Sphene Software Design Document

MATT SPARKS ALI YAHYA SAM D'AMICO

April 16, 2014

Abstract

This document provides a high level overview of the software design of Sphene, our group's dynamic router for CS 344. It addresses the architecture of our software and the design principles and data structures behind it. We also cover our testing strategy and other techniques that were beneficial during the development process. In the last section, we give a post-mortem highlighting some of the lessons learned from this project.

1 High-level architecture

Our router is split into two components, the ControlPlane and the DataPlane. The DataPlane receives Ethernet packets from the wire and forwards those that it can. Packets that the DataPlane cannot forward due to a table miss or error are sent to the ControlPlane for further handling. This division was chosen to mimic the functionality of the NetFPGA hardware component of our router. Specifically, our software DataPlane forwards packets to the ControlPlane for the same reasons that our hardware forwards packets to the software. Similarly, the ControlPlane sends packets to the DataPlane, or the hardware, but not directly to the wire. This architecture, shown in Figure 1, helps clearly define the roles of each component in the system.

1.1 DataPlane

The DataPlane is a software implementation of the functionality that is implemented in our hardware. It is the entity responsible for making decisions about what to do with packets arriving on an inbound interface. The DataPlane does its job by referring to the RoutingTable to determine the IP address of the next hop along the route to the packet's destination. It also refers to the ARPCache to determine the Ethernet address associated with the next hop IP address.

[0]Paragraphs marked with a diamond (⧫) describe key insights about our design.

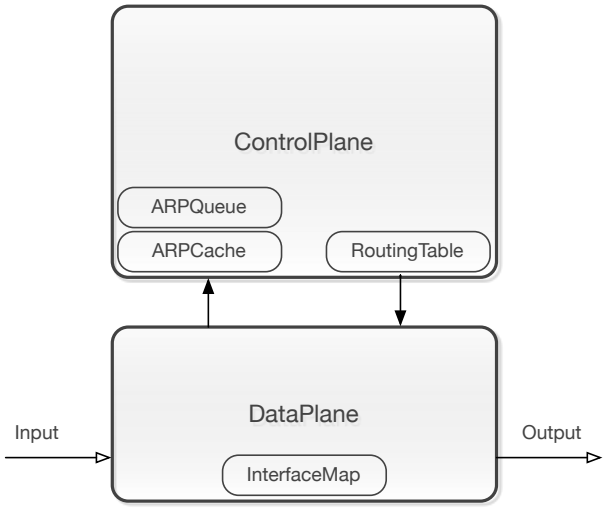

Figure 1: Logical split of the DataPlane and ControlPlane.

The DataPlane logically owns the InterfaceMap, a component that maps interface names and IP addresses to Interface objects. The software DataPlane uses its InterfaceMap similarly to how our hardware uses the IP filter table.

The ARPCache and RoutingTable are contained within the ControlPlane, as it is the ControlPlane that writes to them. However, our ControlPlane exposes access to these components as an optimization so that the DataPlane can have direct access to ARPCache and RoutingTable entries.

The DataPlane receives Ethernet packets from the wire and consults the ARPCache and RoutingTable components. There is a miss in either during the forwarding process, the DataPlane immediately forwards the packet to the ControlPlane and waits for more packets.

1.2 ControlPlane

The ControlPlane is responsible for the complex routing logic. Specifically, the ControlPlane manages the ARPQueue and updates the ARPCache, and generates ICMP messages in error cases. The ControlPlane implements the same logic as the DataPlane for packet processing, but packets causing ARPCache misses are queued in the ARPCache pending ARP replies.

In addition to sending ICMP error messages, the ControlPlane responds to ICMP echo requests and other packets destined for the router's interfaces. The ControlPlane implements the logic for handling GRE packets (see Section 7), and forwarding OSPF packets to the OSPFRouter component (see Section 6).

2 Component design and implementation

This section details the interfaces and implementation of the components that are accessed and managed by the DataPlane and ControlPlane. Only base IP routing components are discussed here; OSPF and GRE design are discussed in Sections 6 and 7 respectively.

2.1 ARPCache

Interface definition: `arp_cache.h`

The ARPCache is a map of `IPv4Addr`s to `ARPCache::Entry` objects. `ARPCache::Entry` objects contain the Ethernet MAC address, IP address, age of the entry, and entry type, static or dynamic. ARPCache entries are inserted through the `entryIs()` method and removed with the `entryDel()` method.

The ARPCache only maintains state and does not have any 'business logic'. Entries are timed out periodically by the ARPCacheDaemon as described in Section 2.5.1. Reactors may implement the `ARPCache::Notifiee` interface to receive notifications from the ARPCache. Notifications are dispatched for the insertion and deletion of ARPCache entries.

2.2 ARPQueue

Interface definition: `arp_queue.h`

The ARPQueue is a map of `IPv4Addr`s to `ARPQueue::Entry` objects. The `ARPQueue::Entry` objects contain a list of packets that are blocked on an ARP lookup. The ARPQueue is responsible for maintaining these lists of packets, along with the number of retries for an ARP request, and the request packet itself. When the ControlPlane encounters an ARP miss, the culprit packet is inserted into an entry in the ARPQueue, keyed on the IP address that caused the ARP miss. Subsequent packets missing on the same IP address will be queued in the existing entry in the ARPQueue. Once an ARP reply is received, the ControlPlane flushes all packets, in the order received, by passing them through the forwarding logic again.

Much like the ARPCache, the ARPQueue is a state-only object. The ControlPlane and the ARPQueueDaemon read and write the ARPQueue. The ARPQueueDaemon is responsible for resending ARP requests, and flushing queues for IP addresses that did not return ARP replies within the timeout. More details about the ARPQueueDaemon are provided in Section 2.5.2.

2.3 InterfaceMap

Interface definitions: `interface_map.h`, `interface.h`

Information about the router's interfaces, both hardware and virtual, are kept in Interface objects and stored in the InterfaceMap. The InterfaceMap is keyed on both the name, e.g. `eth0`, and the IP address of an Interface.

Interface objects contain all relevant details about an interface on the router, including the IP address, subnet mask, and type (hardware or virtual). Additionally, Interface objects provide methods for enabling and disabling an interface, and returning the calculated subnet and broadcast addresses.

Reactors may implement the `InterfaceMap::Notifiee` interface to receive notifications from the InterfaceMap when an interface is added or deleted, or when an interface is enabled or disabled.

2.4 RoutingTable

Interface definition: `routing_table.h`

The RoutingTable is an entity logically owned by the ControlPlane. In software, it is a map of *(subnet, subnet mask)* pairs (`IPv4Subnets`) to `RoutingTable::Entry` objects. The RoutingTable has a similar interface to the ARPCache; entries are added with the `entryIs()` method and deleted with `entryDel()`. Reactors can implement the `RoutingTable::Notifiee` class to receive notifications when the RoutingTable is changed. As discussed in Section 2.6 – HWDataPlane and SWDataPlane, this is the method by which the hardware forwarding table is updated.

RoutingTable entries contain all necessary information to completely describe routes. Specifically, the subnet, subnet mask, gateway, Interface, and type are all contained within `RoutingTable::Entry` objects. Entries can be either static or dynamic.

While the RoutingTable itself is state-only, the RoutingTable contains InterfaceMapReactor, which implements `InterfaceMap::Notifiee`. The InterfaceMapReactor automatically updates the routing table by adding and removing routes for interface subnets as interfaces are added and deleted, or enabled/disabled.

2.5 TaskManager

Interface definition: `task.h`

As detailed in Section 4 – Multithreading Issues, we only have one processing thread in our router. Thus, we needed some method of running periodic tasks for remov-

ing old entries in the ARP Cache, sending OSPF Hello packets, and so on. To solve this problem, we created a simple task simulator called TaskManager. The TaskManager contains a collection of `Task` objects and a 'virtual' time. As the virtual time is increased with the `timeIs()`, the new time is relayed to all Tasks in the collection. Based on the new time, a Task can run, or wait until the time is updated again. We also define a simple `PeriodicTask`, that has a defined period in seconds and runs at most once per period.

♦ The TaskManager approach is convenient in a number of ways. Most of all, we can accomplish the same functionality as multiple threads without worrying about the associated race conditions and deadlock issues. Moreover, using virtual time is very amenable to unit testing; our unit tests are able to exercise the TaskManager by defining tasks and stepping the virtual time of the TaskManager, without increasing the wall clock time by the same duration. As a result, we are able to run deterministic tests almost instantaneously, without having to wait several seconds of wall clock time.

There are a few periodic tasks that run in our router. These are described briefly below.

2.5.1 ARPCacheDaemon

The ARPCacheDaemon runs every second and scans the ARPCache for entries older than the maximum age, which is currently 30 seconds. Entries older than this max age are purged from the ARP Cache by this daemon.

2.5.2 ARPQueueDaemon

The ARPQueueDaemon also runs every second and iterates through the ARPQueue. ARP requests that have been sent fewer than 5 times are resent immediately. Otherwise, if no ARP reply is received after 5 requests, the ARPCacheDaemon removes the entry from the ARPQueue, effectively dropping all packets that were blocked on that ARP reply. In this case, the ARPCacheDaemon sends ICMP Destination Unreachable messages via the ControlPlane.

2.5.3 OSPFDaemon

The OSPFDaemon performs routine tasks to maintain the OSPF state. This includes timing out links and topology entries as well as sending Hello packets and LSUs. More details are provided in Section 6.2 – OSPFDaemon.

2.6 HWDataPlane and SWDataPlane

Our software maintains a `DataPlane::Ptr`, a smart pointer to a `DataPlane` object. However, this actually points to a derived instance of a `HWDataPlane` or a `SWDataPlane`, as shown in Figure 2. The type of DataPlane instantiated is determined

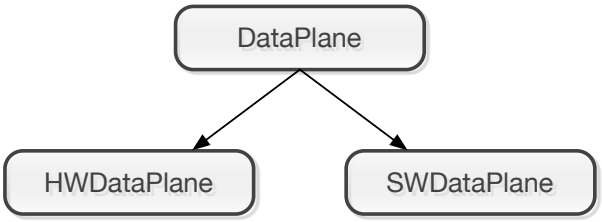

Figure 2: DataPlane inheritance diagram, with DataPlane as the base class.

by a compile-time flag that chooses between VNS [2] mode and NetFPGA mode.

♦ The key advantage to this approach is the separation of hardware-specific logic from VNS-specific logic. The HWDataPlane contains a number of reactors (see Section 3.6 – Notifications) to respond to changes to the software data structures by updating the hardware tables. In particular, `hw_data_plane.cc` defines an ARPCacheReactor within HWDataPlane. ARPCacheReactor implements the `ARPCache::Notifiee` interface and receives notifications when entries are added or removed from the ARPCache. The ARPCacheReactor code updates the hardware ARP table on all changes to the software ARP cache automatically upon notification. With the notifications mechanism, the ARPCache code does not need to know about the hardware tables. Further, this state maintenance logic is separated from the business logic in the ControlPlane, resulting in much cleaner code.

3 Using C++

Going into this class, we were determined to use C++ if all possible. Even before the first lecture, we spent several hours converting the starter code to C++ and get everything working, starting with this commit:

```
commit 7e28f1c3e92099876a3058117f04a82ac74ec107
Author: Matt Sparks <ms@quadpoint.org>
Date:   Tue Mar 29 16:09:45 2011 -0700

    First steps to a complete compile with C++
```

In retrospect, we believe beyond any doubt that this initial time investment was well worth the effort. With C++ , we were able to leverage libraries like the STL and create our own abstractions that ultimately simplified our design and development process. The following subsections discuss some of our design principles and data structures we leveraged.

3.1 State versus logic

Each component mentioned in this document is a C++ class. Our design is very much 'state-oriented' in that most of the components are purely abstractions of state and contain no 'business logic'. For instance, our ARPQueue component stores and provides access to packets queued pending ARP replies. However, the ARPQueue itself is not responsible for sending queued packets when ARP replies are received. The ControlPlane has this job.

♦ All of the ARP, routing, and validation logic is called from within the ControlPlane and DataPlane. All other components are purely stateful. The clear separation of state and logic makes the stateful components amenable to unit testing, as described in Section 8. Further, the code is much easier to reason about if the logic exists in few places.

3.2 Invasive smart pointers

Within the `src/fwk` subdirectory of our router code repository, we have a number of 'framework' sources. The framework includes code we've adapted from Prof. David Cheriton's CS249 courses as well as some of our own creations. We made very heavy use of invasive smart pointers, defined in `ptr.h` and `ptr_interface.h`. These smart pointers differ from other smart pointer implementations, such as the STL `auto_ptr` or boost's `shared_ptr` in that invasive smart pointers store reference counts in the objects themselves, rather than in an auxiliary data structure. The reference increment and decrement operations are provided by the `Fwk::PtrInterface` base class, so objects to which one would like a smart pointer must inherit from this class.

Using smart pointers greatly simplifies memory management. Once the reference count of an object decreases to zero, it is automatically deallocated. This gives us the freedom to return early on error without cluttering the code to free allocated resoures, and to return pointers to objects without having to be extremely concerned about the lifetime of the memory to which they are pointing.

However, smart pointers still suffer from circular reference problems. To avoid circular references, we were careful to keep one 'weak' reference in cases where circular references were possible. One such case where this is needed is the link between ControlPlane and DataPlane. Both objects point to each other, but one has a weak reference to the other.

♦ From experience, we believe the benefits of invasive smart pointers to far outweigh the costs. They are efficient – as the reference count is contained within the object, requiring no additional allocations – and they simplify code structure and project design in a very noticeable way.

3.3 PacketBuffers

Interface definition: `packet_buffer.h`

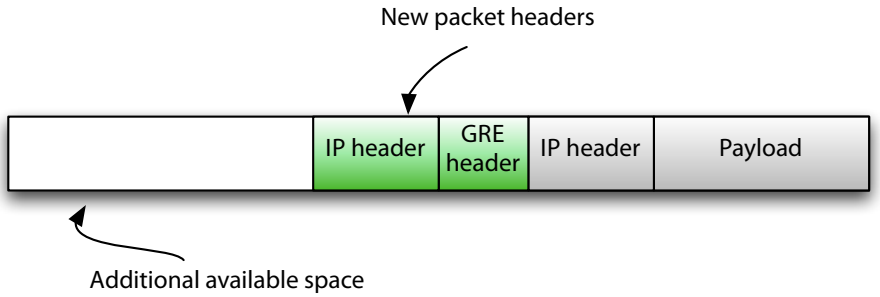

Figure 3: Adding headers to a packet in a PacketBuffer does not require a new packet allocation.

Sphene uses a special structure called a `PacketBuffer` for manipulating packet memory. PacketBuffers use `Fwk::Buffer` objects internally for reference-counted memory, but they are designed specifically for use with packets that can grow at the head. This is useful for packet encapsulation, where headers are added to the beginning of a packet. PacketBuffers allocate `Fwk::Buffer` objects in powers-of-two sizes, with a minimum size of 512 bytes. When a packet is copied into a PacketBuffer, the packet contents are copied to the end of the buffer so that new headers can be added to the beginning. Powers of two are used to be memory-allocator-friendly and also to minimize the growth frequency while also minimizing internal fragmentation.

When a packet is encapsulated, the encapsulating packet headers need to be added. In code, the PacketBuffer is declared to be of a minimum size of the current packet length plus the length of the headers:

```
pkt->buffer()->minimumSizeIs(pkt->size() + encap_header_size);
```

This ensures that the access `pkt->data() - encap_header_size` is a valid address inside the PacketBuffer. If the PacketBuffer is not large enough, its size is automatically doubled to the next largest power of two.

♦ With PacketBuffers, the total packet length need not be known ahead of time during packet creation. Further, adding new headers for encapsulation will not usually require an additional memory allocation and packet copy. We gain performance as well as code readability when using PacketBuffers.

3.4 Packet objects

Interface definition: `*_packet.h`

In one layer up from PacketBuffers are the `Packet` objects. We have classes for each of our packets: EthernetPacket, IPPacket, ICMPPacket, OSPFPacket, GREPacket, etc. All packet types have accessors for the fields within the packet, returned in host byte order. Further, for packets that have payloads such as EthernetPackets and IPPackets, a `payload()` method is defined that returns a `Packet::Ptr` smart pointer that is an instance of the derived packet type as determined by type or protocol fields within the outer packet. For example, an IPPacket inside of an EthernetPacket, `eth_pkt`, would be accessed with:

```
IPPacket::Ptr ip_pkt = Ptr::st_cast<IPPacket>(eth_pkt->payload());
```

♦ Because of the way PacketBuffers work, the returned IPPacket does not duplicate the packet memory. Instead, the IPPacket points to the same internal `Fwk::Buffer` but with a different byte offset within the buffer, so that it starts at the IP header within the packet. Thus, we gain a significant abstraction convenience without the unnecessary performance hit of duplicating memory.

3.5 Functors and double-dispatch

Implementation examples: `control_plane.cc`, `data_plane.cc`

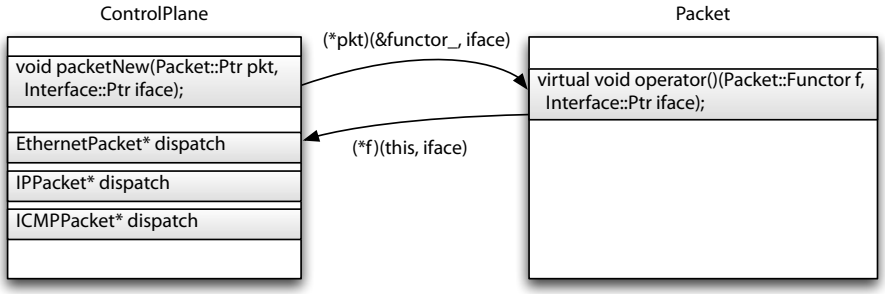

Figure 4: Double-dispatch is used to send packets to their appropriate handlers based on their dynamic types.

Our ControlPlane and DataPlane make use of the visitor pattern [6] to dispatch packet objects to their appropriate handlers based on their dynamic type (EthernetPacket, IPPacket, etc.). This pattern is illustrated in Figure 4.

♦ We use this pattern especially for stripping off headers. Once a packet's headers are inspected, the payload is extracted using the `payload()` method, and the payload is dispatched using the same mechanism. This allows for extremely clean code that is extensible as well. We need only add another method to the `Packet::Functor` class to be able to dispatch a packet of a new type. This is much simpler than a `switch` statement or block of `if-then-else`s.

3.6 Notifications

Interface definition: `fwk/notifier.h`
Implementation example: `arp_cache.h`, `arp_cache.cc`, `hw_data_plane.cc`

As described in Section 2 – Component design and implementation, notifications are a significant part of our design. Sphene defines several `::Notifiee` implementations called *reactors* that respond to notifications from *notifiers*. Reactors register for notifications from a notifier by calling `notifierIs(notifier)` on themselves. Notifiers maintain lists of all notifiees and send notifications to all of them when events occur.

♦ Notifications provide a clear separation between modules. One alternative to notifications is to add logic in the notifier to call the notifiee directly on an event. However, it is easy to see that the notification framework is far more general; it does not require the notifier to be changed when a new object is interested in notifications from it.

3.7 Logging

Interface definition: `fwk/log.h`

Early on in the project, we realized that good log output was important in the development process. Our logger is modeled after the Python `logging` module [7], in which there is a root logger, and subloggers of different names. In full debug mode, our output looks something like the following:

```
2011-06-02 07:28:54 [info] OSPFDaemon: Sending HELLO
2011-06-02 07:28:54 [debug] ControlPlane: outputRawPacketNew() in ControlPlane
2011-06-02 07:28:54 [debug] ControlPlane:   outgoing interface: eth2
2011-06-02 07:28:54 [debug] ControlPlane:   next hop: 224.0.0.5
2011-06-02 07:28:54 [debug] ControlPlane: Forwarding IP packet to 224.0.0.5
2011-06-02 07:28:54 [debug] HWDataPlane: outputPacketNew() in DataPlane
2011-06-02 07:28:54 [debug] HWDataPlane:   iface: eth2
2011-06-02 07:28:54 [debug] HWDataPlane:   src: 00:42:DE:AD:44:02
2011-06-02 07:28:54 [debug] HWDataPlane:   dst: FF:FF:FF:FF:FF:FF
2011-06-02 07:28:54 [debug] HWDataPlane:   length: 66
2011-06-02 07:28:54 [debug] HWDataPlane:   type: 2048 (IP)
2011-06-02 07:28:54 [debug] HWDataPlane:     src: 10.1.0.1
2011-06-02 07:28:54 [debug] HWDataPlane:     dst: 224.0.0.5
2011-06-02 07:28:54 [debug] HWDataPlane:     identification: 0
2011-06-02 07:28:54 [debug] HWDataPlane:     flags: 0
2011-06-02 07:28:54 [debug] HWDataPlane:     fragment offset: 0
2011-06-02 07:28:54 [debug] HWDataPlane:     ttl: 1
2011-06-02 07:28:54 [debug] HWDataPlane:     protocol: 89
2011-06-02 07:28:54 [debug] HWDataPlane:     header checksum: 53098
2011-06-02 07:28:54 [debug] HWDataPlane:     length: 52
2011-06-02 07:28:54 [debug] ControlPlane: outputRawPacketNew() in ControlPlane
2011-06-02 07:28:54 [debug] ControlPlane:   outgoing interface: eth3
2011-06-02 07:28:54 [debug] ControlPlane:   next hop: 224.0.0.5
2011-06-02 07:28:54 [debug] ControlPlane: Forwarding IP packet to 224.0.0.5
```

All of our networking-related values, such as IP addresses and MAC addresses, are special C++ types (`IPv4Addr`, `EthernetAddr`) that define string casting operators.

Because our logger provides an output stream, logging a MAC address in an EthernetPacket is as simple as:

```
DLOG << "src: " << eth_pkt->src();
```

♦ As a result of our logging infrastructure, we were able to determine problems quickly without resorting to opening packet logs in Wireshark. We found this form of logging to be essential in diagnosing problems along the way.

4 Multithreading Issues

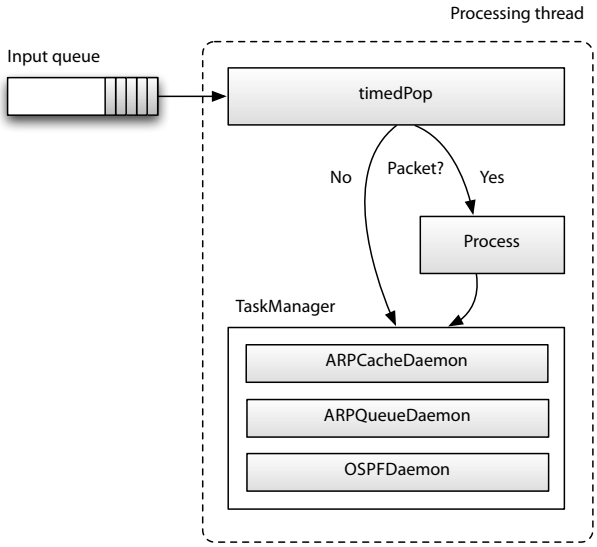

Figure 5: The processing thread processes packets from the input queue in a loop.

Our design uses two threads, one for network input and one for processing. The network input thread pulls data from VNS or the NetFPGA and calls `sr_integ_input()`, which queues data in an input queue in software. The processing thread pops packets from this input queue and processes them. The processing thread is also responsible for performing periodic tasks. The ARP Cache, ARP Queue, and PWOSPF daemons are implemented as periodic tasks executed by the processing thread.

By having a dedicated network thread, we ensure that input packets are received in a timely manner. Using an event loop design for the processing thread simplifies our locking scheme: only the input queue needs synchronization. All other code runs in a single thread. We implemented an atomic types library (`fwk/atomic.h`), a wrapper of Concurrency Kit [8] routines, and used atomic integers for reference counts so that smart pointers could be used in the input queue.

The event loop is non-blocking and performs a number of steps in order. First, the input packet queue is checked. A timed wait for a number of milliseconds is used to ensure we have a reasonable window during which we are available for immediate packet processing. If no packet is received during the window, the daemon intervals are checked next, and any daemon with work to perform is executed. Once all daemons are checked, the event loop restarts with a timed wait on the input queue.

♦ Limiting the number of threads to two was a key design decision. With only a single processing thread, all of our bugs were deterministic and reproducible, and therefore much easier to debug. Our model is far easier to reason about, and with it we freed ourselves from the headache of race conditions and deadlocks.

5 Packet flow

5.1 DataPlane

As packets arrive at the router, the path they take is as follows.

1. Packet is passed to `sr_integ_input()`.
2. `sr_integ_input()` creates `EthernetPacket` instance of the packet.
3. Packet is sent to the `DataPlane::PacketFunctor::operator()(EthernetPacket*)` method in the data plane via double-dispatch.
4. DataPlane extracts the payload of the ethernet packet and performs another double-dispatch to dispatch the packet to the appropriate handler. (ARPPacket handler or IPPacket handler).

   For ARP packets: Pass the packet the control plane, again using double-dispatch.

   For IP packets:

   i. Verify version, checksum, and TTL using `IPPacket` object methods. On errors, send packet to ControlPlane.
   ii. Lookup destination IP address in local address table. If it exists, send packet to ControlPlane.
   iii. Lookup next hop info in the RoutingTable. If not found, send packet to ControlPlane.
   iv. Lookup MAC address for next hop IP address in ARPCache. If not found, send packet to ControlPlane.
   v. Change destination MAC address to next hop MAC address in ethernet header.

vi. Decrement TTL in IP packet.

vii. Recompute checksum in IP packet.

viii. Dispatch output packet with `DataPlane::outputPacketNew()`.

5.2 ControlPlane

The ControlPlane receives packets from the DataPlane. This might happen for any of a number of reasons: a badly formatted packet, forwarding table or ARP cache misses, ARP packets, PWOSPF packets, or local delivery. The ControlPlane receives packets in a similar way as the DataPlane, via double-dispatch.

For ARP reply packets, the control plane must update its local ARP cache. The ControlPlane must also respond to ARP request packets if necessary.

For IP packets, the ControlPlane is responsible for responding with ICMP messages on UDP packets, as it does not implement this protocol. PWOSPF packets are sent to the OSPFRouter to be processed further. Valid IP packets that are not TCP, UDP, or PWOSPF should be forwarded in the usual way, by querying the RoutingTable and ARPCache. If an ARP entry is not available, an ARP request is sent and the packet is inserted into the ARPQueue pending an ARP reply.

6 OSPF

Note: Throughout this section, the term *active endpoint/neighbor* refers to nodes in the topology that are running OSPF; the term *passive endpoint/neighbor* refers to non-OSPF subnets or end-hosts.

The control plane forwards all OSPF packets to Sphene's OSPFRouter instance. The OSPFRouter class makes use of the visitor pattern and double dispatch (see 3.5) to determine each packet's type (HELLO or LSU) and to process it accordingly.

The data model of OSPFRouter consists of two main data structures: The OSPF Interface Map, which keeps track of the router's interfaces and their directly connected neighbors; And the Topology Database, which maintains Sphene's view of the network topology along with subnets connected to each node within the OSPF area.

6.1 OSPF Interface Map

6.1.1 Data Model

The OSPFInterfaceMap object maintains a collection of the interfaces that are owned by Sphene. Each interface contained within OSPFInterfaceMap is represented by an

object of type OSPFInterface. A specific OSPFInterface object can be obtained by passing OPSFInterfaceMap either, the IP address of the interface itself, or the OSPF Router ID of any active neighbor that is directly connected to that interface.

Each OSPFInterface object keeps track of:

1. The list of directly connected active gateways and their corresponding OSPF router IDs;
2. The list of directly connected passive gateways and their corresponding subnets;
3. The HELLO interval for the interface;
4. The time at which the last HELLO packet was broadcasted out the interface;
5. And, a pointer to the main router's Interface object that corresponds to this OSPFInterface.

6.1.2 Integration with OSPFRouter

OSPFInterfaceMap makes use of notifications to keep itself in sync with Sphene's InterfaceMap. It also forwards notifications from InterfaceMap to OSPFRouter. OSPFInterfaceMap also notifies OSPFRouter on addition or deletion of directly connected neighbors. OSPFRouter reacts by updating the routing table accordingly.

♦ This approach works well because any changes to Sphene's interfaces are immediately reflected in OSPF's data structures. Further, because tunnels are represented as virtual interfaces, this model simplifies the integration between OSPFRouter and the ControlPlane.

For more details about notifications, see section 3.6.
For more details about OSPFRouter integration with ControlPlane, see section 6.5

6.2 OSPFDaemon

As mentioned in section 2.5, the OSPFDaemon is responsible for all logic pertaining to OSPF that occurs periodically. OSPFDaemon inherits from PeriodicTask and is registered with Sphene's task manager at startup. At a high level, OSPFDaemon performs the following five operations:

1. Timeout direct links to neighbors from whom no HELLO packet has been received in more than three times their advertised HELLOINT.
2. Timeout indirectly connected nodes from whom no link-state update has been received in more than the link-state update timeout. All links associated with said nodes are also removed from the topology.

3. Prompt OSPFTopology to recompute the optimal spanning tree if needed. This occurs every second. OSPFTopology's dirty-bit scheme ensures that the spanning tree is not recomputed more often than it's required. See section 6.3.3 for more details.

4. Broadcast HELLO packets out every interface (including GRE tunnels, see 7). The interval of this operation is defined by each interface's HELLOINT.

5. Send link-state updates, point-to-point, to all directly connected neighbors. The interval for this operation is defined by the link-state interval.

6.3 OSPF Topology

6.3.1 Data Model

The OSPFTopology object consists of a map of all nodes in the topology, where each node is an active OSPF router; the map is keyed on the node's OSPF router ID. OSPFTopology also encapsulates a pointer to the root node (i.e. this instance of Sphene). The root node is of type OSPFNode.

Each OSPFNode object, including the root node, encapsulates:

1. An OSPF Router ID;

2. The list of links by which it is connected to other nodes in the topology. Each link is of type OSPFLink and contains a pointer to the node at the endpoint of the link and the subnet of the connection to that node;

3. The list of links by which it is connected to passive subnets in the topology (non-OSPF endpoints);

4. The latest link-state sequence number seen from this node;

5. The node's distance (in hops) from the root node.

6.3.2 Optimal Spanning Tree

In addition to keeping track of the network's topology, OSPFTopology also implements Dijkstra's algorithm to compute the shortest path between the root node (i.e. this instance of Sphene) and every OSPF node in the network.

6.3.3 Dirty-bit Scheme

♦ OSPFTopology makes use of notifications (see 3.6) to efficiently detect changes to the topology. OSPFTopology implements OSPFNode's Notifiee interface and registers for notifications with every node in the topology. Whenever any node is updated, OSPFTopology receives a notification and sets its dirty bit to true.

As mentioned above in section 6.2, OSPFDaemon prompts OSPFTopology to recompute the optimal spanning tree, if necessary, every second. OSPFTopology makes use of its dirty bit to determine whether or not the topology has changed since the last time Dijkstra's algorithm was performed.

6.3.4 Integration with OSPFRouter

OSPFTopology defines a Notifiee with a single notification, `onDirtyCleared`. This notification is signaled whenever OSPFTopology's dirty-bit transitions from `true` to `false`. OSPFRouter listens for notifications from OSPFTopology and updates the routing table whenever `onDirtyCleared` is signaled.

6.4 Incoming Packet Flow

OSPF Packets are handed to `OSPFRouter::packetNew()` by Sphene's control plane. `packteNew()`, then handles the packet by making use of OSPFRouter's PacketFunctor, an implementation of the Functor interface declared by the base Packet class (see 3.5). The functor dispatches the packet to the appropriate packet handler (HELLO or LSU), depending on the packet's type.

6.4.1 HELLO Packet Processing

1. Validation; all invalid packets are ignored.
2. Lookup of active gateway that corresponds to the HELLO packet's sender. If no active gateway for the packet's sender exists, one is created.
3. The active gateway's `time_since_hello` attribute is set to 0.

6.4.2 LSU Packet Processing

1. Validation; all invalid packets are ignored.
2. Packets that were originally sent by `this` instance of Sphene are ignored.
3. A packet with a sequence number S that is less than the node's `latest_seqno` is ignored only if S is also $\geq$ `latest_seqno` $-$ `kLinkStateWindow`, where `kLinkStateWindow` has a default value of 10.

   ♦ Packets with sequence numbers that are less the node's `latest_seqno` minus `kLinkStateWindow` are accepted. This takes care of the case in which a node in the topology reboots and resets it's link-state sequence number to zero or one.

   The advantage of this approach is that packets sent by a rebooted router are immediately accepted. Otherwise, it would take up a link-state update timeout for the rebooted packets to be accepted.

The disadvantage of this approach is that, in the incredibly unlikely event that a link-state update from a node is delayed by more than ten times the LSU interval, it may be accepted by Sphene, even though it may no longer represent the latest link-state of the sender.

4. Lookup the node in the OSPFTopology object with the sender's router ID; if no such node exists, create it.
5. Set the node's latest sequence number to that of the incoming packet.
6. Process the packet's link-state advertisements.
   i. LSAs that advertise this instance of Sphene as a direct neighbor are ignored; the HELLO protocol takes precedence.
   ii. LSAs that advertise an endpoint of a GRE tunnel from this instance of Sphene to the sender of the LSU are ignored. This approach prevents infinite recursion on encapsulation. See Section 7.3.2 – OSPF Integration for more details.
   iii. If the LSA advertises a passive endpoint, a new OSPFLink object initialized with the subnet and subnet mask of the advertisement is created and connected to the LSU's sender in the topology.
   iv. Else, if the LSA advertises an active endpoint: Check if the advertised neighbor has also advertised connectivity to the current LSU's sender. If it has, then commit the link between the two nodes to the topology. Otherwise, stage the link until the advertised neighbor confirms connectivity to the LSU's sender.
7. Remove any links connected to the LSU's sender if they were not confirmed by any LSA.

   ♦ This approach reduces OSPF convergence times. Consider a new LSU, L_N, that does not include an advertisement for a neighbor N that had been advertised in an older LSU, L_O. From the absence of said advertisement in L_N, it is possible to infer that the LSU's sender is no longer connected to the neighbor in question. Consequently, it is not necessary to wait for a link-state timeout to remove the link between the LSU's sender and N in the topology; it can be removed immediately.
8. If the LSU packet's TTL is greater than one, decrement the TTL, recompute the packet's checksum, and flood out, point-to-point, to all direct neighbors.

6.5 OSPFRouter: Integration with the Control Plane

♦ OSPFRouter receives notifications from both, the routing table, and Sphene's interface map. This allows OSPFRouter to keep the topology up to date as static routes or interfaces are added or removed from Sphene. There are a number of challenges associated with notifications once a certain level of complexity is reached. For an account of said challenges, see section 9.1.

7 Generic Routing Encapsulation (GRE)

GRE tunneling is our advanced feature. A GRE tunnel is essentially a direct, virtual Ethernet cable between two GRE-supporting devices.

7.1 GRE History

GRE was originally proposed in October 1994 in RFC 1701 by Cisco. The motivation behind GRE is simple. It defines a structure for encapsulation of an arbitrary network protocol. With GRE, tunnels can be created for passing arbitrary network protocol data. Its header has a 16-bit protocol type field that uses the same values as in Ethernet's type field. Encapsulation merely involves adding a small GRE header to an unmodified network layer packet. The GRE packet is then put inside another network layer packet and transmitted as normal. Also in October 1994, RFC 1702 was proposed by the same authors. RFC 1702 discusses some specifics of using IPv4 as the encapsulated protocol (i.e., the payload of the GRE packet), as well as the encapsulating protocol. GRE packets have IP protocol 47 when they are encapsulated by IP. When IP is being encapsulated by GRE, the payload's IP packet must have its TTL decremented upon decapsulation to ensure a finite packet lifetime.

The current proposed standard for GRE is defined in RFC 2784 from March 2000. RFC 2784 has a number of simplifications on RFC 1701, namely the deprecation of many of the flag bits in the header and the source routing functionality defined in RFC 1701 and RFC 1702. The header space for these fields is now reserved and must be zeroed unless the endpoint implements RFC 1701. Backwards compatibility issues are also addressed in RFC 2784.

GRE is currently supported in Linux and many Cisco products. Sphene implements the RFC 2784 standard of GRE and is interoperable with other GRE implementations.

7.2 Packet flow example

This section details how IP packet encapsulation and decapsulation by GRE, and the forwarding and delivery of GRE packets. Suppose router A (172.16.17.18) and router B (172.19.20.21) are GRE-supporting devices on the Internet. Routers A and B are both public routers, but they have private networks behind them. Router A is the gateway for 10.0.1.0/24 at 10.0.1.1, and router B is the gateway for 10.0.2.0/24 at 10.0.2.1. Suppose a host X at 10.0.1.15 sends an ICMP ping packet to a host Y at 10.0.2.49 on router B's private network. For this example, we assume hosts X and Y have the appropriate static routes for 10.0.2.0/24 and 10.0.1.0/24, respectively, or X and Y have default routes through their respective routers. The packet flow is as follows.

1. An ICMP echo request is created at host X, destined for host Y at 10.0.2.49.

2. Host X looks up 10.0.2.49 in its routing table and determines the next hop is 10.0.1.1.
3. Router A receives the ICMP echo request destined for 10.0.2.49. Router A determines the outgoing interface is the virtual interface 'netb'. There is no 'next hop' as would be needed when forwarding on an Ethernet network. The 'next hop' field in Router A's routing table SHOULD contain 0.0.0.0 for this entry.
4. Router A looks up the tunnel information associated with the interface and determines the tunnel remote endpoint to be Router B's public IP address, 172.19.20.21.
5. Router A encapsulates the IP packet containing the ICMP echo request in a GRE packet with protocol type IP (0x0800).
6. The GRE packet is encapsulated in an IP packet with protocol GRE (47) destined for Router B at 172.19.20.21.
7. Router A sends the IP packet containing the GRE payload using its normal IP forwarding pipeline.
8. The GRE-in-IP packet traverses the Internet and arrives at Router B.
9. Router B confirms that the GRE-in-IP packet is from a known tunnel.
10. Router B validates the GRE checksum if it is present, and discards the packet if it is invalid.
11. Router B strips the GRE header and inserts the encapsulated IP packet in its normal incoming IP packet processing pipeline.
12. Router B forwards the ICMP echo request to Y.

7.3 Implementation

GRE encapsulation adds a layer of abstraction that requires a few notable changes at the endpoints. First, the IP packet processing at the endpoints may no longer assume that IP packets are encapsulated by Ethernet, and must support IP encapsulated by GRE encapsulated by Ethernet. Second, the addition of virtual interfaces at the endpoints requires some extra logic in the packet output stage. Specifically, interfaces may no longer be assumed to be Ethernet; GRE tunnel interfaces must also be supported. An IP packet that is to be sent out a GRE tunnel virtual interface must be encapsulated by GRE and then (re-)inserted into the output pipeline. When generating new packets, or encapsulating existing packets, care must be taken to allocate buffers large enough for GRE headers. To accomplish this, we use PacketBuffers, as described in Section 3.3.

When a GRE tunnel is created in Sphene, a 'virtual' Interface is created in the InterfaceMap, with the same name as the name of the tunnel. This Interface has an IP address: the IP address of the machine on the tunnel subnet. Indeed, this interface is

essentially the same as other interfaces. Routes can be added to go out this interface, as one would expect. However, these routes, too, are virtual. When a packet is about to be routed out a virtual route, the ControlPlane triggers encapsulation of the packet and sends the new, larger, GRE packet through the output pipeline.

7.3.1 IP fragmentation

One complication arises with tunneling protocols: encapsulation of maximum-length packets. The standard MTU of Ethernet (i.e., with jumbo frames disabled) is 1500 bytes. A minimum GRE encapsulation requires an IP header (20 bytes without options), and a GRE header (4 bytes without checksum); a size of 24 bytes minimum. Thus, for original packets larger than $1500 - 24 = 1476$ bytes, the encapsulated packet cannot be sent over Ethernet without fragmentation.

Our software implements the IP fragmentation protocol per RFC 791 [5]. That is, Sphene will automatically fragment encapsulated packets if they are larger than 1500 bytes. We were hoping to depend on Path MTU Discovery (PMTUD) [4], but were unsuccessful with large packets from VNS, so we resorted to a simple implementation of IP fragmentation instead.

7.3.2 OSPF Integration

♦ Because Sphene's implementation of GRE represents tunnels as virtual interfaces, the integration between GRE and OSPF was mostly seamless. Whenever a tunnel is created, OSPFRouter receives a notification from Sphene's InterfaceMap signaling the addition of an interface. OSPFRouter reacts to the addition or deletion of interfaces, regardless of whether or not they are virtual, by updating it's internal OSPFTopology. OSPFRouter thereafter treats the tunnel's endpoint as another directly connected neighbor, and all OSPF packet's sent out to that tunnel endpoint are automatically encapsulated on the way out, by the control-plane.

♦ There is one challenge about integrating OSPF with GRE. Recall that each tunnel is treated as just another interface by OSPF, and that, through the tunnel, the tunnel endpoint is a single hop a way (albeit virtual). As a result, the OSPFTopology will have two apparent paths to each tunnel endpoint, one through the tunnel, and one through the real, physical network. This presents a problem because OSPFTopology's implementation of Dijkstra will always treat the tunnel as the shorter path and will discard the real path. Consequently, trying to send a packet out a virtual interface results in infinite recursion because the control plane will attempt to send encapsulated packets to the tunnel endpoint that are meant to be sent through the physical network, through the tunnel itself.

Sphene's solution to this problem is to ignore all LSAs from directly connected neighbors that advertise the subnet that contains the endpoint of any tunnel in Sphene's tun-

nel map. This resolves the problem of dual paths to tunnel endpoints in OSPFTopology in favor of the real, physical path.

7.3.3 Hardware integration

GRE is particulary amenable to hardware integration in that it can be implemented incrementally in hardware. GRE works without a problem with no hardware involvement as all GRE packets are addressed directly to the router, so they are forwarded to software anyway. The first step of hardware acceleration is to handle decapsulation of incoming GRE packets in hardware. Full integration handles both decapsulation and encapsulation. Please refer to our hardware design document for details on our GRE hardware acceleration.

7.4 Tunnel configuration

The tunnels in Sphene are fully configurable through the existing CLI. We added an `ip tunnel` set of commands:

```
sphene% help ip tunnel
ip tunnel {add | del | change} <name> [mode { gre }] [remote ADDR]: modify tunnels
ip tunnel add <name> mode { gre } remote <addr>: add a tunnel to <addr>
ip tunnel del <name>: delete a tunnel
ip tunnel change <name> mode { gre } remote <addr>: change attributes of an existing tunnel
```

Adding a GRE endpoint can be done in as little as two commands:

```
ip tunnel add feynman mode gre remote 10.2.0.10
ip intf feynman 10.4.0.1 255.255.255.0
```

The above example creates a virtual interface called 'feynman' on a GRE tunnel. The GRE tunnel endpoint is 10.2.0.10. The tunnel subnet is 10.4.0.0/24, and the local IP on the tunnel is 10.4.0.1.

8 Testing strategy

As described in Section 3.1 – State versus logic, we took great care in separating state and logic for the purpose of testability. To ensure correctness, we employed a variety of tests to stress our software. In this section, we describe our testing methods, both automated and manual.

8.1 Unit tests

We strived to automate our testing process as much as possible. Doing so ensures that problems are detected early and that correctness is maintained throughout the development process. As a first line of defense against bugs, we implemented a number of unit tests for our stateful components. Our tests are written to ensure that the interface

semantics of each component are upheld, and that edge cases are handled correctly.

For unit testing, we use the googletest [1] framework. It provides great flexibility and simplicity for writing unit tests. We have integrated our unit tests with our build system, which uses GNU Autotools [9], such that a simple `make check` runs all unit tests. Further, we configured buildbot [10] to automatically pull our git changes, recompile and test, and notify us via IRC on any failures. At the time of this writing, the buildbot waterfall display is available at: http://buildbot.quadpoint.org/sphene/waterfall.

8.2 Live tests with VNS

Unit tests are useful, but they are not able to capture more complex interactions between instances of sphene and other PWOSPF-capable routers. To test interoperability, convergence, and connectivity, we developed a system to do live automated tests using VNS. Using the Python nosetests [3] library, we developed tests for each of the topologies.

The first topology we tested was topology 594. Our test for this is in `tests/topo594_test.py`. This is the 1-router, 2-appserver configuration. Our test programmatically first does pings and traceroutes to all of the interfaces. HTTP requests are tested by doing a basic request for the main page of an app server, and the big.jpg file is downloaded and compared with a known size. Bootstrapping the sphene instance is done automatically at test runtime.

The second configuration we tested is topology 602, the 3-router setup. For this test, we created a base class, `tests/topo602_base.py`, and 8 configurations of `topo602_xxx_test.py` where `xxx` $\in \{rrr, rrs, rsr, rss, srr, srs, ssr, sss\}$. Here, r is 'reference' and s is 'Sphene'. The base class defines the real test, which is to disable the one of the links from the border router, and confirm that all connectivity is maintained and that the traceroute shows the path through the intermediate router. The various configurations test all arrangements of Sphene with the reference router to ensure that the two implementations are interoperable.

Finally, we created a test for topology 677, the 6-router mesh. The test breaks links successively, forcing packets to go through more and more routers as additional links are broken, and confirms that traceroutes are correct and pings still succeed. Additionally, we ran this test in three different configurations: one with six Sphene instances, and two with half Sphene and half reference router, and alternating their positions in the mesh.

♦ Combined, these tests exercised a lot of functionality. In fact, it took us a considerable amount of debugging to pass our own tests. We also used these tests against other teams' binaries. Only our team, team 6, and the reference router were able to pass all of

the tests. Other teams likely failed to do semantic disagreements in edge cases such as the ability to ping a disabled interface on the border router. We found these automated tests to be an excellent way to do regression and correctness testing as we perfected our PWOSPF implementation.

8.3 Manual testing

Finally, manual testing was key to ironing out our implementation. Some problems, such as the lack of MAC filtering, were quickly discovered in manual testing with other teams, even though they were not present in our automated tests. We had several interop sessions, particularly with teams 5 and 6, to confirm interoperability been our routers.

9 Post-mortem

Sphene became a massive project for us, totalling over 14,000 lines of C++ of code in 1,300+ git commits. We used this opportunity to experiment with many software engineering techniques as noted in Section 3 – Using C++ . Further, we used many new tools and systems: GNU autotools, googletest, nosetests, and buildbot. Of course, we also used familiar tools like git and gdb throughout the quarter. Some of the lessons we learned follow.

9.1 Notifications are awesome, but they can be tricky

The notifications system is a really useful pattern in cases like the HWDataPlane updating the hardware tables. It *feels* clean and it definitely seems like everything *just works* when notifications are used. They are very easy to think about and easy to use. But in some cases, they are tricky.

In complex notification setups, a potential for mutually recursive reactors exist. Such a situation arises when reactor logic changes the state of another module and causes a chain reaction of notifications. If there is a cycle in the graph of notifications, an infinite loop of reactor execution is possible. This tends to be an easy bug to diagnose, but once this bug is fixed by making operations idempotent to stop notification propagation, other subtle ordering bugs may surface still. The order in which notifications are executed is not well-defined, so one must be careful about making assumptions about state that is changed by notifications, while in a reactor.

9.2 Testing is invaluable

We tried to write high-coverage unit tests for our stateful interfaces. Having these provided two immediate benefits:

(1) A sense of confidence that the software is behaving correctly, at least that the state modules do what they are supposed to do.

(2) Instant notification that a change broke something in the codebase.

Our unit tests revealed bugs several times during the project. Without these unit tests, we would have gone through considerable pain doing unnecessary debugging. Additionally, our live VNS tests helped immensely with running thorough tests quickly and easily, without suffering the manual effort of launching six instances of our router.

9.3 Multithreading is hard

It is common wisdom that writing concurrent applications is not easy. Early on in the project, we were considering a multithreaded approach where the daemon components would run concurrently in separate threads. However, this idea quickly become extremely difficult to reason about, especially when adding in notifications to the mix. Notifications execute in the notifier's thread, so not only would we have to understand what modules are running at the same time and accessing what data, we would have to understand what notifiees are called when and in what thread, and what they access. This is a nightmare.

We concluded that having a daemon that runs once a second in another thread is unlikely to have any measurable performance gain on a multicore machine, so we considered other alternatives. We wanted a way to run all processing in serial so we would know that only one thread[1] is accessing the data structures at a given time. The transition to this model only required a blocking queue with a timed wait, and a second thread to process packets from that queue. Combined with the TaskManager, we had all the functionality we needed to simulate multiple periodic daemons.

Given more time, we would have liked to revisit the necessity of the input thread. At the moment, we have a dedicated thread to pull packets from the NetFPGA and insert them into a queue in software. We are not sure if this input thread is necessary. However, our current model is simple enough, with the packet queue as the only data structure that needs to be truly thread safe.

A Interesting commits

This section is reserved for git commits that the staff might find interesting for future versions of CS344. These commits include fixes for a number of bugs that caused us and other teams grief during development.

Each commit can be examined from one of our public repositories[2] using the command

[1] except perhaps a CLI client thread

[2] `git clone git://quadpoint.org/stanford/sphene.git`

`git show 3bacafe`, substituting an appropriate commit hash prefix.

(1) `adb2fe4ed5f9ab1caaec564dd08208b8d738b726`
Fix `memset()` argument reordering.

(2) `4316d5bb4dda98a883c21586c75fb15a24ea72ea`
Forward declaration was actually a global variable declaration.

(3) `a17fbe192a9dce8bf0fc269a43092fbeecd63f66` (very last diff chunk)
`strncpy()` uses `strlen()` as max length, chopping the null terminator.

The Python modules and shell scripts in our `tests/` directory may be of interest to the staff for automated testing in future CS344s. Please feel free to use them as you wish.

References

[1] *googletest*: Google C++ Testing Framework. http://code.google.com/p/googletest/

[2] *VNS*: Virtual Network System. http://yuba.stanford.edu/vns/

[3] *nosetests*. http://somethingaboutorange.com/mrl/projects/nose/1.0.0/

[4] *Path MTU Discovery*. RFC 1191. November 1990. http://tools.ietf.org/html/rfc1191

[5] *Internet Protocol*. RFC 791. September 1981. http://www.ietf.org/rfc/rfc791.txt

[6] *Visitor pattern*. Wikipedia. http://en.wikipedia.org/wiki/Visitor_pattern

[7] *logging*: Logging facility for Python. http://docs.python.org/library/logging.html

[8] *Concurrency Kit*. http://concurrencykit.org/

[9] *GNU Build System*. Wikipedia. http://en.wikipedia.org/wiki/GNU_build_system

[10] *Buildbot*. http://trac.buildbot.net/