CS434/534: Topics in Network Systems

High-Level Programming for Programmable Networks: Distributed Network OS: Replicated Data Store

Yang (Richard) Yang
Computer Science Department
Yale University
208A Watson
Email: yry@cs.yale.edu

http://zoo.cs.yale.edu/classes/cs434/
Outline

- Admin and recap
- Controller software framework (network OS) supporting programmable networks
  - architecture
  - data model and operations: OpenDaylight as an example
  - distributed data store
    - overview
    - basic Paxos
    - multi-paxos
Recap: Proactive HL Programming

Objective: Proactive, fully generate multi-table datapath pipeline

Basic ideas:

- A function $f$ consists of a sequence of instructions $I_1, I_2, \ldots, I_n$, and one can
  - (conceptually) consider each instruction in the dataflow as a table
  - identify instructions with compact representation (symbolic analysis and direct state plug in)
  - use state propagation to compute compact tables
Map<MAC, Switch> hostTable; // {11->sw1, 22->sw2, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}

0. Route onPacketIn(Packet p) {
    L1. Switch dstSw = hostTable.get(p.dstMac());
    L2. Cond dstCond = condTable.get(p.dstMac());
    L3. Route aRoutes = AllPairCondRoutes();
    L4. return aRoutes.get(dstSw, dstCond);
}

<table>
<thead>
<tr>
<th>Pri</th>
<th>p.dstMac</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11</td>
<td>dstCond=S</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>dstCond=C</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>dstCond=C</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>dstCond=UK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pri</th>
<th>p.dstMac</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11</td>
<td>dstSw=sw1</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>dstSw=sw2</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>dstSw=sw2</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>dstSw=swu</td>
</tr>
</tbody>
</table>

L1, EMPTY

L2, dstSw=sw1, dstMac=11

L2, dstSw=sw2, dstMac=22

L2, dstSw=sw2, dstMac=33

L2, dstSw=swu, dstMac=*  

L4, dstSw=sw1, dstCond=S

L4, dstSw=sw2, dstCond=C

L4, dstSw=swu, dstCond=UK

L∞, aRoutes(sw1, S)

L∞, aRoutes(sw2, C)

L∞, aRoutes(swu, UK)
Pipeline design for constraints and optimization

g1 = dstPort < 1025

dstSw = SW_FW
dstCond = V
egress = Drop
dstSw = hostTable(dstMac)
dstCond = condTable(dstMac)

phi(dstSw)
phi(dstCond)

srcSw = hostTable(srcMac)
egress = f(dstCond, dstSw, srcSw)
Recap: HL NOS Architecture

Key goal: provide applications with high-level views and make the views highly available (e.g., 99.99%), scalable.

Key component - data store:
- Data model
- Data access model
- Data store availability

Logically centralized data store

Program

Network View
Service/Policy

NE Datapath
NE Datapath
Recap: OpenDaylight Software Architecture
Recap: Data Model - Yang Data Tree

Recap: Data Operation Model - Transactions

- Operations included in transactions

```java
ReadWriteTransaction transaction = dataBroker.newReadWriteTransaction();
Optional<Node> nodeOptional;
nodeOptional = transaction.read(
    LogicalDataStore.OPERATIONAL,
    n1InstanceIdentifier);
transaction.put(
    LogicalDataStore.CONFIG,
    n2InstanceIdentifier,
    topologyNodeBuilder.build());
transaction.delete(
    LogicalDataStore.CONFIG,
    n3InstanceIdentifier);
CheckedFuture future;
future = transaction.submit();
```
Recap: Multi-Server NOS

Discussion: why multiple servers?
Recap: A Common Multi-Server (Data Store) Arch: Replicated Log

- Replicated operations log => replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication
- System makes progress as long as any majority of servers are up
Outline

- Admin and recap
- Controller framework supporting programmable networks
  - architecture
  - data model and operations: OpenDaylight as an example
  - distributed data store
Two general approaches to achieving replicated log (consensus in general):

- **Symmetric, leader-less:**
  - All servers have equal roles
  - Clients can contact any server
  - We will see Basic Paxos as an example

- **Asymmetric, leader-based:**
  - At any given time, one server is in charge, others accept its decisions
  - Clients communicate with the leader
  - We will see Raft, which uses a leader
  - OpenDaylight is based on Raft
Outline

- Admin and recap
- Datapath programming
- Controller framework supporting programmable networks
  - architecture
  - data model and operations: OpenDaylight as an example
  - distributed data store
    - overview
    - basic Paxos
Warning

- Leslie Lamport, 1989
- Nearly synonymous with consensus

“The dirty little secret of the NSDI community is that at most five people really, truly understand every part of Paxos ;-).” —NSDI reviewer

“There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system...the final system will be based on an unproven protocol.” —Chubby authors
The Paxos Approach

Basic Paxos ("single decree"):  
- One or more servers propose values (what is a value in our context?)  
- System must agree on a single value (what is chosen in our context?)  
  - Only one value is ever chosen

Multi-Paxos:  
- Combine several instances of Basic Paxos to agree on a series of values forming the log
Requirements for Basic Paxos

- **Safety:**
  - Only a single value may be chosen
  - A server never learns that a value has been chosen unless it really has been

- **Liveness** (as long as majority of servers up and communicating with reasonable timeliness):
  - Some proposed value is eventually chosen
  - If a value is chosen, servers eventually learn about it
Paxos Components

- Proposers:
  - Handle client requests
  - Active: put forth particular values to be chosen

- Acceptors:
  - Passive: respond to messages from proposers
  - Responses represent votes that form consensus
  - Store chosen value, state of the decision process

For this class:
- Each Paxos server contains both components
Outline

- Admin and recap
- Controller framework supporting programmable networks
  - architecture
  - data model and operations: OpenDaylight as an example
  - distributed data store
    - overview
    - basic Paxos
      - structure
      - initial design options
Design I: A Single Acceptor

- A single acceptor chooses value

Problem:
- What if acceptor crashes after choosing?

Solution:
- Quorum: multiple acceptors (3, 5, ...)
- As long as majority of acceptors available, system can make progress
**Design II**

- **Design decision: which value to accept?**
  - Design: Acceptor accepts **only** first value it receives

- **Problem:**
  - Split votes: if simultaneous proposals, no value might have majority

=> **An Acceptor must be able to sometimes “change mind” to accept multiple (different) values**
Design III

- Design decision: which value to accept?
  - Design: Acceptor accepts *every* value it receives

- Problem:
  - Conflicting choices

Solution: If a value has been chosen => future proposals propose/choose that same value
Outline

- Admin and recap
- Controller framework supporting programmable networks
  - architecture
  - data model and operations: OpenDaylight as an example
  - distributed data store
    - overview
    - basic Paxos
      - structure
      - initial design options
      - design details
Basic Paxos Protocol Structure

Two-phase approach:
- **Phase 1**: broadcast Prepare RPCs
  - Find out about any chosen values
  - Block older proposals that have not yet completed
- **Phase 2**: broadcast Accept RPCs
  - Ask acceptors to accept a specific value
Proposal Numbers

- Each proposal has a unique number
  - Higher numbers take priority over lower numbers
  - It must be possible for a proposer to choose a new proposal number higher than anything it has seen/used before

- One simple approach:

  Proposal Number
  
<table>
<thead>
<tr>
<th>Round Number</th>
<th>Server Id</th>
</tr>
</thead>
</table>

  - Each server stores maxRound: the largest Round Number it has seen so far
  - To generate a new proposal number:
    - Increment maxRound
    - Concatenate with Server Id
  - Proposers must persist maxRound on disk: must not reuse proposal numbers after crash/restart
Basic Paxos Protocol

**Proposers**

1) Choose new proposal number $n$
2) Broadcast Prepare($n$) to all servers

4) When responses received from majority:
   - If any acceptedValues returned, replace value with acceptedValue for highest acceptedProposal

5) Broadcast Accept($n$, value) to all servers

6) When responses received from majority:
   - Any rejections (result > $n$)? goto (1)
   - Otherwise, value is chosen

**Acceptors**

3) Respond to Prepare($n$):
   - If $n > \text{minProposal}$ then $\text{minProposal} = n$
   - Return(acceptedProposal, acceptedValue)

6) Respond to Accept($n$, value):
   - If $n \geq \text{minProposal}$ then
     - acceptedProposal = minProposal = $n$
     - acceptedValue = value
   - Return(minProposal)

Acceptors must record minProposal, acceptedProposal, and acceptedValue on stable storage (disk)
Basic Paxos Protocol: Exercise

- Basic setting: single proposer
  - s1 proposes X
Question

- Q: Who knows that a value has been chosen?
- Q: If other servers want to know the chosen value, what should they do?
### Basic Paxos Protocol: Exercise

- **Setting:** A new proposal arrives after a value is already chosen.

  - **Diagram:**
    - **X** → **S1**: Prepare proposal 3.1 (from **s1**).
    - **S2**: Proposal 3.1, Accept 3.1 X
    - **S3**: Proposal 3.1, Accept 3.1 X
    - **S4**: Proposal 4.5, Accept 4.5 X
    - **S5**: Proposal 4.5, Accept 4.5 X

  - **Messages:**
    - "Accept proposal 4.5 with value X (from **s5**)."
Basic Paxos Protocol: Race Condition I

- Previous value not chosen, but new proposer sees it:
  - New proposer will use existing value
  - Both proposers can succeed
Basic Paxos Protocol: Race Condition II

- Previous value not chosen, new proposer doesn’t see it:
  - New proposer chooses its own value
  - Older proposal blocked
Summary of Cases

Three possibilities when later proposal prepares:

1. Previous value already chosen:
   - New proposer will find it and use it
2. Previous value not chosen, but new proposer sees it:
   - New proposer will use existing value
   - Both proposers can succeed
3. Previous value not chosen, new proposer doesn’t see it:
   - New proposer chooses its own value
   - Older proposal blocked
Liveness

- Competing proposers can livelock:

- Potential solutions:
  - randomized delay before restarting, give other proposers a chance to finish choosing
  - use leader election instead
Outline

- Admin and recap
- Controller framework supporting programmable networks
  - architecture
  - data model and operations: OpenDaylight as an example
  - distributed data store
    - overview
    - basic Paxos
    - multi-Paxos
Multi-Paxos

- Separate instance of Basic Paxos for each entry in the log:
  - Add `index` argument to `Prepare` and `Accept` (selects entry in log)

1. Client sends command to server
2. Server uses Paxos to choose command as value for a log entry
3. Server waits for previous log entries to be applied, then applies new command to state machine
4. Server returns result from state machine to client
Multi-Paxos Issues

- Which log entry to use for a given client request?
- Performance optimizations:
  - Ensuring full replication
  - Client protocol
  - Configuration changes

Note: Multi-Paxos not specified precisely in literature
Selecting Log Entries

- When request arrives from client:
  - Find first log entry not known to be chosen
  - Run Basic Paxos to propose client’s command for this index
  - Prepare returns acceptedValue?
    - Yes: finish choosing acceptedValue, start again
    - No: choose client’s command

Logs Before

Logs After
Selecting Log Entries, cont’d

- Servers can handle multiple client requests concurrently:
  - Select different log entries for each
- Must apply commands to state machine in log order
What are inefficiencies of the concurrent, leaderless multi-paxos:

- With multiple concurrent proposers, conflicts and restarts are likely (higher load → more conflicts)
- 2 rounds of RPCs for each value chosen (Prepare, Accept)
Prepare for Next Class

- Stare at Paxos and play with examples to better understand it
- Read Raft
- Start to think about projects