Accurate Timeout Detection Despite Arbitrary Processing Delays

Sixiang Ma and Yang Wang, *The Ohio State University*

https://www.usenix.org/conference/atc18/presentation/ma-sixiang

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Accurate Timeout Detection Despite Arbitrary Processing Delays

Sixiang Ma
The Ohio State University

Yang Wang
The Ohio State University

Abstract

Timeout is widely used for failure detection. This paper proposes SafeTimer, a mechanism to enhance existing timeout detection protocols to tolerate long delays in the OS and the application: at the heartbeat receiver, SafeTimer checks whether there are any pending heartbeats before reporting a failure; at the heartbeat sender, SafeTimer blocks the sender if it cannot send out heartbeats in time. We have proved that SafeTimer can prevent false failure report despite arbitrary delays in the OS and the application. This property allows existing protocols to relax their timing assumptions and use a shorter timeout interval for faster failure detection. Our evaluation shows that the overhead of SafeTimer is small and applying SafeTimer to existing systems is easy.

1 Introduction

This paper presents SafeTimer, a mechanism to enhance existing timeout detection protocols to prevent false failure reports caused by long delays in the OS and the application. With the help of SafeTimer, existing protocols can relax their timing assumptions and thus use a shorter timeout interval for faster failure detection.

Timeout is widely used in distributed systems to detect failures [1, 6, 13, 24, 29, 45]: a node periodically sends a heartbeat packet to others and if the receiver does not receive the heartbeat in time, it may report a failure and may take actions to recover the failure.

Although this idea is simple, delays of packet transfer create a problem: if a receiver misses a heartbeat, is it because the sender has not sent the heartbeat, which indicates a failure, or is it because the heartbeat is delayed somewhere, which should not indicate a failure?

To address this problem, existing systems use one of the following approaches: the first is to prevent false failure reports by setting an appropriate timeout interval. However, such setting requires certain timing assumptions about the communication channel [4, 5, 18] and creates a dilemma: on one hand, these assumptions should be conservative enough to tolerate abnormal events that can cause long delays (e.g., congestion), which means the timeout interval should be long. On the other hand, long timeout interval can hurt system availability, because the system has to wait for a long time before recovering the failure. A recent study shows that inappropriate timeout interval is a major cause of timeout related bugs, leading to various problems like data loss or system hanging [19]. The second approach is to ensure correctness despite false failure reports, using protocols like Paxos [34, 35, 42]. This approach allows short timeout for better availability, but its cost is usually higher.

SafeTimer enhances the first approach to tolerate a subset of those abnormal events, without requiring any timing assumptions. It thus allows existing protocols to relax their timing assumptions to use a shorter timeout interval, without sacrificing the accuracy of timeout detection. It is motivated by two insights.

First, conservative assumptions are only necessary if the communication channel is a blackbox, which cannot provide any additional information other than receiving a packet. If the channel can tell whether a packet is pending or dropped, the receiver can simply check whether there is a pending or dropped heartbeat when missing a heartbeat. This approach can prevent false failure reports without requiring any timing assumptions.

Second, we observe that modeling the whole communication channel as a blackbox is too pessimistic: the routing layer usually does not provide the users with information like packet drops, so it is reasonable to model routing as a blackbox; the OS and the application, however, can provide precise information about its packet processing and thus could be modeled as a whitebox. Furthermore, in today's datacenters, the whitebox part often incurs delays that are comparable to or even larger than those of the blackbox part: on one hand, intra-datacenter networking delays usually range from tens of

microseconds to a few milliseconds and can be further reduced to hundreds of nanoseconds with techniques like Infiniband [31]. Improvement in bandwidth and protocols [14, 46] have significantly reduced the chances of packet drops. On the other hand, a traditional OS can delay processing by several milliseconds because of time sharing or page fault, etc. Such delay can occasionally grow to several seconds for reasons like SSD garbage collection [32] and can grow even higher in abnormal cases (see Section 2).

Because of these two insights—1) the delay of the whitebox part is significant among communication and 2) there exist more effective solutions for the whitebox part—SafeTimer naturally uses a more effective solution for the whitebox part; for the blackbox part, SafeTimer relies on existing protocols and their assumptions.

At the receiver side, SafeTimer guarantees that *as long as the network interface card (NIC) has either delivered or dropped the heartbeat before the deadline, the receiver will not report a failure.* To achieve this property, SafeTimer's receiver module checks whether there are any pending or dropped heartbeats in the system before reporting a failure. Implementing this idea, however, is challenging, because modern OS incorporates a highly concurrent pipeline for fast packet processing. Naive solutions like pausing all its threads requires an intrusive modification to kernel, which is undesirable.

To solve this problem, we propose a non-blocking solution: when the timer expires at t, SafeTimer's receiver module will send a barrier packet to itself. By crafting the barrier packet and configuring the OS properly, SafeTimer ensures that if the receiver module receives the barrier, all heartbeats processed by the NIC before t must have been either delivered to the application or dropped. Therefore, if the receiver module has neither received the heartbeat nor observed any packet drops, it can safely report a failure.

At the sender side, SafeTimer guarantees that *if the sender has not sent out a heartbeat in time, the sender will not be able to send out any new packets.* Such suicide idea is not novel [8, 22], but previous solutions that actively kill or reboot the sender do not work when considering long processing delays, because the kill or reboot operations may be delayed as well, leaving the sender alive. To solve this problem, SafeTimer incorporates a passive design: SafeTimer's sender module maintains a timestamp to identify till when it is valid for the sender to send new packets. The sender module updates this timestamp when successfully sending a heartbeat and checks this timestamp before sending any packets. By doing so, SafeTimer prevents a sender which fails to send heartbeat in time to affect other nodes in the system.

One can enhance an existing timeout detection protocol by applying SafeTimer at both the sender and the receiver. We can prove that, as long as the existing protocol's assumptions about the blackbox part hold, SafeTimer is accurate (i.e., never report failure for a correct sender) despite arbitrary delays in the whitebox part and is complete (i.e., eventually report failure for a failed sender) when the receiver does not experience slow processing or packet drops for sufficiently long [12]. Such properties indicate that one does not need to make conservative assumptions about the whitebox part, and thus can use a shorter timeout interval to improve availability.

Our evaluation shows that the overhead of SafeTimer is negligible when processing big packets and at most 2.7% when processing small packets; SafeTimer can prevent false failure reports when long processing delays are injected; and applying SafeTimer to HDFS [27, 45] and Ceph [9] is easy.

2 Motivation

2.1 Long delays in OS and application

SafeTimer allows existing timeout detection protocols to relax their timing assumptions by excluding delays in the OS and the application. To demonstrate the potential benefits of such relaxation, we present a number of abnormal events that can cause long delays.

- **Disk access.** Disk accesses caused by logging heartbeats [29, 45] or page faults can block heartbeat processing. A typical hard drive has an average latency of tens of milliseconds and an SSD usually has a lower average latency. Worst-case latency, however, is much longer: SSD's internal garbage collection can delay an access by more than one second [32]. Our experiment with hard drives shows that when processing frequent random writes, the buffering mechanism in the file system can occasionally introduce a latency of tens of seconds, when it flushes many random writes.
- **Packet processing.** OS kernel can drop packets at different layers when it runs out of buffer space, which can cause extra delay. Furthermore, handling of abnormal packets may cause a significant delay as well. For example, when Linux receives a packet to an unopened port, it will report "port unreachable" to the router using ICMP [30]. In our experiment, a large number of such abnormal packets can delay the processing of heartbeat by more than two seconds.
- **JVM garbage collection.** Garbage collection in a Java Virtual Machine (JVM) can block the execution of the application. Our experiment on a JVM with 32GB of memory shows that when the memory is close to be fully utilized, a single garbage collection can take up to 26 seconds, even when using parallel GC. A recent survey [19] has observed similar problems in ZooKeeper and HBase (HBase-3273 [26]).

- **Applicaton specific delays.** Applications may have specific logics that can cause long delays occasionally. For example, previous works have reported that HDFS DataNode's heartbeat sending thread may be blocked by the task of scanning local data, which could take long [48]. Although newer versions of HDFS have fixed this problem, our investigation shows that similar problems still exist: the heartbeat sending thread can also be blocked by the task of deleting directories, which can take long as well. A similar problem has been reported in Ceph, in which a heavy rejoin operation can block heartbeat processing [11].

As shown in these examples, some events in the OS and the application can cause delays of tens of seconds, which are comparable to or larger than many systems' default timeout intervals (e.g., 30 seconds in HDFS [28], 5 seconds in ZooKeeper [25], 20 seconds in Ceph [10]). Furthermore, some of these delays may grow longer if a machine has more resource (e.g., more memory for JVM garbage collection).

Existing timeout detection protocols must make their timing assumptions conservative enough to cover all the events mentioned above. For example, to tolerate long garbage collection in ZooKeeper [26], the developers increased their timeout intervals, which will hurt system availability as discussed previously. With the help of SafeTimer, however, they can tolerate these events without requiring any timing assumptions, and thus can use a shorter timeout for faster failure detection.

2.2 Can we provide timing guarantees?

The above problems would be trivial if the OS and the application can provide hard real-time guarantees for heartbeat processing, but during our failed attempts, we find this is a challenging task on commodity OS.

Isolated resource for heartbeats. To prevent other tasks from interfering with heartbeat processing, the application can reserve resources (e.g., a socket) for heartbeat processing. However, this approach cannot prevent such interference in OS kernel. For example, packets from different sockets can be handled by the same thread or CPU core in the kernel; even if heartbeat handling does not need to make disk I/Os, page fault in the kernel may incur a disk I/O, blocking heartbeat processing.

Processing heartbeats at lower layers. To avoid delays in the OS kernel, one can implement heartbeat sending and checking at lower layers, as close as possible to the NIC. This approach can avoid many types of delays, but cannot eliminate them, because heartbeat checking can only happen after the OS reads a packet, which means delays in handling interrupts and reading packets can still cause false failure reports.

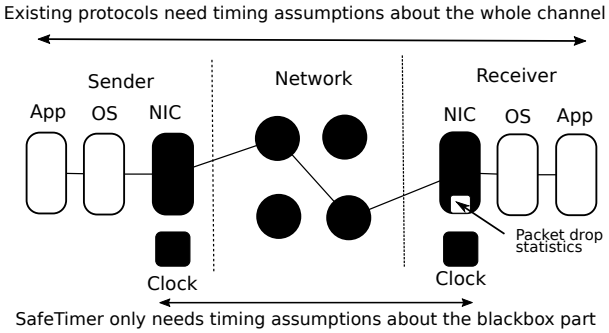

Figure 1: System model: SafeTimer can tolerate long delays in the whitebox part without timing assumptions.

Real-time OS. Real-time Linux [43] and other real-time frameworks for Linux such as RTAI [44] and Xenomai [49] can give higher priority to certain interrupts, so that they wouldn't be delayed by other interrupts. However, this approach can only guarantee an interrupt handler is triggered in time, but cannot guarantee when the OS can finish reading a packet. The latter requires us to analyze the worst-case execution time of handling interrupts and reading packets, which is a challenging task on complicated kernel code with frequent synchronizations.

We find these approaches, even combined, cannot achieve hard timing guarantees for heartbeat processing. The fundamental problem is that commodity OSes are designed with the principles of resource sharing and high concurrency, which is against the goal of strict timing guarantees. Therefore, finally we give up the attempts to provide timing guarantees. Instead, we investigate whether we can prevent false failure reports **assuming delays in the OS and the application can be arbitrary**.

3 Model

The goal of SafeTimer is to enhance existing timeout detection protocols to tolerate long processing delays in the OS and the application. To achieve this goal, SafeTimer makes a few assumptions about the existing protocol: at the receiver side, SafeTimer assumes the receiver defines multiple time intervals and reports a failure if it does not receive any heartbeats during an interval. At the sender side, SafeTimer assumes the application has its own rules to decide when to send heartbeats and whether heartbeats are sent successfully, based on its timing assumptions. Furthermore, SafeTimer assumes these intervals and assumptions are configurable, so that the user can use a shorter timeout interval with the help of SafeTimer.

SafeTimer enhances existing protocols to tolerate a subset of abnormal events without requiring timing as-

sumptions. Figure 1 shows which events SafeTimer can tolerate: the blackbox part includes the network interface cards (NICs) at both sides, the clocks at both sides, and packet routing between two NICs; the whitebox part includes the OS and the application's logic to process packets at both sides. SafeTimer can tolerate long delays in the whitebox part without requiring any timing assumptions. Instead, SafeTimer only assumes that, a node will eventually finish processing a heartbeat and SafeTimer can observe the result (either delivered or dropped). For the blackbox part, SafeTimer relies on existing protocols and their assumptions.

Abnormal events in the whitebox part may affect the processing speed of the blackbox part. SafeTimer assumes such effect can be observed at the boundary: a slow receiver may cause its NIC to drop packets because the receiver's buffer is full and SafeTimer assumes the NICs can provide packet drop statistics. We find this function is commonly provided by modern NICs.

With the help of SafeTimer, existing timeout detection protocols only need to make conservative assumptions about the blackbox part, which means the protocol can use a shorter timeout interval to accelerate failure detection. Note that SafeTimer cannot make concrete suggestions about timeout interval: the user still has to estimate possible delays in the blackbox part. However, considering the various kinds of abnormal events in the whitebox part (Section 2), SafeTimer should be able to reduce timeout interval by at least tens of seconds.

Case studies. We present a few existing timeout detection protocols to show how SafeTimer models them and how they can benefit from SafeTimer.

Budhiraja et al. [5] discuss how to detect failures in primary-backup protocols, given different models. In the simplest model, which assumes clocks are sufficiently synchronized, links are reliable, and packet delay is bounded (δ), the sender can send heartbeats every τ seconds and the receiver reports a failure if it does not receive a heartbeat for $\delta+\tau$ seconds. SafeTimer can model this protocol in the following way: when the receiver receives a heartbeat at t, it creates a new interval from t to $t+\delta+\tau$ and checks whether it receives a heartbeat by the end of the new interval; the sender can define a successful heartbeat sending for interval i as sending a heartbeat at t_i and $t_i \leq t_{i-1}+\tau$. With the help of SafeTimer, this protocol may reduce δ because it does not need to include the delays of the whitebox part. This work also discusses more complicated models, which consider link failures and proposes a gossip protocol to route heartbeats through multiple links, which is adopted in Ceph. SafeTimer can model it accordingly. For example, to tolerate one link failure, the sender can define a successful heartbeat sending as sending two heartbeats to two nodes by $t_{i-1}+\tau$. Similarly, SafeTimer may help to reduce δ.

```
1  /* The application calls safetimer_check when
       missing heartbeats from start_i to end_i */
2  function safetimer_check(start_i)
3      send a barrier to itself
4      wait for barrier (with a timeout)
5      if barrier received and t_lastHeartbeat < start_i
6          read drop count in OS and NIC and reset to 0
7          if (drop count = 0 and t_drop < start_i)
8              return TRUE_FAILURE
9          else if (drop count != 0)
10             t_drop = current_time()
11         end
12     end
13     return FALSE_FAILURE

15 function safetimer_recv_thread()
16     when receiving heartbeat
17         t_lastHeartbeat = current_time()
18     when receiving barrier
19         notify safetimer_check
```

Figure 2: Pseudo code of SafeTimer's receiver module. For simplicity, it assumes there is only one sender, but it can easily be extended to support multiple senders. $t_{lastHeartbeat}$ records the timestamp of the last heartbeat. t_{drop} records the timestamp of the last drop event.

In HDFS, a DataNode sends a heartbeat to the NameNode every three seconds, and the NameNode marks the DataNode as stale if it misses heartbeats for 30 seconds. In the common case, the NameNode will acknowledge a heartbeat to the DataNode; if the DataNode detects errors, it will send heartbeats more aggressively every second. SafeTimer can model it in the following way: when the receiver receives a heartbeat at t, it creates a new interval from t to $t+30$ and checks whether it receives a heartbeat by the end of the new interval (note intervals can overlap in this case); the sender can define a successful heartbeat sending for interval i as 1) getting acknowledgement for one heartbeat or 2) sending heartbeats with an interval of less than one second. SafeTimer may help to reduce the 30-second interval because it does not need to consider delays in the whitebox part.

4 Design

SafeTimer enhances existing timeout detection protocols to tolerate long processing delays in the whitebox part. In this section, we first present SafeTimer's mechanisms and then prove its accuracy and completeness.

4.1 Accurate timeout at the receiver

As discussed in Section 3, SafeTimer assumes the application's heartbeat receiver defines multiple time intervals (interval i from $start_i$ to end_i), and reports a failure if no heartbeat is received during an interval.

SafeTimer guarantees that as long as the receiver's NIC has processed (either delivered or dropped) a heart-

beat during interval i, SafeTimer's receiver module will not report a failure for interval i.

Its key idea is simple: if the receiver module does not receive any heartbeats by the end of an interval, it will check whether there are any pending or dropped heartbeats in its whitebox part, and if not, the receiver module can safely report a failure.

The key challenge, however, is how to implement this idea in modern OS. For fast packet processing, modern OS incorporates a highly concurrent design, which involves a pipeline with multiple threads in each stage. To identify whether some heartbeats are pending, a naive solution is to pause all threads and check all buffers, but this solution will have negative impact on performance and require intrusive modification to the kernel.

To solve this problem, SafeTimer incorporates a non-blocking design as shown in Figure 2: if the application does not receive any heartbeat by end_i, it will check whether any heartbeats are pending or dropped by calling *safetimer_check*, which sends a barrier packet to itself (line 3). By crafting the barrier packet and configuring the system properly, SafeTimer ensures that a barrier will follow the same execution path of heartbeats. Therefore, if the receiver module receives the barrier, it can know that any heartbeats processed by the NIC before end_i must have been processed by the OS and SafeTimer as well, either delivered to the receiver module or dropped. We will present details about how to implement the barrier mechanism in Section 5. For now, the readers can simply assume SafeTimer somehow drives the heartbeats and the barriers into a FIFO channel.

If the receiver module receives the barrier, it will check again whether it has received a heartbeat ($t_{lastHeartbeat} < start_i$ in line 5). If not, the receiver module will read drop statistics from both the OS and the NIC: if $dropcount = 0$ and $t_{drop} < start_i$ (line 7), which means there are no drops in interval i, the receiver module can safely report a failure. If the barrier is dropped as well, the receiver module will not report a failure for interval i. In this case, the application will perform the same check in the following intervals and will eventually report a failure.

4.2 Stop sender when missing heartbeat

As discussed in Section 3, SafeTimer assumes that the application has rules to decide when to send heartbeats and whether they are sent successfully. In particular, without losing generality, SafeTimer assumes for each interval i, the application defines a deadline end'_i to send heartbeats, which should be earlier than end_i at the receiver side because of clock drift and network latency.

SafeTimer guarantees that if a sender cannot successfully send heartbeats by end'_i, the sender will not be able to send out any other packets after end'_i, because the receiver may report a failure at that time. This is necessary because the accuracy property requires that if the receiver reports a failure, the sender must have failed: violating this property can cause correctness issues. Taking the primary backup protocol as an example, a backup should only become active if the primary fails. If a backup receives a failure report and becomes active while the primary is still active, there will be two active nodes, creating a classic "split brain" problem [20].

```
1   function safetimer_send_heartbeat(end'_i, end'_{i+1})
2       send heartbeats
3       if sending succeeded before end'_i
4           t_valid = end'_{i+1}
5       end

7   function safetimer_intercept_sending()
8       if (current_time() > t_valid)
9           drop the packet
10      else
11          perform the send
12      end
```

Figure 3: Pseudo code of SafeTimer sender module. The application defines end'_i as the deadline to send heartbeats for interval i; the application defines whether sending succeeds; SafeTimer maintains a timestamp t_{valid} to identify till when it is safe to send out packets.

Killing a sender when it is slow is not a new idea [8, 22], but how to implement it correctly despite arbitrary processing delays requires careful thought. Existing solutions ask a specific component (e.g., a watchdog [22]) to actively kill the sender. When considering arbitrary processing delays, however, such active solution is incomplete, because the delay of processing the "kill" command may allow the sender to be alive for an arbitrary amount of time, violating the accuracy property.

SafeTimer uses a passive solution by utilizing the idea of output commit [41]: a slow sender may continue processing, but as long as other nodes do not observe the effects of such processing, the slow sender is indistinguishable from a failed sender. As shown in Figure 3 (lines 3-12), SafeTimer's sender module maintains a timestamp t_{valid}, which indicates it is safe for the sender to send packets before t_{valid}. During startup, the sender sets t_{valid} to end'_0. If the sender successfully sends heartbeats for interval i, the sender extends t_{valid} to end'_{i+1} (line 4). Whenever the sender is about to send a packet, SafeTimer will compare the current time with t_{valid}: if current time is larger than t_{valid}, the sender will discard the packet (lines 7-12). Since heartbeat is blocked as well in this case, an invalid sender cannot extend t_{valid} and send packets in the future, unless with recovery operations.

Note that since the sending operation itself may take arbitrarily long, SafeTimer allows a packet generated before t_{valid} to be actually sent out after t_{valid}. This is fine because the packet is generated when the sender is still valid (i.e., when the receiver has not reported the failure).

4.3 Proof of accuracy and completeness

As discussed in Section 3, SafeTimer relies on the existing protocol to send and receive heartbeats in the blackbox part. When the existing protocol's assumptions about the blackbox part hold, we can prove that SafeTimer is accurate (i.e., never report failure for a correct node) despite arbitrary delays in the whitebox part and is complete (i.e., eventually report failure for a failed node) when the receiver does not experience slow processing or packet drops for sufficiently long. We provide the detailed proof in the appendix.

4.4 Benefit of SafeTimer

Because of the accuracy and completeness properties, the users of SafeTimer do not need to make conservative timing assumptions about the whitebox part. They do need to provide a reasonable estimation of such delay in the common case, because the sender needs some time to send out heartbeats. However, this requirement is only for performance: if the actual delay is longer than estimation, which means the sender cannot send the heartbeat in time, SafeTimer will block the sender, which may cause unnecessary recovery and hurt performance, but this will not violate accuracy. Therefore, SafeTimer only requires the user to provide a reasonable estimation to make sure such events are *rare*. As a comparison, in existing protocols, if the actual delay is longer than estimation, system correctness can be violated, and that is why existing systems require conservative assumptions so that such events *never* happen. The gap between "rare" and "never" is where SafeTimer gains its benefit.

5 Implementation

This section presents the barrier mechanism at the receiver and the packet checking at the sender in detail.

5.1 Barrier mechanism at the receiver

The goal of the barrier mechanism is to ensure that if SafeTimer's receiver module sent a barrier to itself at t and received it later, then all heartbeats delivered by NIC before t must have been either delivered to the application or dropped. Achieving this property would be trivial if the OS processes all packets in FIFO order, but unfortunately, this is not true in modern OS. To illustrate the problem and motivate our design, we first present how Linux processes incoming packets.

Background. As shown in Figure 4, Linux incorporates a multi-stage pipeline to process incoming packets.

At the lowest level, an NIC buffers incoming packets in its RX queues and tries to transfer them to kernel's ring buffers: if the ring buffer has empty slots, the NIC will transfer the packet using DMA and fire an interrupt; if the buffer is full, the NIC will retry and may drop packets. For efficiency, modern NIC and Linux incorporate the Receive Side Scaling (RSS) technique [40] to allow parallel packet processing: the NIC creates multiple RX queues and the kernel creates an equal number of ring buffers so that each RX queue is mapped to a unique ring. Furthermore, Linux assigns a unique interrupt request (IRQ) number to each RX queue so that Linux can handle interrupts from different RX queues in parallel.

For efficiency, Linux separates interrupt handling into two parts—hard IRQ and soft IRQ—and invokes hard IRQ first. For an NIC interrupt, its hard IRQ simply sets some registers and triggers a soft IRQ. The soft IRQ reads packets from the ring buffer and executes the logic of the networking protocol, such as TCP/IP. The RSS technique allows Linux to handle IRQs in parallel.

By default, the soft IRQ reads from the ring buffer and executes the protocol logic within a single critical section protected by the lock of the ring. For more parallelism, Linux incorporates the Receive Packet Steering (RPS) technique [40]: when RPS is enabled, a soft IRQ reads a packet from the ring, puts it into a buffer called *backlog*, and then releases the lock of the ring. A separate thread, which may run on another CPU, will retrieve packets from the backlog and execute the protocol logic.

Finally the soft IRQ puts packets into socket buffers and the user-space threads may read from these buffers in parallel.

Such a multi-stage pipeline may re-order packets. Modern NIC and Linux preserve FIFO order for TCP packets with the same (sender IP, sender port, destination IP, destination port) and UDP packets with the same (sender IP, destination IP), by directing packets with same such information to the same RX queue, backlog and socket buffer. For SafeTimer, such guarantee is not enough since heartbeats and barriers are from different senders.

Overview of SafeTimer's solution. Our implementation is driven by three principles: 1) for portability, we hope to minimize modification to OS kernel code; 2) for performance, it should not incur significant overhead; 3) for portability, we hope to minimize dependence on specific NIC features or modification to NIC drivers.

As shown in Figure 4, SafeTimer re-directs heartbeats and barriers to a separate FIFO queue (called STQueue) early in the pipeline, so that they are not affected by re-ordering in later stages. However, since the earliest place we can perform such re-direction is after the soft IRQ reads the packets, RSS technique in the earlier stage may still re-order packets from different ring buffers. To solve this problem, SafeTimer sends a barrier packet to each RX queue/ring. If all of them later go through the

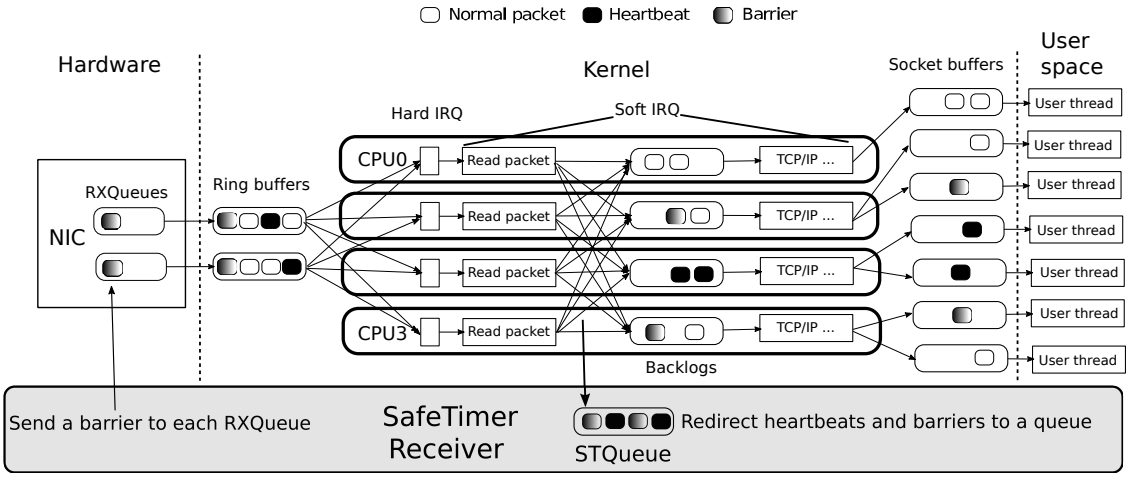

Figure 4: Barrier mechanism at the receiver. The algorithm in Figure 2 reads from STQueue.

STQueue, SafeTimer can know that all previous heartbeats are processed. The key to the correctness of this approach is that a soft IRQ needs to grab the lock of the ring buffer when reading a packet from the ring, and thus packets from each ring are read in a FIFO manner. As long as SafeTimer re-directs a packet before the soft IRQ releases the lock, such per-ring FIFO order will be retained in the STQueue. Therefore, when SafeTimer retrieves a barrier from the STQueue, it knows all previous heartbeats from the same ring must have been processed.

Next we present each step in detail.

Forcing a barrier to go through NIC. SafeTimer requires a barrier packet to follow the same execution path of a heartbeat packet. Putting a barrier in the ring buffer does not work because the OS won't read from the buffer until an NIC interrupt is triggered. Therefore, SafeTimer receiver forces the barrier packet to go through its NIC. This task, however, is challenging for multiple reasons.

First, Linux has the loopback optimization to route a local packet by memory copy instead of sending it to the NIC. SafeTimer bypasses this optimization by sending the barrier directly to the device driver. This approach, however, creates a new problem: the NIC will actually send the packet to the router. To prevent loops, routing protocols usually have a constraint that a router should never forward a packet to the port where the packet is received. Therefore, the router will drop a barrier packet, whose destination and source are the same.

Our prototype uses an NIC with two ports and sends a barrier from one port to the other, which eliminates the above problem. This solution requires the receiver to have at least two links to the router, but considering the fact that redundant links are already widely used for fault tolerance, such requirement often does not incur additional cost. If redundant link is not available, another alternative is to use the virtual LAN (vLAN) technique to virtualize a physical port into two virtual ports [47].

Sending a barrier to a specific RX queue. A few NICs provide the "N-tuple filter" feature to direct packets to specified RX queues, which makes this problem trivial. However, we find this feature is not common so far [21]. Most NICs calculate a hash value based on the IPs and ports information in a packet and then direct the packet to an RX queue based on the hash value. Therefore, we propose a general solution based on the assumption that one cannot control which RX queue a packet is directed to, but packets with same IPs and ports will always be directed to the same RX queue.

SafeTimer uses a brute-force search approach: during initialization, its receiver module sends barriers with different sender ports to its NIC to see which RX queue they are directed to, until SafeTimer can find a port for each RX queue. Since usually there are not many RX queues, such procedure could finish quickly. The challenge, however, is how to know which RX queue (represented by its IRQ number) a packet is directed to. SafeTimer uses netfilter [39], which is a tool provided by Linux, to intercept soft IRQ functions to check whether a packet is a barrier, but soft IRQ functions do not carry the IRQ number of the RX queue. We can modify the driver to pass the IRQ number to the soft IRQ, but this violates our principle to minimize driver-specific modifications.

To solve this problem, we leverage the *irq-cpu affinity* configuration provided by Linux, which can configure the mapping between RX queues and CPUs during RSS. By default, it is configured to be an all-to-all mapping, which means any CPU can execute any IRQ to read from its corresponding RX queue/ring, but Linux also allows one-to-one mapping. We leverage this option to "test" whether a barrier is sent to a specific IRQ i: we map IRQ i to CPU 0 and the other IRQs to the remaining CPUs arbitrarily. When intercepting the soft IRQ function, Safe-

Timer reads the CPU ID: if the packet is a barrier and the IRQ function is run on CPU 0, we can know the barrier must be sent to IRQ *i*; otherwise, SafeTimer tests a different *i* until it can find the right one.

Note that since the NIC always directs packets with same IPs and ports to the same RX queue, we only need to run the inferring procedure once for one machine. Afterwards we can use all-to-all mapping for efficiency.

Re-directing packets to STQueue. As shown in Figure 4, SafeTimer re-directs heartbeats and barriers to a FIFO STQueue after packets are read.

To implement this functionality, SafeTimer uses *netfilter* to hook the *ip_local_deliver* function, and configures iptable to re-direct heartbeats and barriers to a FIFO netfilter queue, which is called STQueue in SafeTimer. SafeTimer hooks *ip_local_deliver* because this is the earliest point packets can be re-directed in *netfilter*. SafeTimer sends heartbeats and barriers to specific ports so that they can be efficiently distinguished from normal packets.

This approach, however, is not fully correct when RPS is enabled: recall that when RPS is enabled, a soft IRQ will put a packet into the backlog and then releases the lock of the ring. In this case, *ip_local_deliver* is called after the lock is released and thus re-direction may not preserve the order of packets from the corresponding ring. To solve this problem, we use kretprobe [33] to intercept *get_rps_cpu* to return -1 for heartbeats and barriers: doing so essentially disables RPS for heartbeats and barriers. As a result, the re-direction will be executed under the protection of the lock of each ring and thus STQueue will preserve the order of packets from each ring. Normal packets, however, are not affected.

The timeout detection protocol (Figure 2) always reads heartbeats and barriers from the STQueue. However, SafeTimer does not remove heartbeats and barriers from later stages of the pipeline, because the OS needs to execute the logic of the network protocol, like congestion control or sending acknowledgements in TCP.

Reading drop count. SafeTimer's receiver module needs to read packet drop counts from both the OS and the NIC. Linux and most NICs have provided such statistics, but their implementation cannot achieve our goal.

In Linux, the NIC device driver periodically reads the drop count from the NIC, which can be fetched by reading /proc files system or using tools such as ethtool. Periodic reading means such statistics may be stale, which can cause SafeTimer's receiver module to miss recent drops and generate a false failure report. To make things worse, the NIC will reset drop count to 0 after it is read, so even if SafeTimer reads the drop count directly from the NIC, it may still get inaccurate results. To solve this problem, SafeTimer reads drop count from the NIC and then merges it with the number reported by the NIC driver. This is the only place SafeTimer requires modification to device drivers and OS kernel.

5.2 Blocking slow sender

As shown in Figure 3, SafeTimer's sender module blocks the sender if it cannot deliver heartbeats to the NIC in time. However, when sending a packet, Linux does not notify users whether or not the packet is delivered to the NIC successfully. Instead, it may write the packet to a buffer, return to the user, and send the packet to the NIC later, which may fail. To solve this problem, we use *kprobe* to intercept the function that the NIC driver invokes to reclaim resources after transmission is complete (e.g., *napi_consume_skb* or *__dev_kfree_skb_any*). As shown in Figure 3, SafeTimer applies the rules of the existing timeout detection protocol to check whether heartbeats are sent successfully. If so, SafeTimer's sender module will update t_{valid}. To block invalid packets, we use *netfilter* to intercept the *ip_output* function: if current time is larger than t_{valid}, the packet will be dropped.

Because of the processing delay, SafeTimer cannot get the exact time when a packet is sent. Instead, SafeTimer conservatively uses the timestamp after sending a packet, t_{after}: when checking whether a heartbeat is sent before end'_i (line 3 in Figure 3), SafeTimer compares t_{after} with end'_i. Such conservative approach ensures a sender failing to send heartbeats in time must be blocked, but it may also block a sender that has sent heartbeats in time, which is unnecessary but does not violate accuracy. Previous works have discussed how to minimize the impact of such unnecessary killing [38].

Since a slow sender process may communicate with other processes on the same machine, SafeTimer needs to block those processes as well, and thus it provides two blocking modes: the first blocks all processes on a machine; the second blocks only the sender process if the user is sure it does not communicate with other processes. Automatically tracking the information flow among different processes is out of the scope of this paper.

5.3 Supporting virtual machine

To maximize the benefit of SafeTimer in a virtual machine architecture, we could implement SafeTimer in the host OS or hypervisor and provide related functions to applications using hypercalls or remote procedure calls. By doing so, we can model the host OS or hypervisor as a whitebox. We plan to implement such support in the future. However, if the user has no control of the host OS or hypervisor, he/she can still deploy SafeTimer to the guest OS and model the host OS/hypervisor as a

blackbox, but this approach of course loses the ability to tolerate long delays in the host OS/hypervisor.

6 Evaluation

Our evaluation tries to answer three questions:

- What is the overhead of SafeTimer?
- Can SafeTimer achieve the expected accuracy property, despite long delays in the OS and the application?
- How much effort does it take to apply SafeTimer to existing systems?

To answer the first question, we have evaluated SafeTimer with a performance benchmark, which can send packets with different sizes, and compared its throughput and latency to those without SafeTimer. For the blackbox part, we use a simple protocol that sends heartbeats periodically with a configurable interval.

To answer the second question, we have injected long delays and packet drops at different layers at both the sender and the receiver to observe whether SafeTimer can prevent false failure report. Of course, this is by no means a complete test: we have proved the accuracy of SafeTimer in the appendix. This set of experiments serves as a sanity check about whether our implementation has actually achieved the expected properties.

To answer the third question, we have applied SafeTimer to HDFS and Ceph to enhance their timeout detection protocols and report our experience.

Testbed setting. We ran all experiments on CloudLab [15]. Each machine is equipped with two Intel Xeon E5-2630 8-core CPUs, 128GB of memory, 1.2 TB of SAS HDD, and a dual-port Intel X520-DA2 10Gb NIC. All machines are connected to a Cisco Nexus C3172PQs switch. Linux 4.4.0 is installed on all machines.

6.1 Overhead

SafeTimer incurs overhead for each packet at both the sender and the receiver: SafeTimer's sender module compares current time with t_{valid} before sending each packet; SafeTimer's receiver module re-directs heartbeats and barriers to the STQueue. To know whether a packet is a heartbeat or a barrier, the receiver module checks the destination port of each packet. When a sender fails, SafeTimer performs additional operations to block the sender, send barriers, and read drop counts, but since failure is rare, we focus on overhead in the failure-free case.

Since SafeTimer incurs overhead for each packet, such overhead should be relatively higher for workloads with smaller packets and thus we measure the overhead of

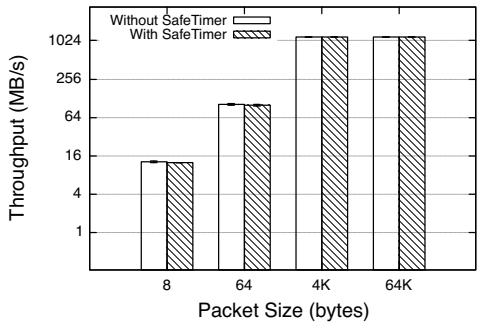

Figure 5: Throughput of the ping-pong benchmark with and without SafeTimer.

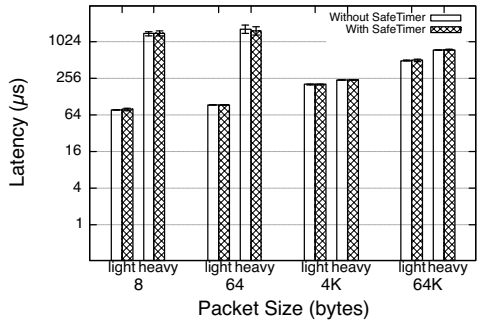

Figure 6: 99 percentile latency of the ping-pong benchmark with and without SafeTimer.

SafeTimer with different packet sizes. However, TCP may merge small packets in the same connection and thus affect our experiment results. To prevent such effect, we use a ping-pong benchmark as suggested in a previous work [2]: we create multiple sender threads at the sender, each creating a connection to the receiver. The sender thread sends a packet to the receiver and waits for the receiver to forward the packet back. In this case, since each connection has only one outstanding packet, TCP has no chance to merge packets. To increase load, we can increase the number of sender threads.

To measure the overhead of SafeTimer, we apply SafeTimer to the ping-pong benchmark and measure how it affects throughput and latency. To measure the maximal throughput, we increase the number of sender threads till we cannot gain higher throughput. To measure the latency, we run experiments under two loads: a light load of about 40% of the maximal throughput and a heavy load of about 90% of the maximal throughput. We do not measure the latency under the maximal throughput because in this case, the latency will be dominated by queuing delay. We run each setting 20 times to compute the average and standard deviation. We set the timeout interval of the blackbox part to be one second.

As shown in Figures 5 and 6, SafeTimer's overhead is small: for 4KB and 64KB packets, the overhead is less than 1%; for 8B and 64B packets, SafeTimer can

Node	Instrument Position	Injected Event	SafeTimer	Vanilla
Receiver	System call (recv)	Delay	No timeout	Timeout
Receiver	Socket (sock_queue_rcv_skb)	Delay/Drop	No timeout	Timeout
Receiver	NFQueue (nfqnl_enqueue_packet)	Delay/Drop	No timeout	N/A
Receiver	IP (ip_rcv)	Delay	No timeout	Timeout
Receiver	RPS (enqueue_to_backlog)	Delay/Drop	No timeout	Timeout
Receiver	Ethernet (napi_gro_receive)	Delay	No timeout	Timeout
Sender	System call (send)	Delay	Blocked	Alive
Sender	Socket (sock_sendmsg)	Delay	Blocked	Alive
Sender	IP (ip_output)	Delay/Drop	Blocked	Alive. Can observe drop.
Sender	Ethernet (dev_queue_xmit)	Delay	Blocked	Alive

Table 1: Verifying accuracy of SafeTimer by injecting long delay or packet drops. Gray cells indicate injection in kernel. N/A means this test case does not apply.

increase 99p latency by 0.7% to 2.7% and decrease throughput by 1.6% to 2.4%. Such low overhead is reasonable because SafeTimer's additional work (i.e., comparing t_{valid} at the sender and reading destination port at the receiver) is small compared to other work the OS has to perform for each packet (e.g., interrupt handling, memory copy). To confirm the result, we run the same benchmark on another set of machines on CloudLab (m510 [16]) with different NICs (Mellanox ConnectX-3 10G) and we find the overhead of SafeTimer is similar.

6.2 Accuracy

Although we have proved the accuracy of SafeTimer, we hope to sanity check whether our implementation has achieved the expected property. For this purpose, we inject long delays and packet drops at different layers at the sender and the receiver. We compare SafeTimer to a vanilla timeout implementation, which has a user thread to periodically send heartbeats at the sender and a user thread to periodically check timeout at the receiver.

Table 1 summarizes the events we injected and how SafeTimer responds to these events. We inject long delays at all positions but only inject drops if the corresponding function can actually drop packets. In these experiments, we set timeout interval to be one second and inject a delay of two seconds. As shown in the table, SafeTimer correctly prevents false failure report at the receiver and blocks the sender in all cases. The vanilla implementation, however, violates accuracy in almost all cases except when a heartbeat is dropped in *ip_output*: in this case, the sender receives an error and can retry.

6.3 Case studies

To evaluate how much effort it takes to apply SafeTimer to real-world applications and its performance overhead, we have applied SafeTimer to HDFS [45] and Ceph [9].

APIs of SafeTimer. At the sender side, SafeTimer provides two APIs: *safetimer_send_HB* to send a heartbeat and check whether it is delivered to the NIC in time; *safetimer_extend* to extend the *t_valid* value. At the receiver side, SafeTimer provides one API: *safetimer_check* to check whether it is safe to report a failure.

HDFS. In HDFS, a DataNode needs to periodically send a heartbeat to the NameNode and if the NameNode misses a number of consecutive heartbeats, the NameNode will mark the DataNode as "stale".

We modified one line of code in NameNode's *isStale* function, which checks whether heartbeats are missing for a DataNode, to perform the additional *safetimer_check*. We modified six lines of code in DataNode to use SafeTimer's APIs to send heartbeats and check whether heartbeats are sent in time. To simplify modification, we do not remove HDFS' original heartbeat mechanism: this leads to duplicate heartbeats but during our experiments, the overhead is negligible.

We killed a DataNode and found the NameNode can correctly mark a failed DataNode as stale. We have measured the performance of an HDFS deployment with three DataNodes by using Hadoop's built-in benchmark tool DFSIO. We ran each experiment five times. Without SafeTimer, DFSIO can achieve a write throughput of 203 MB/s (stdev 12.6) and a read throughput of 627 MB/s (stdev 18.4); with SafeTimer, it can achieve a write throughput of 206 MB/s (stdev 5.5) and a read throughput of 632 MB/s (stdev 8.4). The difference is not statistically significant.

Ceph. In Ceph, an Object Storage Daemon (OSD) sends heartbeats to its two peers every 6 seconds and if they can't receive the heartbeat for 20 seconds, they will send a failure report to the Monitor, which will consider the OSD as failed if receiving two reports.

In this mechanism, an OSD is both the sender and receiver of heartbeats. We modified two lines of code in OSD's *heartbeat_check* function to perform the *safetimer_check* before sending the failure report; we modified five lines of code to use SafeTimer's APIs to send heartbeats and check whether heartbeats are sent in time.

We killed an OSD and found the Monitor can mark it as down. We have measured the performance of a Ceph deployment with three OSDs by using Ceph's inbuit benchmark tool RADOS. We ran each experiment five times. Without SafeTimer, RADOS can achieve a bandwidth of 43.3 MB/s (stdev 1.6); with SafeTimer, it can achieve a bandwidth of 42.2 MB/s (stdev 1.1). The difference is not statistically significant.

7 Related work

Chandra et al. show that many classic problems in distributed system, such as consensus, can be solved with an accurate and complete failure detector [12]. In practice, timeout is widely used for failure detection, whose accuracy depends on their timing assumptions.

Synchronous systems. Under synchronous assumptions (i.e., delay of message transfer and clock deviation are bounded [12]), timeout can achieve both accuracy and completeness for failure detection. Many systems like primary-backup replication and HDFS [4, 5, 18, 24, 45] work under this assumption. To guarantee accuracy, these systems must make conservative assumptions about message delay and clock deviation. Previous works have tried to improve its accuracy by estimating the upper bound adaptively at runtime [3] and by killing a node if the failure detector reports the node has failed [8, 22]. SafeTimer can enhance synchronous systems to tolerate abnormal events in the OS and the application, without requiring any timing assumptions.

Asynchronous systems. Under asynchronous assumptions (i.e., delay of message transfer and clock deviation are unbounded), building a failure detector that is both accurate and complete is proved to be impossible [23]. Paxos [34, 35, 42] is a replication protocol designed for asynchronous environments: it is always correct (i.e., all correct replicas process the same sequence of requests) and is live (i.e., the system can make progress) when the environment is synchronous for sufficiently long. Paxos is used as building blocks in larger systems like Spanner [17] and Microsoft Azure Storage [7]. Compared to synchronous replication systems, Paxos is more expensive in terms of number of replicas and messages. Asynchronous systems don't need accurate failure detection for correctness, but since there is a cost to recover a failure, SafeTimer may help to reduce such unnecessary recovery by reducing the number of false failure reports.

Lease systems. A number of systems [1, 13] install a replicated lease manager (e.g., Chubby [6] and ZooKeeper [29]): a server needs to acquire a lease from the lease manager before it can service clients; the server has to renew the lease before it expires, and if not successful, the server will stop servicing clients. For accuracy, this approach requires the clock speed of servers and the lease manager to be sufficiently close, but it does not require the delay of message transfer to be bounded. Lease systems strike a balance between cost and timing assumptions, but it has its own limitations: first, the centralized nature of the lease manager means if a long delay happens at the lease manager, all leases will expire and all servers will stop servicing, which does not violate the accuracy property, but is certainly undesirable. As a result, lease systems prefer coarse-grained leases [6], which hurts system availability as well, similar as using a long timeout. Second, the requirement of a replicated lease manager makes it less desirable in small-scale systems. Systems using leases can benefit from SafeTimer by installing its sender module to ensure a server will not continue servicing after its lease expires.

Failure detection without timeout. A few systems implement a failure detector without using timeout. For example, Falcon [38] and its following works [36, 37] install probes in routers to monitor servers and install probes at different layers in a server to monitor upper layers. This approach essentially converts the whole communication channel into a white box. As a result, it requires intrusive modification to the routing layer, which makes its deployment challenging and sometimes impossible if the routers are out of the control of the user. To solve these problems, Falcon uses timeout as a backup.

Real-time OS. Real-time Linux [43] and other real-time frameworks for Linux such as RTAI [44] and Xenomai [49] can guarantee important tasks or interrupts are scheduled before given deadlines. However, this is not sufficient to achieve our goal, because long delay is not only caused by untimely scheduling, but also caused by the fact that an important task is occasionally blocked by a heavy task (Section 2). Real-time scheduling can address the former problem, but not the latter one.

8 Conclusion

This paper shows that we do not need to include the maximal local processing delay in timeout interval. Because of the whitebox nature of local processing, we can build efficient and accurate failure detection for this part, despite arbitrary processing delays. Our prototype SafeTimer allows one to use a shorter timeout to improve system availability, without sacrificing accuracy.

Acknowledgements

Many thanks to our shepherd Irene Zhang and to the anonymous reviewers for their insightful comments. This material is based in part upon work supported by NSF grant CNS-1566403.

References

[1] Apache HBASE. http://hbase.apache.org/.

[2] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, CO, 2014. USENIX Association.

[3] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 354–363, Washington, DC, USA, 2002. IEEE Computer Society.

[4] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault Tolerance. *ACM Transanctions on Computer Systems*, 14(1):80–107, February 1996.

[5] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *CDCCA*, 1992.

[6] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[7] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[8] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[9] Ceph. https://ceph.com.

[10] Ceph Default Heartbeat Configuration. http://docs.ceph.com/docs/master/rados/configuration/mon-osd-interaction/.

[11] Ceph MDS heartbeat timeout during rejoin. http://tracker.ceph.com/issues/19118.

[12] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[14] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[15] CloudLab. https://www.cloudlab.us.

[16] Hardware information in the CloudLab manual. http://docs.cloudlab.us/hardware.html.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[18] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, 2008.

[19] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. Understanding real-world timeout problems in cloud server systems. In *IC2E 18*.

[20] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: A survey. *ACM Comput. Surv.*, 17(3):341–370, September 1985.

[21] Overview of Networking Drivers. http://dpdk.org/doc/guides/nics/overview.html.

[22] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, 52(2):99–112, February 2003.

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

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, 2003.

[25] Hadoop Default Configuration. `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/core-default.xml`.

[26] Set the zk default timeout to 3 minutes. `https://issues.apache.org/jira/browse/HBASE-3273`.

[27] Hdfs. `http://hadoop.apache.org/`.

[28] HDFS Default Configuration. `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml`.

[29] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, 2010.

[30] Internet control message protocol. `https://tools.ietf.org/html/rfc792`, 1981.

[31] InfiniBand Performance. `http://www.mellanox.com/page/performance_infiniband`.

[32] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. Hios: A host interface i/o scheduler for solid state disks. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 289–300, Piscataway, NJ, USA, 2014. IEEE Press.

[33] Kernel Probes (Kprobes). `https://www.kernel.org/doc/Documentation/kprobes.txt`.

[34] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[35] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.

[36] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 427–441, Lombard, IL, 2013. USENIX.

[37] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 9:1–9:16, New York, NY, USA, 2015. ACM.

[38] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *SOSP*, 2011.

[39] Netfilter. `http://www.netfilter.org/`.

[40] Scaling in the Linux Networking Stack. `https://www.kernel.org/doc/Documentation/networking/scaling.txt`.

[41] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proc. 7th OSDI*, November 2006.

[42] B. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly-available distributed systems. In *Proc. 7th PODC*, 1988.

[43] Linux Foundation Real-Time Linux Project. `https://rt.wiki.kernel.org`.

[44] RTAI - the RealTime Application Interface for Linux. `https://www.rtai.org`.

[45] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[46] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Holzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in googles datacenter network. In *Sigcomm '15*, 2015.

[47] Patricia Thaler, Norman Finn, Don Fedyk, Glenn Parsons, and Eric Gray. Media access control bridges and virtual bridged local area networks. Technical report, IETF, March 2013.

[48] Yang Wang, Manos Kapritsos, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems. In *NSDI*, 2014.

[49] Xenomai - Real-time framework for Linux. `http://xenomai.org`.

A Appendix: Proof of accuracy and completeness

Assumption of the blackbox part. *The existing protocol can guarantee that if a sender has successfully sent heartbeats for interval i, at least one of the heartbeats will be processed (either delivered to the OS or dropped) by the receiver's NIC by end_i.*

Theorem A.1. *(Accuracy) If SafeTimer's receiver module reports a failure at time t, the sender will not be able to send any packets generated after t.*

Proof. As shown in the protocol, SafeTimer's receiver module reports a failure for interval i if two conditions are both satisfied: first, the receiver module has received the barrier but not any heartbeats for interval i. Because the barrier is sent after end_i and because of the barrier's semantic, any heartbeats processed by the NIC before end_i must either have been delivered to the receiver module or have been dropped. Since the receiver module has not received any heartbeats, we can conclude that it is either because the NIC has not processed any heartbeat by end_i or because some heartbeats are dropped at the receiver side.

The second condition is $dropcount = 0$ and $t_{drop} < start_i$, which means there are no packet drops at the receiver side in interval i. By combining this condition with the first one, we can conclude that the receiver's NIC must have not processed any heartbeat packets for interval i before end_i. This means the sender must have not successfully sent the heartbeats for interval i (Assumption of the blackbox). In this case, the sender will not extend its $t_{valid} = end'_i$, and thus will stop sending any messages after t_{valid}.

Since t is larger than end_i and $t_{valid} = end'_i$ at the sender should be earlier than end_i at the receiver, we can conclude that $t_{valid} < end_i < t$ and thus the sender will not send any packets generated after t. □

Theorem A.2. *(Completeness) If the sender has failed, SafeTimer's receiver module will eventually report a failure if the following two conditions both hold for sufficiently long (five consecutive intervals in the worst case): 1) the receiver's processing speed is normal, which means events (e.g., heartbeat, barrier, and reading drop count) generated before or during an interval can be handled by the end of the interval; 2) the receiver does not experince any packet drops.*

Proof. Suppose the sender fails to send heartbeats in interval i, and afterwards, there are five consecutive intervals j to $j+4$ ($j > i$) during which the receiver's processing speed is normal and the receiver does not experience any packet drops.

Since the receiver's processing speed is normal in interval j, the receiver should be able to handle all delayed heartbeats from the sender, if any, by the end of interval j, which means the receiver won't receive any heartbeats in interval $j+1$. Therefore, the receiver will send a barrier at the end of interval $j+1$. Since the receiver's processing speed is normal and there are no packet drops in interval $j+2$, the receiver will receive the barrier and read drop count by the end of interval $j+2$. If drop count is 0 (t_{drop} must be smaller than $start_{j+2}$ because the receiver does not read drop count in interval $j+1$), the receiver will report the failure; otherwise, the receiver will update t_{drop} (the new t_{drop} must be smaller than $start_{j+3}$) and repeat the above procedure. At the end of interval $j+3$, the receiver must report a failure because both conditions to report a failure can be met: 1) since the processing speed is normal and there are no packet drops in interval $j+4$, the receiver can receive the barrier for $j+3$ but it cannot receive any heartbeats; 2) drop count is 0 because there are no packet drops in interval $j+3$ and $j+4$; $t_{drop} < start_{j+3}$. □

Note that five intervals are the worst case: if previously there are no delayed heartbeats or packet drops, the receiver will report the failure after one interval.