CSC 2229 – Software-Defined Networking

Handout #8:
Simple Distributed Programming for Scalable Software-Defined Networks

Professor Yashar Ganjali
Department of Computer Science
University of Toronto

yganjali@cs.toronto.edu
http://www.cs.toronto.edu/~yganjali

Joint work with Soheil Hassas Yeganeh
ANNOUNCEMENTS

- Presentations
  - Schedule has been emailed
    - Contact me if you are not on the schedule (quite a few people are not).

- The last two weeks of class
  - Final presentations

- Intermediate report
  - Due: Friday, March 10th
  - Please read the guidelines on class website.
TRADITIONAL NETWORKS

Hard to Program Distributed Systems
SOFTWARE DEFINED NETWORKING

Easy
Hard to Program Distributed Systems
Centralized

Application
Controller
Switch
Switch
Switch
SOFTWARE DEFINED NETWORKING

Existing Distributed Controllers
- Excellent in performance & scalability
- Perfect fit for some specific scenarios
SOFTWARE DEFINED NETWORKING

Much better than traditional networks

**still** Hard to Program Distributed Systems

Existing Distributed Controllers

- Don’t hide the boilerplates of distributed programming
- Require significant efforts to instrument and optimize apps
OUR GOAL

Hard to Program Distributed Systems

Easy

similar to centralized controllers

Application
Controller

Application
Controller

Switch

+ optimized placements
+ application analytics
MOTIVATION

Traditional

Dijkstra via Distributed Messaging (115 pages of RFC)

Promised

Dijkstra via Centralized Apps (10s - 100s of LOC)

In Practice

Dijkstra via Distributed Apps (1000s-10000s of LOC)
MOTIVATION

Traditional

Dijkstra via Distributed Messaging (115 pages of RFC)

Promised

Dijkstra via Centralized Apps (10s - 100s of LOC)

In Practice

Dijkstra via Distributed Apps (1000s-10000s of LOC)

Goal

Dijkstra via Distributed Apps (100s of LOC?)
OUR GOAL

centralized Application
OUR GOAL

centralized Application can be automatically transformed into Application
OUR GOAL

centralized Application

can be automatically transformed into

distributed

Application

Application

Application
Our goal

Centralized

Application

Machine

Distributed

Application

Application

Application

Machine

Machine

Machine

deployed on multiple physical machines.

Very challenging for generic control applications.
OVERVIEW

Abstraction

Application

Compiler

Application

Application

Application

Control Platform

Machine

Machine

Machine
ABSTRACTION

what is a control application?

Process async messages in application functions using state dictionaries.
ABSTRACTION  
how do applications communicate?

async messages  
*all functions*

state dictionaries  
*functions of the same application*
EXAMPLE

Initializes dictionary
Queries switches
Collects stat results
Reroutes flows, if needed

Traffic Engineering

Init
Query
Collect
Route

S
T
Statistics
Topology
EXAMPLE

How to transform TE into a distributed application while preserving state consistency?
Functions create an implicit mapping between messages and dictionary entries:

*The entries a function needs to process a message.*
**EXAMPLE**

`Init()`, `Query()` and `Collect()` access $S$ on a per switch basis.
Example

Init(), Query() and Collect() access $S$ on a per switch basis.
EXAMPLE

Traffic Engineering

<table>
<thead>
<tr>
<th>Switch</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>flow1 -&gt; stat</td>
</tr>
<tr>
<td></td>
<td>flow2 -&gt; stat</td>
</tr>
</tbody>
</table>

Machine 1

Flow 1 -> Stat
Flow 2 -> Stat

Traffic Engineering

<table>
<thead>
<tr>
<th>Switch</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>flow3 -&gt; stat</td>
</tr>
<tr>
<td></td>
<td>flow4 -&gt; stat</td>
</tr>
</tbody>
</table>

Machine 2

Flow 3 -> Stat
Flow 4 -> Stat
**EXAMPLE**

**Init()**, **Query()** and **Collect()** access \( S \) on a per switch basis. **Route()** accesses the whole dictionary \( S \) to process the timeout message.
EXAMPLE

This will cause inconsistency.

Machine 1

Machine 2
EXAMPLE

Traffic Engineering

<table>
<thead>
<tr>
<th>Switch</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>flow1 -&gt; stat</td>
</tr>
<tr>
<td></td>
<td>flow2 -&gt; stat</td>
</tr>
<tr>
<td>2</td>
<td>flow3 -&gt; stat</td>
</tr>
<tr>
<td></td>
<td>flow4 -&gt; stat</td>
</tr>
</tbody>
</table>

Machine 1

Machine 2
CONSISTENCY

Application

Function 1

Function 2

msg 3  k1
msg 2  k2  k3  k5
msg 1  k2  k4
CONSISTENCY

Application
Function 1
k2
k3
k4
k5
Function 2

Machine

msg 1
msg 2
msg 3
We need a runtime that steers messages among application instances while preserving consistency.
BEEHIVE CONTROL PLATFORM

Hidden from programmers!

In (raft) consensus about bees and cells
Entries in the dictionaries

Thread of execution

Programming Model
Application

Transformed
Controller

Application
msg
Machine

Application
msg
Machine

Application
msg
Machine
CONTROL PLATFORM
CONTROL PLATFORM

How do we infer the cells?
How do we infer the cells?

`map(app, msg)` is an application defined function that maps a message to the set of cells used to process that message.

```java
func Collect(r, s):
    s.append(flow stats in r)

on StatReply(r):
    Collect(r, S[r.switch])

map StatReply(r):
    return (S, r.switch)
```

Beehive’s compiler can automatically generate the map function.

1-3 lines of code
CONTROL PLATFORM

- Function Composition
- Transactions (State + Messages)
- Bee Migration
- Fault tolerance
- Optimized Placement
- Runtime Instrumentation
- Feedback
- Proxied Hives
- …
MIGRATION
Migration
MIGRATION
MIGRATION

Switch  Switch  Switch

Switch

Switch
This is not optimal and can happen often.
MIGRATION
MIGRATION
MIGRATION

Switch  Switch  Switch

Switch
When/where should we migrate bees?

- NP-Hard problem
- We use a simple heuristic
OPTIMIZED PLACEMENT

Our heuristic

A bee that receives the majority of its messages from bees on another hive is migrated to that hive.
RUNTIME INSTRUMENTATION

• traffic matrix among bees
• resource consumption
• message provenance
ANALYTICS & FEEDBACK

Switch | Switch | Switch

Switch
ANALYTICS & FEEDBACK
ANALYTICS & FEEDBACK

Switch Switch Switch

centralized

Hive

Hives

Hive

TE QCIR

Hives

TE QCIR

Switch
ANALYTICS & FEEDBACK

Switch  Switch  Switch

Hive

Hives

centralized

Switch
ANALYTICS & FEEDBACK

Hive

Switch  Switch  Switch

Hive

Switch

well-balanced
Fault Tolerance

Colony of replicated bees all in consensus about their state.
BEEHIVE CONTROL PLATFORM

Hidden from programmers!

Instrumentation & Optimization

Live Migration

Replication

msg

msg

msg

msg
## Generality

<table>
<thead>
<tr>
<th>Centralized</th>
<th>Kandoo</th>
<th>NIB</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>func Centralized(msg):</code></td>
<td><code>func Local(msg):</code></td>
<td><code>func NIB(msg):</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>map Centralized(msg):</code></td>
<td><code>map Local(msg):</code></td>
<td><code>map NIB(msg):</code></td>
</tr>
<tr>
<td><code>return {(D, 0)}</code></td>
<td><code>return {(D, hiveid)}</code></td>
<td><code>return {(N, nodeid)}</code></td>
</tr>
</tbody>
</table>

### Virtual Networking

<table>
<thead>
<tr>
<th>Virtual Networking</th>
<th>Routing</th>
<th>+ you don’t need to think about placement and load balancing in most cases.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>func VN(msg):</code></td>
<td><code>func Router(msg):</code></td>
<td></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
<td></td>
</tr>
<tr>
<td><code>map VN(msg):</code></td>
<td><code>map Router(msg):</code></td>
<td></td>
</tr>
<tr>
<td><code>return {(VN, vnid)}</code></td>
<td><code>return {(Adv, msg.n[0])}</code></td>
<td></td>
</tr>
</tbody>
</table>
IMPLEMENTATION

- Free & Open Source, written in Go:
  - https://github.com/kandoo/beehive
  - https://github.com/kandoo/beehive-netctrl
- No external dependency in the most recent version
- OpenFlow bindings are generated from high level specs:
  - https://github.com/packet/packet
We have implemented our own SDN controller on top of Beehive:
- Network Object Model (Similar to ONIX NIBs)
- Switching + Routing + Connectivity + Isolation
- The community can extend or propose a new abstraction on top of Beehive.
- Ported application from POX and Beacon:
  - End results are identical in complexity. Differences are programming languages.
  - Our applications are distributed, persistent, fault-tolerance, and scalable.
EVALUATION

Efficient Bee Creation

Low Latency

Fast Failover

Efficient Instrumentation

High Throughput

Seamless Optimization
SDN CONTROLLER - SCALABILITY

- Changed the learning switch to send packet outs instead of flow mods.
- ONOS can handle 6000 msg/sec in a 3-node cluster.
EVALUATION

- The TE application
- Simulated environment
- A 40 node cluster on GCE

These spikes are for instrumentation data (periodic at 10s)
EVALUATION

- The TE application
- Simulated environment
- A 40 node cluster on GCE

This spike is for replicating cells on 40 hives. (~4sec.)

Decoupled

All artificially centralized then dynamically optimized
FINAL REMARKS

• Beehive = Abstraction + Control Platform
  • Almost identical to centralized controllers
  • Dynamically optimized placement
  • Runtime instrumentation and feedback

• Moving forward
  • Strengthen our evaluation
  • Performance optimizations
Distributed programming in SDN doesn’t have to be complicated.
BACKUP SLIDES
BEEHIVE

Programming Model

**app Discovery:**
```go
func Rcv(pin PacketIn):
    ...
```

**app Hub:**
```go
func Rcv(pin PacketIn):
    pout = PacketOut{
        OutPort: FLOOD,
    }
    ReplyTo(pin, pout)
```

```go
app OpenFlow:
    func Rcv(pout PacketOut):
        // write on the socket.
    func Rcv(fmod FlowMod):
        // write on the socket.
    func Rcv(stat StatQuery):
        // write on the socket.

    while data = read(socket)
        switch data:
            case OFPacketIn:
                Emit(PacketIn{...})
            case OFStatReply:
                Emit(StatReply{...})
```

Switch 1  Switch 2
**BEEHIVE**

Programming Model

**app Discovery:**
```go
func Rcv(pin PacketIn):
    ...
```

**app Hub:**
```go
func Rcv(pin PacketIn):
pout = PacketOut{
    OutPort: FLOOD,
}
ReplyTo(pin, pout)
```

**Async Messages**
- Defined by Applications
- Delivered by Beehive
- Processed in Rcv functions

---

**app OpenFlow:**
```go
func Rcv(pout PacketOut):
    // write on the socket.

func Rcv(fmod FlowMod):
    // write on the socket.

func Rcv(stat StatQuery):
    // write on the socket.

while data = read(socket):
    switch data:
        case OFPacketIn:
            Emit(PacketIn{...})
        case OFStatReply:
            Emit(StatReply{...})
```

Switch 1  Switch 2
BEEHIVE

Programming Model

app Discovery:
    func Rcv(pin PacketIn):
        ...

app Hub:
    func Rcv(pin PacketIn):
        pout = PacketOut{
            OutPort: FLOOD,
        }
        ReplyTo(pin, pout)

app OpenFlow:
    func Rcv(pout PacketOut):
        // write on the socket.
    func Rcv(fmod FlowMod):
        // write on the socket.
    func Rcv(stat StatQuery):
        // write on the socket.

while data = read(socket):
    switch data:
        case OFPacketIn:
            Emit(PacketIn{...})
        case OFStatReply:
            Emit(StatReply{...})

OFPacketIn

Switch 1

Switch 2
app Discovery:
  func Rcv(pin PacketIn):
    ...

app Hub:
  func Rcv(pin PacketIn):
    pout = PacketOut{
      OutPort: FLOOD,
    }
    ReplyTo(pin, pout)

app OpenFlow:
  func Rcv(pout PacketOut):
    // write on the socket.
  func Rcv(fmod FlowMod):
    // write on the socket.
  func Rcv(stat StatQuery):
    // write on the socket.

while data = read(socket)
  switch data:
    case OFPacketIn:
      Emit(PacketIn{S1})
    case OFStatReply:
      Emit(StatReply{...})
BEEHIVE

Programming Model

**app** Discovery:
  
  ```
  func Rcv(pin PacketIn):
    ...
  ```

**app** Hub:
  
  ```
  func Rcv(pin PacketIn):
    pout = PacketOut{
      OutPort: FLOOD,
    }
    ReplyTo(pin, pout)
  ```

**app** OpenFlow:
  
  ```
  func Rcv(pout PacketOut):
    // write on the socket.
  func Rcv(fmod FlowMod):
    // write on the socket.
  func Rcv(stat StatQuery):
    // write on the socket.

  while data = read(socket):
    switch data:
      case OFPacketIn:
        Emit(PacketIn{...})
      case OFStatReply:
        Emit(StatReply{...})
  ```

From: OpenFlow
To: *
PacketIn{S1}

From: OpenFlow
To: *
PacketIn{S1}

Switch 1

Switch 2
**BEEHIVE**

Programming Model

```
app Discovery:
  func Rcv(pin PacketIn):
    ...

app Hub:
  func Rcv(pin PacketIn):
    pout = PacketOut{
      OutPort: FLOOD,
    }
  ReplyTo(pin, PacketOut{...})
```

```
app OpenFlow:
  func Rcv(pout PacketOut):
    // write on the socket.
  func Rcv(fmod FlowMod):
    // write on the socket.
  func Rcv(stat StatQuery):
    // write on the socket.

while data = read(socket)
  switch data:
    case OFPacketIn:
      Emit(PacketIn{...})
    case OFStatReply:
      Emit(StatReply{...})
```

From: OpenFlow
To: *

PacketIn{S1}
**BEEHIVE**

Programming Model

**app** Discovery:
  
  ```go
  func Rcv(pin PacketIn):
      ...
  ```

**app** Hub:
  
  ```go
  func Rcv(pin PacketIn):
      pout = PacketOut{
          OutPort: FLOOD,
      }
      ReplyTo(pin, pout)
  ```

**app** OpenFlow:
  
  ```go
  func Rcv(pout PacketOut):
      // write on the socket.
  func Rcv(fmod FlowMod):
      // write on the socket.
  func Rcv(stat StatQuery):
      // write on the socket.
  ```

  ```go
  while data = read(socket)
      switch data:
          case OFPacketIn:
              Emit(PacketIn{...})
          case OFStatReply:
              Emit(StatReply{...})
  ```

  From: Hub
  To: OpenFlow
  PacketOut{...}

  Switch 1
  Switch 2
BEEHIVE

Programming Model

**app Discovery:**
```go
func Rcv(pin PacketIn):
  ...
```

**app Hub:**
```go
func Rcv(pin PacketIn):
  pout = PacketOut{
    OutPort: FLOOD,
  }
  ReplyTo(pin, pout)
```

**app OpenFlow:**
```go
func Rcv(pin PacketOut{...}):
  // write on the socket.
func Rcv(fmod FlowMod):
  // write on the socket.
func Rcv(stat StatQuery):
  // write on the socket.

while data = read(socket)
  switch data:
    case OFPacketIn:
      Emit(PacketIn{...})
    case OFStatReply:
      Emit(StatReply{...})
```

From: Hub
To: OpenFlow

Switch 1
Switch 2
**BEEHIVE**

Programming Model

**app Discovery:**
```go
func Rcv(pin PacketIn):
    ...
```

**app Hub:**
```go
func Rcv(pin PacketIn):
    pout = PacketOut{
        OutPort: FLOOD,
    }
    ReplyTo(pin, pout)
```

**app OpenFlow:**
```go
func Rcv(pout PacketOut):
    // write on the socket.
func Rcv(fmod FlowMod):
    // write on the socket.
func Rcv(stat StatQuery):
    // write on the socket.

while data = read(socket)
    switch data:
    case OFPacketIn:
        Emit(PacketIn{...})
    case OFStatReply:
        Emit(StatReply{...})
```

Switch 1

Switch 2

OFPacketOut
But, what about stateful applications? Applications can use their **local dictionaries** to store state.

```go
app Hub:
    func Rcv(pin PacketIn):
        pout = PacketOut{
            OutPort: FLOOD,
        }
        ReplyTo(pin, pout)

app Discovery:
    func Rcv(pin PacketIn):
        ...
```

```go
app OpenFlow:
    func Rcv(pout PacketOut):
        // write on the socket.
    func Rcv(fmod FlowMod):
        // write on the socket.
    func Rcv(stat StatQuery):
        // write on the socket.

    while data = read(socket)
        switch data:
            case OFPacketIn:
                Emit(PacketIn{...})
            case OFStatReply:
                Emit(StatReply{...})

Switch 1  Switch 2
```
BEEHIVE

Programming Model

```go
app LearningSwitch:
    func Rcv(pin PacketIn):
        switches = Dict(“Switches”)
        m2p = switches[pin.Switch]
        m2p[pin.SMAC] = pin.InPort
        switches[pin.Switch] = m2p
        p = m2p[pin.DMAC]
        if not p:
            ReplyTo(pin, PacketOut{FLOOD})
        else:
            ReplyTo(pin, FlowMod{
                Output{Port: p},
            })
```

```go
app OpenFlow:
    func Rcv(pout PacketOut):
        // write on the socket.
    func Rcv(fmod FlowMod):
        // write on the socket.
    func Rcv(stat StatQuery):
        // write on the socket.

    while data = read(socket)
        switch data:
            case OFPacketIn:
                Emit(PacketIn{...})
            case OFStatReply:
                Emit(StatReply{...})
```

Switch 1

Switch 2
**BEEHIVE**

Programming Model

```go
app LearningSwitch:
  func Rcv(pin PacketIn):
    switches = Dict("Switches")
    m2p = switches[pin.Switch]
    m2p[pin.SMAC] = pin.InPort
    switches[pin.Switch] = m2p
    p = m2p[pin.DMAC]
    if not p:
      ReplyTo(pin, PacketOut{FLOOD})
    else:
      ReplyTo(pin, FlowMod{
        Output{Port: p},
      })
```

```
<table>
<thead>
<tr>
<th>Switches</th>
<th>Mac2Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>01:02:03:04:05:06 P1</td>
</tr>
<tr>
<td>S2</td>
<td>06:05:04:03:02:01 P2</td>
</tr>
</tbody>
</table>
```

```
app OpenFlow:
  func Rcv(pout PacketOut):
    // write on the socket.
  func Rcv(fmod FlowMod):
    // write on the socket.
  func Rcv(stat StatQuery):
    // write on the socket.

while data = read(socket):
  switch data:
    case OFPacketIn:
      Emit(PacketIn{…})
    case OFStatReply:
      Emit(StatReply{…})
```

```
<table>
<thead>
<tr>
<th>Mac2Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>03:03:04:04:05:05 P2</td>
</tr>
<tr>
<td>03:02:01:01:02:03 P1</td>
</tr>
</tbody>
</table>
```
constituent concurrency

```python
app LearningSwitch:
    func Rcv(pin PacketIn):
        switches = Dict("Switches")
        m2p = switches[pin.Switch]
        m2p[pin.SMACK] = pin.InPort
        switches[pin.Switch] = m2p
        p = m2p[pin.DMAC]
        if not p:
            ReplyTo(pin, PacketOut{FLOOD})
        else:
            ReplyTo(pin, FlowMod{
                Output{Port: p},
            })
```

<table>
<thead>
<tr>
<th>Switch</th>
<th>Mac2Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>01:02:03:04:05:06 -&gt; P1</td>
</tr>
<tr>
<td></td>
<td>06:05:04:03:02:01 -&gt; P2</td>
</tr>
<tr>
<td>S₂</td>
<td>03:03:04:04:05:05 -&gt; P2</td>
</tr>
<tr>
<td></td>
<td>03:02:01:01:02:03 -&gt; P1</td>
</tr>
</tbody>
</table>
**CONSISTENT CONCURRENCY**

Thread 1

```go
app LearningSwitch:
    func Rcv(pin PacketIn):
        switches = Dict(“Switches”)
        m2p = switches[pin.Switch]
        m2p[pin.SMAC] = pin.InPort
        switches[pin.Switch] = m2p
        p = m2p[pin.DMAC]
        if not p:
            ReplyTo(pin, PacketOut{FLOOD})
        else:
            ReplyTo(pin, FlowMod{
                Output{Port: p},
            })
```

<table>
<thead>
<tr>
<th>Switch</th>
<th>Mac2Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>01:02:03:04:05:06 -&gt; P1 06:05:04:03:02:01 -&gt; P2</td>
</tr>
<tr>
<td>S2</td>
<td>03:03:04:04:05:05 -&gt; P2 03:02:01:01:02:03 -&gt; P1</td>
</tr>
</tbody>
</table>
**CONSISTENT CONCURRENCY**

**Thread 1**
```
PacketIn{S₁}  PacketIn{S₁}
```

<table>
<thead>
<tr>
<th>Switch</th>
<th>Mac2Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>01:02:03:04:05:06 -&gt; P1</td>
</tr>
<tr>
<td></td>
<td>06:05:04:03:02:01 -&gt; P2</td>
</tr>
</tbody>
</table>

```go
app LearningSwitch:
  func Rcv(pin PacketIn):
    switches = Dict("Switches")
    m2p = switches[pin.Switch]
    m2p[pin.SMAC] = pin.InPort
    switches[pin.Switch] = m2p
    p = m2p[pin.DMAC]
    if not p:    
      ReplyTo(pin, PacketOut{FLOOD})
    else:
      ReplyTo(pin, FlowMod{Output{Port: p}},
```

**Thread 2**
```
PacketIn{S₂}  PacketIn{S₂}
```

<table>
<thead>
<tr>
<th>Switch</th>
<th>Mac2Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₂</td>
<td>03:03:04:04:05:05 -&gt; P2</td>
</tr>
<tr>
<td></td>
<td>03:02:01:01:02:03 -&gt; P1</td>
</tr>
</tbody>
</table>

```go
app LearningSwitch:
  func Rcv(pin PacketIn):
    switches = Dict("Switches")
    m2p = switches[pin.Switch]
    m2p[pin.SMAC] = pin.InPort
    switches[pin.Switch] = m2p
    p = m2p[pin.DMAC]
    if not p:    
      ReplyTo(pin, PacketOut{FLOOD})
    else:
      ReplyTo(pin, FlowMod{Output{Port: p}},
```

---

77
How do we infer the cells?

1. By processing the message handlers at compile-time.
2. By running the message handlers at run-time.
3. Can be explicitly provided by the programmer:

```go
Rcv(p PacketIn):
    m2p := Dict("Switches")[p.Switch]
    m2p[p.SMAC] = p.Port
    Dict("Switches")[p.Switch] = m2p

Map(p PacketIn):
    return ["Switches", p.Switch]
```

**CONSISTENT CONCURRENCY**
FUN FACTS ABOUT BEES

- Bees are single threaded:
  - No locks needed in app functions.
- Bees can be made sticky:
  - The optimizer does not migrate a sticky bee.
- Queen bees assign cells to bees:
  - They act as a router for each application.
TRANSACTIONS

Platform Level

Functions are transactional:
If there is an error, state is unmodified and no message is emitted.

Application Level

Forwarding rules are also transactional (Beyond the scope of this paper).
Applications can proxy remote bees/hives if needed.

Local quorums in LANs, Async messages in WAN
CONSISTENCY

Application

Function 1

k1
k2

Function 2

k3
k4
k5

msg 3  msg 2  msg 1
CONSISTENCY

Application

Function 1

Function 2

k1
k2
k3
k4
k5

msg 1
msg 2
msg 3
CONSISTENCY
CONSISTENCY

Function 1

Function 2

msg 3

msg 1
k2
k4

msg 2
k2
k3
k5

Application

k1
k2

k3
k4
k5
By default we create a local bee to handle requests for cells that are not assigned.

But, we can also enable applications to define where to allocate the new bee:

- Random
- Local
- Rendezvous Hashing
- …
CONTROL PLATFORM
CONTROL PLATFORM
CONTROL PLATFORM
CONTROL PLATFORM
CONTROL PLATFORM

[Diagram showing two半头蜂箱, labeled Hive, with switches marked as "Switch"]

91
EXAMPLE

OpenFlow Driver

SwitchJoined(S)

Traffic Engineering

Init
Query
Collect
Route

func Init(sw, s):
  s.Set(FlowStat(switch))

on SwitchJoined:
  Init(msg.sw, S[msg.sw])
func Init(sw, s):
    s.Set(FlowStat(switch))

on SwitchJoined:
    Init(msg.sw, S[msg.sw])
func Query(s):
    emit(FlowStatQuery(s))

on Timeout(1sec):
    for each s in S:
        Query(s)
func Query(s):
emit(FlowStatQuery(s))

on TimeOut(1sec):
for each s in S:
Query(s)
func Query(s):
    emit(FlowStatQuery(s))

on TimeOut(1sec):
    for each s in S:
        Query(s)
func Query(s):
    emit(FlowStatQuery(s))

on TimeOut(1sec):
    for each s in S:
        Query(s)
func Collect(r, s):
    s.append(flow stats in r)

on StatReply(r):
    Collect(r, S[r.switch])
func Collect(r, s):
s.append(flow stats in r)

on StatReply(r):
Collect(r, S[r.switch])
func Collect(r, s):
s.append(flow stats in r)

on StatReply(r):
    Collect(r, S[r.switch])
func Route(S, T):
    if Change in S > Δ:
        Use T to reroute flows.

on TimeOut(1sec):
    Route(S, T)
**EXAMPLE** Learning Switch

Application

```python
dict S
func switch(pkt, sw):
    sw[pkt.src_mac] = pkt.in_port
    if p = sw[pkt.dst_mac]:
        emit(flow_mod(pkt.dst_mac -> p))
        emit(packet_out(pkt -> p))
    else:
        emit(packet_out(pkt -> flood))

on pkt_in(pkt):
    switch(pkt, S[pkt_in.dpid])
```

102
this application has dictionary $S$ and function $\text{switch}$

```
Application

dict S
func switch(pkt, sw):
    sw[pkt.src_mac] = pkt.in_port
    if p = sw[pkt.dst_mac]:
        emit(flow_mod(pkt.dst_mac -> p))
        emit(packet_out(pkt -> p))
    else:
        emit(packet_out(pkt -> flood))

on pkt_in(pkt):
    switch(pkt, S[pkt_in.dpid])
```
EXAMPLE Learning Switch

Application

```python
dict S
func switch(pkt, sw):
    sw[pkt.src_mac] = pkt.in_port
    if p = sw[pkt.dst_mac]:
        emit(flow_mod(pkt.dst_mac -> p))
        emit(packet_out(pkt -> p))
    else:
        emit(packet_out(pkt -> flood))

on pkt_in(pkt):
    switch(pkt, S[pkt_in.dpid])
```

```python
on pkt_in messages call switch
```
**EXAMPLE**  Learning Switch

**Application**

```python
dict S
func switch(pkt, sw):
    sw[pkt.src_mac] = pkt.in_port
    if p = sw[pkt.dst_mac]:
        emit(flow_mod(pkt.dst_mac -> p))
        emit(packet_out(pkt -> p))
    else:
        emit(packet_out(pkt -> flood))

on pkt_in(pkt):
    switch(pkt, S[pkt_in.dpid])
```

pass the entry for

`pkt_in.dpid in S` to `switch`
**EXAMPLE**  Learning Switch

**Application**

```python
dict S
func switch(pkt, sw):
    sw[pkt.src_mac] = pkt.in_port
    if p = sw[pkt.dst_mac]:
        emit(flow_mod(pkt.dst_mac -> p))
        emit(packet_out(pkt -> p))
    else:
        emit(packet_out(pkt -> flood))

on pkt_in(pkt):
    switch(pkt, S[pkt_in.dpid])
```

`switch` makes decision based on entries in `sw`. 
### Example

**Learning Switch**

**Application**

```python
dict S
func switch(pkt, sw):
    sw[pkt.src_mac] = pkt.in_port
    if p = sw[pkt.dst_mac]:
        emit(flow_mod(pkt.dst_mac -> p))
        emit(packet_out(pkt -> p))
    else:
        emit(packet_out(pkt -> flood))

on pkt_in(pkt):
    switch(pkt, S[pkt_in.dpid])
```

*switch* emits messages to forward the packet.
# Learning Switch

<table>
<thead>
<tr>
<th>Switch</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0B:0E:0E:04:01:05 —&gt; 1</td>
</tr>
<tr>
<td></td>
<td>07:08:09:10:11:12 —&gt; 3</td>
</tr>
<tr>
<td>2</td>
<td>0A:0B:0C:0D:0E:0F —&gt; 1</td>
</tr>
<tr>
<td></td>
<td>0F:0E:0D:0C:0B:0A —&gt; 2</td>
</tr>
</tbody>
</table>
**EXAMPLE**  Learning Switch

To ensure consistency: *each key must be accessed on only one machine.*