Frenetic
A Network Programming Language
Will Clarkson, Raoul Veroy, Matthew Ahrens

Warm-ups

- Domain of Frenetic
  - High-level programming of desired behavior of a network (distributed collection of switches and routers)
  - Make it easier to prototype and implement new network protocols
  - Network router/switch configuration
- Design Goals of Frenetic
  - Declarative
  - Modular (multiple rules can be composed safely)
  - Single-tier (don't need to program switches and controller separately)
  - Abstract away from primitive switch operations without compromising efficiency
  - Avoiding race conditions
  - Cost model
- Architecture and components of OpenFlow networks
  - Switches support only simple rules to dispatch packets
  - Rules are stored in a flow table on the switch along with a timeout, and a set of actions when processing the packet
  - Rules consist of a priority and a pattern
  - If no rule is matched, packet is sent to controller for processing
  - Controller often installs rules on the switch to deal with the rest of a flow of packets and when it encounters new packets
  - Switches are simple and fast, controllers are capable and multi-purpose but much slower
- Advantages of OpenFlow over previous state of the art
  - Replaced a variety of proprietary solutions for programming switches
  - Single interface to program devices from multiple manufacturers across network
  - May have required multiple languages or different manual configurations to achieve a network protocol
- Weaknesses of OpenFlow
  - Not composable (first rule to encounter a packet consumes it, hard for statistics, etc)
  - Low level (commands map directly to switch and router capabilities and hardware concerns, not what the programmer wants)
  - Two-tiered (programmer must reason about writing program for controller and programs for switches)
  - Race conditions (provides no runtime system to ensure that race conditions are avoided)
Design Evaluation

- Abstractions provided by Frenetic
  - Query language with no side effects (good for statistics)
  - Packets represented as a stream (functional reactive)
  - Operations for composing queries
  - Listeners to consume and act on streams
  - Runtime system avoids race conditions, handles installing and uninstalling rules efficiently
- Code in *A First Example*
  - Declare two rules to port forward
  - When a switch joins the network, create a policy tailored to it with the rules and install it on the switch
  - Result is that every switch on the network will forward packets from port 1 -> 2 and 2 -> 1
- Compositionality
  - Read-only nature of queries, multiple rules can be applied to each packet without manually looping them back to the same switch
- Race-free semantics
  - Semantic information of Frenetic queries means that, unlike OpenFlow, the runtime system can know what behavior you want and intelligently drop redundant packets to achieve that goal
- Cost model
  - Packets appear in microflows
  - The number of microflows is typically small compared to the number of packets
  - Only the first packet in a microflow will need to be directed to the controller for special processing, rules will be installed so subsequent packets in the same microflow can be handled directly on switch
  - Remember: switches are fast and controllers are slow
  - The number of round trips to the controller will be low
  - Most packets will stay on fast path of switches
  - Throughput will be high
- Language features that enable cost model
  - Limit() and Every() queries enable controlled sampling of large packet flows to minimize traffic to the controller while still providing desired statistics
- Implementation of Frenetic
  - A combinator library in Python
  - Calls the NOX Python API
  - Runtime system to manage installing rules, maintaining race-free semantics and other abstractions
- Evaluation of Frenetic
  - Compared lines of code with NOX OpenFlow implementations of protocols
○ Compared traffic to controller (since it is the bottleneck, it presumably provides a good proxy for total throughput and efficiency of network protocol implementation)
○ Could have compared performance for a real-world case study (i.e. a real business or campus network)
○ Could have compared with NOX code written by NOX experts, not by the same researchers

Evaluating Frenetic as a DSL

● Advantages of embedding in Python
  ○ Easy to learn and use (familiar base syntax)
  ○ Close integration with C and C++, common networking languages (Python has a good FFI)
  ○ High-level language fits well with paradigm of Frenetic
  ○ Better chance of actually being used
● Disadvantages of embedding in Python
  ○ Type system provides very minimal compile-time guarantees of correctness
  ○ Dynamic type checking has to be implemented yourself
  ○ Python is not known for high performance
  ○ You may have to support your software as people will actually use your software
● Contributions of runtime system
  ○ Layer between abstract description of network and concrete calls to NOX
  ○ Maintains desired language semantics
  ○ Allows for single-tier abstraction
● Contributions of type system
  ○ Streams appear as events where data originates (queries, switch join, etc)
  ○ Events are manipulated by event functions (types like event -> event)
  ○ Events are consumed by listeners which can perform actions like printing to screen or installing a rule on a switch (types like event -> side effect)
● Features allowing compositionality
● Frenetic specific libraries
  ○ No libraries as authors claim compositionality
  ○ Need some basic monitoring and forwarding libraries to use as examples
● Tool support
  ○ Dynamic type checker exists
  ○ Need for a debugger and more libraries
● Is it a DSL?
  ○ Feels like a library with some DSL elements such as the overloading of the >> operator
● Is it easier to write in Frenetic than Nox?
  ○ It is easier to write common functionality in fewer lines in Frenetic than in Nox due to the higher level of abstraction
• Does the language achieve its goals?
  ○ Given the goals listed above, yes.
• How might the design be improved?
  ○ Implement in a language that allows static type checking, one that can actually be used in production/enterprise situations
• Do you like the language? the paper?
  ○ Yes and yes.
  ○ Convinced of good ideas