Efficient failover using a flexible NIC

Kyle DeHolton

May 4, 2017

Abstract

Failover is an important aspect of a database. Failover needs to happen efficiently for the database to be performant. Currently, in Apache Cassandra, it takes a long time for a node to detect that another node has crashed. This is due in part to networks being inherently unreliable in terms of latency. Whether a node has crashed is never known with full certainty. However, if a node were to send out a packet when it died, then when another node got that packet they could be completely certain that the node has died. Instead of interpreting a lack of packets as a node’s death, we can interpret the arrival of a packet as a node’s death.

With more and more services moving to the cloud, high availability and low latency are more important than ever. By offloading failure detection to a flexible network card and daemon, I intend to increase Cassandra’s performance in the event of a failure.
1 Introduction

Distributed databases are relied on heavily in today’s society [1]. Failures are an unfortunate event that must be planned for. Downtime of a cluster can result in very tangible lost profit. While most databases have their own software-based method of failover, these methods can be relatively slow and this can limit performance. By performing parts of the failover in hardware, it will be able to occur much more quickly, enabling the system to recover from a failed node more quickly.

Replication and failover are linked. Data is replicated across multiple machines to protect against the event of failure. If failure does happen, then the system fails over to one of the replicas. If one of the nodes in Cassandra dies, then any other Nodes interacting with it are delayed until they’ve determined that is has crashed.

This paper aims to increase performance of Cassandra by offloading parts of the failover to a programmable network card. This NIC is paired with a daemon running on each node that monitors Cassandra. If Cassandra crashes, the daemon will notify all of the other nodes in the cluster. If the entire machine dies (but still retains power), the NIC will do the notification. By having the NIC assist with failure detection, failures will be detected more quickly and reliably, and less time of the system will be stuck waiting to hear from potentially-dead nodes. Detecting the presence of a message is much easier than detecting the absence of a message.

2 Background

2.1 Apache Cassandra

Apache Cassandra is an open-source NoSQL replicated database system. It is used as the back end in a lot of today’s websites and services, including CERN, GitHub, and Netflix [1].

Each node in the cluster is equal; there is no primary or master node. A client can connect to any of the nodes to make a request. This node is called the coordinator node for that request. The coordinator acts as the interface between the nodes and the client until the client disconnects. The coordinator is the node that determines which writes go to which nodes.

When data is written to a node, it is first written to disk, in the commit
log. Then it is written to memory, to a memtable. Once the memtable has grown past a certain (configurable) size, the writes in the memtable are written to a Sorted Strings Table (SSTable) on disk. Buffering the writes in memory allows for quicker processing of data, as it minimizes the time spent writing to the slow disk.

The nodes in the Cassandra cluster are arranged in a logical ring. Each cluster can have multiple keyspaces, and each keyspace can have its own replication configuration. This includes a value called the replication factor, here written $R$, indicating how many nodes will store a copy of each row. For example, a replication factor of 1 means that data is only stored once in the database. The hash of the primary key determines the node that will store the row in the database. The next $R - 1$ nodes around the ring also store this row. Figure 1 shows an example logical ring of nodes.

Cassandra uses an implementation of the Φ failure detector from *The

---

Figure 1: Cassandra’s nodes are arranged in a logical ring. In this example, the system has a replication factor of 3.
Φ Accrual Failure Detector by Hayashibara et al. In this scheme, a node’s status is not a binary up/down. There is a continuous value representing the suspicion one node has that another is dead. This suspicion level, Φ, is measured on a scale that dynamically adjusts to changing network conditions. [2]

Cassandra uses a gossip protocol to exchange state, determine failures, and help recently-recovered nodes get up to date. Once every second, each node will exchange state information with up to three other nodes in the cluster. Each gossip message has a version associated with it so that during gossip a node can determine if the data it receives is newer or older than data it has already received.

The client, on each request, provides a consistency level that they would like for that request. There are a number of levels here, the most interesting being ALL, QUORUM, ONE, and ANY. Levels ALL, ONE, and ANY are fairly self-explanatory (being the number of nodes on which the operation must succeed). QUORUM may not be. A quorum is a set of nodes defined as follows: given any two quorums in the system, their intersection is always non-empty. If all reads and writes are done at consistency level QUORUM, then the system will provide strong consistency [3].

Let us call the consistency level cl. For writes, the write is sent to all of the machines that should store it (determined by the replication factor). The coordinator then waits for cl acknowledgments. Consistency level works slightly differently for reads. The coordinator only makes requests to exactly cl number of the replicas storing the relevant data for the query. If all of the nodes are operational, this shouldn’t be a problem. If there are crashes during this time, however, this is where there is the potential for the longest delays before failover. Imagine the following scenario. There are four nodes A, B, C, and D, with R = 3. Assume that the client connects to node A, and that nodes B, C, and D store the row the client is requesting. The client asks for a read of this data, at consistency level QUORUM. A, the coordinator node for this request, will contact exactly 2 nodes to start out. Let us say that B is down but has not been recognized as down yet. A requests this row from nodes B and C. It will take a relatively long time for the client to hear back from A, as it will take A a relatively long time to determine that B is down and send a request to D instead. Figure 2 shows a graphical representation of this example.

In both reads and writes, if the consistency level cannot be met, the client is returned an error. [3]
Figure 2: Sample Cassandra cluster with replication factor of 3. On the left: a client attempts to read at consistency level QUORUM. Assume the record is stored on B, C, and D. The coordinator, A, sends this read to B and C. B has recently died and A doesn’t know this yet. A waits for a response from B for a long time, as it is not yet marked as down. On the right: A eventually determines B is dead and sends the read to D instead, who responds.
The coordinator node will route the request to as few replicas as possible to meet the consistency level requested. It uses a dynamic snitch to pick the most responsive replica(s) to send the request to.

In general, snitches determine which datacenter and rack (both logical groupings) a node belongs to. They keep track of network topology information, and aid in routing Cassandra’s requests. Each snitch also has a ‘dynamic snitch’ layer, that monitors the performance of reads and helps the coordinator pick which replicas to talk to, based on this history.

Using data from this dynamic snitch, the coordinator will actually resend a read request if a node is taking significantly longer than usual (this is called rapid read protection). However, this method is still slow and sending excess read requests is not free from a performance standpoint either—in smaller systems, this can hurt throughput, as it places a larger strain on each replica. Also, with no or limited data, the snitch performs less well.

Offloading most of the failure detection mechanisms to the NIC will allow the failure detector to perform more responsively, and limits strain on the system from redundant requests.

2.2 P4

P4 (coming from the 2014 paper Programming Protocol-Independent Packet Processors) is a declarative programming language designed for use with network forwarders such as switches or network cards. A P4 program is broken down into five components: header definitions, a parse graph, table definitions, action definitions, and control flow. A P4 program parses each packet’s headers and matches each packet with an action based on the rules defined in each table. Figure 3 shows a model of the entire P4 flow.

2.2.1 Header definitions

Header structures defined in P4 tell the program how to interpret headers that it strips from the packet. You can write a header for any protocol that exists, as well as create custom headers. Each of the created header instances has a validity indicator, which can be tested through the use of the valid keyword. A header is valid once it has been read in through the parser. Using this keyword allows for the checking of which headers exist in the packet currently being processed. The validity of a header can also be changed with actions.
Figure 3: Abstract model of P4 [4].
2.2.2 Parse graph

In P4, the parser is modeled as a state machine. The parser starts parsing at the first byte of the packet. When a header is extracted, the correct amount of data is read into the header structure, and the parser’s offset into the packet moves forward that amount, making a state transition. Once parsed, the packet is represented as a collection of header and metadata instances called the Parsed Representation, on which the rest of the program will operate. Figure 4 is a possible state graph, with each node representing a header that is parsed out of the packet before continuing.

Each packet also has associated with it some metadata, which can also be read/written by actions. Some of this metadata is critical to the operation of the NIC; this metadata is called Intrinsic Metadata. The always-defined variable standard metadata contains, among others, the following self-explanatory fields: ingress_port, packet_length, and egress_port.

![Figure 4: Graph showing possible state transitions while parsing headers.](image)

At egress, the Parsed Representation is deparsed (serialized), so that it can be sent back out on the wire. A header is only included in the deparsing if it is valid. Metadata is not included in this, so if a metadata value is desired to be attached to an outgoing packet, that value must be copied to a header in one of the actions.
2.2.3 Table definitions

These tables, called *match+action tables*, allow for different actions to be applied depending on the headers or metadata. The tables are written in a YAML file, and a P4 program can have any number of tables. Each table has a set of rules, and each rule has a name, a field on which to match, and an action to perform (potentially with a parameter passed in).

Once a table is selected (see Section 2.2.5), the fields specified in the control flow are sent to the table. If one of the fields matches with a rule, then the action specified in that rule is taken. If no match is made, then each table has a default rule whose action will be applied.

2.2.4 Action definitions

Actions, which are declared as functions, modify the parsed representation of a packet. There are primitive actions, which are built in to the language, and compound actions, which are made up of one or more primitive actions. Figure 5 shows some of the primitive actions relevant to this paper.

<table>
<thead>
<tr>
<th>Primitive action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_header</td>
<td>Add a header to the packet’s Parsed Representation.</td>
</tr>
<tr>
<td>copy_header</td>
<td>Copy one header instance to another.</td>
</tr>
<tr>
<td>remove_header</td>
<td>Mark a header instance as invalid.</td>
</tr>
<tr>
<td>modify_field</td>
<td>Set the value of a field in the packet’s Parsed Representation.</td>
</tr>
<tr>
<td>drop</td>
<td>Drop a packet (in the egress pipeline).</td>
</tr>
<tr>
<td>no_op</td>
<td>Placeholder action with no effect.</td>
</tr>
</tbody>
</table>

Figure 5: List of primitive actions from the P4 specification.

2.2.5 Control flow

The order that match+action tables are applied is determined by the control functions. Control functions can apply tables, call other control functions, or evaluate conditionals.

The *apply* keyword is used to execute a table. Multiple tables can be applied per packet. It is often useful to apply certain tables based on if a conditional statement is met or a certain header is valid.
2.3 Netronome

The network card in use in this system is a Netronome NFP-4000 card. It will run P4 or C code (P4 was used in this paper). P4 cannot process any of the data inside of a packet, therefore all communication with the NIC was implemented using custom packet headers or special fields set in existing headers.

3 P4 code

While I was never able to run the P4 code due to hardware incompatibilities (see Section 6), this section describes what it does and why it would work. Broadly, it processes heartbeats from the running daemon and sends out a broadcast packet if they stop. To all other traffic, it attaches the current time so that they have timestamps to check against their last local heartbeat.

3.1 Headers

In addition to creating Ethernet, IPv4, UDP, and TCP headers (the descriptions of which are not included here and can easily be found), I have also defined a time header, as shown in Figure 6. This header allows for the time to be sent to the NIC with each heartbeat message. This header is also affixed to all outbound traffic. Also defined here is a metadata structure for the NIC to use as a variable to do comparisons with.

3.2 Parser

Figure 7 shows the possible states and state transitions. The IP protocol number is what determines where in the graph the packet goes after IPv4 is parsed. Here, the only type of packet that is allowed to have a time header is a TCP or UDP packet. If there is another type of IP packet in the system it will not have the time header attached to it.

3.3 Actions

Figure 8 shows the actions that I have defined. The drop act and noop actions are wrappers for simple actions built in to P4. The update heartbeat action is what is run when the NIC receives a heartbeat from the daemon.
header_type time_hdr {
    fields {
        time : 32;
    }
}

header_type local_metadata_t {
    fields {
        nic_time : 32;
    }
}

header time_hdr time;
metadata local_metadata_t local_metadata;

Figure 6: Time header definition.

![Figure 7: Graph showing possible state transitions.](image-url)
When the heartbeat is received (including a timestamp), the timestamp is written to a register. Registers are one of the ways to maintain state between packets in P4. The copy_time action copies the register storing the time at which the NIC last received a heartbeat in to the local metadata. The reason for this is that registers cannot be used in comparisons, whereas metadata and headers can.

The last two actions, are slightly different. Here I am creating two new primitive actions. P4 allows for C functions to be written and used as primitive actions. [5] The notify_daemon_remote action is a C function that does a number of things. First it duplicates the packet so the original packet is not lost. The duplicate packet is then changed to have a destination of the broadcast address and a source address of the current machine. The action add_time is a C function that adds a time header to the packet with the current system time.
3.4 Control flow

In this system, there is the NIC, the daemon, and Cassandra itself. The flow of this system is broken into two paths. There is the case where Cassandra dies but the daemon doesn’t, and the case where the entire OS dies.

In normal operation, the daemon sends heartbeats to the NIC, letting it know that the OS and Cassandra are working. If Cassandra dies, the daemon will send out a message to the other nodes, letting it know. If the entire OS dies, the NIC will stop getting heartbeats, and once a pre-determined amount of time has passed it can let the other nodes know about the crash. Since the NIC can only run code when it receives a packet, it can only check if the current time is greater than the last received daemon heartbeat by some threshold amount when it processes a packet. Due to the gossip protocol running every second, there should always be enough traffic flowing across the NIC to trigger the timeout in a reasonable time frame.

The ingress control function, called from the parser is what applies tables to the Parsed Representation of the packet. Here, first it checks if there is a time header. If there is, this means that the packet was sent by another Netronome NIC running this firmware. It then tries to apply the table `time_tbl`. Based on which action was applied from that table (update_heartbeat or copy_time), different blocks of code are run. If the copy_time action was run, then there’s a check. If the time in the packet’s header is larger than the last received heartbeat plus some configurable threshold, then we deem this as the local daemon having timed out. (A simple binary failure classifier here is acceptable, as the variance in packet delays and drops is much lower since the daemon heartbeats never leave the machine.)

After all of that happens, then the in_tbl table is applied. This forwards the packet to the NIC’s virtual interface so that it behaves like a regular network interface card should.

4 Daemon

I wrote a daemon that checks the status of the local Cassandra node. It will notify the other nodes if Cassandra crashes (but the daemon doesn’t). Here, the notification packet is a UDP packet sent out to address 255.255.255.255 (local broadcast), port 4000. This assumes that all of the Cassandra nodes
control ingress {
    if (valid(time)) {
        apply(time_tbl) {
            update_heartbeat { // Heartbeat was updated
                // Do nothing
            }
            copy_time { // Time to check if heartbeats have timed out
                if (time.time > local_metadata.nic_time + TIME_THRESHOLD) {
                    // Cassandra Node has timed out!
                    apply(node_timeout_tbl);
                }
            }
            default {
                // Do nothing
            }
        }
    }
    apply(in_tbl);
}

Figure 9: Ingress control function.
are on the same local network (realistic for a single-data center set up). If there are multiple networks, this assumption can be changed fairly easily. The port number is not important - if there is another service listening on port 4000 it can be changed trivially. The UDP packets contain as their payload the IP address of the node that has just crashed.

Cassandra nodes have their own heartbeats to determine when one node dies. However, as these are across the network, where there is a lot of unreliability, it takes a relatively long time to determine with high reliability that a node has crashed. This daemon also sends heartbeats to the Netronome card. Since these heartbeats never leave the system, there are fewer places that latency or loss can be introduced. If the entire system or OS crashes, then the NIC will stop getting heartbeats from the daemon and send out notifications to the other Cassandra nodes.

Each heartbeat is sent with a header containing the current system time. Additionally, all packets sent by the Netronome card have this header added to them by P4. Any time this header is processed by P4, it is checked against the time of the last heartbeat sent. If this difference is greater than a configurable threshold, the local Cassandra node is assumed to be dead.

In addition to this daemon, Cassandra itself has been changed. Each Cassandra node now creates a UDP socket over which it waits for any packets on port 4000. If it receives a packet over this port, the payload (IP address of the crashed machine) is parsed and Cassandra forces its failure detector to mark that node as down.

## 5 Cassandra benchmarks

Data was collected with Cassandra running. In these tests, NTP is run to synchronize the clocks to an outside server to a close enough degree.

Firstly, we look at a two-node cluster with and without the daemon running. One node is killed, and the time until the other node marks it down is measured (in milliseconds). Without the daemon running, the mean of this time across 30 trials was 18.99 seconds. With the daemon running on both machines, the mean time across 30 trials was reduced to .1593 seconds—an improvement of two orders of magnitude! (see Figure 10)

While there were no benchmarks able to be taken with the Netronome NIC, there were some taken with Cassandra running by itself. This data was taken on two- and three-node Cassandra clusters (with replication factors of
Figure 10: Graph of mean time for a failure to be detected in a two-node Cassandra cluster. Data was taken with and without the daemon running across 30 runs each.

two and three, respectively) and one client with a varying number of threads. Figure 11 shows a graph of latency versus number of threads, while Figure 12 shows operations per second versus threads.

No data was collected measuring the performance of failover (without the P4 program). The best scenario for this, however, would appear be 271 threads. There would need to be (at least) three servers, with a replication factor of three, and a consistency level of QUORUM. This allows for one machine to die and the coordinator to have another server that it can fail over to.

6 Lack of full results

The servers on which this project was run are Dell PowerEdge R230 servers. It was discovered towards the end of the project that these servers do support SR-IOV (single root input/output virtualization), but do no support ARI (Alternative Routing-ID Interpretation), which the Netronome cards rely on to function.

The failover in the case of only the Cassandra process dying is taken care of by the daemon. However, as the P4 code was unable to be run, there are
Figure 11: Latency versus number of client threads in two- and three-node Cassandra clusters.
Two servers, one client
Consistency level = ALL

Three servers, one client
Consistency level = QUORUM

Figure 12: Operations per second versus number of client threads in two- and three-node Cassandra clusters.
no results for the case of both the daemon and Cassandra dying (in the event of an OS failure, for example).

7 Related work

Open-NFP is an organization whose purpose is to help further networking datapath offload techniques. Their website is “a common repository of open source networking datapath P4/C application code and research papers” [6].

One of Open-NFP’s projects is Paxos where the coordinator and accepter are offloaded to the NIC. Paxos is used for solving consensus with unreliable nodes and channels. This NetPaxos is also written using P4, and can increase performance in distributed system. [7]

8 Future work

I am interested in seeing how efficient failover based on the NIC is. Barring any hardware incompatibilities, this should be very straightforward to do in the future, as all of the code is written. Since I was never able to run the P4 code, it is likely that there will be a minor amount of debugging to make it work entirely as desired (little things I have overlooked or are buggy).

Related to this is migrating the daemon entirely to the NIC. Since C code can be run in P4, all aspects of the daemon can happen on the NIC. This would have the benefit of being more robust in the case of failure. However, since with P4 code can only be run when packets arrive, checking Cassandra would potentially happen less frequently when there is less network traffic (though there should always be enough, due to the gossip protocol).

An additional idea for optimization is to offload some of the replication of Cassandra to the NIC. As failure-less operation is the expected behavior of Cassandra, offloading Cassandra’s replication would lead to a better-optimized database in the general case.

This paper was created with no regard for security; if this were to be implemented in a production database, there are a number of security vulnerabilities that must be addressed. For example, none of the special packets exchanged between Netronome NICs employ any sort of authentication or integrity checks. A malicious person could craft packets and erroneously convince nodes that other nodes are down. This could be stopped by restricting
access to the network over which the Cassandra nodes communicate, however it is possible to have a Cassandra database spread across multiple datacenters. Either some form or inter-datacenter packet could be created, or the Netronome NICs could be limited to only communicate in the local datacenter for decreased performance benefit.

Fixing any of these would not be of exceptional difficulty. However, this could be expected to decrease the performance gain somewhat; the extent of this is not known.
References


