SYS: Synchronize Your System with Simple Hardware

Michael Wei Steven Swanson
mwei@cs.ucsd.edu swanson@cs.ucsd.edu
Department of Computer Science and Engineering
University of California, San Diego

Abstract

Modern distributed systems must deal with the real-world problems of asynchrony and failures. Protocols which provide the illusion of synchrony and reliability add significant overhead and complexity, while exposing asynchrony and failures makes programs difficult to reason about. In this paper, we show that we can implement reliable atomic broadcast in hardware over a synchronous network. We show that we can use this network to build primitives which support a wide variety of distributed applications. Our system offers increased reliability and performs an order of magnitude faster than traditional asynchronous systems.

1 Introduction

Distributed systems today must deal with the complexity of asynchrony and failures: a request over the network can take an arbitrary amount of time, or in the case of a failure, never complete. In spite of this unreliability, distributed systems must still coordinate to do work. To that end, most distributed systems either utilize protocols that provide the illusion of synchrony and are robust to failures, or pass failures and asynchrony on to the programmer using relaxed consistency models. Both options have serious drawbacks: not only does providing the illusion of synchrony add significant overhead, but ensuring the correctness of the implementation of any such protocol is not a trivial task [2]. Relaxed consistency models, on the other hand can make distributed programs difficult, even for the experienced programmer, to reason about.

Asynchrony in distributed systems stems from two major sources: the general-purpose processors distributed applications run on, and the network. In a distributed system, processors may crash or simply take a long time to process requests due to load or resource sharing. The network may take a long time to deliver a message over congested network links, deliver messages out of order, or drop a message completely if it is overloaded. Asynchrony complicates implementing atomic broadcast, the process of sending a message so that it appears on all receivers at the same time, since it is uncertain when or if a receiver has received a message.

In this paper, we describe *SYS*, which uses a specialized hardware accelerator and network to avoid these sources of asynchrony while supporting traditional applications. *SYS* implements atomic broadcast in hardware and provides a set of synchronization primitives to applications which are implemented on top of synchronous hardware and a synchronous network. Since the synchronization primitives provided by *SYS* are implemented within a completely synchronous environment, assumptions about the nature of communications and failures are simplified. As a result, *SYS* can utilize more efficient protocols which make assumptions that cannot be made in an asynchronous system. For example, using a synchronous system overcomes the FLP impossibility result [5]. We highlight the advantages and features of *SYS* over traditional asynchronous distributed systems below:

- *SYS* is implemented on top of synchronous logic and network, enabling protocols which make strong assumptions about liveliness and failures in a system.
- *SYS* accelerates communications by implementing the network and atomic broadcast in hardware, eliminating various software overheads traditionally associated with the network stack.
- *SYS* operates without the intervention of the processor, which immunizes *SYS* from application crashes. In fact, *SYS* continues to operate if the host processor resets.

- *SYS* provides familiar synchronization primitives to userspace applications, such as locks, counters, semaphores and registers.
- *SYS* is simple, power-efficient and inexpensive: *SYS* only requires standard Category 6 Ethernet cable. Our prototype switch on a Xilinx Zynq ARM FPGA uses merely $2W$ under load. Likewise, our *SYS* NIC prototype uses only 2W under load, including the processor. Both designs take less than 10% of the area of the FPGA.

We evaluate the performance of *SYS* by comparing the performance of *SYS* to the performance of other atomic broadcast protocols, such as Paxos [7] and Apache Zookeeper's *zab* [6].

2 System Overview

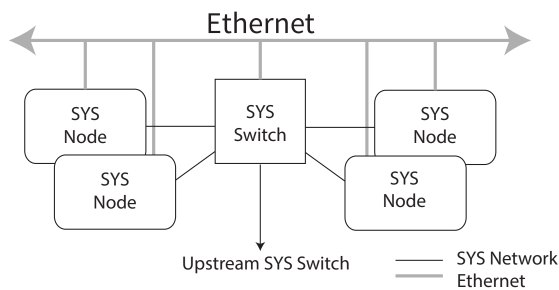

Figure 1: *The SYS architecture.* The *SYS* network coexists with traditional Ethernet. Each *SYS* node is connected to a *SYS* switch, which acts as an arbiter to ensure that updates are linearizable.

The overall architecture of *SYS* is shown in figure 1. *SYS* consists of *SYS* nodes, which applications run on, and *SYS* switches. *SYS* nodes are connected to *SYS* switches via a point-to-point synchronous serial link. *SYS* switches can be daisy-chained to create larger *SYS* networks. Each *SYS* node and switch is also connected to traditional gigabit Ethernet for normal data and management traffic.

Synchronous hardware in the *SYS* nodes and switches enable nodes to atomically broadcast of a small amount of state, which are exposed to the system as 64 *SYS* registers. *SYS* registers are 64-bits wide, and reads and writes to *SYS* registers are linearizable. Registers are used to construct *SYS* primitives, which have specialized semantics. For example, the *counter* primitive is a register which can be either read from or incremented.

The next sections describe the *SYS* system in further detail. Section 2.1 describes the interface *SYS* exposes to applications, and section 2.3 describes the *SYS* network and components in detail.

2.1 Application Interface

SYS exposes a set of primitives, which are built on top of *SYS* registers to applications. This list is by no means exhaustive - it represents the simple primitives we have implemented thus far. Operations on *SYS* primitives are *atomically broadcasted*. *SYS* ensures that operations on *SYS* primitives are linearizable and visible to all *SYS* nodes simultaneously. We describe the primitives and give examples of the applications they enable below.

2.1.1 Registers

Registers are the most basic *SYS* primitive. They implement a fully linearizable distributed shared register that supports `read` and `write` operations. Applications can use *SYS* registers to consistently share a small amount of shared state. For example, the value of the *SYS* register can represent the currently elected leader in a leader-election algorithm.

2.1.2 Counter

Counters support two operations: `increment` and `read`. Counters are an ideal match for applications which require a sequence or total ordering. As an example, a counter can server as a percolator [8] timestamp or as a CORFU [1] sequence number.

2.1.3 AckCounter

The AckCounter is a special purpose counter that tracks acknowledgments. AckCounters support two operations: `acknowledge` and `read`. When all nodes issue an `acknowledge` operation, the AckCounter increments by one. The AckCounter is a perfect counterpart for applications which use the counter. For example, with percolator, the AckCounter can be used to identify the latest transaction seen by all nodes.

2.1.4 Semaphore

The Semaphore acts as a traditional semaphore synchronization primitive. *SYS* sempahores support four operations: `acquire`, `release`, `set` and `read`. Semaphores can be used to coordinate access to a shared or limited resource. For example, a distributed application can use a semaphore to restrict how many requests external clients can make simultaneously.

2.1.5 LockBit

The lockbit acts as a traditional distributed lock which exposes the lock owner. *SYS* lockbits support three operations: `lock`, `unlock` and `read`. Each bit in the exposed register refers a node, and a bit set to "1" indicates that node holds the lock. Lockbits can be used when communicating the owner of a lock is important. For example, a lock bit can be used with distributed storage to indicate which owner holds the most recent updates to a partition.

2.2 *SYS* protocol

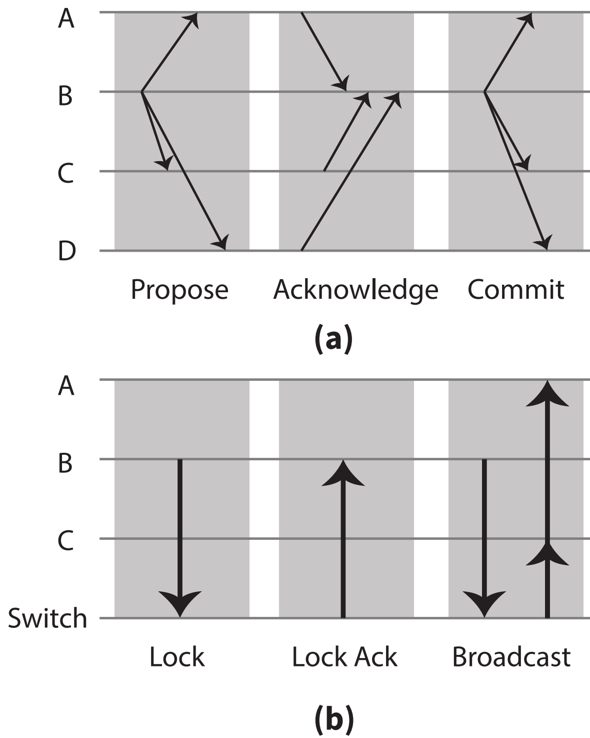

Figure 2: *Asynchronous atomic broadcast protocols and the SYS protocol.* This figure highlights the differences between the steps of a traditional atomic broadcast protocol (a), and the *SYS* protocol (b), which takes advantage of the synchronous atomic broadcast support in *SYS*. In this figure, client B is attempting to atomically broadcast an update to all other nodes.

The *SYS* protocol implements atomic broadcast over the synchronous hardware network provided by the *SYS* switches and nodes. In order to perform an atomic broadcast, nodes first request a *lock* from the switch. The switch arbitrates between nodes and grants locks to exactly one node at a time by sending a *lock ack* as a response. Once a node has been granted the lock, it can broadcast one message, which is then sent to all nodes in the *SYS* network, and each node sends an acknowledgment that it has updated the local copies of its registers. If any node does not send back an acknowledgment, it is considered in error and removed from the system.

Each communication is synchronous and has a well defined time: a lock request requires 3 clock cycles to send, an acknowledge requires 3 clock cycles to receive, and a broadcast of a 64-bit register update requires 72 cycles to send, 72 cycles to receive (3 cycles for the operation code, 5 for the register number, and 64 cycles for data), and 3 cycles to acknowledge the broadcast. Supported *SYS* primitives can reduce the amount of data transmitted (for example, incrementing a counter requires only transmitting an increment operation instead of the entire register).

Figure 2 compares the *SYS* protocol with the typical phases of an atomic broadcast protocol such as *zab* [6] or *Paxos* [7]. The asynchronous atomic broadcast protocol is actually composed of two broadcasts (propose, commit) that are required to make the otherwise asynchronous broadcast appear atomic. In *SYS*, the atomic broadcast is actually broadcast atomically, so only a single broadcast is required.

Unlike asynchronous atomic broadcast protocols, message transmission, reception and processing take a bounded amount of time in *SYS*. Messages that take too long to process can be considered failures in *SYS*, whereas in an asynchronous system it may only be indicative of a transient failure condition. Figure 3 depicts a typical failure condition in an asynchronous system, which requires reconfiguration and a retry, whereas the *SYS* network can continue operating despite the failed node. In order to rejoin the system, the node must load the up-to-date network state, as described in section 2.3.2.

2.3 *SYS* components

As we have previously described, *SYS* is composed of *SYS* nodes, *SYS* switches, and the point-to-point network which connects *SYS* switches and nodes together. We implement *SYS* nodes and switches on low-cost Xilinx Zynq ARM FPGAs. We describe each component below:

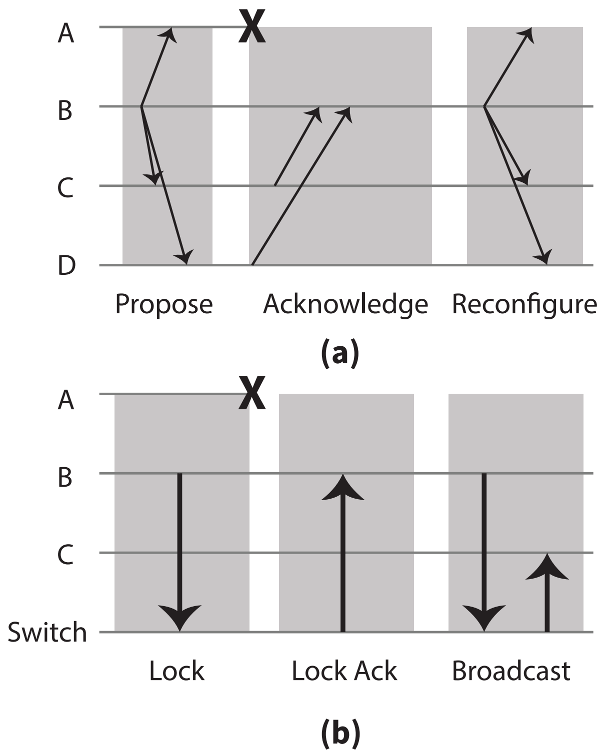

Figure 3: *Failures in asynchronous atomic broadcast protocols and the SYS protocol.* A failure in node A occurs after some amount of time, depicted by the X. An asynchronous protocol cannot be sure if there is a failure, and many protocols wait a certain amount of time for an acknowledgment to be received, and must reconfigure and retry. *SYS* however, detects a failure immediately and removes that node from the system.

2.3.1 Network

The physical *SYS* network is connected via category 6 Ethernet cable. The cable carries four differential signals, *receive*, *transmit*, *clock* and *error*. The network and clock operate at 125MHz, giving the network a theoretical maximum bandwidth of 125 Mb/s.

2.3.2 Peripheral

SYS nodes communicate over the *SYS* network through the *SYS* peripheral, which is a memory-mapped device attached to the processor bus. The *SYS* registers and primitives are exposed via a read-only *register file*. To request an operation on a *SYS* primitive, applications write requests to the *request handler*, which dispatches requests to the *tx engine* and waits for the request to complete. (Figure 4). In order to atomically broadcast requests, the tx engine first requests a *lock* on the network, then after the *rx engine* has received an acknowledgment for the lock, broadcasts the data over the network. The *rx engine* automatically updates the register file based on requests that are broadcast over the network, without intervention from the processor or tx engine. If the network fails for any reason, the rx engine raises an error signal which prevents the processor from sending requests or reading the register file. In order to rejoin the system, the peripheral must reload the network state by requesting a lock and reading up-to-date values of the *SYS* registers.

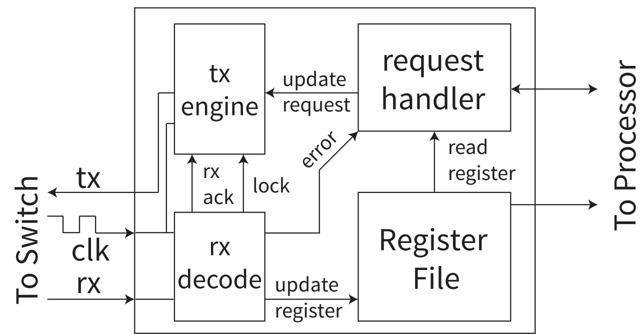

Figure 4: *The SYS peripheral.* The *SYS* peripheral synchronously processes updates from the *SYS* network to the register file. The processor can read the register file and send requests to the request handler.

2.3.3 Switch

The switch is the at the heart of the *SYS* system, and a schematic overview can be found in figure 5. The primary function of the switch is to arbitrate requests on the network so that only one node can broadcast at any given time. The *switch arbiter* decides which port acquires the lock and ensures fairness by granting the lock randomly in the case of a tie. In addition to the arbitration logic, the switch arbiter also implements the *SYS* primitive operations. For example, when a *SYS* counter increment operation is requested, the switch arbiter increments the value of that register by one and broadcasts it to all other nodes.

Switches can be daisy-chained together. When there are multiple switches in a *SYS* network, one *SYS* switch is declared the *top-level switch*, and handles *SYS* primitives. The other switches arbitrate and forward operations to the top-level switch. When switches fail, the system can safely fall back to asynchronous operation using traditional asynchronous atomic broadcast. This

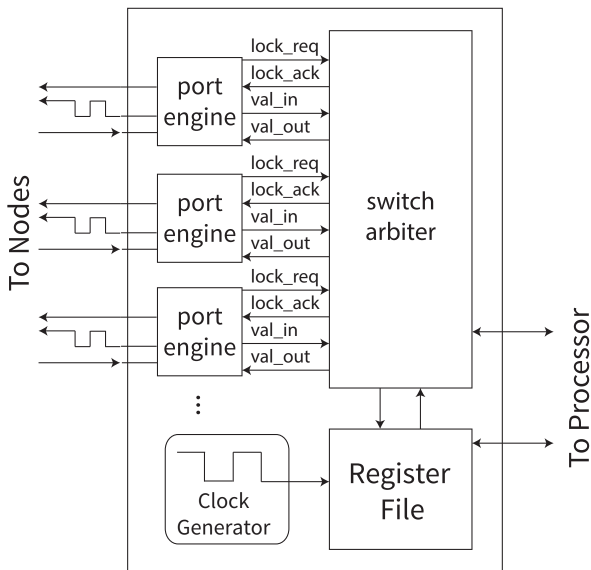

Figure 5: *The SYS switch.* The *SYS* switch decides which nodes are granted locks and keeps its own copy of the *SYS* registers which nodes can request to load the most recent state of the *SYS* registers.

is achieved by clients using a traditional leader-election protocol. Since all the clients have access to the most recent global state during a failure, this state is used to "seed" the asynchronous protocol.

3 Related Work

Synchronous networks have been proposed in the past, such as IEEE 1588 synchronous Ethernet [4]. Atomic broadcast on these networks, however, is still implemented using software-based protocols, which can take a variable amount of time since it is run on a processor. By implementing and processing the atomic broadcast protocol in hardware, *SYS* can rely on operations and requests to take a fixed amount of time.

Distributed real-time operating systems (DRTOSes) have also been proposed in the past [9, 10], which add real-time guarantees to distributed applications. However, re-writing distributed applications to work within the time-constrained environment of a RTOS can prove difficult, especially as a growing number of distributed applications are written in higher-level languages such as JAVA. By exposing primitives which distributed applications can use in software, *SYS* avoids the need to place explicit timing constraints on application code.

Spanner [3] was an effort that used GPS clocks to achieve global consistency. While Spanner can use a synchronized notion of time, it still relies on asynchronous general-purpose processors to process requests, and therefore has weaker guarantees than *SYS* would provide.

4 Results

Our current prototype system with 3 nodes is capable of achieving 700k register update operations per second.

We expect that this number would remain constant, even with additional nodes, and could be increased substantially by running the network at a higher frequency. In contrast, our test Apache ZooKeeper installation can support at maximum 31k operations per second with 3 servers, and the number of operations scaled negatively with each additional servers. We also tested a fast Paxos implementation, and found it could not support more than 2k operations per second with 3 servers, which also scaled negatively with each additional replica.

5 Discussion

The use of hardware to relieve distributed systems from the problems of asynchrony is far from mature, and we do not argue that our system is optimal or even the correct design. The goal of our work is to generate discussion on the use of hardware to deal with asynchrony, and our preliminary results suggest that this is a worthwhile endeavor. To that end, we answer several commonly asked questions about our system in the next sections.

5.1 Does the *SYS* switch form a bottleneck?

While the *SYS* switch (or the top-level *SYS* switch, in a daisy-chained configuration) forms a bottleneck, the overall capacity of the switch is over 20× what traditional solutions can achieve today. It is unlikely that applications will produce enough work to load the switch, and if they did, it may be possible to virtualize the switch at very little cost (so that each application gets its own "VLAN"), or to increase the frequency of the network so it can sustain the workload.

5.2 Is the *SYS* switch a single point of failure?

It is true that the *SYS* switch is a single point of failure (i.e., if it were to fail, the entire synchronous network would not be of any use, since no ports would be granted a lock). However, we contend that such a failure would be analogous to a top-of-rack switch failing, and it would be possible to failover to a second, redundant network. Furthermore, the system could safely failover

to asynchronous operation.

5.3 What about hardware errors?

While hardware errors are rare, especially when compared to software errors, *SYS* protects itself from hardware errors by use of the synchronous network (i.e., if a link fails to respond in a given time frame, the corresponding node is removed from the *SYS* network, and both the node and switch are notified of that failure). This prevents hardware errors from affecting the liveliness of the system. Under most normal operating conditions, this type of failure never occurs unless there is an issue with the link (for example, the cable is cut). In addition, the use of error correction on the links and on the *SYS* registers could be used to further protect the network.

5.4 Does *SYS* only work for local area networks?

Indeed, building a point-to-point synchronous link between two geodistant datacenters would be an expensive undertaking, and the latency between those links would be high (>1 ms). *SYS*, as we have designed, is only relevant for intra-datacenter communications. However, after the writing of this paper, we realized that a synchronous network provided more guarantees than *SYS* requires to operate. We have found that *SYS* only requires a network that provides bounded-time delivery and failure-detection, which many asynchronous point-to-point links provide. Therefore, the *key contribution of SYS is not the synchronous network, but the synchronous hardware* that enables the *SYS* protocol.

6 Conclusion

Distributed systems have been built under the assumption that asynchrony is an unavoidable reality. *SYS* challenges that assumption by providing a synchronous fabric which enables reliable atomic broadcast for a small amount of state. We show that a small of state is sufficient for building flexible synchronization primitives which are familiar to distributed application developers. *SYS* can be built with inexpensive hardware with very low power consumption, using standard category 6 cable. Our preliminary results indicate that *SYS* performs at least an order of magnitude better than traditional atomic broadcast protocols. We believe that this area is ripe for further exploration and is particularly relevant in a world where large companies are considering adding hardware accelerators to their datacenters. We hope to address the issues of scalability in a future work.

7 Acknowledgements

We would like to thank Mahesh Balakrishnan, Marcos Aguilera, Dahlia Malki, Ted Wobber and John Davis for their input on this work. We would also like to thank Xilinx for providing a donation of Zynq FPGA boards. This material is based upon work supported by the National Science Foundation under Grant No. DGE-1144086.

References

[1] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: A shared log design for flash clusters. *Proc. 9th USENIX NSDI, San Jose, CA*, 2012.

[2] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[3] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally-distributed database. In *Proceedings of OSDI*, volume 1, 2012.

[4] J. C. Eidson. *Measurement, control, and communication using IEEE 1588*. Springer Publishing Company, Incorporated, 2010.

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

[6] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[7] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[8] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, volume 10, pages 1–15, 2010.

[9] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, et al. Overview of the chorus distributed operating systems. In *Computing Systems*. Citeseer, 1991.

[10] P. Veríssimo, A. Casimiro, and C. Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 533–542, New York City, USA, June 2000. IEEE Computer Society Press.