CS434/534: Topics in Networked (Networking) Systems

High-Level Programming for Programmable Networks; 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
    - compact per-instruction table (PIT) pipeline
    - optimized pipeline
- Network OS supporting programmable networks
  - overview
  - OpenDaylight
Recap: Black vs Whitebox Approaches

- **Blackbox (trace-tree)**
  - **Pros**
    - Simple, language independent
  - **Cons**
    - Generate only a single table
    - Reactive, with potentially long latency

- **Whitebox (compiler)**
  - **Goal:** proactive, fully generated datapath
Recap: Map Each Instruction to a Flow Table

Map<MAC, Switch> hostTable; // {11->sw1, 22->sw1, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}
Set openPorts; // {22, 53, 80, 8000, 8080, 9090}

Route onPacketIn(Packet p) {
L1  if (p.dstPort < 1025) { // privileged port, SW_FW
L2:  dstSw = SW_FW;
L3:  dstCond = S;
L4:  }
L5:  } else if (openPorts.contains(p.dstPort)) { // spec user
L6:   dstSw = hostTable[p.dstMac];
L7:   dstCond = condTable[p.dstMac];
L8:  } else {
L9:    return Drop; }
L10:  // All pairs paths for all dstConds
L11:  allPaths = AllPairsCond.execute( ... );
L12:  srcSw = hostTable[p.srcMac];
L13:  return allPaths[dstCond].get(srcSw,dstSw); }

Recap: Table-Friendly Instructions

We say L1 is a **compact-mapable (CM)** instruction: Despite large input domain, small # of rules to express.

**L1.**  
\[ g_1 = p.dstPort < 1025; \]

<table>
<thead>
<tr>
<th>Pri</th>
<th>p.dstPort</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0000 00xx</td>
<td>reg(_g_1) = \text{true} jump L2</td>
</tr>
<tr>
<td>2</td>
<td>0000 0100 0000 0000</td>
<td>reg(_g_1) = \text{true} jump L2</td>
</tr>
<tr>
<td>1</td>
<td>xxxx xxxx xxxx</td>
<td>reg(_g_1) = \text{false} jump L2</td>
</tr>
</tbody>
</table>
**Recap: Table-Friendly Instructions**

L5 is another type of CM instruction: Despite large input domain, system state already provides (small) potential values.

### Naïve enumeration

<table>
<thead>
<tr>
<th>p.dstPort</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>Reg_g2 = false; jump L6</code></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td><code>Reg_g2 = true; jump L6</code></td>
</tr>
<tr>
<td>(2^{16})</td>
<td><code>Reg_g1 = false; jump L6</code></td>
</tr>
</tbody>
</table>

### Compact encoding using system state

<table>
<thead>
<tr>
<th>Pri</th>
<th>p.dstPort</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>22</td>
<td><code>Reg_g2 = true; jump L6</code></td>
</tr>
<tr>
<td>2</td>
<td>53</td>
<td><code>Reg_g2 = true; jump L6</code></td>
</tr>
<tr>
<td>2</td>
<td>80</td>
<td><code>Reg_g2 = true; jump L6</code></td>
</tr>
<tr>
<td>2</td>
<td>8000</td>
<td><code>Reg_g2 = true; jump L6</code></td>
</tr>
<tr>
<td>2</td>
<td>8080</td>
<td><code>Reg_g2 = true; jump L6</code></td>
</tr>
<tr>
<td>2</td>
<td>9090</td>
<td><code>Reg_g2 = true; jump L6</code></td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td><code>Reg_g2 = false; jump L6</code></td>
</tr>
</tbody>
</table>

---

\[\text{L5. } g2 = \text{openPorts}.contains(p.dstPort); \]
\[// \text{openPorts} = \{22, 53, 80, 8000, 8080, 9090\};\]
Symbolic analysis/direct state-plugin cannot generate tables for some instructions.

Tables computed in isolation can have unnecessary rules.
Recap: Observation

- Both problems are essentially the same, we need to compute valid inputs to each instruction:
  - \( g_2 = \text{openPorts.contains(p.dstPort)}; \)
    - If we know valid inputs to dstPort (<1025), we can prune the extra rules (22, 53, 80)
  - \( \text{allPaths}\[\text{dstCond}\].get(srcSw,dstSw); \)
    - If we know the valid values of (dstCond, srcSw, dstSw) reaching this instruction, we can compute a compact flow table for it (not all potential values).
Recap: Naïve Approach can Compute Non-Valid Inputs

- L1: \( m_1 \rightarrow s_1, m_2 \rightarrow s_2, \ldots, m_n \rightarrow s_n \)
- L2: \( m_1 \rightarrow c_1, m_2 \rightarrow c_2, \ldots, m_n \rightarrow c_n \)
- L4 sees only \( n \) (sw, cond) combinations, not \( n^2 \) combinations
  - This is called dependent inputs

```java
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);
```
Basic Approach: The FlowExplore Algorithm

- Computes \((I, \text{state})\) to each instruction, using flow tables as edges
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);
}
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);
}

dstMac is already part of state, no need to use flow table, use state propagation

Map<MAC, Switch> hostTable; // {11->sw1, 22->sw2, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}
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);
}

dstMac is a dead variable and hence should be removed.
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);
}

Map<MAC, Switch> hostTable; // {11->sw1, 22->sw2, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}

Same state and hence should merge.

<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>
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);
}
null
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);
}

Map<MAC, Switch> hostTable; // {11->sw1, 22->sw2, 33->sw2, others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}

<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>

L1, EMPTY

L2, dstSw=sw1, dstMac=11
  L4, dstSw=sw1, dstCond=S
    L∞, aRoutes(sw1, S)

L2, dstSw=sw2, dstMac=22
  L4, dstSw=sw2, dstCond=C
    L∞, aRoutes(sw2, C)

L2, dstSw=sw2, dstMac=33
  L∞, aRoutes(sw2, C)

L2, dstSw=swu, dstMac=* 
  L4, dstSw=swu, dstCond=UK
    L∞, aRoutes(swu, UK)
Q: How to obtain the flow table of each instruction?
Assignment

- Compute the reachable inputs of the complete example program
Outline

- Admin and recap
- High-level datapath programming
  - blackbox (trace-tree)
  - whitebox
    - compact per-instruction table (PIT) pipeline
    - optimized pipeline
This example needs 10 tables, but a hardware may support only 2. Unnecessary separate matching resources can be wasteful.

```java
Map<MAC, Switch> hostTable; // {11->sw1, 22->sw1, 33->sw2,others swu}
Map<MAC, Cond> condTable; // {11->S, 22->C, 33->C, others UK}
Set openPorts; // {22, 53, 80, 8000, 8080, 9090}

Route onPacketIn(Packet p) {
    if (p.dstPort < 1025) { // privileged port, SW_FW
        dstSw = SW_FW;
        dstCond = S;
    } else if (openPorts.contains(p.dstPort)) { // spec user
        dstSw = hostTable[p.dstMac];
        dstCond = condTable[p.dstMac];
    } else {
        return Drop;
    }
    // All pairs paths for all dstConds
    allPaths = AllPairsCond.execute( ... );
    srcSw = hostTable[p.srcMac];
    return allPaths[dstCond].get(srcSw,dstSw); }
```
Algorithm Framework

Assume $P$ is the set of pipelines eval’d

$p_0 = \text{empty}; \text{resource}_0 = \text{INF}$

foreach $p$ in $P$

  if $\text{hw\_oracle}(p) \&\& \text{resource}(p) < \text{resource}_0$
  
    $p_0 = p; \text{resource}_0 = \text{resource}(p)$

How many pipelines in $P$ if $P$ is generated from full enumeration. Assume $N$ instructions and $\leq M$ tables?
g1 = dstPort < 1025

!g1

g2 = openPorts. contain(dstPort)

!g2

dstSw = SW_FW

dstCond = V

egress = Drop

dstSw = hostTable(dstMac)

dstCond = condTable(dstMac)

srcSw = hostTable(srcMac)

phi(dstSw)

phi(dstCond)

srcMac

egress = f(dstCond, dstSw, srcSw)
Insight: Suppose we have a table T matching on \((\text{dstPort}, \text{dstMac})\). What should be handled by T?
Summary: Whitebox

- Whitebox (compiler)
  - Pros
  - Cons
Programmable Networks

Program

logically centralized data store

Network View

Service/Policy

NE Datapath

NE Datapath
Outline

- Admin and recap
- High-level datapath programming
  - *Network OS supporting programmable networks*
Discussion

- What support does a network OS (NOS) provide to programmable networks?
NOS Design Goals

- Provide applications with 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
NOX [2008]

- High-level views:
  - Topology (who connects to whom)
  - Topology obtained by capturing packet-misses (flow initiation), DNS, LLDP packets

- Make the views highly available and allow scalable access
  - Distributed cluster
ONOS Design Decisions [2011-]

- High-level views:
  - Topology, initially Blueprint API, Titan graph databases, later customized topology data structures

- Make the views highly available and allow scalable access
  - Distributed cluster using RAMCloud
Outline

- Admin and recap
- High-level datapath programming
  - Network OS supporting programmable networks
    - overview
      - openDaylight

Outline

- Admin and recap
- High-level datapath programming
  - Network OS supporting programmable networks
    - overview
      - OpenDaylight
        - data model and store/views
          (slides based on OpenDaylight tutorial, 2016)
Data Model

- 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
Yang Example

module hello {
    yang-version 1;
    namespace "urn:opendaylight:params:xml:ns:yang:hello";
    prefix "hello";

    revision "2015-01-05" {
        description "Initial revision of hello model";
    }

    rpc hello-world {
        input {
            leaf name {
                type string;
            }
        }
        output {
            leaf greeting {
                type string;
            }
        }
    }
}

container greeting-registry {
    list greeting-registry-entry {
        key "name";
        leaf name {
            type string;
        }
        leaf greeting {
            type string;
        }
    }
}
The Service Development Process

1. YANG Model (s) → Yang Tools → Generated API
2. Generated API → Maven Build Tools → OSGi API JAR
3. Service Implementation → Maven Build Tools → OSGi IMPL JAR
4. Karaf Feature Definition → Maven Build Tools → Karaf KAR
5. Controller

- OSGi API JAR
- OSGi IMPL JAR
- Features.xml

Deploy
Data Tree: Details

- data store as a tree
- Two Logical Data Stores
  - config
  - operational
- Unified View
- InstanceIdentifier:
  - Pointer to a node
Yang Data Tree

Data Store: Read/Write Example

```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();
```
Distributed Data Store for Scalability

- **Datastore**
  - Sharded
  - Replicated
    - But not everywhere
  - **RAFT algorithm for consistency**
Outline

- Admin and recap
- High-level datapath programming
  - Network OS supporting programmable networks
    - overview
      - OpenDaylight
        - data model and store
        - interactions, services, and composition
OpenDaylight Architecture: Big Picture

Software Architecture

Network Devices
- Protocol Plugin
- Protocol Plugin
- Config Subsystem

Applications
- App/Service Plugin
- App/Service Plugin
- RESTCONF

Model-Driven SAL (MD-SAL)

Controller
- Messaging
- Data Store

Remote Controller Instance

RESTCONF
3 Brokers

Data Broker

RPC Broker

Notification Broker

put

store

notify

publish

notify
Data Store: Merging Example

```java
WriteOnlyTransaction transaction = dataBroker.newWriteOnlyTransaction();
InstanceIdentifier<Node> path =
    InstanceIdentifier
    .create(NetworkTopology.class)
    .child(Topology.class,
        new TopologyKey("overlay1");
transaction.merge(
    LogicalDataStore.CONFIG,
    path,
    topologyBuilder.build());
CheckedFuture future;
future = transaction.submit();
```
dataBroker.registerDataChangeListener(
  LogicalDatastoreType.CONFIGURATION,
  myInstanceId,
  myDataChangeListener,
  DataChangeScope.SUBTREE);

myDataChangeListener

AsyncDataChangeEvent
  created  deleted  updated  original

dataBroker.registerDataChangeListener(
  LogicalDatastoreType.CONFIGURATION,
  myInstanceId,
  myDataChangeListener,
  DataChangeScope.SUBTREE);
Backup Slides
**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();
```

Diagram:
- **consumer**
  - getRpcService()
    - return: helloService
  - helloWorld(helloWorldInput)
    - return: future
  - future
    - return: RpcResult<HelloWorldOutput>
    - isDone()
      - false
      - true
    - set(helloOutput)
    - get()
public class HelloWorldImpl implements HelloService {
    public HelloWorldImpl(ProviderContext session){
        session.addRpcImplementation(
            HelloService.class, this);
    }
    @Override
    public <RpcResult>HelloWorld helloWorld(HelloWorldInput input) {
        SettableFuture future = new SettableFuture();
        process (input, future);
        return future;
    }
}
Global RPCs - Processing a Message - Async

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
Discussion: OpenDaylight NOS Design

- What you like and do not like about the design?
ONOS Architecture: Big Picture

Northbound - Application Intent Framework
(policy enforcement, conflict resolution)

Distributed Core
 scalability, availability, performance, persistence

Southbound
(discover, observe, program, configure)

OpenFlow
(pluggable, extensible)

More API in future release

Source: ON.LAB
ONOS Architecture: Some Key Abstractions

https://wiki.onosproject.org/display/ONOS/System+Components
Controller Architecture

Controller

Service Adaptation Layer

Service Functions
- Statistics Manager
- Forwarding Rules Manager
- PCEP

Base Network Functions
- Topology Exporter
- Inventory Manager

Configuration Subsystem
- NETCONF

REST APIs

Controller Platform

Network Applications Orchestration & Services

Southbound Interfaces & Protocol Plugins

Network Devices

Applications

Controller Architecture