadDecl.add(myID, expSrc, lastReceived)
59: procedure DOBOOKKEEPING
60:     // Figure out who has been declared dead by all live replicas
61:     D = max_D ∀d ∈ D, o ∈ ( alive \D) : (o, d, _) ∈ deadDecl
62:     for d ∈ D do
63:         alive = alive \{d}
64:         highWater.remove(d)
65:         for f ∈ N \ D do
66:             termSeq.put(d, max{q : (_, d, q) ∈ deadDecl })
67:     while true do                                      ▷ Compute longest contiguous prefix of received messages
68:         expSrc = sender[myHighWater+1 mod L]
69:         if (myHighWater+1, _, _) ∉ execWait and (expSrc ∈ alive or
70:      termSeq.get(expSrc) > myHighWater+1) then break
71:         myHighWater ++
72: procedure INIT
73:     timeout(0)
```

Figure 6 Server-side algorithm for Watchmaker.

values of δ_A and δ_B for which each case arises. We can then simply "fork" four copies of the simulation, one for each case, with the transmission times expressed symbolically in terms of δ_A and δ_B, forking additional copies as necessary; we can reject a candidate schedule if *any* scenario results in a queue overflow, and we can set `maxDelay` to the highest one-way delay in any scenario. This computation is expensive, but it only needs to be done once for each topology, and it can be done offline.

3.5 Occasional packet losses and crash faults

Next, we admit the possibility of occasional packet losses (A4), as long as not more than k consecutive packets from a given sender are lost. (This is the key "leap of faith" that the quasi-synchronous model requires.) One simple way to leverage A4 would be to just send copies of every request in $k+1$ consecutive slots. However, that would be quite wasteful – especially given that, to be conservative, one may want to choose k to be somewhat large. (For instance, $k = 4$ would result in a 75% loss of potential throughput.) However, we are able to do better because, since transmissions are coordinated, nodes know when to expect packets and can thus reliably *detect* when a loss has occurred. Moreover, we can use A4 to distinguish between corruption losses and crash faults: a sequence of k or fewer missing packets is due to corruption, and any longer sequence is because the sender has crashed.

Thus, we can adjust the server-side algorithm to the one in Figure 6. (We keep the client-side algorithm from Figure 3, except that we now use the client's timestamps only as unique identifiers; the clients' clocks need not be synchronized with the replicas' clocks.) Whenever a replica i broadcasts a packet, it includes 1) a sequence number, 2) the original sequence number of the packet, if it is being retransmitted; and 3) the highest sequence number s such that i has received *all* transmissions with sequence numbers up to s (with an important exception we will discuss in a moment). We refer to this as i's *high-water mark*. When another replica notices that i's high-water mark s_i is lower than it should be, it retransmits the packet with sequence number $s_i + 1$, assuming it has a local copy. This retransmission is given a new, higher sequence number $s'_i > s_i$, but, to prevent replicas from "falling behind", it counts for both slots s_i and s'.

But what if a replica crashes and stops broadcasting packets? To prevent the other replicas from becoming "stuck", we must allow them to skip the crashed replica's slots. We can reliably detect crashes using assumption A4, so, if a replica i is missing $k+1$ transmissions from another replica j and s'' was the sequence number of the last packet i has received from j, it seems tempting to just let i skip all of j's slots after s''. However, even though in this situation i can be sure that j has crashed, it cannot be sure that s'' was truly the last transmission; other replicas may have received a few more packets with higher sequence numbers. To handle this case correctly, we require i to broadcast a DEAD(j,s") message in this situation, to pronounce j dead only after it has received similar messages from all the other live replicas, and to skip j's slots only after the *highest* sequence number that *any* live replica has received from j. In this (rare) case, we can afford to ensure delivery of this key message by sending $k + 1$ copies back-to-back.

If losses are possible, we can no longer rely on timestamps alone when executing requests. Suppose a replica i executes a request r, responds to the client, and then crashes immediately. If all the other copies of r happen to have been dropped in the network, the request is lost. Instead, we execute requests once they are covered by the high-water marks of all the live replicas; recall that, if losses are extremely rare, this almost always happens after one RTT. As before, requests are executed in the order of their sequence numbers, and they can be garbage-collected immediately after execution, since at this point all other replicas have

received them.

3.6 Link failures and partitions

We now remove the last two of our idealized assumptions and admit link failures (A5) and partitions (A6). This adds another possible explanation for why a replica i might see k consecutive losses from another replica j: instead of having crashed, j might still be alive but is unable to reach i. If a link failure disconnects a link (a, b) but does not disconnect the entire network, there must be a node c that can reach both a and b. In this situation, both a and b will be stuck in the WAIT state, since they both consider each other faulty but their LIVE sets will never match c's, who can still reach both of them. This is not a safety issue, and liveness is preserved for the remaining maximum clique; although one could in principle keep a and b alive by redirecting their traffic through c, we choose not to do this here, to keep things simple.

If a link failure partitions the network, the challenge, as usual, is to prevent the state in different partitions from diverging. A simple way to do this is to simply stop executing requests once the `live` set no longer contains a majority of the replicas; that way, at most one partition can be live at any given time. Notice that, if this last change is made, Watchmaker requires $2f + 1$ replicas to tolerate f faults. In contrast, all previous versions require only $f + 1$ replicas, which is the lower bound for synchronous systems [33]. We have included a formal proof of correctness in Appendix A.

3.7 View changes

In a long-running deployment, handling replica crashes (Section 3.5) is not enough – there must also be a way to perform a view change that replaces dead replicas with new ones. As usual in SMR protocols, this can be done by having the "old" set of replicas process the view-change command as a special request, much like a normal client request, except that its execution changes the set of replicas. In Watchmaker's case, the special request should also contain a new transmission schedule (Section 3.3) and a restart time, sufficiently far in the future, at which this new schedule is to be activated. Until then, the old replicas continue with the protocol, to handle any individual packet losses (Section 3.5) and any messages that are still in flight, but they hold any new client requests. Any new replicas would have to already be connected to the clock network and the replica network, although they could of course perform other functions until they are needed.

4 Implementation

In this section, we describe the two prototype implementations of Watchmaker that we built for our experiments.

- **Custom kernel:** For best results, Watchmaker needs tight control over the timing of network transmissions and computations – more than commodity kernels typically provide. (Even most real-time kernels focus more on meeting deadlines than on synchrony: they will guarantee that certain things are done *before* a certain point in time, but not necessarily *at* a certain point in time.) Hence, we decided to write a small kernel of our own. Our kernel consists of 1,434 lines of C++ code and 132 lines of assembler; it runs on standard x86 hardware and has a driver for the Intel X710 NIC, as well as for the RS-232 port we use to keep the clocks in sync. We omit 1) graphics and HID

drivers because the replicas typically run headless, 2) storage drivers because our state machines are not persistent, and 3) support for the many advanced feature of the X710 NIC that we do not use. With this custom kernel, we can control most sources of "time noise" [22] and can thus get tight bounds for timing.

- **Linux version:** To demonstrate that our approach does not fundamentally *depend* on a custom kernel, we also implemented a version of Watchmaker on Linux 6.0.12 and DPDK 20.11.0. This cannot match the performance of our custom kernel because Linux was not designed for predictable timing and thus produces a lot of time noise. We took a variety of measures to reduce this "noise" as much as we could; for instance, we disabled TurboBoost and pinned the threads to specific cores. (See Section 5.1 for a list of measures.) Additionally, since we were not able to access the CTS pin from user level, we used periodic beacon packets, sent from an additional machine over another dedicated switch to a separate NIC port, as the shared clock signal in the Linux experiments. Despite all this, we would not recommend our prototype for production use without a much closer look for sources of time noise that we may have missed.
- **Clock synchronization:** Watchmaker needs 1) a rock-solid way to synchronize clocks, without estimates or heuristics, and 2) control over precisely when a given network packet will be sent. Ideally, these features would be provided by the hardware, perhaps in the form of a simple "clock port" (see Section 2.2) and an extra "time" field in the NIC's transmit descriptors. Since these features are not yet available on commodity hardware – certainly not on the hardware we have – we used two simple workarounds.
  For the "clock port" we used the CTS pin on one of the replicas' RS-232 ports. This is not an ideal choice, since the pin can only be read with an `inb` instruction, which can take a few microseconds (!) on our Xeon CPUs. We use one CPU core to read the pin in a tight loop; whenever this core sees the pin change from 0 to 1, it increments an epoch counter E and saves the CPU's time-stamp counter (TSC) in a variable $T[E]$. Then, whenever Watchmaker needs to know the current time, we use $\pi \cdot E + (\texttt{tsc} - T[E]) \cdot c$, where π is the pulse width of the external clock signal on the CTS pin in nanoseconds and c is a correction factor that both converts from clock cycles to nanoseconds and accounts for the local clock drift. We set c to $\frac{\pi \cdot n}{T[E]-T[E-n]}$; this is possible because we have a high-quality external clock signal, and it also accounts for the fact that c can vary a bit over time, e.g., in response to temperature changes. For our experiments, we use $n = 1$; our CPUs do support Intel's invariant TSC, which ensures that the TSC advances at a constant rate. Notice that our drift bound Φ must include the multi-μs execution time of the `inb` instruction, which is unfortunate; however, the queues in our switch can easily handle overlapping transmissions for that amount of time, so this does not create a risk of queue drops, it merely increases latency.
- **Timed send:** For timed transmissions, we use a second core that constantly watches the transmission buffers, reads the transmission-time field as soon as the buffer is ready, and then keeps reading the TSC until the specified transmission time arrives; then it issues the transmission command to the NIC. With this method, there is a small delay τ between this point and the time the NIC actually starts transmitting, but since this delay is the same for all of our machines, it merely shifts the network schedule backward by τ.
  These workarounds create an overhead of two CPU cores for Watchmaker, but this overhead could easily be removed by adding these two simple features to the hardware. In fact, our NICs support IEEE 1588 time synchronization and therefore already *do* have clocks with nanosecond-level precision, but they use these clocks to record the

transmission time after the fact, which is not sufficient for our purposes. We speculate that this could be changed with a firmware update.

- **KVS:** Our implementation of Watchmaker runs on a single core. It has a hard-coded transmission schedule: each replica is given a fixed-size transmission window in which to broadcast on the replica network, and the windows are assigned round-robin. We also implemented the two standard workloads for evaluating SMR protocols: a NOP state machine, to evaluate the raw performance of the SMR protocol itself, and a simple key-value store (KVS), to measure performance with a somewhat more realistic workload. To enable comparisons with prior work, we did not make the KVS persistent, so the keys and values are stored in memory on each replica.

5 Evaluation

The main questions we wanted to answer experimentally were: 1) How does Watchmaker compare to state-of-the-art algorithms in terms of latency and throughput?, 2) How well does its throughput scale with the number of replicas?, 3) How quickly can it recover from faults?, 4) How well can it adjust to changing and unbalanced workloads?, 5) How much of its benefits are due to having a custom kernel?, and 6) How good is its end-to-end performance with the key-value store as an application?

5.1 Experimental setup

- **Hardware:** We ran our experiments on a cluster of Dell PowerEdge R440 servers. Each server had two 2.1 GHz Xeon Silver 4216 CPUs, 64 GB of RAM, and two NICs: one with two 1 Gbps copper ports (a Broadcom BCM5720), and one with four 10 Gbps optical ports (an Intel X710). We used some servers as replicas and some as clients. Both clients and replicas were connected to the access network (Section 2) via one of their optical ports; the replicas were additionally connected to the replica network using a second optical port. We used two Juniper EX4550-32F switches, one for each network, plus a third for the Linux experiments (Section 5.7). The copper ports were used only during startup and to download results, but not during the experiments themselves.
- **Baselines:** We chose NOPaxos [68], Nezha [38], Fast Paxos [62] and Multi-Paxos [60] as baselines. NOPaxos and Nezha are both state-of-the-art protocols; they use approaches that are related to ours – NOPaxos leverages the network for ordering, while Nezha leverages synchronized clocks – and they are the highest-performing protocols we are aware of. The Paxos variants are classics, and other SMR papers often include them for reference. There are literally dozens of other SMR protocols we could have chosen, but in many cases the comparison would not have been very fair and/or very interesting: for instance, TOQ-EPaxos [99] and Domino [108] do use synchronized clocks, but they are designed for WANs, so their throughput is naturally lower.
- **Software:** For the Watchmaker experiments, the servers ran our custom kernel (Section 4), but the clients simply ran Linux 6.0.12. For the other experiments, the servers ran Linux as well. We obtained the codebases for NOPaxos, Fast Paxos, and Multi-Paxos from the NOPaxos github repository, which is mentioned in its source paper. We did the same for Nezha. Since we did not have a P4 switch, we ran the NOPaxos sequencer in software on a separate machine instead. We used the clients included with each source repository to generate requests on the clients; we made a small modification to the NOPaxos client to send requests directly to the software sequencer instead of broad-

casting them. Since we did not have access to Huygens, we used Nezha with aggressively configured NTP; the reported clock sync accuracy was on the order of a few hundred ns.

- **Workloads:** For most of our experiments, we used a simple "Echo-RPC" state machine that receives 32-byte requests and echoes them back to the client. For the KVS workload, we use requests with a one-byte opcode (GET or PUT), a 4-byte key, and, for PUTs an 8-byte value. The responses consist of a one-byte status code plus, for GET responses, an 8-byte value. Additionally, Multi-Paxos takes a packet batching parameter for its replicas, which we set to 3.
- **Synchronization:** In the Watchmaker experiments, we synchronized the clocks by connecting the CTS pin on each server's RS-232 port to an external signal generator, a Siglent SDG1032X. (We did not experiment with redundant signal generators.) This device cost about $350 at the time of writing; it has a special circuit for square waves that supports up to 60 MHz and has a jitter of less than 300ps plus 0.05ppm of period. We configured it to generate square waves at 66 kHz, which we used for clock synchronization as described in Section 4. Based on the results from Figure 2 in Section 2.2, we set $\Delta = 30\,\mu$s and $\Phi = \pm 3\,\mu$s. We configured Watchmaker with a simple round-robin transmission schedule, with a $15\,\mu$s transmission window for each replica; this corresponds to 725K MTU-sized packets per second, or about 8.7 Gbps.
- **Time noise mitigations:** Our Linux experiment ran on Fedora 35 servers (Linux 6.0.12-100.fc35.x86_64) with a 2.1 GHz CPU (SMT off). Core 0 was reserved for OS bookkeeping. All DPDK and SMR threads were isolated to cores 14 via `isolcpus=1-4` and pinned using `pthread_setaffinity_np`. At boot, we applied the following kernel flags to eliminate jitter sources and background activity:

  ```
  intel_iommu=on isolcpus=nohz,domain,managed_irq,1-4 nohz_full=1-4
  rcu_nocbs=1-4 rcu_nocb_poll kthread_cpus=0 irqaffinity=0 nosmt nowatchdog
  tsc=nowatchdog preempt=none processor.max_cstate=0 intel_idle.max_cstate=0
  idle=poll mitigations=off iTCO_wdt.disable=1 mce=off split_lock_detect=off
  ```

  `mce=off` suppresses Machine Check Exception interrupts that could asynchronously preempt timingcritical threads. In the BIOS, we disabled Turbo Boost, monitor/MWAIT, and memory patrol scrubbing. In user space, we used DPDK 20.11.0 with hugepages. On startup, the main thread:

  1. called `iopl(3)` to gain raw I/O privileges.
  2. invoked `mlockall(MCL_CURRENT|MCL_FUTURE)` and performed stack prefaulting to avoid page faults.

  Meanwhile, the other functional threads each:

  1. executed `asm volatile("cli")` to disable interrupts.
  2. invoked `mlockall(MCL_CURRENT|MCL_FUTURE)` to lock their working set in memory.

  Additionally, we removed real-time scheduling quotas via:

  ```
  echo -1 | sudo tee /proc/sys/kernel/sched_rt_runtime_us
  ```

  and wrapped the SMR code in a systemd service configured with `SCHED_FIFO@98`, `LimitRTPRIO=99`, `LimitMEMLOCK=infinity`, and `CPUAffinity=1-4`. This ensures uninterrupted, high-priority execution on the isolated cores.

5.2 Latency vs. throughput

A standard experiment in the SMR literature is to measure throughput with a NOP state machine, so that performance is really only limited by the SMR protocol and not by the state machine itself. We followed the approach in the Nezha paper and ran experiments both

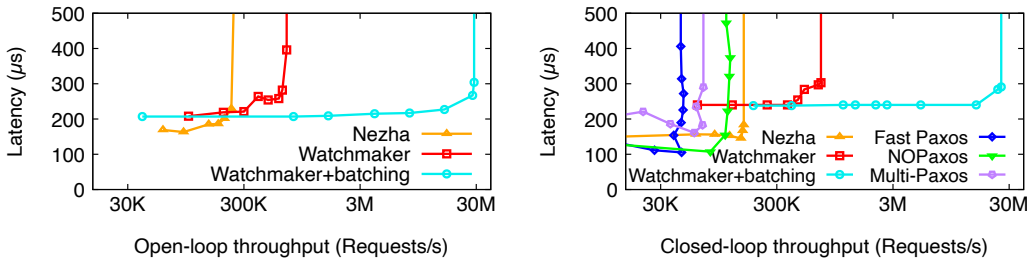

Figure 7 Throughput for Watchmaker and our baselines, with open-loop (left) and closed-loop (right) workloads.

with closed-loop and open-loop workloads. In the closed-loop workload, clients send a new request whenever they receive a response to the previous one; in the open-loop workload, clients instead use a Poisson process to generate requests, without waiting for responses. We varied the request rate by changing the number of clients and/or their parameters; we measured the median latency and the average throughput. We used $N = 3$ replicas for all systems and ran experiments with and without batching.

Figure 7 shows our results. We make two observations. First, Watchmaker's open-loop throughput without batching is 700K requests/s, which is about three times as high as Nezha's. At this rate, the bottleneck is actually not the SMR protocol but the bandwidth of our 10 Gbps links.

Second, batching can boost Watchmaker's throughput to 28M requests/s – two orders of magnitude above Nezha's throughput – *without a significant latency increase*. This is because Watchmaker's replica-network transmission schedule is fixed and consists of MTU-sized packets. If the requests are small (in our case, 32 bytes), these packets will be sent mostly empty; filling up the unused bits with extra requests does not change much. We report these results separately so that the reader can decide on her own which comparison makes more sense. However, we note that, in the configuration we used, Nezha does use some batching as well [39, §6.2].

Watchmaker can achieve this high throughput because of our stronger, quasi-synchronous assumptions, which capture more of the power of the underlying hardware. Having closely synchronized clocks and fine-grained control over the replica network enables Watchmaker to carefully choreograph packet transmissions and thus to "run like clockwork"; this, in combination with ruling out certain very pessimistic packet-loss patterns, enables it to achieve hard bounds on propagation delays, which is fundamentally impossible in the asynchronous model. Thus, Watchmaker has a key advantage over algorithms that merely optimize for such conditions but are not willing to assume them.

5.3 Tail latencies

By reporting only the median latency, Figure 7 obscures an important detail of the latency distribution that distinguishes quasi-synchronous protocols from partially synchronous ones. To show this, we sent open-loop requests to Nezha and Watchmaker, at a constant rate that was 95% of each protocol's peak throughput, and we captured the per-request latencies in steady state, without the first 100 ms and the last 100 ms of the trace, to avoid startup and shutdown artifacts.

Figure 8 shows our results as a CDF. Since the replicas are not fully saturated at this rate, Watchmaker's latencies are quite stable: the median was 262 μs and the 99th percentile

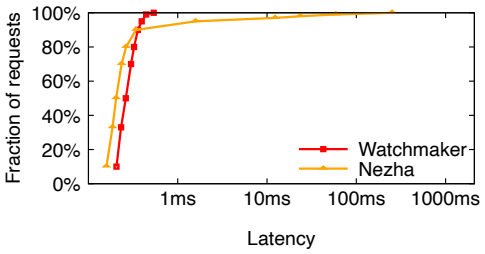

Figure 8 CDF of the end-to-end latency for Nezha and Watchmaker.

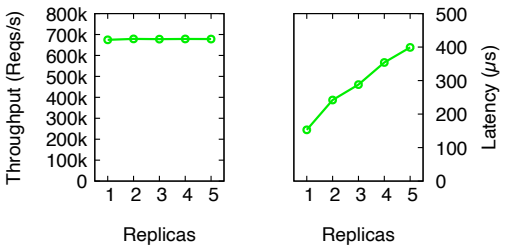

Figure 9 Scalability of Watchmaker: average throughput (left) and median latency (right).

was only 443 μs. In contrast, Nezha's 99th-percentile latency was over 59 ms. This is because Watchmaker closely coordinates the replicas and executes "like clockwork", with very little protocol-induced variation. In contrast, Nezha, like any other asynchronous protocol, experiences variable latencies and thus generates a heavy latency tail.

5.4 Scalability

Next, we ask how Watchmaker's throughput and fault-tolerance properties change with the number of replicas. To find the answer to the first question, we ran experiments with two open-loop clients that together generated 680K requests/s (roughly 95% of the maximum without batching). We varied the number of replicas from $N = 1$ (no replication) to $N = 5$, and we measured both latency and throughput.

Figure 9 shows our results: average throughput (left) and median latency (right). The throughput remains constant as we increase the number of replicas. This is expected for Watchmaker because there is almost no coordination between the replicas, other than the constant-size field for the high-water mark. However, it is interestingly different from many existing protocols, where throughput often degrades with the number of replicas (e.g., in Paxos and Fast Paxos).

Latencies are lowest for the unreplicated case ($N = 1$) and increase linearly when Watchmaker is enabled. This is because a Watchmaker replica executes a request only once it has positive confirmation that each of the other live replicas has received the request, and this takes $O(N)$ transmission slots. We speculate that this effect could be avoided by using NAKs in the rare case that a packet is lost, instead of ACKs in the common case that it is received.

5.5 Recovery from faults

Next, we examine how Watchmaker recovers from faults. Given our quasi-synchronous assumptions, there are two different scenarios that seem interesting: what happens when a replica crashes, and what happens when a packet is lost. We examine each in turn.

Figure 10 (left) shows throughput over time; at $t = 0$, we intentionally crashed one of the replicas. As expected, this causes Watchmaker to briefly get "stuck", since it initially assumes that some of the now-missing packets were lost and need to be retransmitted. (Throughput does not immediately fall to zero because the remaining replicas still have some responses queued up.) However, after $k+1$ transmission "rounds", the crashed replica is declared dead and its transmission slots are adopted by the other replicas; as a result, throughput quickly returns to normal. The downtime is about 540 μs. For comparison, NOPaxos [68] reported a downtime of about 300 ms and Nezha [39] reported a downtime of

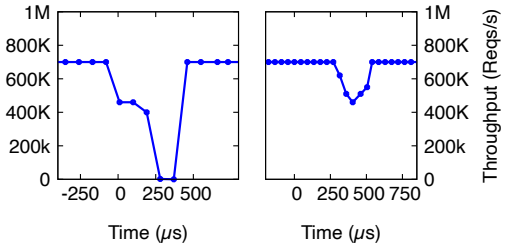

Figure 10 Recovery from faults: response to a node crash (left) and to a physical-layer packet loss (right).

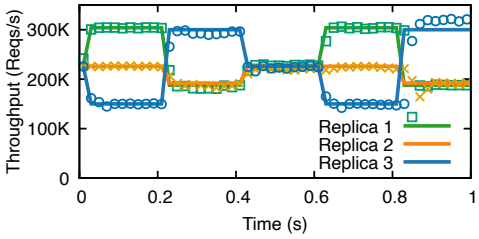

Figure 11 Response to an imbalanced workload: request rate received by each of the three replicas (lines) and response rate (points).

about 4 s. Recall from Section 3 that quick and reliable fault detection is one of the benefits of the quasi-synchronous model.

Figure 10 (right) shows our results from a similar experiment, except that at $t = 0$, we intentionally dropped a single packet. Just like the crash, this causes Watchmaker to get "stuck", but only until another replica retransmits the missing packet. This happens quickly, after a brief glitch of around 270 μs; after that, throughput returns to normal. Recall from Section 2.1 that, at least in the environment for which we have designed Watchmaker, the probability of a physical-layer loss is very low, perhaps on the order of 10^{-8}. If a Watchmaker instance processes $7 \cdot 10^5$ requests per second, we would expect such a glitch to occur roughly once every 71 seconds.

5.6 Dynamic workload

In the experiments we have shown so far, the replicas were receiving requests at the same rate. To test the dynamic scheduling mechanism from Section 3.3, we performed an experiment in which the replicas were receiving requests at different rates, which randomly changed every 200 ms but always added up to the full capacity of the system. The schedules were adjusted every 1 ms. Once every 10 ms, we measured the total rate at which the clients sent requests to each replica, as well as the rate at which the clients received the responses.

Figure 11 shows results from an example run. As expected, the response rates closely track the request rates. Sometimes there is a small delay when a change happens right in the middle of a 10ms window, but overall, the system continued to fully utilize the available bandwidth while quickly adjusting to the imbalance in the workload.

5.7 Custom kernel vs. Linux

A natural question is how much of Watchmaker's high performance is due to the careful coordination that the quasi-synchronous model enables, and how much is due to our custom kernel. To answer this question, we re-ran the open-loop experiment from Section 5.2 with our Linux implementation of Watchmaker. Under these conditions – on unmodified Linux and stock DPDK, in combination with our time-noise mitigations mentioned in Section 5.1 – Watchmaker sustained an average throughput of 545K requests/s with a median latency of 4.5 ms.

The key difference in this setting is that the synchronization is less precise than with the custom kernel and the signal generator. Because of this, we cannot transmit quite as aggressively. Recall from Section 3.4 that queue overflows occur when transmissions from different replicas overlap for too long; we avoid this by using larger transmission slots with

some extra spacing in between. The spacing leads to a somewhat lower throughput and the larger slots to a higher latency. However, in absolute terms, the throughput was still more than twice as high as Nezha's, and the 99th percentile latency was almost an order of magnitude lower.

These results suggest that 1) the benefits are due to Watchmaker and not to our kernel, and that 2) with some additional due diligence to rule out other sources of time noise, a deployment on Linux could be an option. We speculate that performance on Linux could be improved, e.g., by adding a custom driver to access the shared clock signal.

5.8 Application: KVS

Finally, we ask how well Watchmaker performs with a more realistic application as a state machine, instead of the NOP state machine we have used for the experiments so far. This question is more subtle than it may seem at first: the slower the application, the less important the performance of the SMR algorithm will be. For instance, if the SMR protocol can handle 200k requests/s, its overhead will be negligible for an application that can handle only 20k requests/s, but it will massively limit an application that can handle 1M requests/s. Because of this, we decided against using Redis, as in some of the earlier work [38, 68]: with Redis, even slower SMR protocols can perform close to the unreplicated case. Instead, we wrote our own simple in-memory KVS, as described in Section 4. We ran open-loop workloads with Watchmaker and $N = 3$ replicas, for reads and writes in different proportions. We used 4-byte keys and 8-byte values, so the packet size is five bytes for read requests and write responses, which are both small, and 13 bytes for write requests and read responses, which both contain a value.

We found that the throughput for our KVS is essentially the same as with the NOP workload, so we did not include a separate graph. This is expected: Watchmaker is so fast that the bottleneck shifts from the SMR protocol to the link capacity, even for high-throughput applications like our KVS.

6 Related Work

To the best of our knowledge, Watchmaker is the first *practical* SMR protocol that relies on fine-grained synchrony for correctness: earlier solutions either 1) rely on a weaker form of synchrony, 2) use synchrony only as an optimization while remaining asynchronous, or 3) are theoretical and do not have a practical implementation.

- **Synchrony in theory:** The theoretical underpinnings of the synchronous model are well-established, including lower bounds, such as the $N > 2f$ bound for the asynchronous and partially synchronous models [33], the limitations of synchronized clocks by themselves [91], the famous FLP impossibility [37], and the failure detector abstraction [20] as a way to capture synchrony assumptions.
- **State-machine replication:** Since the original Paxos paper [59], a rich and growing literature on consensus protocols and SMR has emerged [3, 8, 13, 14, 16, 21, 30, 31, 34, 35, 56, 60–62, 75, 77, 79–83, 86, 90, 96, 98, 100, 106, 108, 112]. Most of these protocols are for the partially synchronous or the asynchronous model and must thus pay a price (in terms of overhead and the number of replicas) for the *ability* to tolerate high delays and losses, even when these do not actually occur. Protocols that use synchronized clocks or special network features are discussed separately below.

- **Synchrony in practice:** Some SMR implementations have used the synchronous model, e.g,, most recently IGOR [74]. But IGOR simply assumes that the network is reliable and that packets are never lost. This may be fine for embedded applications such as IGOR's, which presumably send very few messages to begin with, but not for data-center environments, where sporadic losses *will* happen once in a while. There has been some debate about the role synchrony should play in this setting. On the one hand, Aguilera and Walfish [7] warns against synchronous model and considers it "dangerous"; they instead suggest the asynchronous model with a perfect failure detector, as later realized in [64]. On the other hand, some of us have pointed out in a prior workshop paper [109] that data-center hardware is more reliable than it is typically given credit for. A possible compromise is Verissimo et al. [104]'s "timely computing base", which confines synchrony to some key parts of the system; this partly inspired our distinction between replica and access networks.
- **Synchronized clocks:** There are many ways to synchronize clocks using network messages, and recent solutions, such as Sundial [70] and DPTP [50], can get uncertainty bounds on the order of 100 ns, and Firefly [78] achieves sub-10 ns device-to-device sync accuracy while being scalable and failure-resilient. (Synchronous Ethernet [48] is aiming for nanosecond-level accuracy [72, 93], but it is not yet widely available.) However, most solutions rely on heuristics and estimates, so this is not a hard guarantee. Perhaps because of this, many systems have followed Liskov's advice [73] and used synchronized clocks only for performance but not for correctness. Systems like Spanner [27], CLOCC [5], and TAPIR [111] have used loosely synchronized clocks to establish a total order; CRaft [105] and CockRoachDB [97] have use them to speed up Raft, Domino [108] to accelerate Fast Paxos, and TOQ [99] to improve EPaxos. Nezha [38] also uses them for its deadline-ordered multicast primitive. The key difference to quasi-synchrony and Watchmaker is that none of these protocols takes the final step and relies on the synchrony for correctness. Thus, despite their excellent fast-path performance, they all technically remain asynchronous and thus are subject to the known limitations of that model.
- **Special hardware features:** A number of recent systems have focused specifically on data centers. One line of work relies on special hardware: NOPaxos [68], Eris [67], 1pipe [66], and Hydra [23] use programmable switches for ordering, and Waverunner [10] uses FPGA-based SmartNICs to accelerate data replication. This is somewhat orthogonal to our approach: the hardware definitely helps with speed, but it does not fundamentally change the equation because it cannot do anything that a normal node could not also do. A second line of work uses topology information: for instance, Speculative Paxos [88] leverages it to ensure that most packets arrive in order; Watchmaker uses the same information (A9), but it relies on it for correctness.
- **Bounded network delays:** Watchmaker critically relies on the fact that, in a data-center network, queueing delays are not inevitable but can be controlled with coordination. One line of work has used this fact to rein in tail latencies: for instance, SILO [49], Orchestra [24], Pulsar [12], Clockwork [45], Baraat [32], and Chronos [51] fall into this category, and they all achieve latencies that are remarkably stable. However, these systems are fundamentally best-effort and cannot *guarantee* a hard bound on message delivery times. Another line of work does provide such a guarantee: for instance, time-triggered Ethernet [57] does so with special hardware, while QJump [42] uses QoS features in commodity hardware, albeit only for a small fraction of the traffic. A third line of work can eliminate queueing in the network almost entirely, e.g., with a central arbitrator, as in

Fastpass [87], or with RSVP-like reservations, as in Harmony [6]. Vattikonda et al. [102] proposes a form of TDMA for data-center networks, which resembles the transmission schedules in Watchmaker. But on their own, these solutions are only one of the pieces that are necessary for synchrony.

- **Byzantine consensus:** There is a rich literature on tolerating Byzantine [63] faults. Early solutions, such as Rampart [89] and SecureRing [54], did rely on synchrony for correctness, but PBFT [17], and much of the work that followed it [1,9,11,18,25,26,28,43, 47,58,65,69,76,94,101,107,110], relied on it only for liveness. Recently, fully-synchronous solutions have seen renewed interest in the context of cryptocurrencies [2,52,53], but these protocols typically operate in WANs, where synchrony is threatened, e.g., by network outages and denial-of-service attacks. As a result, their synchrony assumptions are at very different timescales: for instance, Algorand [41] assumes a bound on the length of asynchronous periods that is on the order of days or weeks. Unlike this like of work, Watchmaker focuses on tolerating crash faults and on data centers, where fine-grained synchrony is easier to achieve.
- **Weak synchrony:** Several weaker versions of the fully-synchronous model have been proposed, including partial synchrony [33]. Another example is the mobile sluggish model [46], which allows the delay bound to be violated by a certain number of sluggish replicas; PiLi [19] and Sync HotStuff [4] rely on this model. The sleepy model [84] instead allows replicas to be temporarily offline; Thunderella [85] is an example of a solution for this model. Other models have assumed probabilities on failures [15, 44, 71, 103] or have treated timing failures as exceptions that must be handled [36], or have assumed WCETs and bounded clock drift while allowing for packet loss [29]. Among these, our proposal – quasi-synchrony – is the most restrictive, in that it only makes sense for tightly-controlled environments, such as single-tenant data centers. But in return, it comes closest to the fully synchronous model and thus to the benefits this model provides.

7 Conclusion

At first glance, this paper presents a particular SMR protocol (Watchmaker), with a particular set of unusual assumptions that the reader may or may not be comfortable with. But to us, Watchmaker is also meant to illustrate three larger points: 1) the asynchronous model is not the *only* model that can be used to build practical distributed systems; 2) the asynchronous model has both a price, in terms of overhead and complexity, and specific benefits; and 3) there exist at least some deployment scenarios where it is not obvious that the benefits justify the costs. As designers of distributed systems, we really do have a choice when it comes to the fundamental assumptions, and, even though most textbooks focus almost exclusively on the asynchronous model, that should not be the only option we consider.

For Watchmaker, we chose a particular assumption – that at most k consecutive packets can be dropped on an otherwise working link. To us, this seemed like a reasonable starting point, but it is certainly not the only option. We speculate that there is an entire space of possible models, each with interesting consequences for the classical distributed-systems challenges (certainly not just for SMR!). We think that exploring this space further could be an interesting opportunity for the community.

References

1 Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. SOSP*, 2005. doi:10.1145/1095810.1095817.

2 Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Efficient synchronous Byzantine consensus. Cryptology ePrint Archive, Paper 2017/307, 2017. URL: https://eprint.iacr.org/2017/307.

3 Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous Byzantine agreement with expected $O(1)$ rounds, expected $O(n^2)$ communication, and optimal resilience. In *Proc. International Conference on Financial Cryptography and Data Security*, pages 320–334. Springer, 2019. doi:10.1007/978-3-030-32101-7_20.

4 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and practical synchronous state machine replication. Cryptology ePrint Archive, Paper 2019/270, 2019. URL: https://eprint.iacr.org/2019/270.

5 Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1995. doi:10.1145/568271.223787.

6 Saksham Agarwal, Qizhe Cai, Rachit Agarwal, David Shmoys, and Amin Vahdat. Harmony: A congestion-free datacenter architecture. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024. URL: https://www.usenix.org/conference/nsdi24/presentation/agarwal-saksham.

7 Marcos Aguilera and Michael Walfish. No time for asynchrony. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS)*, 2009. URL: https://www.usenix.org/legacy/events/hotos09/tech/full_papers/aguilera/aguilera.pdf.

8 Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/aguilera.

9 Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, 2005. doi:10.1145/1095810.1095816.

10 Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho, Michael Ferdman, Peter Milder, and Shuai Mu. Waverunner: An elegant approach to hardware acceleration of state machine replication. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023. URL: https://www.usenix.org/conference/nsdi23/presentation/alimadadi.

11 Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011. doi:10.1109/TDSC.2010.70.

12 Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/angel.

13 Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017. URL: https://ieeexplore.ieee.org/abstract/document/8023110/.

14 Tigran Bantikyan, Jonathan Zarnstorff, Te-Yen Chou, Lewis Tseng, and Roberto Palmieri. Pineapple: Unifying Multi-Paxos and atomic shared registers. In *Proceedings of the 22nd*

USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2025. URL: https://www.usenix.org/conference/nsdi25/presentation/bantikyan.

15 Ian Broster, Alan Burns, and Guillermo Rodriguez-Navas. Timing analysis of real-time communication under electromagnetic interference. *Real-Time Systems*, 30(1):55–81, 2005. doi:10.1007/s11241-005-0504-z.

16 Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/burke.

17 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398461, November 2002. doi:10.1145/571637.571640.

18 Benjamin Y. Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020. doi:10.1145/3419614.3423256.

19 T.-H. Hubert Chan, Rafael Pass, and Elaine Shi. PiLi: An extremely simple synchronous blockchain. *Cryptology ePrint Archive*, page 980, 2018. URL: https://eprint.iacr.org/2018/980.

20 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, March 1996. doi:10.1145/226643.226647.

21 Aleksey Charapko, Ailidani Alijiang, and Murat Demirbas. Linearizable quorum reads in Paxos. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2019. URL: https://www.usenix.org/conference/hotstorage19/presentation/charapko.

22 Ang Chen, W. Brad Moore, Hanjun Xiao, Andreas Haeberlen, Linh Thi Xuan Phan, Micah Sherr, and Wenchao Zhou. Detecting covert timing channels with time-deterministic replay. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, October 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chen_ang.

23 Inho Choi, Ellis Michael, Yunfan Li, Dan R. K. Ports, and Jialin Li. Hydra: Serialization-free network ordering for strongly consistent distributed applications. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023. URL: https://www.usenix.org/conference/nsdi23/presentation/choi.

24 Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with Orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference*, 2011. doi:10.1145/2043164.2018448.

25 Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, 2007. doi:10.1145/1294261.1294280.

26 Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, Mirco Marchetti, et al. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation (NSDI)*. The USENIX Association, 2009. URL: https://www.usenix.org/legacy/events/nsdi09/tech/full_papers/clement/clement.pdf.

27 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Googles globally

distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3), August 2013. doi:10.1145/2491245.

28 James A. Cowling, Daniel S. Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006. URL: http://www.usenix.org/events/osdi06/tech/cowling.html.

29 Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed systems*, 10(6):642–657, 2002. URL: https://ieeexplore.ieee.org/abstract/document/774912/.

30 Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review*, 46(2):1824, May 2016. doi:10.1145/2935634.2935638.

31 Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at network speed. In *Proceedings of the 1st ACM Symposium on SDN Research (SOSR)*, 2015. doi:10.1145/2774993.2774999.

32 Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. In *Proceedings of the ACM SIGCOMM 2014 Conference*, 2014. doi:10.1145/2740070.2626322.

33 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2), April 1988. doi:10.1145/42282.42283.

34 Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient replication via timestamp stability. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, 2021. doi:10.1145/3447786.3456236.

35 Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, 2020. doi:10.1145/3342195.3387543.

36 Christof Fetzer and Flaviu Cristian. Fail-awareness: an approach to construct fail-safe applications. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 282–291, 1997. doi:10.1109/FTCS.1997.614101.

37 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

38 Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. Nezha: Deployable and high-performance consensus using synchronized clocks. *Proceedings of the VLDB Endowment*, 16(4):629642, December 2022. doi:10.14778/3574245.3574250.

39 Jinkun Geng, Anirudh Sivaraman, Balaji Prabhakar, and Mendel Rosenblum. Nezha: Deployable and high-performance consensus using synchronized clocks. *arXiv preprint arXiv:2206.03285*, 2022. URL: https://arxiv.org/abs/2206.03285.

40 Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosunblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018. URL: https://www.usenix.org/system/files/conference/nsdi18/nsdi18-geng.pdf.

41 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017. doi:10.1145/3132747.3132757.

42 Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't matter when you can JUMP them! In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/grosvenor.

43 Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, 2010. doi:10.1145/1755913.1755950.

44 Arpan Gujarati and Björn B Brandenburg. When is CAN the weakest link? A bound on failures-in-time in CAN-based real-time systems. In *2015 IEEE Real-Time Systems Symposium*. IEEE, 2015. URL: https://ieeexplore.ieee.org/abstract/document/7383582/.

45 Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/gujarati.

46 Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. In *Advances in Cryptology CRYPTO 2019*, 2019. doi:10.1007/978-3-030-26948-7_18.

47 Chi Ho, Robbert Van Renesse, Mark Bickford, and Danny Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008. URL: https://www.usenix.org/legacy/events/nsdi08/tech/full_papers/ho/ho.pdf.

48 International Telecommunication Union. ITU-T G.8262/Y.1362 standard, 2018. URL: https://www.itu.int/rec/T-REC-G.8262.

49 Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *Proceedings of the ACM SIGCOMM 2015 Conference*, 2015. doi:10.1145/2785956.2787479.

50 Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. Precise time-synchronization in the data-plane using programmable switching ASICs. In *Proceedings of the 5th ACM Symposium on SDN Research (SOSR)*, 2019. doi:10.1145/3314148.3314353.

51 Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, 2012. doi:10.1145/2391229.2391238.

52 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for Byzantine agreement. *Journal of Computer and System Sciences*, 75(2):91–112, 2009. doi:10.1016/j.jcss.2008.08.001.

53 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. Cryptology ePrint Archive, Paper 2016/889, 2016. URL: https://eprint.iacr.org/2016/889.

54 Kim Potter Kihlstrom, Louise E Moser, and P Michael Melliar-Smith. The secureRing protocols for securing group communication. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 3, pages 317–326. IEEE, 1998. URL: https://ieeexplore.ieee.org/document/656294/.

55 James C. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, 1975. doi:10.1145/390016.808444.

56 Marios Kogias and Edouard Bugnion. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, 2020. doi:10.1145/3342195.3387545.

57 Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The time-triggered ethernet (TTE) design. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2005. URL: https://ieeexplore.ieee.org/abstract/document/1420949/.

58 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007. doi:10.1145/1294261.1294267.

59 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133169, 1998. doi:10.1145/3335772.3335939.

60 Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

61 Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2004. URL: https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/.

62 Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006. doi:10.1007/s00446-006-0005-x.

63 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382401, July 1982. doi:10.1145/3335772.3335936.

64 Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the Falcon Spy Network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011. doi:10.1145/2043556.2043583.

65 Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009. URL: https://www.usenix.org/legacy/events/nsdi09/tech/full_papers/levin/levin.pdf.

66 Bojie Li, Gefei Zuo, Wei Bai, and Lintao Zhang. 1Pipe: scalable total order communication in data center networks. In *Proceedings of the ACM SIGCOMM 2021 Conference*, 2021. doi:10.1145/3452296.3472909.

67 Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017. doi:10.1145/3132747.3132751.

68 Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li.

69 Jinyuan Li and David Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007. URL: https://www.usenix.org/legacy/events/nsdi07/tech/full_papers/li/li.pdf.

70 Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/li-yuliang.

71 George MA Lima and Alan Burns. A consensus protocol for CAN-based systems. In *Proc. 24th IEEE Real-Time Systems Symposium (RTSS)*, 2003. URL: https://ieeexplore.ieee.org/abstract/document/1253289/.

72 Maciej Marek Lipiski. *Methods to Increase Reliability and Ensure Determinism in a White Rabbit Network*. PhD thesis, Warsaw University of Technology, 2016. URL: https://repository.cern/records/tkh30-4ja15.

73 Barbara Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC)*, 1991. doi:10.1145/112600.112601.

74 Andrew Loveless, Ronald Dreslinski, Baris Kasikci, and Linh Thi Xuan Phan. IGOR: Accelerating Byzantine fault tolerance for real-time systems with eager execution. In *Proc. 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021. doi:10.1109/RTAS52030.2021.00036.

75 Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Symposium on Operating*

Systems Design and Implementation (OSDI), 2008. URL: https://www.usenix.org/event/osdi08/tech/full_papers/mao/mao_html.

76 Zarko Milosevic, Martin Biely, and André Schiper. Bounded delay in Byzantine-tolerant state machine replication. In *Proc. 32nd International Symposium on Reliable Distributed Systems (SRDS)*, 2013. doi:10.1109/SRDS.2013.15.

77 Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013. doi:10.1145/2517349.2517350.

78 Pooria Namyar, Yuliang Li, Weitao Wang, Nandita Dukkipati, Kk Yap, Junzhi Gong, Chen Chen, Peixuan Gao, Devdeep Ray, Gautam Kumar, Yidan Ma, Ramesh Govindan, and Amin Vahdat. Firefly: Scalable, ultra-accurate clock synchronization for datacenters. In *Proceedings of the ACM SIGCOMM Conference*, 2025. doi:10.1145/3718958.3750502.

79 Harald Ng, Seif Haridi, and Paris Carbone. Omni-Paxos: Breaking the barriers of partial connectivity. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, 2023. doi:10.1145/3552326.3587441.

80 Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. Tolerating slowdowns in replicated state machines using copilots. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/ngo.

81 Rui Oliveira, Rachid Guerraoui, and André Schiper. Consensus in the crash-recover model. Technical Report 97/239, EPFL, Département dInformatique, August 1997.

82 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

83 Seo Jin Park and John Ousterhout. Exploiting commutativity for practical fast replication. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019. URL: https://www.usenix.org/conference/nsdi19/presentation/park.

84 Rafael Pass and Elaine Shi. The sleepy model of consensus. In *Proc. Advances in Cryptology (ASIACRYPT)*, 2017. doi:10.1007/978-3-319-70697-9_14.

85 Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology – EUROCRYPT 2018*, 2018. doi:10.1007/978-3-319-78375-8_1.

86 Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016. URL: https://ieeexplore.ieee.org/abstract/document/7579738.

87 Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the ACM SIGCOMM Conference*, 2014. doi:10.1145/2619239.2626309.

88 Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ports.

89 Michael K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, 1995. doi:10.1007/3-540-60042-6_7.

90 Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. SwiftPaxos: Fast geo-replicated state machines. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024. URL: https://www.usenix.org/conference/nsdi24/presentation/ryabinin.

91 Nicola Santoro and Peter Widmayer. Time is not a healer. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, 1989. doi:10.1007/BFb0028994.

92 Fred B. Schneider. The state machine approach: A tutorial. In *Fault-Tolerant Distributed Computing*, 1990. doi:10.1007/bfb0042323.

93 J. Serrano, P. Alvarez, M. Cattin, E. Garcia Cota, J. Lewis, P. Moreira, T. Wlostowski, G. Gaderer, P. Loschmidt, J. Dedic, R. Bär, T. Fleck, M. Kreider, C. Prados, and S. Rauch. The White Rabbit project. In *12th International Conference On Accelerator And Large Experimental Physics Control Systems*, 2009. URL: https://cds.cern.ch/record/1215571/files/CERN-ATS-2009-096.pdf.

94 Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros Maniatis. Zeno: Eventually consistent Byzantine-fault tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009. URL: http://www.usenix.org/events/nsdi09/tech/full_papers/singh/singh.pdf.

95 SiTime. SiT5811 MEMS OCXO, 2024. Specs. URL: https://www.sitime.com/products/holdover-ocxos/sit5811.

96 Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*, 2022. doi:10.1145/3492321.3519579.

97 Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020. doi:10.1145/3318464.3386134.

98 Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galinanes, and Bryan Ford. QuePaxa: Escaping the tyranny of timeouts in consensus. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023. doi:10.1145/3600006.3613150.

99 Sarah Tollman, Seo Jin Park, and John Ousterhout. EPaxos revisited. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021. URL: https://www.usenix.org/conference/nsdi21/presentation/tollman.

100 Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3), 2015. doi:10.1145/2673577.

101 Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007. doi:10.1145/1294261.1294268.

102 Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C. Snoeren. Practical TDMA for datacenter Ethernet. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*, 2012. doi:10.1145/2168836.2168859.

103 Paulo Verissimo and Carlos Almeida. Quasi-synchronism: A step away from the traditional fault-tolerant real-time system models. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 7(4):35–39, 1995. URL: https://www.dsc.ufcg.edu.br/~fubica/papers/FD/VA.pdf.

104 Paulo Veríssimo and António Casimiro. The timely computing base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, 2002. URL: https://ieeexplore.ieee.org/abstract/document/1024739.

105 Feiran Wang. *Building High-Performance Distributed Systems with Synchronized Clocks*. PhD thesis, Stanford University, 2019. URL: https://search.proquest.com/openview/392dfbcd9ca4a64c717ee2ca95c44738/1?pq-origsite=gscholar&cbl=18750&diss=y.

106 Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated

state machines with compartmentalization. *Proceedings of the VLDB Endowment*, 14(11), 2021. doi:10.14778/3476249.3476273.

107 Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. ZZ and the art of practical BFT execution. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, 2011. doi:10.1145/1966445.1966457.

108 Xinan Yan, Linguan Yang, and Bernard Wong. Domino: Using network measurements to reduce state machine replication latency in WANs. In *Proceedings of the 16th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2020. doi:10.1145/3386367.3431291.

109 Tian Yang, Robert Gifford, Andreas Haeberlen, and Linh Thi Xuan Phan. The synchronous data center. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2019. doi:10.1145/3317550.3321442.

110 Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003. doi:10.1145/945445.945470.

111 Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4), December 2018. doi:10.1145/3269981.

112 Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. SDPaxos: Building efficient semi-decentralized geo-replicated state machines. In *Proceedings of the 9th ACM Symposium on Cloud Computing (SoCC)*, 2018. doi:10.1145/3267809.3267837.

A Proof of correctness for Watchmaker

In the following, we make the standard assumption that duplicate requests can be removed and that the state machine will execute them only once. When using line numbers in this proof, we refer to Figure 6.

▶ **Lemma 1.** *In every slot, exactly one packet is sent over the replica network.*

Proof. Since we have assumed that nodes have unique IDs (A12), the condition in line 16 can be true for at most one replica at a time, and, given the way the schedule is constructed, it will also be true for one of the replicas. So the SEND procedure is invoked on exactly one replica in each slot, and this procedure has an unconditional BROADCAST command in line 26. This is the only BROADCAST command in the entire algorithm; the only other transmission, in line 34, sends responses to clients, which goes over the access network and not over the replica network. (Recall from Figure 1 that we assume two separate networks: an access network, which connects clients to replicas, and a replica network, which connects the replicas to each other.) ◀

▶ **Lemma 2.** *If a packet has been sent in slot x (at time T), over a path whose links are all working, and does not experience corruption loss, it arrives no later than $T + \Delta$.*

Proof. Because clocks are synchronized up to some bound Φ (A7), the packets will be sent exactly at the scheduled time, plus or minus Φ. We have assumed that there is no cross traffic (A10) and that the topology (A9), queue lengths (A8), propagation delays (A2), and link capacities (A3) are known, so the effects of Φ are captured by the simulation procedure from Section 3.4. Since this procedure does not output any schedule where queue drops can happen, the packet will make it to the destination unless it experiences a corruption loss, and the maximum delay will be the Δ from the simulation procedure. ◀

▶ **Lemma 3.** *Replicas can tell when packets from other replicas are missing, after at most time Δ.*

Proof. This follows directly from Lemmas 1 and 2: since the transmission schedule is common knowledge and clocks are synchronized, each replica knows when each of the other replicas will send a packet (up to Φ), and, at most Δ after the last possible transmission time, each of the other replicas has either received the packet or can be certain that it is missing. The only possible explanations are that the sender has crashed, the link has failed, or, if at least one of the preceding $k-1$ packets from the same sender has arrived, that the packet was lost due to corruption. ◀

▶ **Lemma 4.** *When a replica i is alive and all links on the paths from i to the other replicas are working, no other replica will send a* DEAD*(i,s) message, for any value of s.*

Proof. Suppose i is alive and another replica $j \neq i$ does send such a message. This message can only be sent in line 21, after being put into the `ctrlWait` set in line 57. But for this to happen, j's `numOmit` map would have had to contain the value $k+1$ for i (line 54), and this could only have happened if $k+1$ consecutive packets from i were missing – the value is incremented in lines 52–53, when j is missing a packet from i (which is possible according to Lemma 3), and it would have been reset in line 43 if a packet from i had arrived. But since all the links on the path $i \to j$ are working and we have assumed that $k+1$ corruption losses to the same destination cannot occur in a row (A4), the only way this could have happened is if i had crashed, which contradicts our assumption. ◀

▶ **Lemma 5.** *Suppose that, in some execution e, a replica i crashes at time T. Let A be the set of replicas that do not crash in e. Then every replica $j \in A$ eventually receives a* DEAD*(i,s_q) message from every other replica $q \in A$, where s_q is the highest sequence number of any packet that q has received from i.*

Proof. The first step is roughly analogous to the argument for Lemma 4: if i crashes, the `numOmit` counter for i on each $j \in A$ will eventually reach $k+1$, at which point $k+1$ DEAD(i,s) messages will be added back-to-back to j's `ctrlWait` queue in line 57. Since SEND is invoked periodically and will send the first message from `ctrlWait` when invoked, and since `ctrlWait` is a FIFO queue, these $k+1$ DEAD(i,s) messages will eventually be broadcast on the replica network back-to-back. And since we have assumed that no $k+1$ broadcasts from the same sender will be lost (A4), at least one of these messages will make it to each $q \in A$. The claim about the s_q follows directly from the way the value is computed in line 55. ◀

▶ **Theorem 6** (Agreement)**.** *If a replica i broadcasts a request r over the replica network, then eventually either 1) all live replicas have received a copy of r, or 2) both i and all other replicas that have received a copy of r will crash.*

Proof. Suppose the opposite is true – that is, i has broadcast r, and some replica q (possibly i itself) has a copy of r and remains alive, but there is some other replica $j \neq i$ that never crashes and never receives r.

Let x be the first sequence number of any packet that j has not received. Notice that, whenever a replica broadcasts a packet to the replica network in line 26, it includes a "high water mark" `myHighWater`. Our first goal is to show that `myHighWater` will eventually reach $x-1$. The number starts at -1 and is increased in the bookkeeping procedure, in line 71. It would not reach $x-1$ if the condition in line 69 were true – but this would mean that some other packet with an earlier sequence number than x is missing, which contradicts our assumption. So `myHighWater` will eventually reach $x-1$ and stay there, because the condition in line 69 would prevent it from being incremented further.

Next, notice that `myHighWater` is included in every single packet j sends – the only transmissions to the replica network are made in SEND (line 26) and, since each replica has at least one slot in the transmission schedule, SEND is invoked periodically. According to assumption A4, every other live replica, including q, will have received a packet with this value after at most $k+1$ transmissions. But when q receives this transmission, it will notice, in line 47, that j's `myHighWater` has not advanced as expected. q will have put r into its `execWait` set in line 44, when it first received it; although packets are eventually deleted again from this set in line 31, this would not have happened yet because packets are deleted only when their sequence number is smaller than the high-water marks from all the live replicas (the `safe` value in line 29). So q still has the packet in its `execWait` set, and it will put a copy into `ctrlWait` in line 48. Because `ctrlWait` is a FIFO queue and its contents have the highest priority for transmission, this copy will be transmitted eventually. It could be lost to corruption, but since j's high-water mark does not advance, q would keep retransmitting copies, and according to assumption A4, either j eventually receives one or q crashes. This contradicts our assumption, so we conclude that the claim does hold at least for the request with sequence number x.

Our remaining proof goal is to show that the high-water mark eventually reaches r's sequence number. This is not obvious a priori because, if x is smaller than r's sequence number and option 2) from the lemma applies to x, the first condition in line 69 (the arrival of a packet with sequence number x) will not be sufficient to increase the high-water mark.

However, the high-water mark is *also* increased when 1) the expected sender (let us call it y) has been removed from j's `alive` set, and 2) the sequence number is higher than y's entry in the `termSeq` map. But in order for y to be removed from `alive` in line 63, j would have had to receive DEAD(y,s) messages from all other live replicas (see line 61), and the `termSeq` entry would have been set to the *highest* sequence number that any live replica has received from y. (The replica that did receive this sequence number could have subsequently crashed as well; this is why `termSeq` is adjusted after every new crash, in line 65.) If x is smaller than this sequence number, some other replica still has a copy of the packet and will retransmit it eventually; if x is larger, the second condition in line 69 will hold. Either way, the high-water mark will continue to be increased and will eventually reach r's sequence number. At that point, the generic argument we have made for an arbitrary x will also hold for r's sequence number, and thus either j eventually receives a copy or all other nodes that have a copy of r will crash. ◀

▶ **Theorem 7** (Liveness). *When a client sends a request and can reach at least one live replica (A6') or a quorum of the replicas (A6), it will receive at least one response eventually.*

Proof. If the client does not receive a response to its request r, it will time out and try the replica (see Figure 3). Since at least one replica is alive and reachable from the client, that replica will eventually receive a copy of the request. Let us call this replica i.

i will append the request to its `sendWait` queue in line 37. However, this does not necessarily mean that it will ever be broadcast over the replica network, because in SEND, messages from `ctrlWait` have higher priority. But notice that only two kinds of messages can be put into `ctrlWait`: DEAD messages and retransmissions. The number of replicas is finite, and the death of each replica can cause only $k+1$ DEAD messages to be added to `ctrlWait` (line 57), so these messages cannot delay r's transmission forever. But retransmissions cannot either, because of Theorem 6, which shows that the live replicas cannot be "stuck" with retransmissions forever. True, i's own `sendWait` queue may never get to r in the unlikely event that there are *lots* of losses, but even then, *some* other replica will be making progress, and the client will eventually try that replica instead. So we conclude that r will eventually be broadcast on the replica network.

But once r has been broadcast, we know from Theorem 6 that eventually all the live replicas will have a copy. They will eventually increase their high-water marks past r's sequence number, at which point r will be executed in line 33; dead replicas would not increase their high-water marks but would be removed from `highWater` in line 64. (In fact, line 33 could be reached for more than one copy of r, if there were retransmissions, but we have assumed that duplicates can be filtered out.) And once that happens, each live replica will send a response to the client in line 34. ◀

▶ **Theorem 8** (Ordering). *If two replicas i and j execute two requests r_1 and r_2, then they will both execute the requests in the same order.*

Proof. Requests are executed in the order of their sequence numbers, so this claim follows by construction. ◀

▶ **Theorem 9** (Validity). *If a client receives more than one response to the same request, the responses will be identical.*

Proof. This follows directly from Theorem 8 and the standard assumption that the state machine is deterministic and starts in the same state on all replicas. ◀