Dataplane Programming
Outline

- Example use case
- Introduction to data plane programming
- P4 language
Example Use Case: Paxos in the Network
The Promise of Software Defined Networking

- Increased “network programmability” allows ordinary programs to manage the network.

- Applications can leverage SDNs to improve performance through data plane configuration (e.g., route selection and QoS).

- Can application logic be moved into the network?

- This work focuses on the widely-deployed Paxos protocol.
Why Paxos?

- Paxos is a fundamental building block for distributed applications
  - e.g., Chubby, OpenReplica, and Ceph
- There exists extensive work on optimizing Paxos (e.g., Fast Paxos)
- Paxos operations can be efficiently implemented in hardware
Outline of This Talk

- Motivation
- Paxos Background
- Consensus in the Network
  - Paxos in SDN Switches (and required OpenFlow extensions)
  - Alternative consensus protocol (without OpenFlow changes)
- Evaluation
- Conclusions
Paxos Background
**Paxos Protocol**

- **Goal:** Have a group of participants agree on one result (i.e., consensus)
- Protocol proceeds in rounds, each round has two phases
- Performance is measured in message hops
- Classic Paxos requires 3 hops
Paxos Protocol

**Goal**: Have a group of participants agree on one result (i.e., consensus)

- Protocol proceeds in rounds, each round has two phases
- Performance is measured in message hops
- Classic Paxos requires 3 hops

Issues: Proposer ➔ Coordinator ➔ Acceptor ➔ Learner

Issues requests
Paxos Protocol

- **Goal:** Have a group of participants agree on one result (i.e., consensus)
- Protocol proceeds in rounds, each round has two phases
- Performance is measured in message hops
- Classic Paxos requires 3 hops
Paxos Protocol

**Goal:** Have a group of participants agree on one result (i.e., consensus)

- Protocol proceeds in rounds, each round has two phases

- Performance is measured in message hops

- Classic Paxos requires 3 hops
Paxos Protocol

- **Goal**: Have a group of participants agree on one result (i.e., consensus)
- Protocol proceeds in rounds, each round has two phases
- Performance is measured in message hops
- Classic Paxos requires 3 hops
Paxos Protocol

**Goal:** Have a group of participants agree on one result (i.e., consensus)

- Protocol proceeds in rounds, each round has two phases
- Performance is measured in message hops
- Classic Paxos requires 3 hops

*Proposer* → *Coordinator* → *Acceptor* → *Learner*
**Paxos Protocol**

- **Goal**: Have a group of participants agree on one result (i.e., consensus)
- Protocol proceeds in rounds, each round has two phases
- Performance is measured in message hops
- Classic Paxos requires 3 hops
Paxos Protocol

- **Proposer**
- **Coordinator**
- **Acceptors**
- **Learner**

- **Goal:** Have a group of participants agree on one result (i.e., consensus)
- **Protocol proceeds in rounds, each round has two phases**
- **Performance is measured in message hops**
- **Provides replication**

- Classic Paxos requires 3 hops
Paxos Protocol

**Goal**: Have a group of participants agree on one result (i.e., consensus)

- Protocol proceeds in rounds, each round has two phases
- Performance is measured in message hops
- Classic Paxos requires 3 hops
Fast Protocol

- **Key idea**: Optimize for the case when proposals don’t collide
- Optimistically uses *fast rounds* that bypass coordinator
  - Only 2 message hops
  - Requires 1 more acceptor
- If there is collision, revert to Classic Paxos
Observations

- Performance metric (hops) does not account for network topology
  - 1 “classic” message hop has to travel through multiple switches
- Coordinators and acceptors are typically bottlenecks
  - Must aggregate or multiplex messages
- “Fault tolerance” does not include the network devices
Moving Paxos into the Network
Paxos on Network Devices

Key idea: Move coordinator and acceptor (phase 2) logic into switches

OpenFlow API not sufficient
Required Extensions

- Proposer
- Coordinator
- Acceptor
- Learner
Required Extensions

1. Generate round and sequence numbers
Required Extensions

1. Generate round and sequence numbers
2. Persistent storage
3. Stateful comparisons
4. Storage cleanup
## Hardware Feasibility

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Device</th>
<th>Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round &amp; Sequence Generator</td>
<td>Netronome NFP-6xxx, NetFPGA, Arista 7124FX</td>
<td>Stateful flow processing</td>
</tr>
<tr>
<td>Stateful Comparisons</td>
<td>Arista 7124FX</td>
<td>50 GB SSD logging</td>
</tr>
<tr>
<td>Persistent Storage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Storage Cleanup</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Consensus without OpenFlow Extensions
Can We Avoid Extensions?

- Make optimistic assumptions about message order
- Like Fast Paxos, have “fast” and classic rounds
No Sequence Numbers Needed

- Rely on optimistic message ordering
- Use switch to increase probability
No On-Device State Needed

Use additional servers as storage
No On-Device Logic Needed

- No coordinator
- Acceptors always “accept”
- Need an additional “minion”
Performance Assumption

- Messages from serializer to minion are in the same order
- If not, execute slow round
Correctness Assumption

- Messages from minion to servers are in same order
- If not, protocol breaks
NetPaxos Summary

- **Latency**: Fewer “true” network hops, including switches
- **Throughput**: Avoids potential bottlenecks, reduced logic
- **Fault tolerance**: Serializer can easily be made redundant, other devices can fail, as with Classic Paxos
Evaluation
Experiments

Focus on two questions:

- Do our ordering assumptions hold?
- What is the *potential* benefit of NetPaxos?

Testbed:

- Three Pica8 Pronto 3290 switches, 1Gbps links
- Send messages of 1 MTU size, with sequence numbers
Assumptions Mostly Hold

- Performance assumption held up to 70% link capacity
- Correctness assumption was never violated
- Traffic should not be bursty
High Potential for Performance

Disclaimer: Best case scenario
- 9x increase in throughput
- 90% reduction in latency
Outlook

- Formalizing protocol with Spin model checker
- Implementing a NetFPGA-based prototype
- Investigating root causes for packet reordering
Conclusions

- SDNs enable tight integration with the network, which can improve distributed application performance.
- Proposed two approaches to moving Paxos logic into the network.
- Paxos is a fundamental protocol. Performance improvements would have a great impact on data center applications.
Introduction to Data Plane Programming
SDN is Not Enough

- SDN allows you to program the control plane
- Many large data centers program host network stacks (hypervisors), roughly edge-based SDN
- Not yet able to program the data plane
Data Plane Opportunities

- Simplify and improve network management
  - Extensions for debugging and diagnostics
  - Dynamic resource allocation
- Enable critical new features
  - Improved robustness
  - Port-knocking
  - Load balancing, enhanced congestion control
Suppressing Innovation

- OpenFlow provides an (intentionally) limited interface
  - No state
  - No computation
  - Restricted to a fixed set of headers
- May need to customize hardware support
  - Match tables usually have fixed width, depth, and execution order
Demand for New Features

- SDNs and white boxes set the stage
- Large private networks want new features
- Rate of new feature arrivals exceeds rate of hardware evolution
No DIY solution. Must work with vendors at the “feature” level

- Hard to get consensus on the feature
- Long time to realize the feature
- Need to buy the new hardware
- What you get is not usually what you want
Extensions to OpenFlow

- OpenState project, G. Bianchi et al.
  - http://openstate-sdn.org
- Mealy machine abstraction

ONF Working Groups
- EXT-WG focused on extensions
- FAWG focused on forwarding abstractions
Vision

- What about the next version of OpenFlow? Or custom protocol?
- We all know how to program CPUs
  - Supporting tools and infrastructure
  - Allows fast iteration and differentiation
  - Let’s you quickly realize your own ideas
- Challenge: How can we replicate this in the network?
## Networking is Late to the Game

<table>
<thead>
<tr>
<th>Domain</th>
<th>Target Hardware</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computers</td>
<td>CPU</td>
<td>C, Java, OCaml, JavaScript, etc.</td>
</tr>
<tr>
<td>Graphics</td>
<td>GPU</td>
<td>CUDA, OpenCL</td>
</tr>
<tr>
<td>Cellular Base Station</td>
<td>DSP</td>
<td>C, MATLAB, Data Flow languages</td>
</tr>
<tr>
<td>Networks</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Two Questions

- What do we need at the hardware level?
- Once we have that, how do we program it?
Hardware Trend

- PISA (Protocol Independent Switch Architecture)
  - Fundamental departure from switch ASICS
  - Programmable parsing
  - Protocol independence (i.e., generic match-action units)
  - Parallelism across multiple match-action units, and within each stage
- Power, size, or cost penalty is negligible
Why Now?

- I/O, memory, and bus dominate chip size
- Logic is getting proportionally smaller
- Programmability means larger logic. The rest stays the same.
- Very little power, area, or performance penalty for programmability.
PISA Chip

Logical Dataplane View

Switch Pipeline

Parser

Match Table  Action Macro

Match Table  Action Macro

Match Table  Action Macro

Match Table  Action Macro

Queues

ACL
PISA Chip

Mapping Logical Dataplane Design to PISA Chip

Logical Dataplane View

Switch Pipeline

Parser
L2 Table
L2 Action Macro
IPv4 Table
IPv4 Action Macro
IPv6 Table
IPv6 Action
ACL Table
ACL Action Macro

Queues
PISA Chip

Re-configurability

Logical Dataplane View
Switch Pipeline

Parser
L2 Table
L2 Action Macro
MyEncap
IPv4
Action 1
IPv4
Action 2
IPv6
Action 3
ACL Table
ACL Action Macro
Queues
PISA Chip

**PISA** (Protocol Independent Switch Architecture)
- Parallelism across pipelined stages
- Parallelism within each stage
Key Players

- Hardware manufacturers
  - Proto-PISA chips already available: FlexPipe (Intel) and others (Cisco and Cavium)
  - Full-fledged PISA chips on the horizon (Barefoot)
- High-level language
  - P4 (p4.org)
- Compiler and development tools
  - “Hour-glass design” with IR, profilers, debuggers
Abstract Forwarding Model

Parser

Hdr Fields

Ingress Stages

Match Action

Match Action

Match Action

Match Action

Match Action

Egress Stages

Match Action

Match Action

Match Action

Match Action

Match Action
P4 Language Components

- **Parser Program**
  - State-machine;
  - Field extraction

- **Match Tables + Actions**
  - Table lookup and update;
  - Field manipulation;
  - Control flow

- **Control Flow**

- **Assembly (“deparser”) Program**
  - Field assembly

No: memory (pointers), loops, recursion, floating point
P4 Examples

- Header Fields
- Parsing
- Tables
- Actions
- Control Flow
Header Field and Parsing

```c
header_type ethernet_t {
  fields {
    dstAddr : 48;
    srcAddr : 48;
    etherType : 16;
  }
}

parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType) {
    0x8100 : parse_vlan;
    0x800  : parse_ipv4;
    0x86DD : parse_ipv6;
  }
}
```
Table (Match)

table ipv4_lpm
{
   reads {
      ipv4.dstAddr : lpm;
   }
   actions {
      set_next_hop;
      drop;
   }
}

<table>
<thead>
<tr>
<th>ipv4.dstAddr</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.*</td>
<td>drop</td>
</tr>
<tr>
<td>10.0.0.*</td>
<td>set_next_hop</td>
</tr>
<tr>
<td>224.*</td>
<td>drop</td>
</tr>
<tr>
<td>192.168.*</td>
<td>drop</td>
</tr>
<tr>
<td>10.0.1.*</td>
<td>set_next_hop</td>
</tr>
</tbody>
</table>
Actions

<table>
<thead>
<tr>
<th>ipv4.dstAddr</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.*</td>
<td>drop</td>
</tr>
<tr>
<td>10.0.0.*</td>
<td>set_next_hop</td>
</tr>
<tr>
<td>224.*</td>
<td>drop</td>
</tr>
<tr>
<td>192.168.*</td>
<td>drop</td>
</tr>
<tr>
<td>10.0.1.*</td>
<td>set_next_hop</td>
</tr>
</tbody>
</table>

```java
action set_next_hop(nhop_ipv4_addr, port)
{
    modify_field(metadata.nhop_ipv4_addr, nhop_ipv4_addr);
    modify_field(standard_metadata.egress_port, port);
    add_to_field(ipv4.ttl, -1);
}
```
control ingress
{
    apply(port);
    if (valid(vlan_tag[0])) {
        apply(port_vlan);
    }
    apply(bridge_domain);
    if (valid(mpls_bos)) {
        apply(mpls_label);
    }
    retrieve_tunnel_vni();
    if (valid(vxlan) or valid(genv) or valid(nvgre)) {
        apply(dest_vtep);
        apply(src_vtep);
    }
}
Compilation

Parser Program | Match Tables + Actions | Control Flow

Compiler

PISA Chip
Protocol API

My (running) switch

MPLS (source)

Parser Program

Match Tables + Actions

Control Flow

Compiler

Forwarding

Table Population

Switch Program API

Switch Runtime API

MPLS-TE

Switch Driver

Linux

Add/delete

Program

PISA Chip