High Performance Functional Programming for Software-Defined Network Control

Andi Voellmy

Yale University

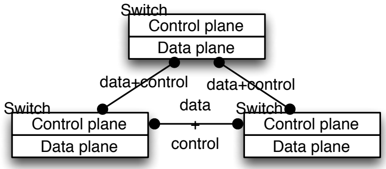
November 5, 2012

Software-Defined Networking (SDN)

SDN argues that networks should be easy to program.

Today's network infrastructure lacks programmability:

- Many commercial devices lack expressive, programmatic API.
- Typically requires programming distributed protocol, because control plane is distributed.



Traditional Network

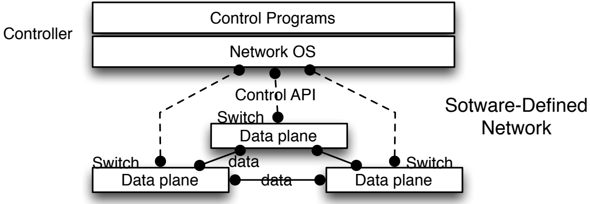
The SDN Paradigm

Write a program, not a protocol.

- Input: network state
- Output: device configurations

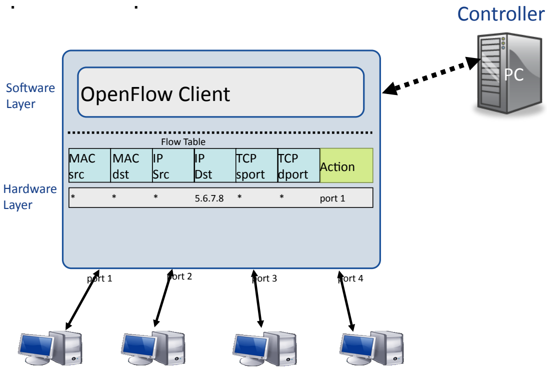
E.g. Program Dijkstra algorithm, not distance-vector routing protocol.

Introduce a “Network OS” layer to implement programming abstraction.



SDN Technology: OpenFlow (OF)

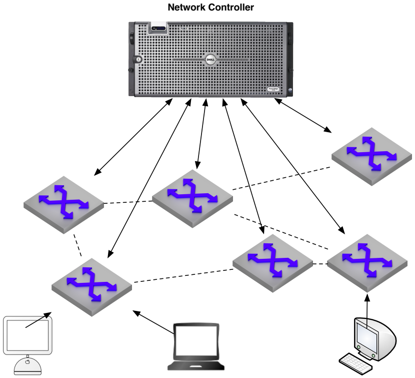
- Hardware-independent abstraction of today's Ethernet switches.
- Flexible forwarding abstraction.
- Specification statistics that are collected.
- Network protocol to configure forwarding behavior and read state.



Flow Table

MAC src	MAC dst	IP Src	IP Dst	TCP sport	TCP dport	Action
*	*	*	5.6.7.8	*	*	port 1

One Controller to Rule Them All



Write a single, ordinary program with global visibility to control the entire network.

Control Load in a Data Center

Tavakoli (HotNets'09) estimate data center demands:

- Hundreds to thousands of switches.
- 2 million virtual machines.
- 20 million new TCP sessions per second.
- 10 ms latency required.
- Best single-threaded controller (C++, CPU bounded) served 30K requests per second with 10ms latency.
- ∴ 667 controllers needed.

But there is some reason for hope

- Optimize controller's single-threaded speed: use fewer CPU cycles.
- Multicores: core counts are rapidly increasing and the entire multicore architecture is highly parallelized.

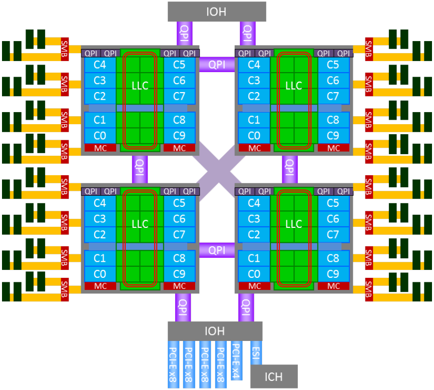


Figure : Intel Xeon Westmere E74 4 node, 10 core system

Multicore-Nettle (McNettle)

A library that allows users to write control programs with a simple concurrent programming model and that automatically executes effectively on multicore NUMA servers.

Simple control programs written with McNettle can serve over 20 million flow requests per second with sub 10ms response time, making use of about 50 cores.

Control programs are written in Haskell, a high-level functional programming language with many convenient programming features, especially high-level concurrency constructs.

Multicore-Nettle (McNettle)

A library that allows users to write control programs with a simple concurrent programming model and that automatically executes effectively on multicore NUMA servers.

Simple control programs written with McNettle can serve over 20 million flow requests per second with sub 10ms response time, making use of about 50 cores.

Control programs are written in Haskell, a high-level functional programming language with many convenient programming features, especially high-level concurrency constructs.

Multicore-Nettle (McNettle)

A library that allows users to write control programs with a simple concurrent programming model and that automatically executes effectively on multicore NUMA servers.

Simple control programs written with McNettle can serve over 20 million flow requests per second with sub 10ms response time, making use of about 50 cores.

Control programs are written in Haskell, a high-level functional programming language with many convenient programming features, especially high-level concurrency constructs.

Outline

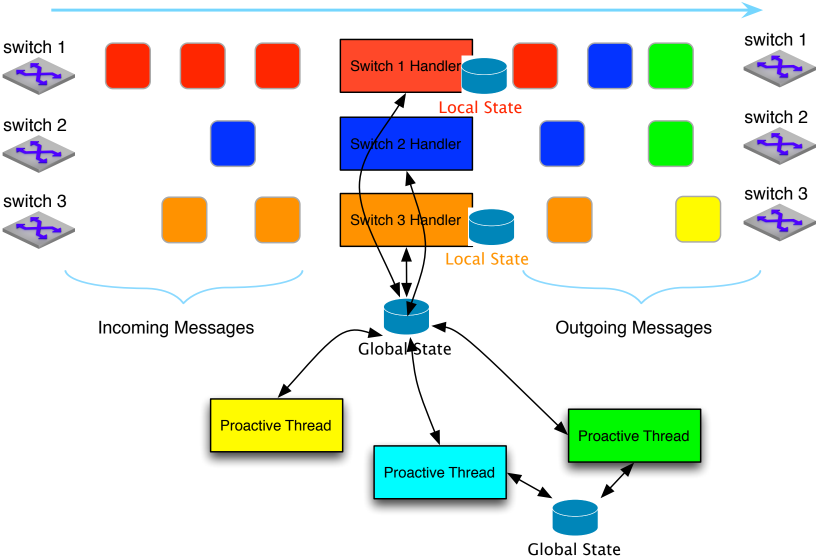
1 Introduction

2 McNettle Programming Model

3 McNettle Implementation

4 Evaluation

Programming Model: Proactive Threads



McNettle API in a Nutshell

Starting the control server:

$$startServer :: ServerParams \to (Features \to IO\ Handler) \to IO\ ()$$

A *Handler*s is a function from messages to computations that output commands:

$$\textbf{type}\ Handler = SwitchMessage \to Reaction\ ()$$

Messages represented by a data type:

$$\textbf{data}\ SwitchMessage = Init\ Features \mid PacketIn\ PacketIn \mid \ldots$$

Computations that send commands to switches:

$$forward :: PacketIn \to ForwardingAction \to Reaction\ ()$$
$$forwardWithExactRule :: PacketIn \to ForwardingAction \to \ldots$$
$$\ldots$$

McNettle API in a Nutshell

Starting the control server:

$$\mathit{startServer} :: \mathit{ServerParams} \to (\mathit{Features} \to \mathit{IO}\ \mathit{Handler}) \to \mathit{IO}\ ()$$

A *Handler*s is a function from messages to computations that output commands:

$$\textbf{type}\ \mathit{Handler} = \mathit{SwitchMessage} \to \mathit{Reaction}\ ()$$

Messages represented by a data type:

$$\textbf{data}\ \mathit{SwitchMessage} = \mathit{Init}\ \mathit{Features} \mid \mathit{PacketIn}\ \mathit{PacketIn} \mid \ldots$$

Computations that send commands to switches:

$$\begin{array}{l} \mathit{forward} :: \mathit{PacketIn} \to \mathit{ForwardingAction} \to \mathit{Reaction}\ () \\ \mathit{forwardWithExactRule} :: \mathit{PacketIn} \to \mathit{ForwardingAction} \to \ldots \\ \ldots \end{array}$$

McNettle API in a Nutshell

Starting the control server:

$$\mathit{startServer} :: \mathit{ServerParams} \rightarrow (\mathit{Features} \rightarrow \mathit{IO}\ \mathit{Handler}) \rightarrow \mathit{IO}\ ()$$

A *Handlers* is a function from messages to computations that output commands:

$$\textbf{type}\ \mathit{Handler} = \mathit{SwitchMessage} \rightarrow \mathit{Reaction}\ ()$$

Messages represented by a data type:

$$\textbf{data}\ \mathit{SwitchMessage} = \mathit{Init}\ \mathit{Features} \mid \mathit{PacketIn}\ \mathit{PacketIn} \mid \ldots$$

Computations that send commands to switches:

$$\mathit{forward} :: \mathit{PacketIn} \rightarrow \mathit{ForwardingAction} \rightarrow \mathit{Reaction}\ ()$$
$$\mathit{forwardWithExactRule} :: \mathit{PacketIn} \rightarrow \mathit{ForwardingAction} \rightarrow \ldots$$

...

McNettle API in a Nutshell

Starting the control server:

```
startServer :: ServerParams → (Features → IO Handler) → IO ()
```

A *Handlers* is a function from messages to computations that output commands:

```
type Handler = SwitchMessage → Reaction ()
```

Messages represented by a data type:

```
data SwitchMessage = Init Features | PacketIn PacketIn | ...
```

Computations that send commands to switches:

```
forward :: PacketIn → ForwardingAction → Reaction ()
forwardWithExactRule :: PacketIn → ForwardingAction → ...
...
```

McNettle Examples

Flood everything:

```
main = runServer params makeHandler
makeHandler features = return handler
handler (PacketIn pkt) = forward pkt flood
handler _              = return ()
```

Clear table at startup:

```
handler (Init features) = clearTable (switchID features)
```

McNettle Examples

Flood everything:

```
main = runServer params makeHandler
makeHandler features = return handler
handler (PacketIn pkt) = forward pkt flood
handler _              = return ()
```

Clear table at startup:

```
handler (Init features) = clearTable (switchID features)
```

McNettle Examples

Install a flood rule:

```
handler (Init features) =
  do clearTable (switchID features)
     installRule (switchID features) any flood defaultOpts
```

McNettle Examples: MAC Learning

```
handler table (PacketIn pkt) =
  do let hdr = etherHeader pkt
     insertMac (etherSrc hdr) (inPort pkt) table
     result ← lookupMac (etherDst hdr) table
     case result of
       Nothing  → forward pkt flood
       Just port →
         forwardWithExactRule pkt (phyPort port) defaultOpts
```

McNettle Examples: Local vs. Global State

Local:

```
main = runServer params makeHandler
makeHandler features =
   do table ← newMacTable
      return (handler table)
```

Global:

```
main = do table ← ConcurrentMacTable.newMacTable
          runServer params (makeHandler table)
makeHandler table features = return (handler table)
```

McNettle Examples: Local vs. Global State

Local:

```
main = runServer params makeHandler
makeHandler features =
  do table ← newMacTable
     return (handler table)
```

Global:

```
main = do table ← ConcurrentMacTable.newMacTable
          runServer params (makeHandler table)
makeHandler table features = return (handler table)
```

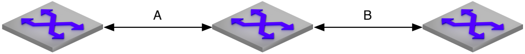
Programming Concurrent Shared State

Haskell provide us with several options; use each where appropriate:

- CAS, atomic-update for busy-waiting synchronization.
- *MVar* for blocking synchronization.
- Software Transactional Memory (STM) offers high-level, composable concurrency constructs.

STM Example: Bandwidth Reservation

- Applications request a path with reserved bandwidth.
- Many requests are received concurrently at origin different switches.
- Programming with fine-grained locks on each link capacity can lead to deadlocks, e.g.
  - executing *reservation* (A, B) in thread 1
  - *reservation* (B, A) in thread 2
  - *lock* (t_1, A); *lock* (t_2, B) deadlock



STM Example: Bandwidth Reservation

STM makes this easy: use a transaction variable *TVar* to hold the capacity on each link; put them all in a dictionary indexed by link.

To reserve a link:

```
reserveLink vars amt link =
  do let v = varForLink vars link
     current ← readTVar v
     let remaining = current − amt
     when (remaining < 0) retry
     writeTVar v remaining
```

Executing *retry* causes transaction to try again later, blocking the caller.

Implementation waits until a *TVar* accessed by the transaction is updated.

To reserve a path:

$$\begin{aligned}&\mathit{reservePath}\ \mathit{vars}\ \mathit{amt}\ \mathit{path}\\ &\quad = \mathit{forM}\ \mathit{path}\ (\lambda \mathit{link} \rightarrow \mathit{reserveLink}\ \mathit{vars}\ \mathit{amt}\ \mathit{link})\end{aligned}$$

To execute it, blocking until it succeeds:

$$\mathit{atomically}\ (\mathit{reservePath}\ \mathit{vars}\ \mathit{amt}\ \mathit{path})$$

Notice:

- Deadlock impossible
- Atomicity: either every link on the path or no links are updated.
- Isolation: a partially completed transaction can't cause another to fail.
- Composable: Path reservation composed of smaller transactions.

Outline

1 Introduction

2 McNettle Programming Model

3 McNettle Implementation

4 Evaluation

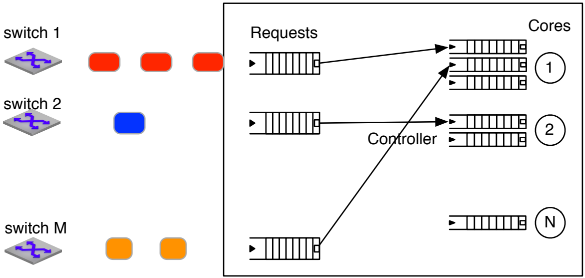
McNettle Implementation

Fast and scalable implementations of key components:

- Scheduling work on cores
- IO event notification
- OF protocol implementation
- Garbage collection (GC)

McNettle Implementation: Scheduler

Schedule work on cores efficiently, subject to the constraint that messages from each switch are processed in order.

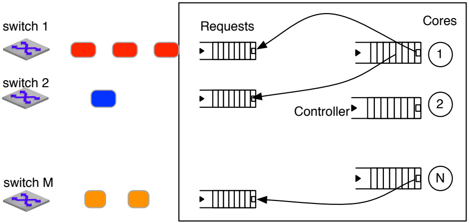


McNettle Implementation: Scheduler

Piggyback on GHC's thread scheduler: use one Haskell thread per switch, exclusively servicing that switch's messages.

GHC Scheduler:

- One run queue per core.
- Each core services queue in round robin order.
- Work-pushing load balancing.



Scheduler Optimization: Stabilize Load Balancing

Between running threads, scheduler checks for idle cores, divides work evenly among idle cores.

Problem: Under light load, cores are often idle and switch handlers are transferred back and forth between cores; bad for cache locality of switch handlers and causes contention on run queues.

Solution: Keep running average of run queue lengths of each core and transfer only to improve the balance.

GHC's IO Manager

To wait for a socket to be ready, a thread:

- creates an empty *MVar*
- inserts the *MVar*, file descriptor and event in a callback table, and registers with global epoll object,
- blocks on *MVar*, causing scheduler to remove the thread from run queue.
- when unblocked, remove callback and unregister from epoll object.

GHC includes an IO manager thread that loops on *epoll_wait*

- For each ready FD: fill the *MVar*s of waiters, causing scheduler to move waiters to run queues.

Problems in GHC's IO Manager

1. Global callback table is protected by a single, heavily contended lock
   - unblocked threads have to remove their *MVar*s and then interfere.
   - threads with nothing to do prevent IO loop from notifying threads with something to do!
2. Single IO loop is sequential bottleneck

Bottlenecks caused large latencies.

Parallelized IO Manager

We implemented a new IO manager:

- Parallelized IO loop: one loop per core.
- Reduce contention on callback table: a waiter registers with the callback table on the core it is running on.
- Streamline IO loop

70x reduction in latency using work balancing stabilization & parallel IO manager versus GHC 7.4

Garbage Collection

GC is a significant cost.

Using large (128MB) nursery size reduces GC frequency, improving performance.

But then pause times became large:

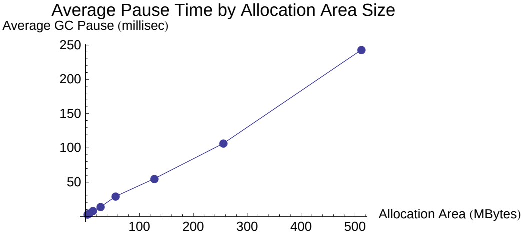


Figure : Average GC pause time for different allocation areas (generation 0) when running the local learning controller with 40 cores and 500 switches.

GC Pause Times

- GHC's GC is parallel, stop the world, generational collector.
- Each core has a private nursery to allocate from.
- At end of GC, nurseries are cleared; requires traversing block descriptors for the entire nursery; worse on large nurseries.
- Sequential in GHC 7.4.1
- Embarassingly parallel. Modify to have each core clear its nursery in parallel. Parallelizes and avoids cache thrashing.
- Reduces pause times by 10x.
- With 128 MB allocation area: under 10ms average pause time.

Outline

1 Introduction

2 McNettle Programming Model

3 McNettle Implementation

4 Evaluation

Hardware: Control Server

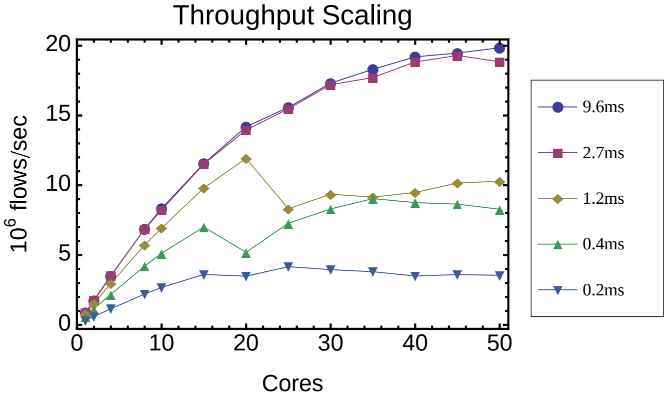
- 8 Intel(R) Xeon(R) CPU E7- 8850 2.00GHz,
- each with 10 cores, 24MB smart cache, 32MB L3 cache
- Four 10Gbps Intel NICs
- Linux 3.4.0.0, ixgbe 3.9.17 drivers

Hardware: Workload Servers and Switch

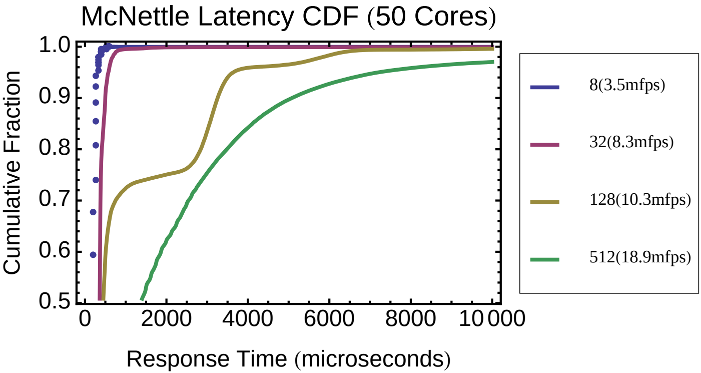
Generate workload by simulating switches on up to 9 workload servers and 12 NICs.

All servers networked with 10Gbps or 1Gbps ports through an HP switch.

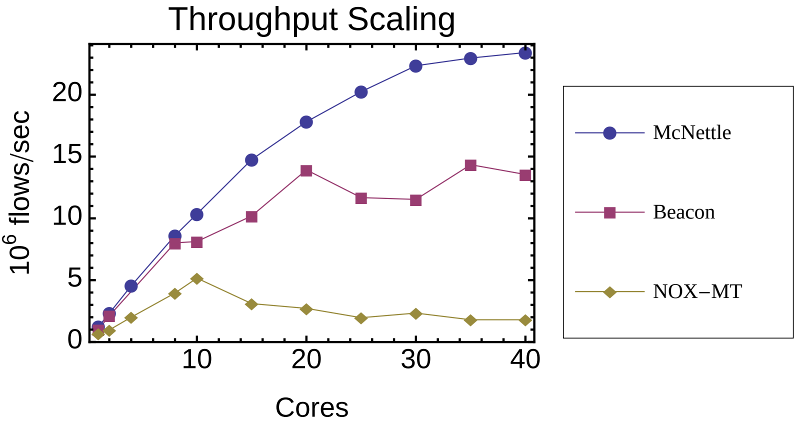
Throughput for MAC Learning



Latency for MAC Learning



Comparison with Beacon, NOX-MT


Throughput Scaling
10^6 flows/sec
Cores
McNettle
Beacon
NOX−MT