BEEHIVE

Towards a simple abstraction for scalable software-defined networking

SOHEIL HASSAS YEGANEH    YASHAR GANJALI
University of Toronto
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

Application
Controller

Switch

Application
Controller

Switch

Switch
Existing Distributed Controllers

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

Much better than traditional networks

Still Hard to Program Distributed Systems
Our goal is to make it easy to program distributed systems, similar to centralized controllers, with optimized placements and application analytics.
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

what is a control application?

Process async messages in application functions using state dictionaries

![Diagram showing application, function, and dictionaries relationship]
ABSTRACTION

how do applications communicate?

async messages
all functions

state dictionaries
functions of the
same application
EXAMPLE

**Initializes dictionary**

- SwitchJoined\(s_i\)
- Timeout
- StatQuery\(s_i\)
- StatResult\(s_i\)
- Timeout
- FlowMod

**Queries switches**

**Collects stat results**

**Reroutes flows, if needed**

**Traffic Engineering**

- **Init**
- **Query**
- **Collect**
- **Route**

**Statistics**

**Topology**
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.*
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>
</table>
| 1      | flow1 -> stat  
|        | flow2 -> stat  |

Machine 1

Switch Entry

1
S
T
Init
Query
Collect
Route

Traffic Engineering

<table>
<thead>
<tr>
<th>Switch</th>
<th>Entry</th>
</tr>
</thead>
</table>
| 2      | flow3 -> stat  
|        | flow4 -> stat  |

Machine 2

Switch Entry

2
S
T
Init
Query
Collect
Route
**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

<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>

Traffic Engineering

- Init
- Query
- Collect
- Route

Machine 1

Machine 2
**CONSISTENCY**

<table>
<thead>
<tr>
<th></th>
<th>msg 3</th>
<th>k1</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg 2</td>
<td>k2 k3 k5</td>
<td></td>
</tr>
<tr>
<td>msg 1</td>
<td>k2 k4</td>
<td></td>
</tr>
</tbody>
</table>

Application

**Function 1**

k1
k2

**Function 2**

k3
k4
k5
CONSISTENCY
We need a runtime that steers messages among application instances while preserving consistency.
CONTROL PLATFORM

Hive + Cell + Bee
CONTROL PLATFORM

Hive
- is the controller
- provides the boilerplates (e.g., locking, consistency, …)
- can run on a separate machine
Cell • an entry in a dictionary of a specific application
• e.g., (TE, S, $s_i$, stats of $s_i$)
CONTROL PLATFORM

- a lightweight thread of execution
- process messages
- exclusively owns a set of cells
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.

```go
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

Switch  Switch  Switch

TE  CI

Hive

TE  CI

Hive

Switch
MIGRATION

Switch  Switch  Switch

Switch
MIGRATION

Switch  Switch  Switch
MIGRATION

Switch  Switch  Switch

Switch
This is not optimal and can happen often.
MIGRATION

Switch  Switch  Switch

Switch
MIGRATION

Switch  Switch  Switch

Switch
MIGRATION
When/where should we migrate bees?

- NP-Hard problem
- We use a simple heuristic
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
ANALYTICS & FEEDBACK

Switch  Switch  Switch

Switch
ANALYTICS & FEEDBACK
ANALYTICS & FEEDBACK
ANALYTICS & FEEDBACK

Switch  Switch  Switch

well-balanced

Switch
Fault Tolerance

Colony of replicated bees all in consensus about their state.
**Generality**

**Centralized**

```go
func Centralized(msg):
    ...
    map Centralized(msg):
        return {(D, 0)}
```

**Kandoo**

```go
func Local(msg):
    ...
    map Local(msg):
        return {(D, hiveid)}
```

**NIB**

```go
func NIB(msg):
    ...
    map NIB(msg):
        return {(N, nodeid)}
```

**Virtual Networking**

```go
func VN(msg):
    ...
    map VN(msg):
        return {(VN, vnid)}
```

**Routing**

```go
func Router(msg):
    ...
    map Router(msg):
        return {(Adv, msg.n[0])}
```

+ you don’t need to think about placement and load balancing in most cases.
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
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.)

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.