Scalable Software Defined Network Controllers

Andreas Voellmy
Yale University
New Haven, CT, U.S.A.
andreas.voellmy@yale.edu

Junchang Wang
University of Science and Technology of China
Hefei, China
wangjc@mail.ustc.edu.cn

ABSTRACT

Software defined networking (SDN) introduces centralized controllers to dramatically increase network programmability. The simplicity of a logical centralized controller, however, can come at the cost of control-plane scalability. In this demo, we present McNettle, an extensible SDN control system whose control event processing throughput scales with the number of system CPU cores and which supports control algorithms requiring globally visible state changes occurring at flow arrival rates. Programmers extend McNettle by writing event handlers and background programs in a high-level functional programming language extended with shared state and memory transactions. We implement our framework in Haskell and leverage the multicore facilities of the Glasgow Haskell Compiler (GHC) and runtime system. Our implementation schedules event handlers, allocates memory, optimizes message parsing and serialization, and reduces system calls in order to optimize cache usage, OS processing, and runtime system overhead. Our experiments show that McNettle can serve up to 5000 switches using a single controller with 46 cores, achieving throughput of over 14 million flows per second, near-linear scaling up to 46 cores, and latency under 200 μs for light loads and 10 ms with loads consisting of up to 5000 switches.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Centralized networks* ; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages, Haskell*; D.1.3 [**Programming Techniques**]: Concurrent Programming; C.5.5 [**Computer System Implementation**]: Servers

Keywords

Software-defined Networking, Haskell, OpenFlow, Multicore

1. INTRODUCTION

Network systems are becoming more feature-rich and complex, and system designers often need to modify network software in order to achieve their requirements. Software-defined networking attempts to move as much network functionality as possible into user-definable software, making more of the network system components programmable. In particular, SDN architectures introduce a centralized control server (controller) to allow potentially dramatically simplified, flexible network programming.

Unfortunately, as the network scales up—both in the number of switches and the number of end hosts—the SDN controller can become a key bottleneck. Specifically, Tavakoli et al. [7] estimate that a large data center consisting of 2 million virtual machines may generate 20 million flows per second. On the other hand, current controllers, such as NOX [2] or Nettle [8], about 10^5 flows per second [7].

Previous work [7, 4] has made use of distributed controllers to achieve scalability, preventing programmers from sharing state at high transaction rates. With McNettle, we demonstrate a highly scalable SDN control framework that executes on shared-memory multicore servers, preserving a simple and natural programming model for controller developers.

2. PROGRAMMING MODEL

McNettle programs consist of a collection of message handlers, one for each switch in the network. The message handlers include a function which is applied whenever a packet-miss message is sent by a switch in the network. The packet-miss function updates *switch-local* and *network* state variables, and may decide on actions to provision flows in the network. Messages from each switch are handled sequentially, while messages from different switches are executed concurrently. As a result, switch-local state can be accessed without synchronization, while access to network state must be typically be synchronized in order to preserve correctness. To simplify programming concurrent global state access, we emphasize the use of *memory transactions*, which allows programmers to delineate sections of code that should execute atomically. Memory transactions are especially appropriate in this domain, since shared state is often accessed in unpredictable ways, depending on dynamically maintained graph data structures. Consider a controller which accepts requests to reserve bandwidth between pairs of hosts. Such a controller must keep track of available bandwidth on each link in the network, and must update these amounts when reservations are fulfilled. The particular links updated for a reservation depend on the network routing, which typically varies dynamically, and is therefore difficult to predict.

Memory transactions allow us to easily implement a concurrent reservation program that uses fine-grained, optimistic concurrency. Figure 1 shows the code to reserve bandwidth along a path, and consists of two functions. *reservePath* ex-

SIGCOMM'12, August 13–17, 2012, Helsinki, Finland.
ACM 978-1-4503-1419-0/12/08.

```
reservePath capTable amt path
  = forM path (λlink → reserveLink capTable amt link)
reserveLink capTable amt link =
  do current ← linkCapacity capTable link
     let remaining = current − amt
     when (remaining > 0) retry
     updateCapacity capTable link remaining
```

Figure 1: Bandwidth reservation using STM.

ecutes the *reserveLink* function on each link in a path and *reserveLink* checks if the link has enough available bandwidth and if so updates the link's available bandwidth. These functions execute as part of a transaction, which can be delineated and executed using an *atomically* statement:

atomically (*reservePath capTable amt path*)

McNettle provides a library of concurrent data structures that can be used to track network state and execute efficiently on multicore architectures. These libraries include several different concurrent hash tables that provide different atomicity guarantees and performance. We rely on GHC's STM system [3] to implement memory transactions.

3. IMPLEMENTATION

While McNettle draws on our earlier work on Nettle [8], achieving scaling on ccNUMA systems with tens of cores required redesigning many system components, from high-level programming APIs, to Haskell standard libraries, and Haskell runtime system components. In particular, McNettle avoids contention on sockets and ensures that each message is processed on a single CPU core, reducing inter-core synchronizations to only those required by user-specified controller logic. McNettle's API is redesigned to allow the McNettle implementation to safely reuse various message buffers, allowing the implementation to improve cache behavior and reduce load on the garbage collector, and more than double throughput of McNettle controllers. Furthermore, we eliminated a bottleneck in GHC's runtime system by fully parallelizing the waiting and dispatching of threads on IO devices, and stabilizing the multi-core load balancing algorithm of GHC to prevent excessive cache-thrashing due to repeated work migration. These improvements lead to a 50-fold reduction in latency in our controllers.

4. RESULTS

We evaluated throughput and latency of McNettle using a lightly modified version of the cbench [6] program, which simulates a collection of OpenFlow switches. We ran our controllers on a DELL Poweredge R815 server with 48 cores of AMD Opteron 1.7GHz 6164 processors and 64 GB memory, using 8 1Gbps and two 10Gbps NICs.

Figure 2 shows maximum throughput scaling results for McNettle, Beacon [1], a multithreaded OpenFlow controller platform in Java, and multi-threaded NOX [5], a multi-threaded OpenFlow platform written in C++, on a workload of 1000 switches with a very simple "learning switch" controller. We see that the McNettle controller outperforms the NOX controller for all loads with performance up to over six times of NOX. We see that the McNettle controllers scale up through 46 cores, while NOX controllers scale to 10 cores. Beacon obtains better throughput than McNettle for fewer than 30 cores, but stops scaling at 20 cores and obtains lower peak throughput than McNettle.

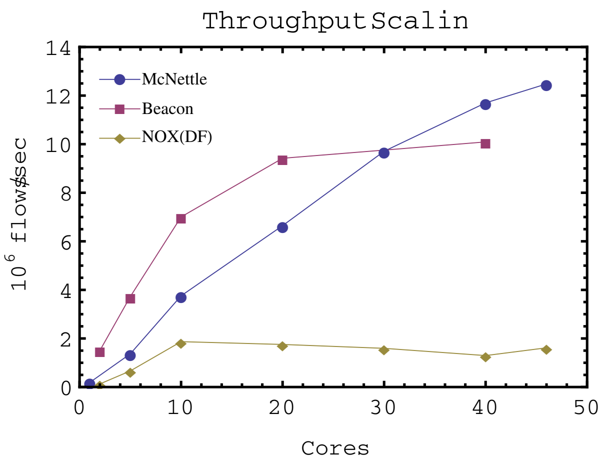

Figure 2: Throughput for McNettle, NOX destiny-fast branch, and Beacon learning controllers as a function of the number of cores.

5. DEMO

We will demonstrate the features of McNettle using example controllers including learning switch controllers, a topology discovery module, a bandwidth-on-demand controller using STM, and a parallelized shortest path routing computation. These controllers demonstrate (1) the basic structure of controllers, (2) switch-local and network state, (3) non-blocking synchronization, (4) transactional memory, and (5) deterministic parallel programming. We will run controllers on a local laptop and on a multicore server running at Yale University.

Acknowledgements This work was supported in part by a gift from Microsoft Research.

6. REFERENCES

[1] D. Erickson. Beacon. `https://openflow.stanford.edu/display/Beacon/Home`, 2012. [Online].

[2] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[3] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[4] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, 2010.

[5] noxrepo.org. Nox destiny-fast. `http://www.noxrepo.org/`, 2012. [Online; git commit e9c3da6bb12ad3fa0d2b609e697a50ce44ca19f4].

[6] R. Sherwood and K.-K. Yap. Cbench. `http://www.openflow.org/wk/index.php/Oflops/`, 2012. [Online; accessed 10-April-2012].

[7] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the datacenter.

[8] A. Voellmy and P. Hudak. Nettle: taking the sting out of programming network routers. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, pages 235–249, Berlin, Heidelberg, 2011. Springer-Verlag.