Consensus in a Box: Inexpensive Coordination in Hardware

Zsolt István, David Sidler, and Gustavo Alonso, *ETH Zürich;*
Marko Vukolić, *IBM Research—Zürich*

https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/istvan

This paper is included in the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16).
March 16–18, 2016 • Santa Clara, CA, USA
ISBN 978-1-931971-29-4

Consensus in a Box: Inexpensive Coordination in Hardware

Zsolt István, David Sidler, Gustavo Alonso
Systems Group, Dept. of Computer Science, ETH Zürich

Marko Vukolić[*]
IBM Research - Zürich

Abstract

Consensus mechanisms for ensuring consistency are some of the most expensive operations in managing large amounts of data. Often, there is a trade off that involves reducing the coordination overhead at the price of accepting possible data loss or inconsistencies. As the demand for more efficient data centers increases, it is important to provide better ways of ensuring consistency without affecting performance.

In this paper we show that consensus (atomic broadcast) can be removed from the critical path of performance by moving it to hardware. As a proof of concept, we implement Zookeeper's atomic broadcast at the network level using an FPGA. Our design uses both TCP and an application specific network protocol. The design can be used to push more value into the network, e.g., by extending the functionality of middleboxes or adding inexpensive consensus to in-network processing nodes.

To illustrate how this hardware consensus can be used in practical systems, we have combined it with a main-memory key value store running on specialized microservers (built as well on FPGAs). This results in a distributed service similar to Zookeeper that exhibits high and stable performance. This work can be used as a blueprint for further specialized designs.

1 Introduction

Data centers face increasing demands in data sizes and workload complexity while operating under stricter efficiency requirements. To meet performance, scalability, and elasticity targets, services often run on hundreds to thousands of machines. At this scale, some form of coordination is needed to maintain consistency. However, coordination requires significant communication between instances, taking processing power away from the main task. The performance overhead and additional resources needed often lead to reducing consistency, resulting in less guarantees for users who must then build more complex applications to deal with potential inconsistencies.

[*] Part of this work was done while the author was at ETH Zürich.

The high price of consistency comes from the multiple rounds of communication required to reach agreement. Even in the absence of failures, a decision can be taken only as quickly as the network round-trip times allow it. Traditional networking stacks do not optimize for latency or specific communication patterns turning agreement protocols into a bottleneck. The first goal of this paper is to *explore whether the overhead of running agreement protocols can be reduced* to the point that it is no longer in the performance critical path. And while it is often possible to increase performance by "burning" more energy, the second goal is to *aim for a more efficient system*, i.e., do not increase energy consumption or resource footprint to speed up enforcing consistency.

In addition to the performance and efficiency considerations, there is an emerging opportunity for smarter networks. Several recent examples illustrate the benefits of pushing operations into the network [16, 41, 54] and using middleboxes to tailor it to applications [52, 9, 61, 49]. Building upon these advances, the following question arises: could agreement be made a property of the network rather than implementing it at the application level? Given the current trade off between complexity of operations and the achievable throughput of middleboxes, the third goal of this work is to *explore how to push down agreement protocols into the network in an efficient manner*.

Finally, data center architecture and the hardware used in a node within a data center is an important part of the problem. Network interface cards with programmable accelerators are already available from, e.g., Solarflare [55], but recent developments such as the HARP initiative from Intel [25] or the Catapult system of Microsoft [50] indicate that heterogeneous hardware is an increasingly feasible option for improving performance at low energy costs: the field programmable gate arrays (FPGAs) used in these systems offer the opportunity of low energy consumption and do not suffer from some of the traditional limitations that conventional CPUs face in terms of data processing at line-rate. Catapult also demonstrates the benefits of having a secondary, specialized network connecting the accelerators directly among themselves. When thinking of agreement protocols that are bound by round trip times and jitter, such a low latency dedicated network seems quite promising in terms of efficiently reducing overhead. We do not argue for FPGAs as the only way to solve the problem, but given their increasing adoption in the data center, it makes sense to take advantage of the parallelism and performance/energy advantages they offer. This leads us to the fourth and final question the paper addresses: *can an FPGA with specialized networking be used to implement consensus while boosting performance and reducing overall overhead*?

Contribution. In this paper we tackle the four challenges discussed above: We implement a consensus protocol in hardware in order to remove the enforcement of consistency from the critical path of performance without adding more bulk to the data center. We create a reusable solution that can augment middleboxes or smart network hardware and works both with TCP/IP and application-specific network protocols using hardware and platforms that are starting to emerge. Therefore the solution we propose is both a basis for future research but also immediately applicable in existing data centers.

Results. In the paper we show how to implement Zookeeper's atomic broadcast (ZAB [43]) on an FPGA. We expose the ZAB module to the rest of the network through a fully functional low latency 10Gbps TCP stack. In addition to TCP/IP, the system supports an application-specific network protocol as well. This is used to show how the architecture we propose can be implemented with point-to-point connections to further reduce networking overhead. For a 3 node setup we demonstrate 3.9 million consensus rounds per second over the application specific network protocol and 2.5 million requests per second over TCP. This is a significant improvement over systems running on commodity networks and is on par even with the state of art systems running over lower latency and higher bandwidth Infiniband networks. Node to node latencies are in the microsecond range, without significant tail latencies. To illustrate how this hardware consensus can be used in practical systems, we have combined it with a main-memory key value store running on specialized microservers (built as well on FPGAs). This results in a distributed service similar to Zookeeper that exhibits a much higher and stable performance than related work and can be used as a blueprint for further specialized designs.

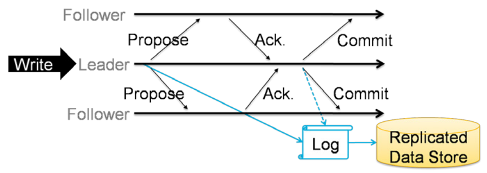


Figure 1: Zookeeper's Atomic Broadcast

2 Background

2.1 Zookeeper's Atomic Broadcast

There are many distributed systems that require some form of coordination for achieving their core services, and since implementing distributed consensus [34, 35] correctly is far from trivial [46], reusable libraries and solutions such as Zookeeper have emerged. Zookeeper is a centralized service that provides distributed synchronization, store configuration, and naming services for distributed systems. It achieves fault tolerance and high availability through replication.

At the core of Zookeeper is an atomic broadcast protocol (ZAB [33]) coupled with leader election that is used to ensure the consistency of modifications to the tree-based data store backing Zookeeper. ZAB is roughly equivalent to running Paxos [35], but is significantly easier to understand because it makes a simplifying assumption about the network. The communication channels are assumed to be lossless and strongly ordered (thus, Zookeeper in principle requires TCP).

We briefly describe the ZAB protocol in a 3 node setup (Figure 1): The atomic broadcast protocol of Zookeeper is driven by a leader, who is the only node that can initiate proposals. Once the followers receive proposals, they will acknowledge the receipt of these proposals thus signaling that they are ready to commit. When the leader received an acknowledgment from the majority of followers it will issue a commit message to apply the changes. Committed messages are persisted by default on a disk, but depending on the nature of the data stored in the service and failure scenarios, writing the log to memory can be enough. The order of messages is defined using monotonically increasing sequence numbers: the "Zxid'," incremented every time a new proposal is sent, and the "epoch" counter, which increases with each leader election round.

Zookeeper can run with two levels of consistency: strong [26] and relaxed (a form of prefix consistency [56]). In the strong case, when a client reads from a follower node, it will be forced to consult the leader whether it is up to date (using a *sync* operation), and if not, to fetch any outstanding messages. In the more relaxed case (no explicit synchronization on read) the node might return stale data. In the common case, however, its state mirrors the global state. Applications using Zookeeper often opt for relaxed consistency in order to increase read performance. Our goal is to make strong consistency cheaper and through this to deliver better value to the applications at a lower overall cost.

2.2 Reconfigurable Hardware

Field programmable gate arrays (FPGAs) are hardware chips that can be reprogrammed but act like application-specific integrated circuits (ASICs). They are appealing for implementing data processing operations because they allow for true dataflow execution [45, 57]. This computational paradigm is fundamentally different from CPUs in that all logic on the chip is active all the time, and the implemented "processor" is truly specialized to the given task. FPGAs are programmed using hardware description languages, but recently high level synthesis tools for compiling OpenCL [6], or domain specific languages [24] down to logic gates are becoming common.

FPGAs typically run at low clock frequencies (100-400 MHz) and have no caches in the traditional sense in front of the DDR memory. On the other hand the FPGA fabric contains thousands of on-chip block RAMs (BRAM) that can be combined to form different sized memories and lookup tables [13]. Recent chips have an aggregated BRAM capacity in the order of megabytes.

There are good examples of using FPGA-based devices in networks, e.g., smart NICs from Solarflare [55] that add an FPGA in the data path to process packets at line-rate, deep packet inspection [11] and line-rate encryption [51]. It has also been proposed to build middleboxes around FPGAs [9] because they allow for combining different functional blocks in a line-rate pipeline, and can also ensure isolation between different pipelines of different protocols. We see our work as a possible component in such designs that would allow middleboxes to organize themselves more reliably, or to provide consensus as a service to applications.

3 System Design

For prototyping we use a Xilinx VC709 Evaluation board. This board has 8 GB of DDR3 memory, four 10Gbps Ethernet ports, and a Virtex-7 VX690T FPGA (with 59 Mb Block RAM on chip). Our system consists of three parts: networking, atomic broadcast, and to help evaluate the implementation of the latter two, a key-value store as consensus application (Figure 2). The network stack was implemented using high level synthesis [60], the other two modules are written in Verilog and VHDL.

The Atomic Broadcast module replicates requests sent to the application (in our case the key-value store). Since it treats the actual requests as arbitrary binary data, it requires a thin header in front of them. The structure of the 16 B header is explained in Table 1: It consists of an operation "code" and ZAB-specific fields, such as

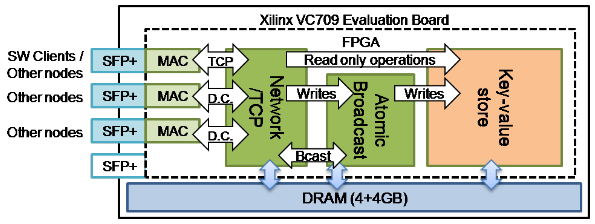


Figure 2: The target platform and system architecture

Bits	Description	Bits	Description
[15:0]	Magic number	[63:32]	Length of message
[23:16]	Sender Node ID	[95:64]	Zxid (req. sequence no.)
[31:24]	Operation code	[127:96]	Epoch number

Table 1: Structure of request header

epoch-number and Zxid. This is because the same header structure is used for communication between nodes and clients, and different node's atomic broadcast units. This means that not all messages will have a payload. As explained in Section 2.1, Zookeeper provides two levels of consistency, from which in our system we implement strong consistency by serving both reads and writes from the leader node. This setup simplifies the discussion and evaluation, however, serving strongly consistent read on followers is also possible.

When the atomic broadcast unit is used in conjunction with the key-value store, one can distinguish between two types of client requests: local ones (reads) and replicated ones (writes). Local requests are read operations that a node can serve from its local data store bypassing the atomic broadcast logic completely. Write requests need to be replicated because they change the global state. These replicated requests are "trapped" inside the atomic broadcast module until the protocol reaches a decision and only then are sent to the application, which will process them and return the responses to the client. For reaching consensus, the atomic broadcast module will send and receive multiple messages from other nodes. Since the atomic broadcast unit does not directly operate on the message contents, these are treated as binary data for the sake of replication.

4 Networking

The FPGA nodes implement two networking protocols: TCP/IP and an application specific one, used for point-to-point connections. As Figures 2 and 3 show, the network module connects to the Ethernet Network Interface provided by the FPGA vendor that handles Layer 1 and 2 (including MAC) processing before handing the packets to the IP Handler module. This module validates IP checksums and forwards packets to their protocol handlers. Additionally, data arriving from other FPGAs, using the application specific network protocol, shortcut

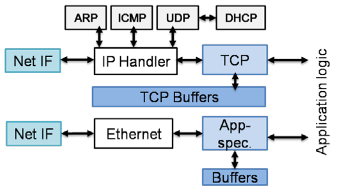


Figure 3: Network stack overview

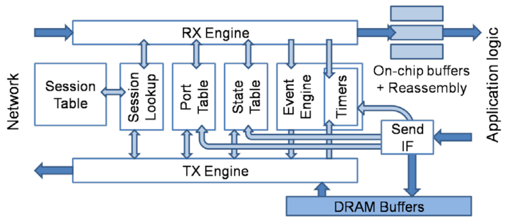


Figure 4: TCP stack details

the TCP/IP stack and are instead processed by an application specific network module. The ARP, ICMP and DHCP modules in Figure 3 provide the functionality necessary for integrating the FPGA into a real network.

4.1 TCP/IP in Hardware

The network stack is a modified version of earlier work [53]. The original design addresses the problem of very low connection counts in existing network stacks for FPGAs (typically in the tens or low hundreds). The changes introduced in this paper aim to further reduce the latency of the stack and reduce its memory footprint by tailoring the logic to the consensus and replication traffic.

The main benefit of using hardware for implementing a network stack is that it allows for building true dataflow pipelines and also for the isolation of send and receive paths so that they do not influence each other's performance negatively. Figure 4 shows the resulting architecture of the TCP stack in hardware: two pipelines that share connection state through the data structures. These data structures are: session lookup, port and state table, and an event engine backed by timers. The session lookup table contains the mapping of the 4-tuple (IP address and TCP port of destination and source) to session IDs, and is implemented as a content-addressable memory directly on the FPGA [32]. It holds up to 10k sessions in our current configuration. The port table tracks the state of each TCP port and the state table stores meta-information for each open TCP connection. The event engine is responsible for managing events and incoming requests from the Send Interface, and instructs the TX Engine accordingly.

4.2 Application-aware Receive Buffers

TCP operates on the abstraction of data streams, however data packets on the application level are usually very well defined. We take advantage of this application knowledge to reduce the latency of our network stack. The original version [53] of the network stack implemented "generic" receive buffers. In this version we replaced the DRAM buffer on the receive path with low latency on-chip BRAM. The smaller buffer space has no negative impact on throughput due to two reasons: 1) The application logic is designed to consume data at line-rate for most workloads, 2) In the datacenter TCP packets are in the common case rarely reordered [49]. Consequently, a smaller on-chip BRAM buffer will lower the latency without negatively impacting performance and frees up DRAM space for the consensus and application logic. Internally the BRAM buffers are organized as several FIFOs that are assigned dynamically to TCP sessions. By pushing down some knowledge about the application protocol (header fields), the BRAM buffers can determine when a complete request is available in a FIFO and then forward it to the application logic. In case all FIFOs fill up, we rely on TCP's built in retransmission mechanisms in order to not lose any data. For this reason on the transmit path a much larger buffer space is required, since packets have to be buffered until they are acknowledged. Therefore the high capacity DRAM buffer from our original design was kept.

4.3 Tailoring TCP to the Datacenter

TCP gives very strong guarantees to the application level, but is very conservative about the guarantees provided by the underlying network. Unlike the Internet, datacenter networks have well-defined topologies, capacities, and set of network devices. These properties, combined with knowledge about the application, allow us to tailor the TCP protocol and reduce the latency even further without giving up any of the guarantees provided by TCP.

Starting from the behavior of consensus applications and key-value stores we make two assumptions for the traffic of the key-value store and consensus logic to optimize the TCP implementation: a client request is always smaller than the default Ethernet MTU of 1500 B and clients are synchronous (only a single outstanding request per client). Additionally, we disable Nagle's algorithm which tries to accumulate as much payload from a TCP stream to fill an entire MTU. Since it waits for a fixed timeout for more data, every request small than MTU gets delayed by that timeout. The combination of disabling Nagle's algorithm, client requests fitting inside an MTU, and synchronous clients means that we can assume that in the common case and except for retransmission between the FPGAs, requests are not fragmented over multiple MTUs and each Ethernet frame holds a single request. Disabling Nagle's algorithm is quite common in software stacks through the TCP_NODELAY flag. Having our own hardware implementation we did an additional optimization to reduce latency by disabling Delayed Acknowledgments as described in RFC1122 [4], which says that a TCP implementation should delay an Acknowledgment for a fixed timeout such that it either can be merged with a second ACK message or with an outgoing payload message, thereby reducing the amount of bandwidth which is used for control messages. Since in our setup the network latencies between the FPGAs are in the range of a few microseconds, we decided to not just reduce the timeout but completely remove it. This way each message sent is immediately acknowledged. Obviously removing Delayed Acknowledgments and disabling Nagle's algorithm comes with the tradeoff that more bandwidth is used by control messages. Still, our experiments show that even for small messages we achieve a throughput of more than 7 Gbps considering only "useful" payload.

4.4 Application Specific Networking

In addition to the regular TCP/IP based channels, we have also developed a solution for connecting nodes to each other on dedicated links (direct connections, as we will refer to them in the paper, labeled D.C. in Figure 2), while remaining in-line with the reliability requirements (in-order delivery, retransmission on error). Packets are sent over Ethernet directly, and sequence numbers are the main mechanism of detecting data loss. These are inserted into requests where normally the ZAB-specific magic number is in the header – so the sequence number is actually increased with each logical request, not with each packet. Since the links are point-to-point, congestion control is not necessary beyond signaling backpressure (achieved through special control packets). If data was dropped due to insufficient buffer space on the receiving end, or because of corruption on the wire, the receiver can request retransmissions. To send and receive data over this protocol, the application uses special session numbers when communicating with the network stack, such that they are directly relayed to the application specific network module in Figure 3.

The design of the buffers follows the properties of the connections as explained above: the sending side maintains a single buffer (64 KB) per link from which it can resend packets if necessary, and the receiving side only reserves buffer space for request reassembly. Since the latency between nodes is in the order of microseconds, this buffer covers a time window of 50 μs on a 10 Gbps link, more than enough capacity for our purposes.

At the moment, our design only accommodates a limited number of nodes connected together with this protocol because there is no routing implemented and the FPGAs have only four Ethernet interfaces each. The Catapult platform [50] is a good example of what is possible over such secondary interconnects: it includes a 2D torus structure where FPGAs route packets over the dedicated

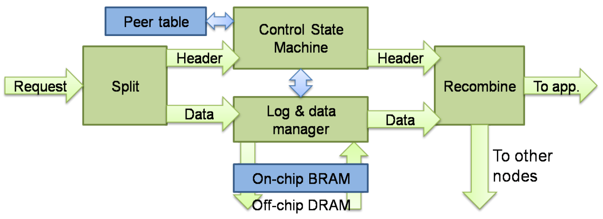


Figure 5: Overview of the atomic broadcast module

network, while using a simple application-specific protocol. We plan to eventually evaluate our system at larger scale using such a network interconnect.

5 Atomic Broadcast in Hardware

The overall benefit of using hardware for implementing consensus is that nodes have predictable performance, thereby allowing the protocol to function in the"best case scenario" most of the time. Latencies are bound and predictable, so with careful adjustments of on-chip buffers and memories, the hardware solution can for instance avoid in most cases to access the log in DRAM and read the cached head from on chip memory instead. Even the "less common" paths in the algorithm can perform well due to the inherent parallelism of FPGA resources, and the ability to hide memory access latencies through pipelining for instance. Another example is the timeout used for leader election that is much lower than what would be feasible in software solutions. In conclusion, the high determinism of hardware, low latency and inherent pipeline parallelism are a good fit for ZAB and there was no need to write a new solution from scratch.

By design, the atomic broadcast logic treats the data associated with requests as arbitrary binary data. This decouples it from the application that runs on top. For the purpose of evaluation, in this paper we use a key-value store but integrating other applications would be straightforward as well.

Inside the consensus module the control and data planes are separated, and the Control State Machine and the Log/Data Manager shown in Figure 5 can work in parallel to reduce latencies more easily. There are two additional blocks in the figure to complete the consensus functionality. The Splitter splits the incoming requests into command word and payload, and the Recombine unites commands with payloads for output. Headers (i.e., command words) are extracted from requests and reformatted into a single 128 bit wide word, so that they can be manipulated and transmitted internally in a single clock cycle (as compared to two on the 10Gbps data bus that is 64 bits wide). Similarly, payload data is aggregated into 512 bit words to match the memory interface width. When the control state machine (controller) issues a command (header) that has no encapsulated data,

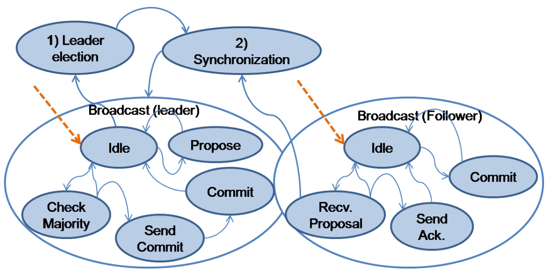


Figure 6: Abstract states used for implementing the Zookeeper Atomic Broadcast in the FPGA

Operation/State	**Cost** (clock cycles)	**Max for: 3 nodes**	**Max for: 7 nodes**
L_1 Create and send proposal	2+C_{nodes}x2	19.5 M/s	9.75 M/s
F_1 Recv. proposal and send acknowledgment	2	78 M/s	78 M/s
L_2 Recv. acknowledgment and check majority	2+C_{nodes}+L_{cLog}	17.3 M/s	13 M/s
F_2 Commit	1+L_{cLog}	39 M/s	39 M/s
L_3 Commit	3+C_{nodes}	26 M/s	15.6 M/s
Consensus round (leader)	$L_1 + L_2 + L_3$	7.1 M/s	4.1 M/s
Consensus round (follower)	$F_1 + F_2$	26 M/s	26 M/s

Table 2: Cost of ZAB operations and the theoretical maximum consensus rounds per second over 10GbE

such as the *acknowledgment* or *commit* messages of the ZAB protocol, this passes through the Recombine module without fetching a payload from the log. If, on the other hand, there is encapsulated data to be sent then the controller will request the data in parallel to creating the header. The system is pipelined, so it is possible to have multiple outstanding requests for data from memory.

5.1 State Machine

The state machine that controls the atomic broadcast is based on the ZAB protocol as described in [43]. Figure 6 shows the "super states" in which the state machine can be (each such oval on in the figure hides several states of its own). Transitions between states are triggered by either incoming packets or by means of event timers. These timers can be used for instance to implement timeouts used in leader election, detection of failed nodes, etc., and operate in the range of tens of μs.

Table 2 shows an overview of how many clock cycles the more important states of the state machine take to be processed. Most of them are linear in cost with the number of nodes in a setup (C_{nodes}), or the time it takes to seek in the command log as the parameter L_{cLog} (3 cycles in the average case in our current design). The theoretical maximum throughput achievable by the control unit shown in Table 2 for the 3 node setup we use in the Evaluation is higher than the maximum throughput in reality as our system is limited by 10Gbps networking most of the time. If we wanted to scale our system up to 40 Gbps networking, this component could be clocked up to 300 MHz (independently from the rest of the pipeline) and then it would have enough performance to handle the increased message rate. The rest of the logic inside the atomic broadcast module handles the payloads only, and these paths could be made wider for 4 x throughput.

On each node there is a table to hold the state of the other nodes in the cluster. The table resides in BRAM and in the current implementation holds up to 64 entries. Each entry consists of information associated with Zookeeper atomic broadcast (Zxid, epoch, last acknowledged Zxid, etc.), a handle to the session opened to the node, and a timestamp of the node's last seen activity. Since the mapping from session number to network protocol or even network interface is made in the networking module, the controller is independent of the network details and works the same for messages received over any protocol or connection.

5.2 Quorums and Committing

Zookeeper atomic broadcast allows the pipelining of requests, so when the leader's controller receives a client request that needs to be replicated it will send out the proposal and mark its Zxid as the highest that has already been sent but not acknowledged or committed. When an acknowledgment is received from another node, the leader's controller will test if a quorum (majority) has been reached on that Zxid. This is done by iterating through the active nodes in the state table: if enough nodes have already acknowledged, the leader's controller will send out commit messages to all nodes that already acknowledged the proposal. Then the leader will instruct the log unit to mark the operation as successful and to return the payload so that the application can process the original request. On the follower, the receipt of a commit message will result in the same operations of updating the log and preparing the data for the application. In case a node sends its acknowledgment after the operation has already been committed, the leader will issue a commit to that node as a response.

The system offers tunable consistency by allowing the quorum-checking function to be updated at runtime. To be more specific, one can change between either waiting for a majority or waiting for all nodes to respond. The latter behavior could be useful in cases when failures of nodes are assumed to be transient, but updates have to happen absolutely at the same time on all nodes (like changing a policy on a set of middleboxes). While in software this could lead to much higher response times, in the Evaluation section we show the benefits of the low latency hardware.

5.3 Maintaining a Log

After the payload is separated from the command it is handed to the Log and Data Manager (bottom half of Figure 5). The payload is added to an append-only log, and read out later to be sent to other nodes with the proposal commands. When an operation commits, this event is also added to the log (a physically different location, reserved for command words) with a pointer to the payload's location. In case a synchronization has to happen, or the node needs to read its log for any other reason, the access starts with the part of the log-space that contains the commands. Since each entry is of fixed size, it is trivial to search a command with a given Zxid.

To reduce the latency of accessing this data structure we keep the most recent entries in on-chip BRAM, which is spilled to DRAM memory in large increments. Of course if the payloads are large only a small number will fit into the first part of the log. However, this is not really an issue because in the common case each payload is written once to the log when received and then read out immediately for sending the proposals (it will still be in BRAM at this point), and then read again later when the operation commits. The aspect where the atomic broadcast unit and the application need to work together is log compaction. In the case of the key-value store, the log can be compacted up to the point where data has been written to the key-value store, and the key-value store notifies the atomic broadcast unit of each successful operation when returning the answer to the client.

Our design is modular, so that the log manager's implementation could change without requiring modifications in the other modules. This is particularly important if one would want to add an SSD to the node for persistent log storage. We have mechanisms to extend the FPGA design with a SATA driver to provide direct access to a local SSD [59]. Although we have not done it in the current prototype, this is part of future work as part of developing a data appliance in hardware. Alternatively, one can use battery backed memory [5], which in the context of the FPGA is a feasible and simpler option.

5.4 Synchronization

When a node fails, or its network experienced outages for a while it will need to recover the lost messages from the leader. This is done using *sync* messages in the ZAB protocol. In our implementation, when a follower detects that is behind the leader it will issue a *sync* message. The leader will stream the missing operations from the log to the follower. These messages will be sent with an opcode that will trigger their immediate commit on the other end. In the current design the leader performs this operation in a blocking manner, where it will not accept new input while sending this data. It is conceivable to perform this task on the side, but for simplicity we implemented it this way for this prototype design.

If for some case the other node would be too far behind the leader, and syncing the whole log would take longer than copying the whole data structure in the key-value store (or the log has already been compacted beyond the requested point) there is the option of *state transfer* at bulk: copying the whole hash table over and then sending only the part of the log that has been added to the table since the copy has begun. Of course this tradeoff depends on the size of the hash table, and is an experiment that we defer to our future work.

5.5 Bootstrapping and Leader Election

On initial start-up each node is assigned a unique ID by an outside entity, e.g., a configuration master [36]. This ID is a sequence number that is increased as nodes are added. If a node joins after the initial setup, it gets the next available ID and all other nodes are notified. If a node fails and then recovers, it keeps its ID. When thinking of smart network devices or middleboxes connected together in a coordination domain, it is reasonable to expect much less churn than with regular software nodes.

We implement the leader election following the algorithm in the ZAB paper [33], with the optimization that the followers will propose prospective leaders in a round-robin fashion, i.e., proposing the next higher ID once the current leader is unreachable. Nodes transition to leader election once no message or heartbeat has been received from the leader for a given timeout (based on the maximum consensus time in our use-case we set this to $50\mu s$). We perform the synchronization phase after the leader election (discovery phase in the ZAB paper) in a pull-based manner. This means that the newly elected leader will explicitly ask the most up to date follower to send it the requests with which it might be behind instead of followers actively sending their histories. Requests arriving from clients during leader election and synchronization will be dropped by default, to allow the clients to reconfigure based on a timeout mechanism. One simple optimization that we implement is responding to requests arriving during the leader election with a message that will prompt the client to switch over to the next leader directly without timeouts. Further, more sophisticated optimizations are possible, but are deferred to future work.

6 Use-case: Key-value Store

In order to test the atomic broadcast unit with a realistic application running on the same FPGA we implemented a key-value store that at its core uses the hash table design from our earlier work [29]. It is compatible with memcached's ASCII protocol and implements the *set*, *get*, *delete* and *flush all* commands. In addition, it supports memcached's check-and-set (*cas*) as well. The design is aggressively pipelined and handles mixed workloads well. As we previously demonstrated, the internal pipeline can process more than 11 million memcached requests per second, enough to saturate a 10Gbps connection even with keys and values as small as 16 B.

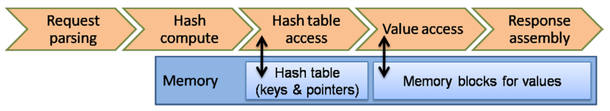


Figure 7: Internal pipeline of the key-value store

	32B Request	512B Req.	1KB Req.
Ethernet loopback	0.6μs	1.4μs	2.2μs
TCP loopback	1.5μs	3.8μs	6.5μs
Direct Conn. loopback	0.7μs	1.5μs	2.3μs
DRAM access latency	0.2μs		
Ping from client	15μs	35μs	–

Table 3: Latencies of different components of the system

Keys are hashed with a hash function that is based on multiplication and shifting and processes the input one byte at a time (similar to a Knuth's hash function). To meet the line-rate requirements we rely on the parallelism of the FPGA and create multiple hashing cores that process the keys in the order of arrival. We solve collisions by chaining, and the first four entries of every chain are pre-allocated and stored in memory in such a way that they can be read in parallel. This is achieved by dividing each physical memory line (512b on our platform) into four equal parts belonging to different entries, and tiling keys over multiple physical memory addresses. The start address of each bucket is at a multiple of 8 memory lines, which allows for keys of up to 112 B long (the header of each entry is 16 B) to be stored. This size should be enough for most common use-cases [10].

In order to hide the memory latency when accessing the data structures in off-chip DDR memory, the hash table is implemented as series of pipelined stages itself. Multiple read commands can be issued to the memory, and the requests will be buffered while waiting for the data from memory. While with this concurrency there is no need for locking the hash table entries in the traditional sense, the pipeline has a small buffer on-chip that stores in-flight modifications to memory lines. This is necessary to counter so called read-after-write hazards, that is, to make sure that all requests see a consistent state of the memory. A benefit of no locking in the software sense, and also hiding memory latency through pipelining instead of caching, is that the hash table is agnostic to access skew. This is an improvement over software because in the presence of skew any parallel hash table will eventually become bottlenecked on a single core.

Similarly to the related work in software [7] and hardware [30], we store the values in a separate data structure from keys. This allows for more flexible memory allocation strategies, and also the option to provide more complex ways of managing memory in the future without modifying the hash table data structure. At the moment we use a simple block based memory allocation scheme that allocates memory linearly. When a key is inserted into the hash table, and its value placed in memory, its slot in the value store is reused for subsequent updates as long as the modified value is smaller than or equal in size to the previous one.

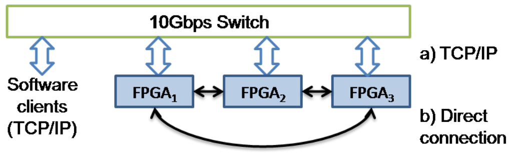


Figure 8: Evaluation setup of our prototype system

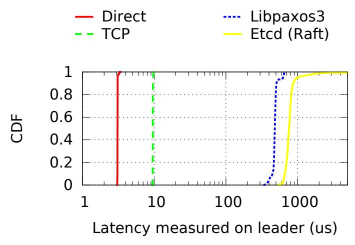
7 Evaluation

7.1 Setup

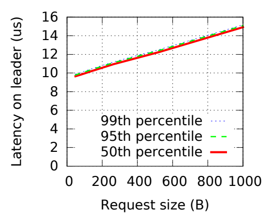
For evaluation, we use 12 machines connected to a 10Gbps 48 port Cisco Nexus 5596UP switch and three FPGAs connected to the same switch (Figure 8). FPGAs communicate either over TCP or the specialized network, i.e., direct connections. The three node setup mirrors the basic fault-tolerant deployment of Zookeeper that can tolerate one faulty node. The client machines have dual-socket Xeon E5-2609 CPUs, with a total of 8 cores running at 2.4 GHz, 128 GB of memory and an Intel 82599ES 10Gbps network adapter. The machines are running Debian Linux (jessie/sid with kernel version 3.12.18) and use the standard ixgbe drivers. Our load generator was memaslap [2] with modifications to include our ZAB header in the requests.

7.2 Baselines

The performance of consensus protocols is sensitive to latency, so we performed a series of micro-benchmarks and modeling to determine the minimal latencies of different components of our system with differently sized packets. As the results in Table 3 show, the transmission of data through TCP adds the most latency (ranging between 1 and 7μs), but this is expected and is explained by the fact that the packet goes through additional checksumming and is written and read from a DRAM buffer. An other important result in Table 3 is that round trip times of ping messages from software are almost an order of magnitude higher than inter-FPGA transmission times, which highlights the shortcomings of the standard networking stack in software. The measurements were taken using the ping-flood command in Linux. In the light of this, we will mainly report consensus latencies as measured on the leader FPGA (we do this by inserting two timestamps in the header: one when a message is received and the other when the first byte of the response is sent), and show times measured on the client for experiments that involve the key-value store more.



(a) Small requests, related work



(b) Increasing size over TCP

Figure 9: Consensus round latency on leader

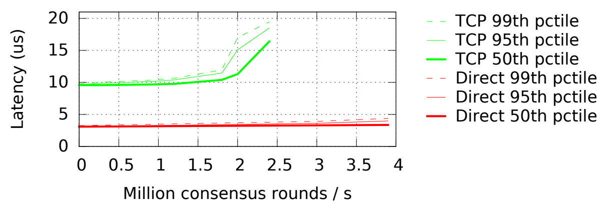


Figure 10: Load vs. Latency on leader

7.3 Cost of Consensus

Systems such as Zookeeper require hundreds of microseconds to perform a consensus round [48, 20] even without writing data to disk. This is a significant overhead that will affect the larger system immediately, and here we explain and quantify the benefits of using hardware for this task. Instead of testing the atomic broadcast module in isolation, we measure it together with the application on the chip, using memaslap on a single thread sending one million consecutive write requests to the key-value store that need to be replicated. We chose very small request size (16 B key and 16 B value) to ensure that measurements are not influenced by the key-value store and stress mostly the atomic broadcast module.

Figure 9(a) depicts the probability distribution of a consensus round as measured on the leader both when using TCP and direct connections to communicate with followers. Clearly, the application-specific network protocol has advantages over TCP, reducing latencies by a factor of 3, but the latter is more general and needs no extra infrastructure. Figure 9(b) shows that the latency of consensus rounds increase only linearly with the request size, and even for 1KB requests stay below 16μs on TCP. To put the hardware numbers in perspective we include measurements of Libpaxos3 [3] and the Raft implementation used in Etcd [1]. We instrumented the code of both to measure the latency of consensus directly on th leader and deployed them on three nodes in our cluster. Unsurprisingly the software solutions show more than an order of magnitude difference in average latency, and have significantly higher 99th percentiles even for this experiment where the system handles one request at a time.

Figure 10 shows how the FPGA system fares under increasing load of replicated requests. As later shown in the experiments, with small payloads (<48 B) the system can reach up to 2.4 million consensus rounds per second over TCP and almost 4 million over direct connections before hitting bottlenecks. In our graph the latencies do not increase to infinity because we increased throughput only until the point where the pipeline of the consensus module and the input buffers for TCP could handle load without filling all buffers, and the clients did not need to retransmit many messages. Since we measured latency at the leader, these retransmitted messages would lead to false measurements from the leader's perspective.

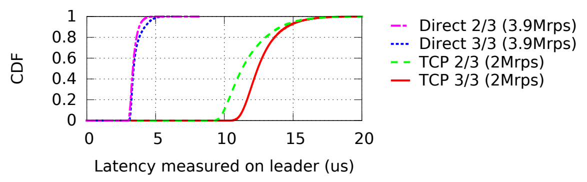


Figure 11: Latency of consensus on leader when waiting for majority or all nodes, on TCP and direct connections

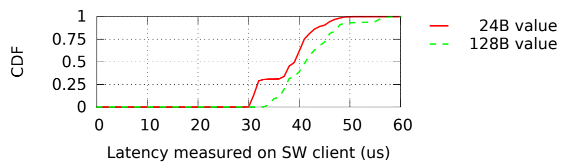


Figure 12: Round-trip times with replication measured on client

7.4 Quorum Variations

The leader can be configured at runtime to consider a message replicated either when a majority of nodes acknowledged or all of them. The second variant leads to a system of much stronger consistency, but might reduce availability significantly. We measured the effect of the two strategies on consensus latency, and found that even when the system is under load waiting for an additional node does not increase latencies significantly. This is depicted in Figure 11 both for TCP and direct, "2/3" being the first strategy committing when at least two nodes agree and "3/3" the second strategy when all of them have to agree before responding to the client.

7.5 Distributed Key-value Store

The rest of the evaluation looks at the atomic broadcast module as part of the distributed key-value store running on the FPGAs. We measure the round trip times (latency) on the clients and report maximum throughput with multiple client machines. As seen in Figure 12, for a single threaded client round trip times are 30μs larger than the measurements taken on the leader. The reason for this is the inefficiency of the software network stack, and is in line with the ping micro-benchmarks. Interestingly, even though these numbers are measured on a system with a single client and one TCP connection, software still adds uncertainty to the high percentiles.

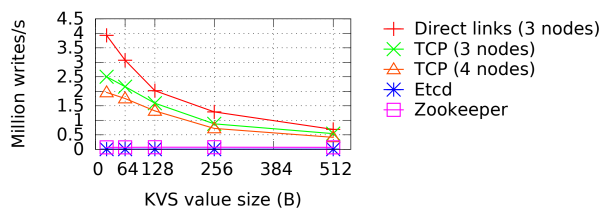


Figure 13: Consensus rounds (writes operations) per second as a function of request size

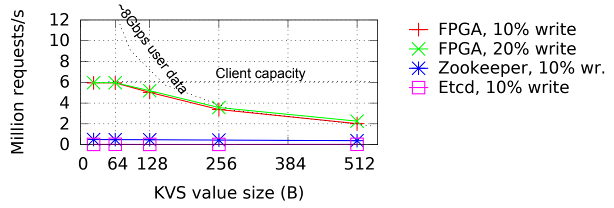


Figure 14: Mixed workload throughput measured on leader without and with replication (3 nodes)

When varying the size of the values in the write requests to the key-value store, we can exercise different parts of the system. The key size is kept at a constant 16 B for all experiments. Figure 13 and Figure 14 show the achievable throughput for write-only and mixed workloads (for the former all requests are replicated, for the latter only a percentage). Naturally, the write-only performance of the system is limited by the consensus logic when using direct connections, and by the combination of the consensus logic and the TCP/IP stack otherwise. This is because the transmit path on the leader has to handle three times as many messages as the receive path, and random accesses to DRAM limit performance. To further explore this aspect we ran experiments with a 4 node FPGA setup as well, and seen that the performance scales as expected.

For mixed workloads and small requests the clients' load generation capability is the bottleneck, while for larger messages performance is bound by the network capacity. This is illustrated by the two workloads in Figure 14, one with 10% writes and the other with 20%. Since they show the same performance, the atomic broadcast logic is not the bottleneck in these instances. The workload with 20% writes is actually slightly faster because the average size of responses to the clients gets smaller (each read request to the key-value store will return the key, value and headers, while write requests only return headers and a success message).

We ran Zookeeper and Etcd on three machines each and performed the same mixed-workload experiment as before. To make sure that they are not impacted by hard drive access times, we set up ram disks for persisting logs. Figure 14 shows that their performance is not bound by the network, and is mostly constant even for large messages. Both are almost an order of magnitude slower for small messages than hardware.

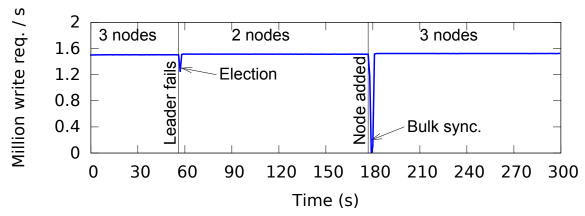


Figure 15: Leader election triggered while client issue an all-write workload, nodes connected via TCP links

7.6 Leader Election

To exercise leader election and recovery we simulate a node failure of the leader, which results in a leader election round without significant state transfer since the FPGAs do not drift much apart under standard operation. Hence leader election over TCP/IP takes approximately as long as a consensus round (10μs in average), not counting the initial timeout of the followers (50μs) Figure 15 depicts the experiment: we generate write-only load from several clients for a three node FPGA cluster communicating over TCP and at the 56 s mark the leader node fails and a new leader is elected. To make client transition possible we modified memaslap and added a timeout of 100 ms before trying an other node (the clients retry in the same round robin order in which the FPGAs try to elect leaders). The graph indicates that the dip in performance is due to the 100 ms inactivity of clients, since leader election takes orders of magnitude less.

Synchronization of state between nodes happens for instance when a new node joins the cluster. In Figure 15 we shows the previously failed node recovering after 2 minutes and prompting the new leader to synchronize. Since at this point the log has been compacted, the leader will bulk transfer the application state that consists of the hash table and value area, occupying 256MB and 2GB, respectively. During synchronization the leader will not handle new write requests to keep the state stable, hence the clients will repeatedly time out and resume normal operation only once the leader has finished the state transfer. The results show that, as expected, this step takes between 2-3 seconds, the time necessary to send the state over 10Gbps network plus clients resuming.

This experiment allows us to make two observations. First, leader election can be performed very quickly in hardware because detecting failed nodes happens in shorter time frames than in software (i.e., in order of 10s of μs). Hence, leader-change decisions can be taken quickly thanks to low round-trip times among nodes. Second, the cost of performing bulk transfers shows that in future work it will be important to optimize this operation. The hardware could benefit from the approach used by related work, such as DARE [48], where the followers synchronize the newly added node. This leads to smaller performance penalty incurred by state transfer at the cost of a slightly more complex synchronization phase.

Component	LUTs	BRAM	DSPs
PHY and Ethernet (x3)	16k	18	0
TCP/IP + App-spec.	23k	325	0
Memory interface	47k	127	0
Atomic broadcast	10k	340	0
Key-value store	28k	113	62
Total used *(% of available)*	124k *(28%)*	923 *(63%)*	62 *(2%)*

Table 4: Detailed breakdown of the resource usage

7.7 Logic and Energy Footprint

To decide whether the hardware consensus protocol could be included in other FPGA-based accelerators or middleboxes we need to look at its logic footprint. FPGAs are a combination of look-up tables (LUTs) that implement the "logic", BRAMs for storage, and digital signal processors (DSPs) for complex mathematical operations. In general, the smaller the footprint of a module, the more additional logic can fit on the chip besides it. Table 4 shows that the ZAB module is smaller than the network stack or the key-value store, and uses only a fraction of the LUTs on the chip. The resource table also highlights that a big part of the BRAMs are allocated for networking-related buffers and for log management. While it is true that some of these could be shrunk, in our current setup where there is nothing else running on the FPGA, we were not aiming at minimizing them.

One of the goals of this work is to show that it is possible to build a Zookeeper-like service in an energy efficient way. The power consumption of the FPGAs, even when fully loaded, is 25 W – almost an order of magnitude lower than the power consumption of a x86 server.

8 Related Work

8.1 Coordination in Systems

Paxos [34, 35] is a family of protocols for reaching consensus among a set of distributed processes that may experience failures of different kinds, including ones in the communication channels (failure, reordering, multiple transmission, etc.). While Paxos is proven to be correct, it is relatively complex and difficult to implement, which has led to alternatives like Raft [46], ZAB [27, 43] or chain replication [58]. There is also work on adapting consensus protocols for systems that span multiple physical datacenters [38, 40, 15], and while they address difficult challenges, these are not the same problems faced in a single data-center and tight clusters.

Paxos and related protocols are often packaged as coordination services when exposed to large systems. Zookeeper [27] is one such coordination service. It is a complex multi-threaded application and since its aim is to be as universal as possible, it does not optimize for either the network or the processor. Related work [48, 20] and our benchmarks show that its performance is capped around sixty thousand consensus rounds per second and that its response time is at least an order of magnitude larger than the FPGA (300-400μs using ram disks). Etcd [1], a system similar to Zookeeper, written in Go and using Raft [46] at its core has lower throughput than Zookeeper. This is partially due to using the HTTP protocol for all communication (both consensus and client requests) which introduces additional overhead.

Many systems (including e.g., the Hadoop ecosystem) are based on open source coordination services such as Zookeeper and Etcd, or proprietary ones (e.g., the Chubby [14] lock server). All of them can benefit from a faster consensus mechanisms. As an illustration, Hybris [20] is a federated data storage system that combines different cloud storage services into a reliable multi-cloud system. It relies on Zookeeper to keep metadata consistent. This means that most operations performed in Hybris directly depend on the speed at which Zookeeper can answer requests.

8.2 Speeding up Consensus

Recently, there has been a high interest in speeding up consensus using modern networking hardware or remote memory access. For instance DARE [48] is a system for state machine replication built on top of a protocol similar to Raft and optimized for one-sided RDMA. Their 5 node setup demonstrates very low consensus latency of $<15\mu$s and handles 0.5-0.75 million consensus rounds per second. These numbers are similar to our results measured on the leader for 3 nodes (3-10μs) and, not surprisingly, lower than those measured on the unoptimized software clients. While this system certainly proves that it is possible to achieve low latency consensus over Infiniband networks and explores the interesting idea of consensus protocols built on top of RDMA, our hardware-based design achieves higher throughput already on commodity 10 GbE and TCP/IP.

FaRM [23] is a distributed main-memory key value store with strong consensus for replication and designed for remote memory access over 40Gbps Ethernet and Infiniband. It explores design trade-offs and optimizations for one-sided memory operations and it demonstrates very high scalability and also high throughput for mixed workloads (up to 10M requests/s per node). FaRM uses a replication factor of three for most experiments and our hardware solution performs comparably both in terms of key-value store performance (the hardware hash table reaches 10 Gbps line-rate for most workloads [29]) and also in terms of consensus rounds per second, even though the FPGA version is running on a slower network.

NetPaxos [18] is a prototype implementation of Paxos at the network level. It consists of a set of OpenFlow extensions implementing Paxos on SDN switches; it also offers an alternative, optimistic protocol which can be implemented without changes to the Open- Flow API that relies on the network for message ordering in low traffic situations. Best case scenarios for NetPaxos exhibit two orders of magnitude higher latency than our system, FARM, or DARE. It can also sustain much lower throughput (60k requests/s). The authors point out that actual implementations will have additional overheads. This seems to indicate that it is not enough to push consensus into the network but it is also necessary to optimize the network and focus on latency to achieve good performance. In more recent work [17] the same authors explore and extend P4 to implement Paxos in switches. While P4 enables implementing complex functionality in network devices, the high level of abstraction it provides might make it difficult to implement the kind of protocol optimizations we describe in this paper and that are necessary to achieve performance comparable to that of conventional systems running over Infiniband.

Similar to the previously mentioned work, Speculative Paxos[49] suggests to push certain functionality into the network, e.g., message ordering. The design relies on specific datacenter characteristics, such as the structured topology, high reliability and extensibility of the network through SDN. Thereby, it could execute requests speculatively and synchronization between replicas only has to occur periodically. Simulations of the proposed design show that with increasing number of out-of-order messages the throughput starts to decrease quickly, since the protocol and application have to rollback transactions.

8.3 Quicker and Specialized Networking

One of the big challenges for software applications facing the network is that a significant time is spent in the OS layers of the network stack [47, 31] and on multicore architectures response times can increase as a result of context switching and memory copies from the NIC to the right CPU core. As a result, there are multiple frameworks for user-space networking [28, 31], and on the other end of the spectrum, operating systems [12, 47] that aim to speed up networking by separating scheduling and management tasks. The use of RDMA [22, 44] is also becoming common to alleviate current bottlenecks, but there are many (legacy) systems that rely on the guarantees provided by TCP, such as congestion control, in-order delivery and reliable transmission. Although some functionality of the network stack is offloaded to the NIC, processing TCP packets still consumes significant compute resources at the expense of the applications. Hardware systems, as we present in this paper, are implementing network processing as a dataflow pipeline and thereby can provide very high performance combined with the robustness and features of TCP.

A good example of what can be achieved with user-space networking is MICA [37], a key-value store built from the ground up using Intels DPDK [28] library. The results of this work are very promising: when using a minimalistic stateless protocol the complete system demonstrates over 70 million requests per second over more than 66Gbps network bandwidth (using a total of 8 network interfaces and 16 cores). It is important to note however that in MICA and similar systems skewed workloads will experience slowdowns due to the partitioned nature of the data structures. Additionally, a significant part of the servers logic (for instance computing the hash function on keys, or load balancing) is offloaded to clients. Our aim with the hardware solution on the other hand was to offer high throughput, low latency while relying on simple clients and commodity networks.

8.4 Hardware for Middleboxes

There is a wide spectrum of middlebox implementations ranging from all-software [42, 21, 8, 39], through hybrid [52, 9], to all-hardware [19]. One advantage of using FPGA-based solutions over software is that data can be processed at line-rate and only a minimal overhead in terms of latency is added. In ClickOS [42], for instance, adding a 40ms delay to get load balancing or congestion control is considered a good tradeoff. A hardware-based solution like the one we propose can perform even more complex operations, possibly involving coordination and consensus, in a fraction of that overhead.

9 Conclusion

In this paper we have explored a number of research questions aiming at determining whether the overhead of consensus can be removed as a bottleneck in distributed data processing systems. First, we have shown that it is possible to reduce the cost of reaching consensus without compromising reliability or correctness, through the means of specialized hardware. Second, based on the low latency and high throughput achieved, we have shown how to use the hardware consensus to implement a fully functional version of Zookeeper atomic broadcast with a corresponding key-value store. Third, we have argued that the proposed consensus module is agnostic to the actual request contents sent to the application and, hence, it could easily be integrated with middleboxes or other accelerators/microservers built with FPGAs. Finally, we have explored the benefits of using a custom messaging protocol for reducing latency, establishing the basis for further research into application specific protocols over secondary networks.

Acknowledgments

We would like to thank Xilinx for the donation of the FPGA boards used in this paper. The work reported in this paper is partially funded by Xilinx, Microsoft Research and the Swiss State Secretariat for Education Research and Innovation (SERI) under contract number 15.0025 (as part of the SUPERCLOUD H2020 project). We would also like to thank our shepherd, Wyatt Lloyd, for his help in improving our paper.

References

[1] Etcd repository in the CoreOS project. `https://github.com/coreos/etcd`.

[2] Libmemcached-1.0.18. `https://launchpad.net/libmemcached/`.

[3] LibPaxos3 repository. `https://bitbucket.org/sciascid/libpaxos`.

[4] Network working group: Requirements for internet hosts – communication layers. `https://tools.ietf.org/html/rfc1122`.

[5] Viking Technology. http://www.vikingtechnology.com/.

[6] Altera. Programming FPGAs with OpenCL. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01173-opencl.pdf`.

[7] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP'09*.

[8] James W Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xOMB: extensible open middleboxes with commodity servers. In *ANCS'12*.

[9] Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, and Nick Feamster. Switchblade: a platform for rapid deployment of network protocols on programmable hardware. *ACM SIGCOMM CCR*, 40(4), August 2010.

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS'12*.

[11] Michael Attig and Gordon Brebner. 400 Gb/s programmable packet parsing on a single FPGA. In *ANCS'11*.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *OSDI'14*.

[13] Stephen Brown and Jonathan Rose. FPGA and CPLD architectures: A tutorial. *IEEE Design & Test*, 13(2), June 1996.

[14] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI'06*.

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Googles globally distributed database. *ACM TOCS*, 31(3), August 2013.

[16] Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. NaaS: Network-as-a-Service in the Cloud. In *Hot-ICE'12*.

[17] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM CCR*, 2016.

[18] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at network speed. In *SOSR'15*.

[19] Lorenzo De Carli, Yi Pan, Amit Kumar, Cristian Estan, and Karthikeyan Sankaralingam. Plug: flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *SIGCOMM'09*.

[20] Dan Dobre, Paolo Viotti, and Marko Vukolić. Hybris: Robust hybrid cloud storage. In *SOCC'14*.

[21] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *SIGOPS'09*.

[22] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *NSDI'14*.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP'15*.

[24] Nivia George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *FPL'14*.

[25] PK Gupta. Xeon+FPGA platform for the data center. In *CARL'15*.

[26] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), July 1990.

[27] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC'10*.

[28] Intel. DPDK networking library. `http://dpdk.org/`.

[29] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. A hash table for line-rate data processing. *ACM TRETS*, 8(2), March 2015.

[30] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees A. Vissers. A flexible hash table design for 10Gbps key-value stores on FPGAs. In *FPL'13*.

[31] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *NSDI'14*.

[32] Weirong Jiang. Scalable ternary content addressable memory implementation using FPGAs. In *ANCS'13*.

[33] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN'11*.

[34] Jonathan Kirsch and Yair Amir. Paxos for system builders: An overview. In *LADIS '08*, 2008.

[35] Leslie Lamport. Generalized consensus and paxos. Technical report, Microsoft Research MSR-TR-2005-33, 2005.

[36] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *PODC'09*.

[37] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: a holistic approach to fast in-memory key-value storage. In *NSDI'14*.

[38] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI'13*.

[39] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. ServerSwitch: A programmable and high performance platform for data center networks. In *NSDI'11*.

[40] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *PVLDB*, 6(9), July 2013.

[41] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *CoNEXT'14*.

[42] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *NSDI'14*.

[43] André Medeiros. Zookeeper's atomic broadcast protocol: theory and practice. Technical report, 2012.

[44] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *USENIX ATC'13*.

[45] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on FPGAs. *PVLDB*, 2(1), August 2009.

[46] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC'14*.

[47] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *OSDI 14*.

[48] Marius Poke and Torsten Hoefler. DARE: high-performance state machine replication on RDMA networks. In *HPDC'15*.

[49] 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 *NSDI'15*.

[50] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jordan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA'14*.

[51] Safenet. Ethernet encryption for data in motion. `http://www.safenet-inc.com/data-encryption/network-encryption/ethernet-encryption/`.

[52] Alan Shieh, Srikanth Kandula, and Emin Gun Sirer. Sidecar: building programmable datacenter networks without programmable switches. In *HotNets'10*.

[53] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware. In *FCCM'15*.

[54] Solarflare. Accelerating memcached using Solarflare's Flareon Ultra server I/O adapter. December 2014. `http://www.http://solarflare.com/Media/Default/PDFs/Solutions/Solarflare-Accelerating-Memcached-Using-Flareon-Ultra-server-IO-adapter.pdf`.

[55] Solarflare. Application OnLoad Engine (AOE). `http://www.solarflare.com/applicationonload-engine`.

[56] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12), December 2013.

[57] Jens Teubner and Louis Woods. Data processing on FPGAs. *Morgan & Claypool Synthesis Lectures on Data Management*, 2013.

[58] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI'04*.

[59] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex - an intelligent storage engine with support for advanced SQL off-loading. *PVLDB*, 7(11), July 2014.

[60] Xilinx. Vivado HLS. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[61] Wei Zhang, Timothy Wood, KK Ramakrishnan, and Jinho Hwang. Smartswitch: Blurring the line between network infrastructure & cloud applications. In *HotCloud'14*.