CS434/534: Topics in Networked (Networking) Systems

Distributed Network OS

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
- High-level datapath programming
  - blackbox (trace-tree)
  - whitebox
- Network OS supporting programmable networks
  - overview
  - OpenDaylight
  - distributed network OS (Paxos, Raft)
Admin

- PS1 posted on the Schedule page
- Make sure to make appointment w/ me to discuss ps1
Recap: Network OS Design Goals

- Provide applications w/ high-level views
  - Make the views highly available (e.g., 99.99%)
  - Make view access and update highly-scalable

- Allow/control the interactions among applications
Recap: OpenDaylight

- Not a fixed model/view, but models described by Yang and loaded into the system by OSGi

- Yang:
  - a modeling language describing data as a tree, as well as
  - RPCs and notifications
  - Based on RFC 6020

- Module system
  - OSGi:
    - Allows dynamically loading bundles
    - Allows registering dependencies and services exported
    - For exchanging information across bundles
  - Karaf: Light-weight Runtime for loading modules/bundles
    - OSGi based. Primary distribution mechanism since Helium

Diagram:
- FeatureA
- FeatureB
- SAL
- Karaf
- OSGi Framework (Equinox)
Recap: Yang Data Tree

Recap: Interactions (3 Brokers)

Data Broker
- put
- notify
- store

RPC Broker

Notification Broker
- publish
- notify
Recap: DataChangeListener

dataBroker.registerDataChangeListener(
    LogicalDatastoreType.CONFIGURATION,
    myInstanceId,
    myDataChangeListener,
    DataChangeScope.SUBTREE);

myDataChangeListener

AsyncDataChangeEvent
created  deleted  updated  original

transaction

Datastore

Recap: DataChangeListener
RPC Messages

- Using RPCs to:
  - Send a message
  - Receive a response
  - Asynchronously
  - Without knowledge of provider of implementation

- RPCs come in two flavors:
  - Global - One receiver
  - Routed - One receiver per context
Sending an Asynchronous RPC Message

```java
HelloService helloService = session.getRpcService(HelloService.class);

Future<RpcResult<HelloWorldOutput>> future = helloService
    .helloWorld(helloWorldInput);

while(! future.isDone()) {
    /* Do other work */
}

HelloWorldOutput helloWorldOutput = future.get().getResult();
```

```
getRpcService()
return: helloService

helloWorld(helloWorldInput)
return: future
isDone()
false
isDone()
true
get()
set(helloOutput)

return: RpcResult<HelloWorldOutput>
```
Global RPCs – Processing a Message – Async

```
public class HelloWorldImpl
    implements HelloService {
    public HelloWorldImpl(ProviderContext session){
        session.addRpcImplementation(
            HelloService.class,
            this);
    }
    @Override
    public Future<RpcResult<HelloWorldOutput>>
        helloWorld(HelloWorldInput input) {
        SettableFuture future = new SettableFuture();
        process (input,future);
        return future;
    }
}
```
public class HelloWorldImpl implements HelloService {
    /*
     * see previous slide for
     * calls to addRpcImplementation
     * and the helloWorld method
     */
    private process(HelloWorldInput input, SettableFuture future) {
        /* process in new thread */
        future.set(RpcResultBuilder
            .success(helloWorldOutput)
            .build());
    }
}
Routed RPCs: What are They?

- **A Unicast Message**
  - Well defined Input/Output
  - Processor is context dependent

- **Input includes 'Context'**
  - InstanceIdentifier
    - Pointer to a place in the tree defining message context
  - Consumer is unaware RPC is routed

- **Registration includes 'Context'**

- **MD-SAL 'routes' to correct message processor for 'Context'**
**Clustering RPCs**

- **RPCs**
  - Routed across the cluster
Recap: Software Architecture

Model-Driven SAL (MD-SAL)

Controller

Messaging

Data Store

Remote Controller Instance

Remote Controller Instance

Network Devices

Applications

Protocol Plugin

Protocol Plugin

Config Subsystem

App/Service Plugin

App/Service Plugin

RESTCONF

Messages

Network Devices

Applications

Remote Controller Instance

Remote Controller Instance

Model-Driven SAL (MD-SAL)
Outline

- Admin and recap
- High-level datapath programming
  - blackbox (trace-tree)
  - whitebox
- Network OS supporting programmable networks
  - overview
  - OpenDaylight
  - distributed network OS using Paxos, Raft
    (Slides borrowed heavily from Raft user study)
Setting

Clients

Servers

request/response
Replicated 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

Failure model: fail-stop (not Byzantine), delayed/lost messages
Approaches to Consensus

Two general approaches to consensus:

- **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
Outline

- Admin and recap
- High-level datapath programming
- Network OS supporting programmable networks
  - overview
  - OpenDaylight
  - distributed network OS
    - Overview
    - Basic Paxos
The Paxos Approach

- 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
  - System must agree on a single value as chosen
  - Only one value is ever chosen
  - We will focus on this part only

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

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

For this class:
- Each Paxos server contains both components
Design I: A Single Acceptor

- A single acceptor chooses value

Problem:
  - What if acceptor crashes after choosing?

Solution:
  - Quorum: multiple acceptors (3, 5, ...)
  - If one acceptor crashes, other acceptors still available
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 sometimes accept multiple (different) values
Design III

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

- Problem:
  - Conflicting choices

=> Needs to detect if a value has been chosen; if so, future proposals must propose/choose that same value
Basic Paxos

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

**Proposers**

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

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

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 $n >$ minProposal then minProposal = $n$
   - Return(acceptedProposal, acceptedValue)

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

Acceptors must record minProposal, acceptedProposal, and acceptedValue on stable storage (disk)
1. Previous value already chosen:
   ⬤ New proposer will find it and use it

```
<table>
<thead>
<tr>
<th></th>
<th>P 3.1</th>
<th>A 3.1 X</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>P 3.1</td>
<td>A 3.1 X</td>
</tr>
<tr>
<td>s3</td>
<td>P 3.1</td>
<td>A 3.1 X</td>
</tr>
<tr>
<td>s4</td>
<td>P 4.5</td>
<td>A 4.5 X</td>
</tr>
<tr>
<td>s5</td>
<td>P 4.5</td>
<td>A 4.5 X</td>
</tr>
</tbody>
</table>
```

“Prepare proposal 3.1 (from s₁)"

“Accept proposal 4.5 with value X (from s₅)”
2. Previous value not chosen, but new proposer sees it:
   - New proposer will use existing value
   - Both proposers can succeed
Liveness

- Competing proposers can livelock:

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

- Admin and recap
- High-level datapath programming
- Network OS supporting programmable networks
  - overview
  - OpenDaylight
  - distributed network OS
    - Overview
    - Basic Paxos
    - Raft
Raft Overview

- Designed in 2013, with a main goal to be easier to understand
- Implemented in multiple systems

<table>
<thead>
<tr>
<th>Name</th>
<th>Primary Authors</th>
<th>Language</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>RethinkDB/clustering</td>
<td>Blake Mizerany, Xiang Li and Yicheng Qin</td>
<td>C++</td>
<td>AGPL</td>
</tr>
<tr>
<td>etcd/raft</td>
<td>Armon Dadgar (hashicorp)</td>
<td>Go</td>
<td>Apache 2.0</td>
</tr>
<tr>
<td>LogCabin</td>
<td>Diego Ongaro (Stanford)</td>
<td>C++</td>
<td>ISC</td>
</tr>
<tr>
<td>go-raft</td>
<td>Ben Johnson (Sky) and Xiang Li (CMU, CoreOS)</td>
<td>Go</td>
<td>MIT</td>
</tr>
<tr>
<td>hashicorp/raft</td>
<td>Armon Dadgar (hashicorp)</td>
<td>Go</td>
<td>MPL-2.0</td>
</tr>
<tr>
<td>hoverbear/raft</td>
<td>Andrew Hobden, Dan Burkert</td>
<td>Rust</td>
<td>MIT</td>
</tr>
<tr>
<td>ckite</td>
<td>Pablo Medina</td>
<td>Scala</td>
<td>Apache2</td>
</tr>
<tr>
<td>verdi/raft</td>
<td>James Wilcox, Doug Woos, Pavel Panchekha, Zach Tatlock, Xi Wang, Mike Ernst, and Tom Anderson (University of Washington)</td>
<td>Coq</td>
<td>BSD</td>
</tr>
<tr>
<td>OpenDaylight</td>
<td>Moiz Raja, Kamal Rameshan, Robert Varga (Cisco), Tom Pantelis (Brocade)</td>
<td>Java</td>
<td>Eclipse</td>
</tr>
<tr>
<td>zraft_lib</td>
<td>Gunin Alexander</td>
<td>Erlang</td>
<td>Apache2</td>
</tr>
<tr>
<td>kanaka/raft.js</td>
<td>Joel Martin</td>
<td>Javascript</td>
<td>MPL-2.0</td>
</tr>
<tr>
<td>akka-raft</td>
<td>Konrad Malawski</td>
<td>Scala</td>
<td>Apache2</td>
</tr>
<tr>
<td>rafter</td>
<td>Andrew Stone (Basho)</td>
<td>Erlang</td>
<td>Apache2</td>
</tr>
<tr>
<td>floss</td>
<td>Alexander Flatter</td>
<td>Ruby</td>
<td>MIT</td>
</tr>
<tr>
<td>willem/raft</td>
<td>Willem-Hendrik Thiart</td>
<td>C</td>
<td>BSD</td>
</tr>
</tbody>
</table>
Raft Protocol Summary

<table>
<thead>
<tr>
<th>Followers</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Respond to RPCs from candidates and leaders.</td>
</tr>
<tr>
<td>• Convert to candidate if election timeout elapses without either:</td>
</tr>
<tr>
<td>• Receiving valid AppendEntries RPC, or</td>
</tr>
<tr>
<td>• Granting vote to candidate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Increment currentTerm, vote for self</td>
</tr>
<tr>
<td>• Reset election timeout</td>
</tr>
<tr>
<td>• Send RequestVote RPCs to all other servers, wait for either:</td>
</tr>
<tr>
<td>• Votes received from majority of servers: become leader</td>
</tr>
<tr>
<td>• AppendEntries RPC received from new leader: step down</td>
</tr>
<tr>
<td>• Election timeout elapses without election resolution: increment term, start new election</td>
</tr>
<tr>
<td>• Discover higher term: step down</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Leaders</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Initialize nextIndex for each to last log index + 1</td>
</tr>
<tr>
<td>• Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts</td>
</tr>
<tr>
<td>• Accept commands from clients, append new entries to local log</td>
</tr>
<tr>
<td>• Whenever last log index ≥ nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful</td>
</tr>
<tr>
<td>• If AppendEntries fails because of log inconsistency, decrement nextIndex and retry</td>
</tr>
<tr>
<td>• Mark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of servers</td>
</tr>
<tr>
<td>• Step down if currentTerm changes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Persistent State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each server persists the following to stable storage synchronously before responding to RPCs:</td>
</tr>
<tr>
<td>currentTerm</td>
</tr>
<tr>
<td>votedFor</td>
</tr>
<tr>
<td>log[]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Log Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
</tr>
<tr>
<td>index</td>
</tr>
<tr>
<td>command</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RequestVote RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoked by candidates to gather votes.</td>
</tr>
<tr>
<td><strong>Arguments:</strong></td>
</tr>
<tr>
<td>candidateId</td>
</tr>
<tr>
<td>term</td>
</tr>
<tr>
<td>lastLogIndex</td>
</tr>
<tr>
<td>lastLogTerm</td>
</tr>
<tr>
<td><strong>Results:</strong></td>
</tr>
<tr>
<td>term</td>
</tr>
<tr>
<td>voteGranted</td>
</tr>
<tr>
<td><strong>Implementation:</strong></td>
</tr>
<tr>
<td>1. If term &gt; currentTerm, currentTerm ← term (step down if leader or candidate)</td>
</tr>
<tr>
<td>2. If term == currentTerm, votedFor is null or candidateId, and candidate's log is at least as complete as local log, grant vote and reset election timeout</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AppendEntries RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat.</td>
</tr>
<tr>
<td><strong>Arguments:</strong></td>
</tr>
<tr>
<td>term</td>
</tr>
<tr>
<td>leaderId</td>
</tr>
<tr>
<td>prevLogIndex</td>
</tr>
<tr>
<td>prevLogTerm</td>
</tr>
<tr>
<td>entries[]</td>
</tr>
<tr>
<td>commitIndex</td>
</tr>
<tr>
<td><strong>Results:</strong></td>
</tr>
<tr>
<td>term</td>
</tr>
<tr>
<td>success</td>
</tr>
<tr>
<td><strong>Implementation:</strong></td>
</tr>
<tr>
<td>1. Return if term &lt; currentTerm</td>
</tr>
<tr>
<td>2. If term &gt; currentTerm, currentTerm ← term</td>
</tr>
<tr>
<td>3. If candidate or leader, step down</td>
</tr>
<tr>
<td>4. Reset election timeout</td>
</tr>
<tr>
<td>5. Return failure if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm</td>
</tr>
<tr>
<td>6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry</td>
</tr>
<tr>
<td>7. Append any new entries not already in the log</td>
</tr>
<tr>
<td>8. Advance state machine with newly committed entries</td>
</tr>
</tbody>
</table>
Raft: Big Picture

- Time divided into terms:
  - Election
  - Normal operation under a single leader
- At most one leader per term
  - Some terms have no leader (failed election)
- Each server maintains current term value
Raft Components

1. Basic leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. Basic log replication (Normal operation)
3. Revised leader election and log replication for safety and consistency
4. Neutralizing old leaders
5. Client interactions
   - Implementing linearizeable semantics
6. Configuration changes:
   - Adding and removing servers
Basic Election: Server States

- At any given time, each server is either:
  - **Leader**: handles all client interactions, log replication
    - At most 1 viable leader at a time
  - **Follower**: completely passive (issues no RPCs, responds to incoming RPCs)
  - **Candidate**: used to elect a new leader

- Normal operation: 1 leader, N-1 followers
Basic Election: Server State Transitions

- Servers start up as followers
- Followers expect to receive RPCs from leaders or candidates
- Leaders must send heartbeats (empty AppendEntries RPCs) to maintain authority
- If electionTimeout elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500ms

[Diagram showing state transitions from Follower, Candidate, and Leader]

- start
- timeout, start election
- timeout, new election
- receive votes from majority of servers
- "step down"
- discover current server or higher term
Basic Election Alg

- Increment current term
- Change to Candidate state
- Vote for self
- Send RequestVote RPCs to all other servers, retry until either:
  1. Receive votes from majority of servers:
     - Become leader
     - Send AppendEntries heartbeats to all other servers
  2. Receive RPC from valid leader:
     - Return to follower state
  3. No-one wins election (election timeout elapses):
     - Increment term, start new election
Basic Elections, cont'd

- **Safety**: allow at most one winner per term
  - Each server gives out only one vote per term (persist on disk)
  - Two different candidates can't accumulate majorities in same term

- **Liveness**: some candidate must eventually win
  - Choose election timeouts randomly in \([T, 2T]\)
  - One server usually times out and wins election before others wake up
  - Works well if \(T \gg\) broadcast time

Need refinement to basic election. See later.
Neutralizing Old Leaders

- Deposed leader may not be dead:
  - Temporarily disconnected from network
  - Other servers elect a new leader
  - Old leader becomes reconnected, attempts to commit log entries

- Terms used to detect stale leaders (and candidates)
  - Every RPC contains term of sender
  - If sender’s term is older, RPC is rejected, sender reverts to follower and updates its term
  - If receiver’s term is older, it reverts to follower, updates its term, then processes RPC normally

- Election updates terms of majority of servers
  - Deposed server cannot commit new log entries
- Log entry = index, term, command
- Log stored on stable storage (disk); survives crashes
- Entry **committed** if known to be stored on majority of servers
  - Durable, will eventually be executed by state machines
Basic Log Operation

- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to followers

Once new entry committed:
  - Leader passes command to its state machine, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers pass committed commands to their state machines

- Crashed/slow followers?
  - Leader retries RPCs until they succeed
Log Consistency Invariant

High level of coherency between logs:

- If log entries on different servers have same index and term:
  - They store the same command
  - The logs are identical in all preceding entries

- If a given entry is committed, all preceding entries are also committed
AppendEntries Consistency Check

- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request (see later for how this is used)
- Implements an induction step, ensures coherency

---

**AppendEntries succeeds:** matching entry

**AppendEntries fails:** mismatch
At beginning of new leader’s term:
- Old leader may have left entries partially replicated
- Multiple crashes can leave many extraneous log entries:
- Design goal
  - No special steps by new leader: just start normal operation
  - Leader’s log is “the truth”
  - Will eventually make follower’s logs identical to leader’s

---

Leader Election and Commit Refinement

<table>
<thead>
<tr>
<th>log index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Safety with Leader Change

**Safety**: Once a log entry has been applied (committed) to a state machine, no other state machine can apply a different value for that log entry.