Paxos in a Network Card

Jonathan Ibers
The University of Texas at Austin

May 3, 2017

Abstract

Paxos is a widely used and efficient algorithm for implementing consensus between a set of nodes. While there have been efforts to optimize the algorithm, there are certain aspects of the algorithm that create bottlenecks in implementations of Paxos. In this paper, we aim to identify those bottlenecks and mitigate their impact by moving parts of the algorithm to hardware. We attempt to utilize the Netronome NFP-4000 network card and the P4 programming language to implement Paxos in hardware to decrease latency and increase throughput of the system.

1 Introduction

In order to solve more computationally expensive tasks, many are turning to cloud computing and distributed systems. A key component of these distributed systems is ensuring consensus. Consensus is the process of validating that all the separate members of a distributed system all agree on a particular result. Consensus is extremely difficult because different nodes could have events happen simultaneously, and each believes their event happened first. Because these events happened on separate nodes, there typically isn’t a central clock or time for each node to confirm that their event occurred first. Consensus is even more difficult in an environment where messages may be lost or members may fail. Paxos was designed as an algorithm that would be tolerant of failures, and guarantees consistency.

While safe, the algorithm does notably create various bottlenecks in its implementation. For this paper, we tested for these performance bottlenecks using the Libpaxos library, an implementation of Paxos that allows for nodes to be classified as one particular part of the algorithm or a generic node.[5] We treat each node as generic, so our members could deal with every part of the algorithm and we could more easily identify bottlenecks.

Once we have identified parts of the algorithm that bottleneck our system, we worked on moving those parts down to the Netronome NFP-4000
network card. The Netronome card allows for custom behavior using the P4 programming language. We utilize P4 to implement Paxos at the hardware level, and we aim for improved performance when compared to an implementation developed in software.

An implementation in the card should be faster than an implementation in software. With a hardware design, there is no need for PCIe bus communication, which cuts down on the time it takes to respond to a message. Hardware designs are also faster than a general purpose CPU, as they are more integrated into the hardware. Utilizing P4, a packet gets processed using the same optimized hardware that a NIC normally uses. This represents a “best case” scenario where we can parse packets using only functions that are implemented efficiently on the hardware. We hope to be able to utilize these advantages in our implementation of Paxos.

2 Background

2.1 Paxos

Before describing our solution to speed up Paxos using hardware, we describe the Paxos protocol.

Paxos is a robust and widely used consensus protocol introduced by Leslie Lamport in 1989[1]. Paxos is always safe, which means that multiple learners cannot learn different values, even in asynchronous environments. It also is live within reasonable assumptions of synchrony, which means that under an expected number of failures, it will make progress. Both of these properties are very valuable for a consensus algorithm, as it guarantees that different members cannot learn different values (the safety property) and in periods of synchrony, it will have members eventually learn proposed values (the liveness property).[2]

In Paxos, different replicas can play different roles, namely a proposer, an acceptor, or a learner. A client will propose values to the proposer, who begins rounds with the intention of assigning the proposed value to a slot. The acceptors are queried by the proposer. The proposer waits for a quorum of acceptors to signal that they are ready to accept the value. The learner can execute actions or other behaviors once values are learned and an order is established.[2]

The client proposes a value to a replica, which acts as a proposer for that round of Paxos. From there, the algorithm continues in two phases.

2.1.1 Phase 1

In the first phase, the proposer works to submit the value sent by a client. It selects a unique ballot number to identify that proposal, and sends a Prepare message to a quorum of acceptors of the proposer’s choosing. The
ballot number must be greater than any of the previous numbers selected by that proposer. [2]

When an acceptor receives this Prepare message, it checks any previously accepted ballot numbers, which could be from any proposer, including the one it just received from the Prepare message. If the ballot number is lower than a previously accepted ballot number, the acceptor can ignore the Prepare message. For the sake of optimization, it could send a Nack or some other rejection to let the proposer know.

If the ballot number is greater than any previously accepted values, the acceptor updates its ballot number, and crafts a Promise message. This message includes the ballot number, as well as any previously accepted values from phase 2.

2.1.2 Phase 2

Once the proposer has received a quorum of Promise messages for its proposed ballot number, it proceeds to the second phase. The proposer selects a slot for the proposed value. It has to choose an empty slot no other value has been assigned. Because they have said that the proposed ballot is the highest they will accept, and the acceptors sent previously accepted values, the proposer is safe in assigning the lowest empty slot. It crafts an Accept message, which contains the ballot for this proposal, the assigned slot, and the value proposed by a client.

Upon receiving an Accept message, the acceptors once again check the ballot number. If it has accepted a higher ballot from another Prepare message, it can ignore this Accept message. If the proposed ballot is the same as the one it received in the Prepare message, it crafts an Accepted message to the proposer. The accepted message contains the same three values as the Accept message, and acks as an acknowledgement that it has been accepted.

If the proposer receives an Accepted message from a quorum of acceptors, it can inform the learner that the proposed value can be assigned to the given slot. The learner can then perform whatever task is relevant to the situation. A common behavior of the learner is to execute instructions proposed by the client. Once these instructions are assigned to a slot number, the learner can execute them in that assigned order, and inform the client about the output of the instructions.

2.1.3 Analysis

All four of these different message types have different overheads. For example, an acceptor that received a Prepare message needs to compare it to its stored ballot number, and then store and send all of its previously accepted values. Upon receiving an Accept message, however, the acceptor
needs only to check its stored ballot number, before it responds with an *Accepted* message.

The proposer has even more extreme differences in message types. A received *Promise* message might contain more accepted values than the proposer has seen before, so it must parse and store these values into its own memory. When the proposer receives an *Accepted* message, however, it only needs to check if it has received a quorum. Otherwise, it must wait for another message.

Because of this disparity between the processing for different messages, there are natural bottlenecks that exist in the algorithm. Speeding up one of these different messages or phases would improve the performance of Paxos as a whole.

### 2.2 The Netronome Card and P4

We targeted the Netronome NFP-4000 card to create an implementation of Paxos. The card supports standard 10G, x40G and x100G interfaces. Most importantly, the card has the ability to have a customized network data path. This data path can be controlled using a language known as P4, and allows us to customize how packets are processed by the card. It has internal memory which can be used to save state.[4]

P4 stands for *Programming Protocol-Independent Packet Processors*. It is a high level language that is used to express how packets are processed by some network forwarding element, such as a switch, router, or, in our case, a network interface card. Any P4 program specifies five parts: headers, parsers, tables, actions, and control flow.[3]

P4 is written to customize how the data plane of a programmable device (such as a NIC, switch, or FPGA). It does not change any other part of the hardware, which allows the card’s control plane to continue to interact with the data plane as usual. P4 does not include many basic features of other programming languages, such as loops or pointers. This is because all functions available in P4 are designed to be implemented efficiently on the hardware. Because of these limitations, programs designed with P4 will run quickly on hardware when compared to a similar implementation running on a general purpose CPU.

A diagram of the pipeline of P4 is given below. P4 modifies the parser and match+action tables, leaving the rest of the card unchanged.
2.2.1 Header definitions

Header definitions describe the format of any packet headers that the program intends to parse and use. A header definition specifies the name and size of each of the fields to be interpreted by the P4 program. The list of fields is ordered, and interpreted from incoming packets.[3] The header type definition defines the list of fields and their length in bits. An example of the IPV4 header is given below.

```c
header_type ipv4_t {
    fields {
        version : 4; // IP version, always 4
        ihl : 4; // Header length
        diffserv : 8; // Differentiated services field
        toLen : 16; // The total size of the headers and data
        identification : 16; // Identification field
        flags : 3; // Flags for fragmentation and more fragments
        fragOffset : 13; // Offset of fragmentation
        ttl : 8; // Time to live
        protocol : 8; // The protocol used in the data portion
        hdrChecksum : 16; // A checksum used for error checking
        srcAddr : 32; // IPV4 address of the source
        dstAddr : 32; // IPV4 address of the destination
    }
}
```
Other custom packet headers can also be defined, which we will use in our program in the next section.

2.2.2 Parse Graph

P4 defines the parser as a state machine. The parser extracts the values from the headers, which are used in the Match + Action tables defined in the next section. An example is given below, as well as a parse graph that the given parser defines.[3]

```plaintext
parser start {
    return parse_ethernet;
}

parser parse_ethernet {
    extract(ethernet); // Extract ethernet headers
    return select(latest.ethertype) { // Choose next parser
        0x8100: parse_vlan;
        0x800: parse_ipv4;
        default: ingress;
    }
}

parser parse_vlan {
    extract(vlan);
    return select(latest.ethertype) {
        0x800: parse_ipv4;
        default: ingress;
    }
}

parser parse_ipv4 {
    extract(ipv4);
    return ingress;
}
```

In this simple example, we parse ethernet headers, then check and see if the ethernet type is `ipv4` or `vlan`. If it is `vlan`, we check the ethertype of the `vlan` header to see if we use `ipv4`. The reference to ingress is a call to the ingress control flow function, and indicates that parsing is over.
The graph shown above is a visual representation of the parse graph, with the parsing always proceeding downward. All incoming packets are eventually passed to the ingress control flow, and may include many different types of headers, which are parsed in order.

2.2.3 Table Definitions

Tables define the type of lookup to perform on incoming packets. They define the fields that are used as input, the actions that could be performed, and the dimensions of each of the tables. An example of a table is shown below.

```
table ipv4_table {
  reads {
    ipv4.protocol: exact;
    ipv4.srcAddr mask 0xFF000000: ternary;
  }
  actions {
    mark_visited;
    route_ipv4;
  }
}
```

The table definition specifies the headers that are read to determine actions. A separate user config file supplies how to match the parsed headers, and what actions to take based on those matches. A matched value can be matched to one value (exact), to a range of values (range), masked and then compared (ternary and the mask value), or can be checked to be valid (valid).[3]

In our example, we match the protocol exactly, and perform a masking operation on the source address before we match it. The values to match on are defined in the aforementioned user config file.
2.2.4 Actions

Actions are defined as actions that the tables perform on incoming packets. They are declared imperatively as functions. They can take parameters, which are defined by the config file. In addition to the values passed to the functions, an action has access to packet headers that have been extracted.[3]

Actions are built from a set of pre-defined primitive actions. Some primitive actions include add_header, modify_field, drop, and remove_header. An example of an action is shown below.

```plaintext
action route_ipv4(dst_mac, src_mac, vid) {
    modify_field(ethernet.dst_addr, dst_mac);
    modify_field(ethernet.src_addr, src_mac);
    modify_field(vlan_tag.vid, vid);
    add_to_field(ipv4.ttl, -1);
}
```

In this example, dst_mac, src_mac, and vid were passed to the action by the table. The action modifies some headers in the packet’s parsed representation, and then changes the time to live.

2.2.5 Control Flow

A control flow definition is defined as the layout and order of tables to be run over incoming packets. Each packet is processed by a sequence of match and action tables, and the control function may call action tables, call other control flow functions, or test certain conditions of the packet.[3]

A basic example of a control flow function is defined below.

```plaintext
control ingress {
    if(valid(vlan)) {
        apply(vlan_table);
    }
    if(valid(ipv4)) {
        if(ipv4.ttl > 0){
            apply(ipv4_table);
        } else {
            apply(drop_table);
        }
    }
}
```

Tables are applied by the control flow pipeline, and checks can be done in the pipeline to check fields of the parsed representation view of the packet. An example of checking these fields is done here, where we check the ttl of the ipv4 packet, and either apply our ipv4 table or a drop table.
The *valid* keyword checks if the given fields are valid for the incoming packet. This is done to make sure that the given fields are available to use in the called tables.

3 Libpaxos Bottleneck Analysis

The first thing we did was measure Libpaxos, an implementation of Paxos that allows the user to define different processes as generic replicas in a Paxos system, or individually as proposers, acceptors, or learners.[5] In our example, we decided to have each node be a replica, which would act as a proposer, acceptor, or learner as needed.

We ran the system on three machines, with two of them using an Intel 10-Gigabit SFI/SFP+ network card and the third using a Mellanox Technologies MT27700 card. A fourth machine passed client messages to one of the machines, who would act as a proposer for those rounds of Paxos.

The goal of these measurements was to detect bottlenecks in the algorithm. Specifically, which phases took the longest to process that could potentially be improved by moving the algorithm to the NFP-4000 cards. Libpaxos was run for approximately 10 minutes, and the average time to process the different phases of the algorithm were collected every 5 seconds.

The proposer had an average latency of 47.55 milliseconds from initial client request to learned response. Each of the other replicas had an average of 1.41 milliseconds from first prepare request to learned value. So the proposer was the highest bottleneck of the system, and a good place to target for modifications involving P4 and the NFP-4000 card.

We then broke down the work done by the proposer, and measured the average latency of each message type received by the proposer. Below is a graph of the average latencies of the different messages received by a proposer (promise, accepted, preempted, client proposals, or a message about acceptor state). Preempted and the acceptor state messages aren’t really found in this algorithm, because there is a single proposer, and no replicas crash, so there is never need for leader election.
The promise phase took the longest to process, and seemed like a good target for moving it to the NIC. The promise phase had a mean latency of 1.67 microseconds, almost double the 0.87 microseconds it took to process client requests.

## 4 Paxos in a Network Card

Our goal with this paper was to define a P4 program that would be run on the NFP-4000 devices. The original design was meant to be a modification of Libpaxos. It was intended to replace parts of the algorithm that were identified as higher latency or lower throughput. However, Libpaxos is built on Libevent, which buffers messages, and identifies message types based on data within the packet itself. Both of these made it difficult to utilize P4, which is designed to parse packet headers only, in the original design.

Instead, we designed a completely new implementation of Paxos within P4. A client sends a request, and then a P4 program running on the expected network card acts as a node in Paxos. In our prototype, there are only two different behaviors of replicas: Acceptors and Proposers.

### 4.1 Headers

These are the set of packet headers defined for our P4 program.
header_type ethernet_t {
    fields {
        ...
    }
}

header_type ipv4_t {
    fields {
        ...
    }
}

header_type udp_t {
    fields {
        ...
    }
}

header_type paxos_t {
    fields {
        ballot : 32;
        acceptor : 32;
        slot : 32;
        msg_type : 16;
        val : 32;
    }
}

Our implementation of Paxos uses the UDP protocol, so we define these headers so P4 can parse them.

For any Paxos related messages, we define these headers to be utilized by each phase of the algorithm.

msg_type is used to identify which phase of the algorithm was sent. P1A is prepare, P1B is promise, P2A is accept, and P2B is accepted. More numbers can be defined for other possible actions, such as any setup messages or informing a potential learner.

The table below shows how each phase uses these headers.
<table>
<thead>
<tr>
<th>Step</th>
<th>Message Type</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client request</td>
<td></td>
<td>val = proposed value</td>
</tr>
<tr>
<td>Prepare (P1A)</td>
<td>msg.type = P1A</td>
<td>val = client value, ballot = assigned ballot</td>
</tr>
<tr>
<td>Promise (P1B)</td>
<td>msg.type = P1B</td>
<td>ballot = highest ballot previously accepted, slot = highest slot previously accepted, acceptor = acceptor id of current acceptor</td>
</tr>
<tr>
<td>Accept (P2A)</td>
<td>msg.type = P2A</td>
<td>ballot = ballot of the round, slot = slot proposed by proposer, val = proposed client value</td>
</tr>
<tr>
<td>Accepted (P2B)</td>
<td>msg.type = P2B</td>
<td>ballot = ballot of the round, slot = slot accepted by acceptor, val = value accepted by acceptor, acceptor = id of acceptor</td>
</tr>
</tbody>
</table>

### 4.2 Acceptor

The acceptor works much like the acceptor described above. It accepts monotonically increasing ballot numbers, and refuses ballot numbers below any previously accepted ones.

Because we were designing the implementation entirely in P4, we needed to have our implementation handle all of the important parts of Paxos. The program would be entirely self-contained in the cards, so our acceptor needed to be able to correctly respond to prepare and accept messages, and correctly ignore any messages that had a ballot below any previously accepted values.

Our acceptor needed to make use of stateful memory available via P4, which is done in the form of registers. Our registers needed to store information about the current ballot number, and learned values. Because every phase of Paxos is now implemented directly at the hardware level, we can avoid any potential overhead needed to send information through PCIe buses. Also because of P4’s specialization within hardware, we hopefully have better performance than software written on a general purpose CPU.

This is the control flow of the acceptor code.

```plaintext
control ingress {
    if (valid(ipv4)) {
        apply(fwd_tbl);
    }

    if (valid(paxos)) {
        apply(find_ballot_tbl);
    }
```
if (paxos.ballot >= local_data.ballot) {
    apply(acceptor_tbl);
} else {
    apply(drop_tbl);
}

If it is a valid ipv4 packet, we perform the functions of a standard NIC. The packet’s destination is set to its intended destination, and forwarded either from the host to the net or from the net to the host. If the packet is not a paxos packet, our control stops there, and behaves like a simple NIC. If the packet is a paxos packet, however, we proceed with our algorithm.

The find_ballot_table loads the highest previously accepted ballot into a local variable. This variable is compared to the incoming ballot number, and if the ballot incoming is greater than or equal, the acceptor handles the packet. Otherwise, the packet is dropped.

The acceptor table is defined below.

```
table acceptor_tbl {
    reads {
        paxos.msg_type : exact;
    }
    actions {
        p1a;
        p2a;
        no_op;
    }
}
```

It checks the message type, and then performs the appropriate action. If the message type is P1A or P2A, it calls p1a or p2a respectively. If the message type is something else, it performs a no op.

The p1a action is defined below.

```
action p1a() {
    // Because we got here, we know this is the highest ballot
    // So we can safely set the ballot number
    // and highest previously accepted slot
    register_write(ballot_register, 0, paxos.ballot);
    register_read(paxos.slot, highest_slot, 0);

    register_read(paxos.acceptor, acceptor_id, 0);

    modify_field(paxos.msg_type, P1B);
    modify_field(ipv4.dst, ipv4.src);
}
```

The acceptor loads the proposed ballot number into the highest ballot register. It also sets the slot header to be the previously accepted slot.
number. It informs the proposer about its acceptor id, sets the message type to 1 (indicating a promise message) and sets the destination to the source, which responds back to the proposer with the now modified packet.

The p2a action is defined below.

```p4
action p2a() {
    register_write(ballot_register, 0, paxos.ballot);
    register_write(accepted_values, paxos.slot, paxos.val);
    register_write(highest_slot, paxos.slot, 0);

    register_read(paxos.acceptor, acceptor_id, 0);
    modify_field(paxos.msg_type, P2B);
    modify_field(ipv4.dst, ipv4.src);
}
```

The p2a action writes the ballot and slot to its set of "highest" values, and assigns the value to its slot number in the accepted values register. The accepted values register has a set of different values, which are used here as slots to define the values that have been accepted. Once again, the acceptor, msg_type, and destination are set like they were in the p1a action.

### 4.3 Proposer

The proposer also works as defined above. Much like the acceptor, the implementation entirely within P4 needs to handle all the different parts of Paxos. Our proposer acts as a learner, and simply informs the client when it has gotten a quorum of values from acceptors. Upon any client request, it sends out a prepare message to its acceptors.

The proposer must store more information than the acceptor. It needs to know the next ballot to use, as well as all the learned values.

The proposer should see a great increase in performance. Because all of the logic exists within the parser and match+action tables of the NIC, it can be performed quickly and efficiently within hardware. Each phase handled by the proposer now goes through a single pipeline written in P4.

This is the control flow of the proposer.

```p4
control ingress {
    if (valid(ipv4)) {
        apply(fwd_tbl);
    }

    if (valid(paxos)) {
        apply(find_ballot_tbl);
        // Not a cloned packet
        if (standard_metadata.instance_type == 0) {
            if (paxos.ballot > local_metadata.ballot) {
                apply(reset_paxos);
            }
        }
    }
}
```
else {  
  apply(proposer_tbl);
  if(local_metadata.promises_received == NUM_ACCEPTORS){
    apply(proposer_tbl);
  }
}

// A cloned packet. Send this to all the other acceptors.
else if(standard_metadata.instance_type == 2){
  apply(propagate_tbl);
}

}

In order to send messages to all acceptors, the proposer makes use of packet cloning. A packet can be cloned either from the ingress or egress. Here, we clone packets from egress back to ingress, so we can propagate them to all known acceptors. The propagate table simple modifies the destination to the next known acceptor defined in a list. Once it sends it to all acceptors, it stops modifying that particular packet.

The propagate table is defined below.

```
table propagate_tbl {
  reads {
    local_metadata.acceptor_num : exact;
  }
  actions {
    send_to_next_acceptor;
    drop_act;
  }
}
```

The local variable `acceptor_num` keeps track of the next acceptor to send it to. The `send_to_next_acceptor` function is very simple.

```
action send_to_next_acceptor() {
  register_read(ipv4.dst, acceptors, local_metadata.acceptor);
  register_write(packets_sent, 0, local_metadata.acceptor + 1);
  clone.egress_pkt_to_ingress();
}
```

The proposer simply sets the destination port to the next acceptor, and increases the register `packets_sent`. If this register is more than the number of acceptors, the proposer doesn’t call this function anymore.

The proposer table handles the different parts of the Paxos algorithm.

```
table proposer_tbl {
  reads {
    paxos.msg_type : exact;
    local_metadata.promises_received;
  }
  actions {
    client_request;
  }
}
```
p1b;
send_p2a;
p2b;
setup_client;
reply_to_client;
}

Setup_client is a setup method, sent by some program (potentially the client) to inform the proposer of other acceptors and information about the client.

The client_request function handles client requests.

```java
action client_request () {
    register_write(ballot_register, 0, paxos.ballot);
    register_read(local_metadata.client_request_num, client_number, 0);
    register_write(client_number, 0, local_metadata.client_request_num + 1);
    register_write(packets_sent, 0, 0);
    modify_field(paxos.msg_type, P1A);
    clone_egress_pkt_to_ingress();
}
```

The proposer assigns the next ballot in its sequence, stores the client request, and sets the message type to be P1A. The message is then cloned, so it can be propagated to all the acceptors.

Upon receiving a promise message, the proposer calls the action p1b.

```java
action p1b(){
    modify_field(local_metadata.acceptor_num, paxos.acceptor);
    register_write(acceptor_ballots, paxos.acceptor, paxos.ballot);
    register_read(local_metadata.slot_num, next_slot, 0);
    register_write(next_slot, 0, max(local_metadata.slot_num, paxos.slot));
    modify_field(local_metadata.promises_received, local_metadata.promises_received + 1);
}
```

It updates the next available slot based on the response, stores the ballot in a register labeled for acceptors ballots. If promises_received is equal to number of acceptors, the proposal table is called again, and the proposer creates an accept message to send.

```java
action send_p2a(){
    modify_field(paxos.msg_type, P2A);
    register_read(paxos.slot, next_slot, 0);
    clone_egress_pkt_to_ingress();
}
```
The proposer assigns the correct message type, slot number, and then clones it to be propagated to all acceptors.

Upon receiving an *accepted* message, the proposer knows that a value was learned, and it can store it and inform the client.

```plaintext
action p2b()
    modify_field(local_metadata.accepteds_received,
                 local_metadata.accepteds_received + 1);
}
```

When it has learned a value by receiving from a quorum of acceptors, it can reply to the client.

```plaintext
action reply_to_client()
    register_read(local_metadata.client, client_address, 0);
    register_write(learned_values, paxos.slot, paxos.val);
    modify_field(local_metadata.promises_received,
                 local_metadata.promises_received + 1);
    register_read(ipv4.dst, client_number, 0);
}
```

The proposer acts as a learner in this instance. When it learns of a learned value, it stores it and responds to the client with the proposed value and slot set in the paxos headers.

### 4.4 Integration with Software

While the entirety of the algorithm runs on hardware, our solution relies on a client which runs at the application level. The client sends initial messages and receives values from the proposer when values are learned. The client must craft its initial messages to contain the *paxos* protocol described above. This can be done simply using a python script with the scapy package, as shown below.

```plaintext
p = IP(src="127.0.0.1", dst="128.83.143.68") / UDP(sport=55000, dport=0x6000)
p = p / Paxos(ballot=0, acceptor=0, slot=0, msg_type=0, val=100)
```

This sends a client request message to the address 128.83.143.68, and has a proposed value of 100. We can identify a Paxos client message by the destination port value of 6000. If the proposer receives a message on this port, it can check the Paxos headers to see if it is a valid client proposal.

The client, in our simple example, waits for any responses from the proposer to indicate that it has learned a value and a slot number. A more sophisticated application could execute these instructions once they are learned, or have a separate learner that performs some other task. For now, the client simply sends requests and listens for learned values.
4.5 Implementation of Learners

We also decided to have proposers act as learners. This was done for a number of reasons, most importantly being that we wanted to offload the entire algorithm to hardware. A notable bottleneck of Dang et. al’s work was the learner, which was still running at the application layer.[9] We instead have our proposers act as learners. They store learned values in the correct slots using the NICs on-board memory. From there, the proposer can act on the learned values as needed. In our example, we simply inform the client.

It would be an easy modification to move the learner logic to a separate node. This could be done if the learner needed to perform some other action once values were assigned to slots. By keeping the learner logic entirely within the hardware, we can utilize the same speed up expected by moving code to hardware.

5 Lack of Results

Unfortunately, due to circumstances outside of our control, we weren’t able to run our P4 code on the NICs themselves. The servers we were using supposedly supported hardware virtualization and these specific cards, but we were unable to enable virtualization of the cards. When attempting to create hardware virtualization of the cards, the OS would fail to recognize the virtual interface of the cards, and we had no way of accessing them. We found after much testing that the servers did not actually support the hardware virtualization that we needed to be able to access the cards.

We were able to compile and upload the P4 code as firmware to run on the NICs, but we were never able to run it and compare the benchmarks to the Libpaxos implementation. This would be the final part of our goal, and could give an indication that this is a valid alternative implementation of consensus.

6 Future Work

The most obvious next step is to successfully execute the programs on a working server and collect benchmarks about its performance. This would allow us to actually determine if this is a viable implementation of Paxos, and if moving it to the hardware level gives us significant performance improvements.

Outside of the obvious, the proposed implementation has some drawbacks. It is a very limited application that can only run a single proposer and a set of acceptors. Also, each node is designated as either a proposer or acceptor, without the ability to perform other tasks. In the event that the proposer crashes, it is impossible to have another node take up the tasks
The proposer is also unable of modifying the acceptors it knows about. It requires a setup message at the beginning to initialize the connections to all the other proposers and the client.

The proposed implementation also requires certain information to be written in packet headers. In an environment that requires either large client proposals or variable length data, this may not be a practical solution. Because P4 can only inspect the headers of a packet, it is fairly limited in how it could be used for more real world applications.

7 Related Work

Istvan et al. described a hardware implementation of Zookeeper’s atomic broadcast using an FPGA. Atomic broadcast is a form of consensus that ensures that all participants receive messages reliably and in the same order. Istvan wrote the algorithm to utilize the TCP protocol to implement atomic broadcast. The found that moving the algorithm to hardware represented the best case scenario for all messages, as latencies have a defined upper bound and are very predictable. This allowed them to achieve a significant performance speed up when compared to similar software solutions.[6]

Microsoft’s Project Catapult was an effort to improve performance in response to diminishing CPU improvements. They targeted various different hardware devices, and currently uses a customizable FPGA. Microsoft uses an FPGA at each data center node that can be harnessed for services as an add-on to existing data center nodes. Instead of acting as a separate network, the FPGAs are reconfigurable and can be customized as needed. It is now running in nearly every new production server.[7]

The NetFPGA project is an open source hardware and software project that is designed to give researchers the ability to rapidly prototyping computer network devices. It allows users to develop designs to process packets at line-rate.[8] Dang et al. used NetFPGA to build a consensus implementation as a network service, where they found reduced latency and increased throughput when compared to a software implementation, namely Libpaxos.[9]

Our solution differs from Istvan et al. in that it utilizes the UDP protocol, which relaxes assumptions about synchronization, and it implements Paxos instead of atomic broadcast. We are different from Microsoft’s Project Catapult in that we are implementing our version of Paxos on a NIC instead of an FPGA, and is a single purpose design instead of a reconfigurable setup.

Our implementation is similar to Dang et al.’s solution, but we have our proposer act as a learner and as a coordinator and inform the client about any learned value. Dang et al.’s implementation used an FPGA for coordinators and acceptors, but found a bottleneck still existed in the
application layer, where they implemented learners and proposers. Our solution is aimed to move the entirety of the algorithm to hardware.

8 Conclusion

P4 and customizable NICs could potentially be a solution for improving performance of consensus implementations. It allows the NIC to perform tasks and react according to different messages, and because it is running on the hardware, it likely has much better performance. It generally could be used to improve the performance of Paxos, or at least the parts identified as bottlenecks.

However, we were unable to measure the performance of this implementation, so it’s still unclear if this is a viable alternative. Given working hardware, we could accurately test the algorithm to see its improvements.

References


