Chapter 2

Architecture & pac.c Prototypes

In the previous chapter, we introduced the concept of a unified control architecture for the converged operation of packet and circuit switched networks [11]. Our control architecture (Fig. 2.1) is based on two abstractions: the common-flow abstraction and the common-map abstraction.

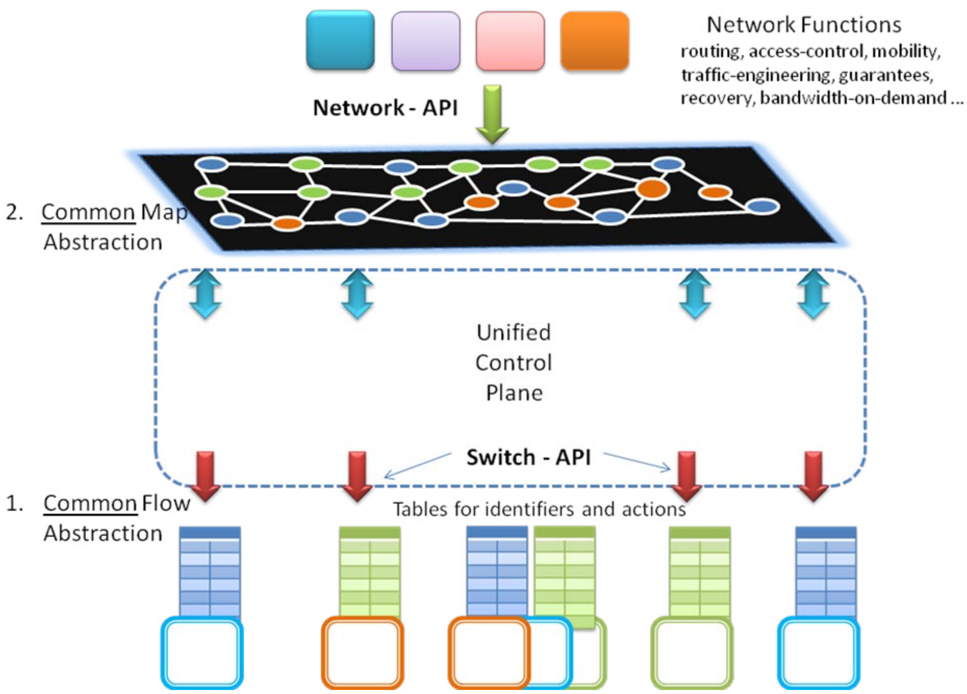


Figure 2.1: Unified Control Architecture

The common-flow abstraction is based on a data-abstraction of switch *flow tables* manipulated by a *switch-API*. The flow tables take the form of lookup-tables in packet switches and cross-connect tables in circuit switches; and together with the common switch-API, abstract away layer and vendor specific hardware and interfaces.

The common-map abstraction is based on a data-abstraction of a network-wide *common-map* manipulated by a *network-API*. The common-map has full visibility into both packet and circuit switched networks, and allows creation of network-applications that work across packets and circuits. Implementing network-functions in such applications is simple and extensible, because the common-map abstraction hides the details of state distribution from the applications.

In this chapter we give details of the design of the two abstractions. We discuss the flow abstraction as applied to packet-switches [12] and then give design-details on what is needed to apply such an abstraction to circuit-switches [13]. Further we show how a common-switch API can be designed to manipulate this data-abstraction. For the common-map abstraction, we give design details on how we represent such a map, how it can be built and maintained, and briefly what a network API that manipulates such a map could look like.

Next we present implementations of our architectural approach. We developed prototypes to help us validate our architectural constructs and improve on their implementation. We give details of three prototypes systems we built (and named **pac.c** for **p**acket **a**nd **c**ircuit **.c**onvergence). Two early prototypes helped us design and validate the circuit-switch abstraction and common switch-API. Accordingly they use different kinds of circuit switches – a wavelength switch and a time-slot based switch – together with packet switches. The third, more complete prototype helped us understand the intricacies of building a converged packet-circuit network with a common-map and network-API. We discuss the ideas we demonstrated and the lessons we learned with each prototype.

2.1 The Common-Flow Abstraction

The objective is to develop a generic data-plane abstraction based on *flows*, for all kinds of packet and circuit switches. We first discuss packet-switches and then extend those ideas to circuit switches to develop a common-flow abstraction and switch-API.

2.1.1 Packet-switching and the Flow Abstraction

Our discussion of the packet-flows is based on a generic packet-switch abstraction that is implicit in the design of the OpenFlow protocol [12, 14]. Here we wish to give an overview of the abstraction and the reasoning behind it.

We start by discussing the various kinds of packet-switches used today and their common characteristics. We then discuss a generalization of these switches into a single representation; and detail the relationship of that representation with the concept of a 'flow', both within a switch as well as across multiple switches in the network. Finally we outline the main functions of an interface (a switch-API) used to manipulate such a representation.

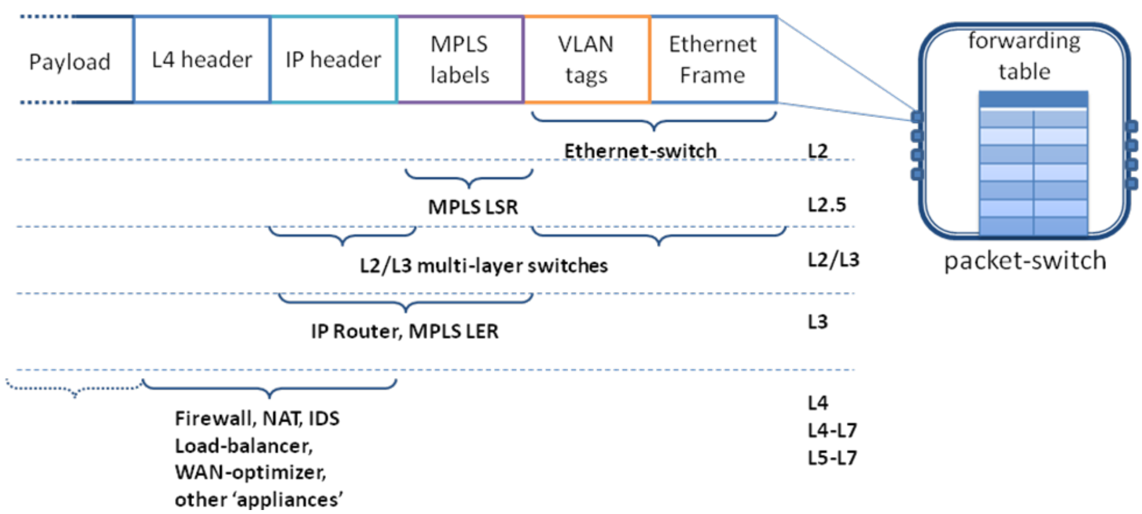


Figure 2.2: Different kinds of packet-switches

Packet-Switching: Switching in packet-networks is often defined in a layer-specific way. The network layer (or Layer 3 or L3) originally represented the switching (or routing*) layer. But switching exists in the lower layers (L2), higher layers (L4-L7), and new intermediate layers (L2.5) have been coined. And so it is worth noting that today a) the layer terminology simply refers to the different parts of the packet-header; and b) packet switches in different layers make forwarding decisions based on the layer they are part of (Fig. 2.2). For example:

- An IP router (L3) may typically forwards a packet based on the IP destination address in the IP header of the packet.
- A traditional Ethernet bridge (L2) forwards packets based on MAC addresses and VLAN ids. If the bridge includes 'routing' functionality i.e if it looks at IP packets to route between VLANs, it becomes a L2/L3 multi-layer switch.
- Some routers forward packets based on 'tags' such as MPLS labels. Since the label is inserted in a packet typically between a MAC header (L2) and an IP header (L3), it is referred to as L2.5 switching.
- And there are other, more special-purpose switches (or appliances or middleware) that forward packets (forward, drop, modify-and-forward etc.) based on yet other parts of the packet header.

And so, it is clear that irrespective of the type of packet switch, all of them perform the same basic functionality of identifying the part of the packet-header they are interested in; matching that part to related-identifiers in a lookup-table (or forwarding table); and obtaining the decision on what-to-do with that packet from the entry it matches in the lookup-table. Some other aspects of packet-switching in today's networks:

- In most cases packets are switched independently *within a switch,* without regard to what packets came earlier or which ones might come afterwards. Each packet is dealt with in isolation while ignoring the logical-association between packets that are part of the same communication^.

* Routing is just another name for switching in L3, just like bridging is another name for switching in L2.
^ Examples of such communications and logical-associations were discussed in Chapter 1 (Table 1.1).

- As packets travel from switch to switch, each switch makes its own independent forwarding-decision. Packets correctly reach their destination because the switches base their forwarding decision on some form of coordination mechanism between the switches. But such coordination mechanisms typically tend to be a) restricted to the layer/network in which the switch operates; and b) only give information for the part of the packet-header related to that network. For example,
    - STP co-ordinates Ethernet networks by preventing loops in the network topology, so that switches can learn about destinations *only* from packets coming in from un-blocked ports, and also forward packets to only those un-blocked ports.
    - IP networks use completely different coordination-mechanisms (routing protocols OSPF, IS-IS, I-BGP) that only refer to IP-destination prefixes.
    - MPLS networks use LDP or RSVP as co-ordination mechanisms similar to signaling. Here too the co-ordination is layer specific – bindings of MPLS labels to IP addresses.
- Finally, because packets are switched in isolation within and across switches, and the logical association between packets is typically not processed; it becomes very hard to perform accounting and resource management in a packet network. For example, if it is difficult to get a common handle for a stream of packets between two servers travelling across an Ethernet network; it is very difficult to tell which how much bandwidth the stream is consuming (accounting); or make resource decisions for just that stream (a specific path, bandwidth-reservation, etc.).

Packet-Flow Abstraction: From the previous discussion, it is clear that there are advantages to defining a *generic* (layer independent) data-plane abstraction for packet-switches based on 'flows':

- A packet-flow is a logical association (or classification) between packets that are part of the same communication and are given the same treatment in the network (within a switch and from switch-to-switch);

- The data-abstraction is the representation of a packet-switch as flow-tables, ports and queues. The flow (or the logical-association) is defined in the flow-tables which have the ability to indentify the flow generically (in a layer independent way) from multiple parts of the packet header. For example:
    - If the logical association is simply a destination MAC or IP address, then the generic flow table should be able to behave like layer-specific switch-tables (eg. L2-tables in Ethernet switches or L3 tables in IP routers);
    - But if the logical association requires a mix of packet-header fields for identification, the table should be able to process this as well (for example the flow identification could require a mix of IP and TCP fields).
- Once the logical association has been identified, then all packets that have the same association are treated the same way *within* the switch; where the flow-table applies the same set of actions on all packets that are identified as part of the same flow.
- Furthermore, each switch that 'sees' the flow, does not make an independent, isolated decision on the treatment to the same flow. The decision on the treatment given to all packets in the flow is communicated in a layer-independent way to all switches through which packets in the flow traverse;
- The flow-definition serves as a common-handle with which accounting and resource-management in the network can be performed on a flow-level (not packet-level).
- Finally, the data-abstractions in all switches are manipulated by a layer-independent switch-API, which we discuss next.

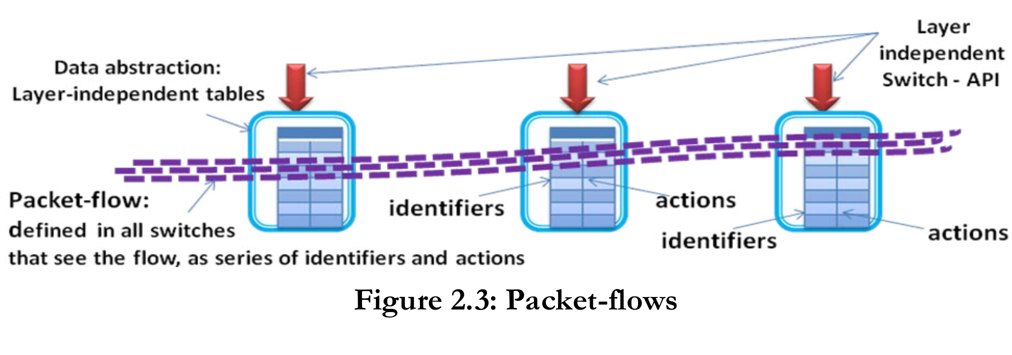


Figure 2.3: Packet-flows

Switch API: With the aforementioned definition of packet-flows in switches based on data abstractions of <flow-tables, ports, queues>; any entity that makes decisions on flows needs a layer-independent switch-API to manipulate the data-abstractions. For example, such an entity decides on what constitutes a flow (the logical association); determines how to identify the flow (packet-header fields); and how to treat all packets that match the flow-definition in all switches that the packets flow through. In order to enable the entity to make these decisions, it needs to a) understand the capabilities of the data-abstractions (the flow-tables, ports, queues) and have control over its configurable parameters; b) have full control over the forwarding path; and c) have the ability to monitor or be notified of changes in switch state. Thus the layer independent switch-API includes the following set of functions:

- Get/Set Capabilities and Configuration:
  - Methods to get the representation of the switch as data-abstractions: ports, queues, tables and their features.
  - Methods to get the capabilities of the flow-tables: for example, the flows-identifiers and actions the table can process.
  - Methods to set configurable parameters for the data-abstractions: a) locally on port, tables or queues; and b) globally on a switch level.
  - Methods to query default or current values of these configurable parameters.
- Control forwarding state:
  - Method for adding flow definitions by specifying the flow-identifiers and related set of actions. Method for deleting the flow definition.
  - Method for changing the actions applied to a flow-definition.
  - Methods to set advanced forwarding state eg. logical ports.
- Monitor: Statistics and Status
  - Methods for querying flow-table state and flow statistics.
  - Methods for querying tables, port, queue and switch statistics.
  - Methods for setting traps for change in switch-state.

The methods described above may or may not elicit a response from the switch. In some cases, there are explicit-replies; for example - to request methods (get). In other cases there may be explicit or silent acknowledgements, or replies to indicate errors.

2.1.2 Circuit-switching and the Flow Abstraction

Circuit-Switching: Like packet-switches, circuit-switches can also be defined in layer-specific ways (Fig. 2.4). For example, time-slot switching, using SONET/SDH or OTN standards, are often described as Layer 1 (or L1) or physical-layer switching. Additionally, wavelength or fiber switches are described as L0 switches, not because a new layer has been included in the OSI model, but because it is a convenient way to describe switching at an even coarser physical granularity than time-slots.

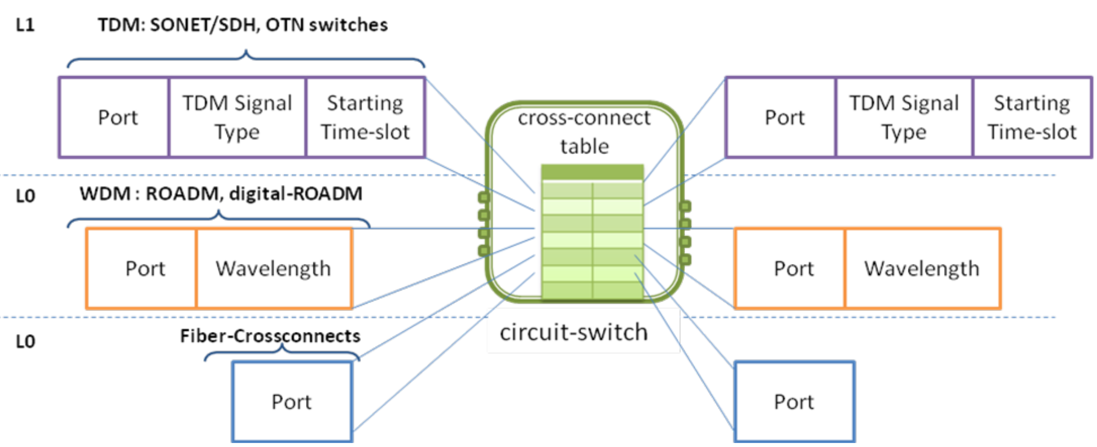


Figure 2.4: Different kinds of circuit switches

Nevertheless, all circuit switches maintain forwarding tables in the form of cross-connect tables with entries that are suited to the switching-type of the switch. In the following discussion, we consider two kinds of circuit switches - TDM and WDM; describe the creation of a circuit in each layer; and then relate a circuit to a ‘flow’. We then develop the circuit-flow abstraction; show how to map packet-flows to circuit-flows and describe a common switch-API can be used to manipulate both data-abstractions.

TDM switching: A time-slot based switch has a time-synchronous switching fabric. It cross-connects time-slots on incoming ports to time-slots on outgoing ports. The following factors need to be considered for TDM switches†:

- Framing: There are different framing standards for TDM switch ports – SONET, SDH and OTN – frames differ in sizes and overhead bytes
- Line-rate: In the SONET standard, an interface with ~ 10Gbps line-rate (actually 9.95328 Gbps) is an OC192, but in OTN it is OTU2 (10.709225 Gbps).
- Time-slots: In an OC192, the number '192' comes from the fact that the line-rate can be divided into 192 time-slots each with a data-rate of roughly 50 Mbps.
- Signal-type: The smallest signal is the 50Mbps time-slot (STS-1). But larger signals are defined which use more time-slots. Thus the OC192 can carry multiple signals concurrently – 192 STS-1s; or 20 STS-1s and 10 STS-3s with 142 unused time-slots; or even 1 big STS-192c. Thus a SONET 'signal' is defined by the number of time-slots it uses and its starting (lowest) time-slot in an optical carrier (OCx).
- Concatenation: In SONET/SDH, signals can further be concatenated using contiguous (STS-3c, STS-12c, VC-4-4c etc.) or virtual concatenation (VCAT, ODUflex);
- Switching-granularity: It is necessary to understand the minimum switching granularity of the switching fabric – for example, if it has the ability to switch time-slots as small as STS-1s.

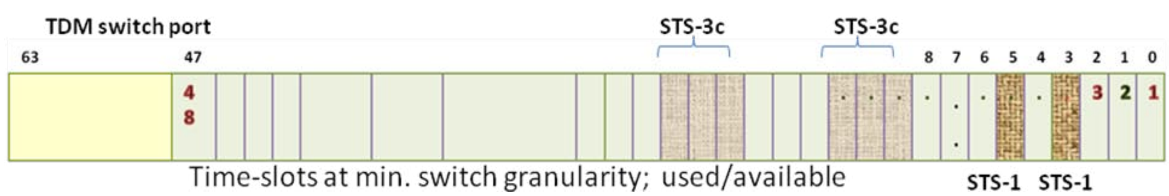


Figure 2.5: Representing bandwidth in TDM switches

- Bandwidth representation: Given the line-rate & the minimum switching granularity, a bandwidth representation of a time-slotted port can be drawn up (Fig. 2.6). It is not enough to give cumulative numbers for reserved and un-reserved bandwidth; instead

† Other factors such as signal-multipliers, transparency, and rules for contiguous and virtual concatenation have been left out for clarity

each minimum-granularity time-slot on a port can be represented by a bit field in a bit-map. Then a 1 or 0 value for the bit signifies the availability of that time-slot.

- Cross-connection: To specify a cross-connection, the input and output time-slots have to be specified (Fig. 2.4). The time-slots can be specified as a 3-tuple of <port, TDM signal-type, starting-time-slot>. The port identifies the physical port; the signal-type identifies the number of time-slots; and the starting-time-slot identifies the lowest (or earliest-in-time) time-slot in the carrier where the signal starts. In Fig. 2.5, one of the STS-3cs starts at time-slot #10 (bit number 9) in the OC-48 the bit map represents, and since it is an STS-3c signal it occupies 3 time-slots (10-12). In the same example, we see that the carrier currently has 2 STS-1s and 2 STS-3cs (shaded) and 40 free (unused) time-slots.
- TDM circuit: Therefore a circuit in a TDM network is simply a series of cross-connections in switches. As an example, consider the STS-3c circuit (150 Mbps) in Fig. 2.6. The signal-type (STS-3c) must remain the same throughout the circuit definition. The port numbers have significance only to the switch the port belongs. And the time-slots can interchange in a connection if the switch supports such behavior. But the time-slots on a link must be the same. For example, the outgoing STS-3c signal on the first switch starts on the 10th time-slot (on port 4). Therefore the cross-connection on the second switch must specify the same start-time-slot on port 5, which connects to port 4 on the first switch.
- Bi-directionality: Circuits are always bidirectional. Thus specifying a cross-connection from an 'in' 3-tuple to an 'out' 3-tuple, simultaneously specifies exactly the same cross-connection in the reverse direction.

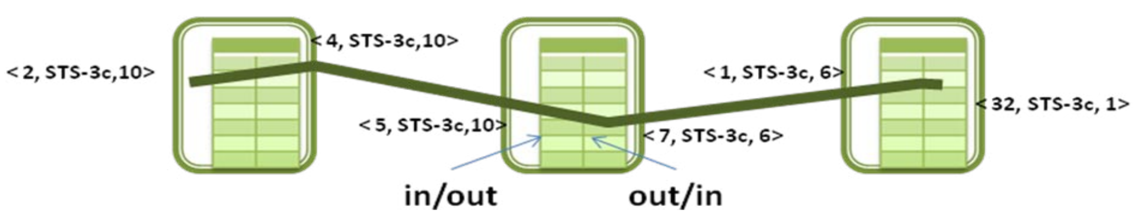


Figure 2.6: TDM circuit

WDM switching: A wavelength switch cross-connects an incoming wavelength (or set of wavelengths) on a port connected to an outgoing same-or different wavelength (or set of waves) on a different port. Typically it is implemented as a combination of a wavelength-demultiplexer, a switching-fabric that switches light-beams and a wavelength multiplexer. The following factors need to be considered for WDM switches:

- Switching-granularity: Does the switch support a minimum-switching granularity of a single wavelength or band of wavelengths. If it is the latter, then how is the band defined (in terms of the number of wavelengths). A fiber-switch may be considered as a special case of a wavelength switch, where the 'band' is defined as all the wavelengths that can be supported in the fiber.
- Line-rate: A wavelength switch has mux/demux filters designed to operate in at a certain line-rate, which must be specified. Optical fiber-switches are typically agnostic to whatever signals are carried in the incoming fiber. However, there may be transponders or WDM filters attached to the ports that may limit the switch to a single line-rate. Then the line-rate and wavelengths of the signals have to be considered.
- Fabric-technology: For WDM switches, depending on whether a wavelength is switched electronically or optically, a number of additional factors crop up. If the switching fabric is electronic, the framing used has to be taken into consideration. Also the incoming wavelength can be switched to a different out-going wavelength. There may also be other technology dependant feature support – eg. out-going wavelength tunability; variable line rate; optical supervisory channel (OSC) support.
- Bandwidth representation: A wavelength switch port can be represented as a bit-map. The bits in the bit-map represent the ITU grid frequencies [30]. Flags can be used to identify the spacing of the frequency channels – 25 GHz, 50 GHz, 100GHz etc.; and to identify a C, L or S band system. Fig. 2.7 shows a 100 GHz spaced C-band system (191.3 THz to 196.7 THz). Using multiple bit-maps, the switch can report the waves it supports on a port, and the ones that are currently cross-connected (in-use).

Figure 2.7: Representing bandwidth in WDM switches

- Cross-connections: To specify a cross-connection, the input and output wavelengths have to be specified (Fig. 2.4). The wavelengths can be specified as a 2-tuple of <port, ITU-grid-frequency-numbers>. The port identifies the physical port; the wavelength or a set of contiguous wavelengths (for a waveband) on the physical port are identified by their corresponding bits in the relevant ITU-grid-bitmap. All other technology dependant factors have to be (implicitly) considered when specifying this cross-connection. For example, two different wavelengths cannot be cross-connected when wavelength conversion is not supported; if there are transponders or wavelength filters then different line-rates should not be cross-connected; wavebands have to be of the same size (same number of lambdas) etc.
- Wavelength circuit: A circuit in a WDM network is then simply a series of cross-connections in switches. Fig. 2.8 shows a single-wavelength circuit (not waveband) in a network of electronically-switched wavelength switches, where each hop of the circuit comprises of a different wavelength.

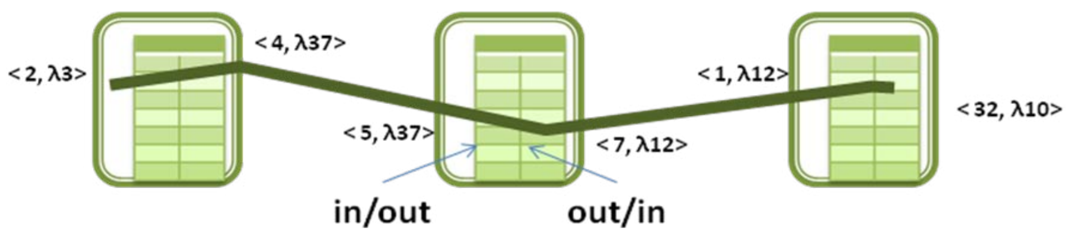


Figure 2.8: Wavelength circuit

- Bi-directionality: Similar to TDM circuits, wavelength-circuits or fiber-circuits are always bidirectional. Specifying a cross-connection from an in-lambda to an out-lambda simultaneously specifies exactly the same cross-connection in the reverse direction.

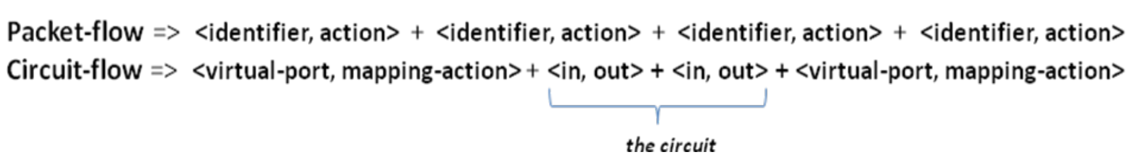
Viewing a Circuit as a Flow: It is worth comparing Figs. 2.6 and 2.8 which show TDM and wavelength circuits, to Fig. 2.3 which shows a packet flow. We note the following similarities:

- Both packet switches and circuit switches can be represented as forwarding-tables that support translations – in the packet case, an incoming <packet, port> is translated to an outgoing <packet′, port′>; in the circuit case, an incoming < λ, time-slot, port> is translated to an outgoing <λ′, time-slot′, port′>.
- A packet-flow is series of <match-identifiers, actions> in all packet-switches that the flow traverses; Likewise a circuit is a series of <in-x-tuple, out-x-tuple> cross-connections in all the switches the circuit passes through. Furthermore a decision to create a cross-connection within a switch is not-independent of cross-connections in other switches that make up the circuit (similar to our definition of packet flows).

And so, given these similarities, it is easy to see that the data-abstraction and switch-API that we discussed in the previous section can potentially be used in similar ways in the circuit-switching context.

There is however one important distinction. A packet-flow is the logical-association between packets that are part of the same communication. The packet-flow exists *in itself* to bind-together the packets as they flow in the network. On the other hand, a circuit as defined so far is a *carrier*; and it only becomes a logical-association for something between two-communication end-points, when that something is mapped into the circuit.

And so, to complete our analogy to packet-flows, a circuit becomes a *circuit-flow* only when we account for the end-points of the circuit in relation to what gets mapped into the circuit. Such an end-point is represented as a virtual-port with associated mapping-actions.



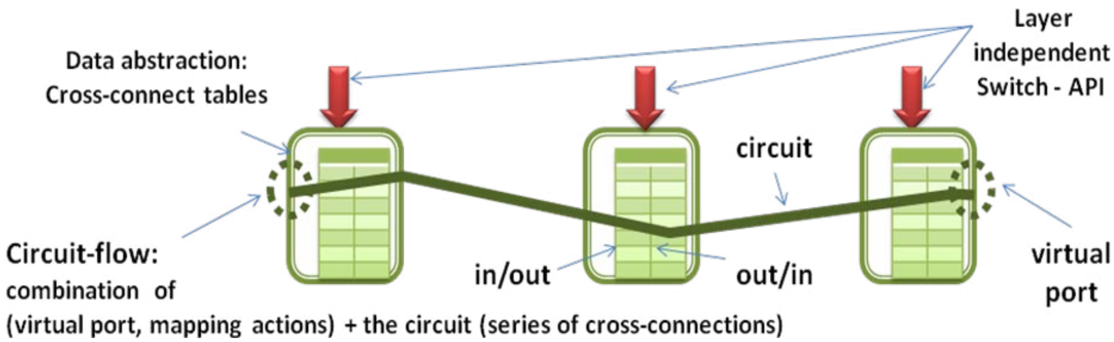


Figure 2.9: Circuit-flows

Circuit-Flow Abstraction: We are now ready to define a generic data-plane abstraction of circuit switches based on flows:

- A circuit-flow is a logical association between a payload that is carried by a circuit and is therefore given the same treatment in the network (from one end of the circuit to the other);
- The data-abstraction is the representation of a circuit switch as flow-tables and ports: the flow is defined by a) virtual-ports that identify the end-points of the circuit with mapping-actions to map payloads into-and-out of the circuit; and b) a bidirectional cross-connection which translates an incoming circuit-identifier to an out-going circuit-identifier;
- The decision on the treatment given to the payload in the circuit-flow is communicated in a layer-independent way to all switches which 'see' the flow, ie. the switches through which the circuit passes;
- The circuit-flow definition serves as a common-handle with which accounting and resource-management in the network is performed on the circuit-level.
- Finally, the data-abstraction in all circuit-switches is manipulated by a layer-independent switch-API. Given the similarities between the data-abstraction for packet and circuit switches, it is easy to see that the API can be a common one for both kinds of data-abstractions.

Mapping Packet-flows to Circuit-flows: Representing circuit-flows as combinations of virtual-ports and cross-connections presents a way to map packet-flows to circuit-flows, irrespective of the way in which packet and circuit switches are interconnected. Consider the different ways in which packet and circuit switches can be interconnected. In Fig. 2.10,

- If the packet switch P is connected to the TDM circuit switch via a Packet over SONET (POS) interface, then the virtual port is manifested by the PoS port.
- If P is connected to a hybrid switch (with both packet/circuit switching fabrics) via an Ethernet interface (ETH), then the hybrid switch has the capability to adapt Ethernet frames to TDM frames (SONET or OTN). The virtual port is then manifested by a mapper that performs this mapping.

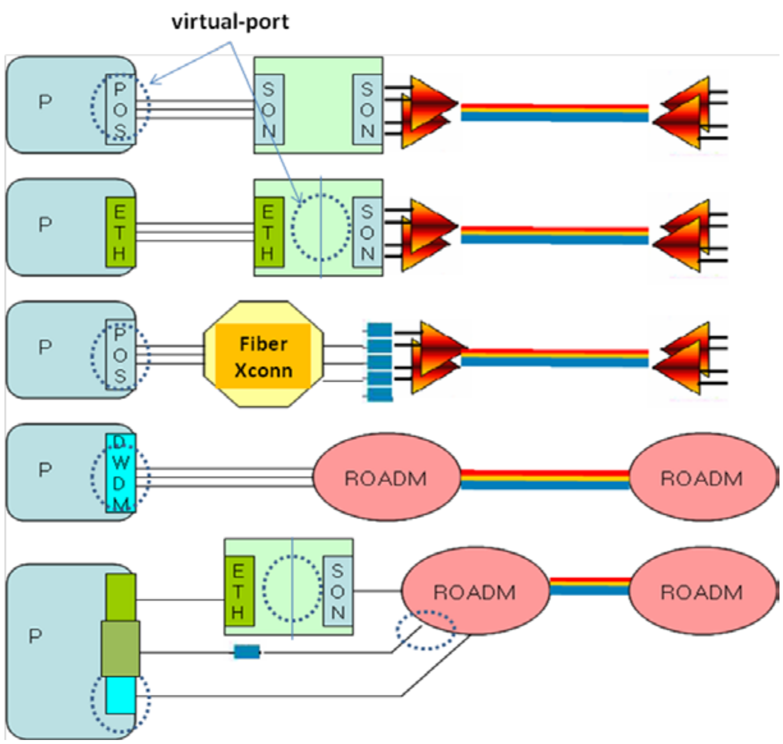


Figure 2.10: Various ways to interconnect packet and circuit switches

- If P is connected via a fiber cross-connect (Xconn) to a DWDM line-system, then again the POS port is an instance of the virtual-port. This PoS port typically does not use high quality transceivers needed for long distance communications neither does it use the standardized ITU grid wavelengths– hence DWDM transponders are needed before the signal is transmitted over the DWDM line systems. Such transponders are reported as technology-specific switch-features by the Fiber-Xconn.
- If P uses DWDM transceivers (with suitable framing), it could connect to the DWDM line-system via a wavelength-switch (ROADM). The virtual port is the DWDM interface on the packet switch. The wavelength on the transmitters may even be tunable. If instead P uses an Eth interface with a transponder to connect to the ROADM, the virtual port would be the transponder in the ROADM interface.
- Lastly, different ports on the packet switch could use different combinations of the above, in which case the virtual ports for each of the circuit-flows could be in different switches.

To map a packet-flow into a circuit- flow (Fig. 2.11), the last action in the action-set defined for a packet-flow, forwards all packets that match the packet-flow definition to the virtual-port. This way multiple packet-flows can be mapped into the same circuit-flow. At the other-end of the circuit-flow, the packet-flow identifiers match on the virtual-port and any other packet-header-fields to distinguish between and de-multiplex the packet-flows coming out of the virtual port.

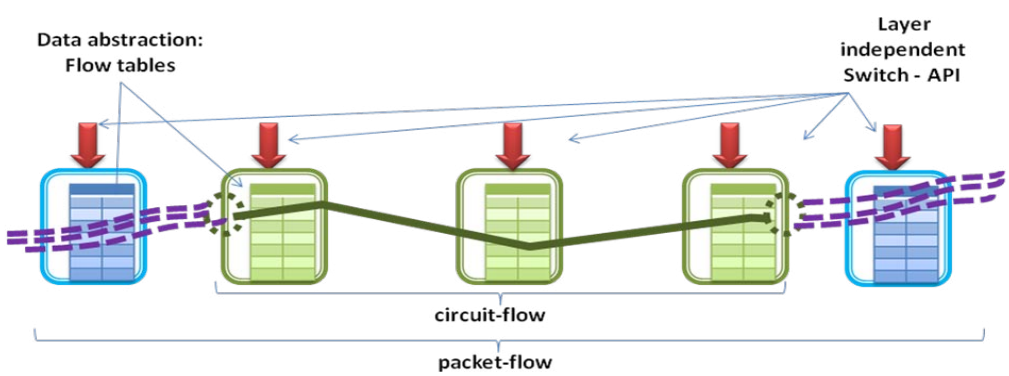


Figure 2.11: Mapping packet-flows to circuit-flows (and back)

Switch API: The circuit-flow abstraction holds remarkable similarities to the packet-flow abstraction. Unsurprisingly, the switch-API that manipulates a circuit switch abstracted as <flow-tables, ports>, also has a similar set of functions – the ability to understand the capabilities of the data-abstraction and have control over its configuration; full control over the forwarding state; and the ability to monitor status and obtain statistics. In fact a common switch-API can be developed for both packet and circuit switches, with small modifications to account for technology-dependant features. We details such modifications below:

- Get/Set Capabilities and Configuration:
  - Methods to get the capabilities of the cross-connect-tables. Features include:
    - Switching fabric type – time-slot (TDM), wavelength (WDM) or fiber;
    - Switching-granularity – TDM: smallest signal-type that can be switched; WDM: single wavelengths or band of wavelengths;
    - TDM signal support, WDM wavelength range
    - Mapping-Actions support: TDM: virtual-concatenation support using VCAT (SONET/SDH) or ODUflex (OTN), LCAS support etc; WDM: variable line-rate support, tunable wavelength support etc.
  - Methods to get the switch make-up: port, tables and bandwidth representations.
    - Port representations – line rates (OC48, OC192, OTU0/1/2/3 etc.), framing types (SONET, OTN, Ethernet etc.)
    - Bandwidth representations – TDM ports: used and available time-slots at minimum switch-granularity; WDM ports: used and available wavelengths for a defined grid spacing and range
    - Recovery support – most circuit switches have in-built hardware support for link recovery. Such recovery types are known as 1+1, 1:1, N:1 etc.
    - Neighbor discovery support – some circuit-switches (digital ones) have the ability to discover their neighbors on the links they share with them.

    - Technology dependant switch features –eg. WDM: wavelength conversion capabilities; transponders; WDM filters etc.
  - Methods to set configurable parameters on a local port or table basis as well as on a global switch basis.
  - Method to query default or current values of these configurable parameters.
- <u>Control forwarding state</u>:
  - Method for adding circuit-flow definitions by specifying the virtual-port and mapping-actions.
  - Method for creating cross-connections between: physical fiber-ports, wavelengths and time-slots. Also means for connecting virtual-ports to any physical fiber-port, wavelength or time-slot.
  - Method for changing the mapping-actions applied to a circuit-flow's virtual-port.
  - Method for deleting the circuit-flow definition – both cross-connections and virtual-ports. Deleting a virtual port is equivalent to deleting the cross-connections that support it within a switch.
  - Methods to set recovery state eg. link-protection.
- <u>Monitor:</u> Statistics and Status
  - Methods for querying cross-connect table state
  - Methods for querying virtual-port state (example: the packet-flows mapped into the port) and statistics (transmitted/received bytes etc.).
  - Methods for setting traps for change in switch-state: example physical port up/down, or virtual-port queue depth (queue of packets entering circuit)

As before, the methods described above may elicit a reply from the switch with requested information, positive acknowledgements or error messages, or the switch may simply process the function with silent acknowledgements.

In our work, we have implemented the common-switch-API by extending the OpenFlow protocol to manipulate circuit switch flow-tables [36].

2.2 The Common-Map Abstraction

The common-map abstraction is based on a data-abstraction of a network-wide *common-map* manipulated by a *network-API* (Fig. 2.12a). The common-map has full visibility into both packet and circuit switched networks, and allows the creation of network-applications that work across packets and circuits. The common-map is created and kept updated and consistent with network state by the unified-control-plane (UCP).

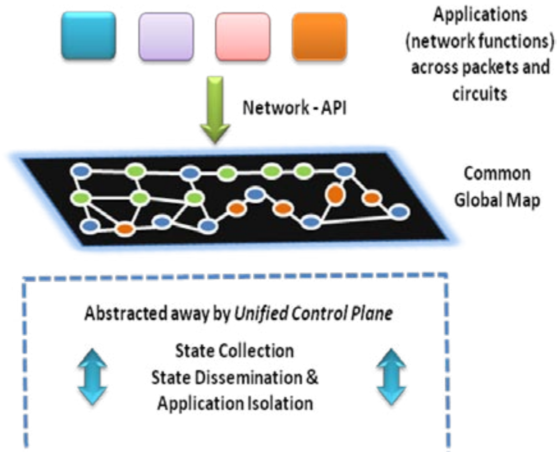


Figure 2.12(a): Common-Map Abstraction (same as Fig. 1.12)

The common-map makes it simpler to implement network functions by abstracting away (hiding) the means by which network state is collected and disseminated. Today network functions are implemented as distributed applications tightly coupled to the state-distribution mechanisms that support them. By breaking this coupling, the common-map abstraction allows applications to be implemented in a centralized manner. Not only does this make the applications simpler, it also improves extensibility, as inserting new functions into the network becomes simpler. Moreover, the network itself becomes programmable, where functionality does not have to be defined up-front by baking it into the infrastructure. A network-API can be used to write programs that introduce new control-functionality as the need for it arises. With its full visibility, the common-map allows new features to be supported that take advantage of both packets and circuits.

Importantly it gives the application-programmer the choice to treat packets and circuits together or as separate layers, or even completely ignore one or the other in the application context. Together the common-map abstraction benefits of programmability, simplicity, extensibility and choice ease the path to convergence and innovation in packet and circuit networks.

In the following discussion we first briefly describe a representation of the common-map. We then describe means by which it can be constructed and maintained, with emphasis on the important aspects of link-discovery and layering. Finally we detail possible features of the network-API.

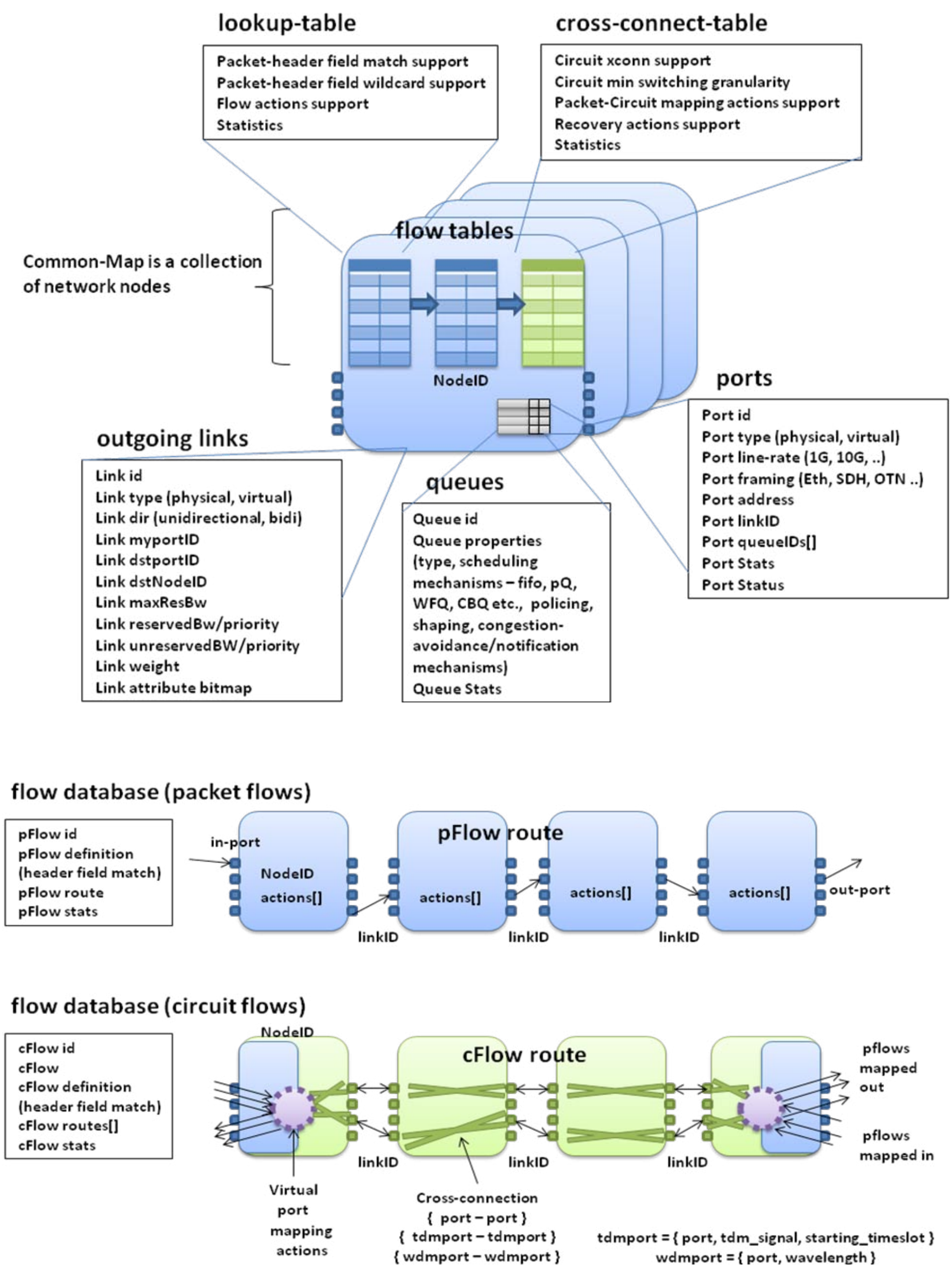
2.2.1 Common-Map Representation

In Chapter 1, we introduced the common-map as an annotated graph (or database) of the network topology (Fig. 2.12b). This graph is a collection of network Nodes. In the context of wide area networks, the nodes are essentially switches[†]; packet, circuit, and hybrid-switches which have both packet and circuit switching features.

Nodes: Each node is a collection of the following: flow-tables (both lookup-tables and cross-connect tables); outgoing links (physical and virtual); ports (physical and virtual); and queues. The node is a data-structure that represents the switch abstracted as <tables, ports, queues>. In itself there is not much information regarding the node other than a unique identifier such as a Node-id. However, the collections held within a Node give more details of the data-abstraction.

Ports: The port data-structure includes information on the type of port – either physical or virtual. It has a unique identifier (PortId) and one or more network addresses and names. A physical port can have meaningful line-rates and framing type and a LinkId for the physical Link for the connected link. It can also have a number of Queues attached it (or at least indexes into the Queue collection). A virtual-port can be of many types. It can represent the end-point of a circuit flow into which packet flows are mapped.

† Note that in other networking contexts such as enterprise or campus LANs, 'Nodes' can also include end-hosts or middleware connected to the switches.



2.12(b): Common-Map Databases (same as Fig. 1.10 and Fig. 1.11) I

In some cases it can be manifested by a physical port. In other cases, it can represent a group of physical-ports for a variety of purposes, such as broadcast or multipath load-balancing. Finally, port level statistics and status indicators are also maintained as part of this collection.

Tables: The flow-table information for the lookup (packet) and cross-connect (circuit) tables mainly includes information of the capabilities and features supported by the tables. For example, the lookup table matching and wildcarding support for packet-header-fields, and the actions that can be applied to a packet are stored here. For the cross-connect table, the particular kind of switching fabric, minimum switching granularity, mapping-actions and recovery types supported are stored here. Lastly statistics are maintained on a table basis.

Queues: The queue-data structure can include information such as the QueueId; the PortId of the port it is assigned to; Queue statistics; and any other information regarding the type of queue – min-rate, max-rate, associated scheduling mechanisms like FIFO, WFQ, priority queueing, class-based-queueing, policing and shaping support, congestion avoidance (RED) and other AQM support. It can also include information on traps set by the user to be notified for queue occupancies cross set thresholds.

Links: While links in the network can be maintained in a database separate from the nodes-database, there are many advantages to maintaining it as part of the nodes data-structure. For example, it is a convenient way to quickly get the outgoing links from a node which is required by many routing algorithms; it is also convenient to represent unidirectional links (for eg. unidirectional MPLS tunnels or other virtual-links) or link features which are different in either directions (such as bandwidth reservations in MPLS tunnels). The link data-structure includes information on the destination portId and nodeId. It also includes information of max bandwidth, reserved bandwidth, unreserved bandwidth, and actual bandwidth usage for the out-going direction. The actual bandwidth representation depends on the type of link – eg a packet-link (used for MPLS-TE) or a TDM circuit link or a WDM fiber link. In all cases the notion of bandwidth and its reservation holds. Additionally the links database can maintain a link-cost (or weight) useful in certain path-calculations, as well as a set of attributes that the user can define for similar purposes. Ultimately the links data-structure is a repository of all link information. It is up to the user to use-or-ignore parts of the structure in the context of the application or its corresponding networking domain.

2.2.2 Common-Map Construction & Maintenance

To create a common-map, the unified control-plane learns about switch ports, queues, tables, and other switch characteristics and capabilities using the switch-API (discussed in the previous section as Get/Set Capabilities and Configuration). However, the network database is incomplete without information on network *links*. Thus as an important part of the construction of the common-map, we discuss various kinds of packet and circuit link-discovery mechanisms.

Link Discovery: With reference to Fig. 2.13, we define a packet link to be one that interconnects interfaces on two packet switches (or the packet switching part of multi-layer switches). These interconnections could be physical (for example two Gigabit Ethernet ports inter-connected by an Ethernet cable) or they could be virtual (eg. two PoS or GE interfaces connected over the wide-area by an underlying circuit). Circuit links are *always* physical links – essentially fiber or wavelengths that interconnect interfaces on circuit switches (or the circuit switching part of multi-layer switches).

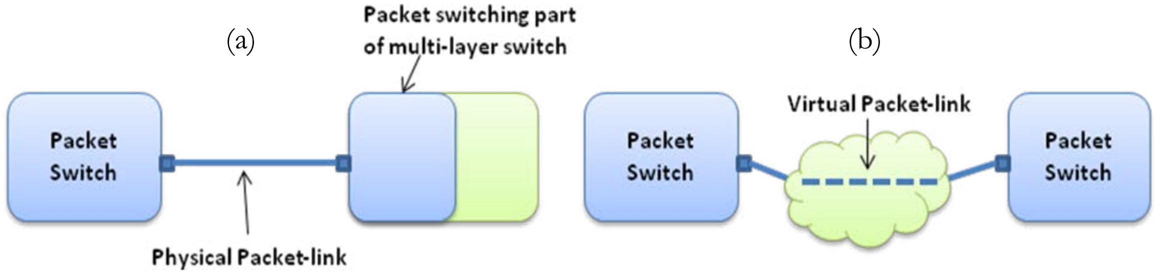


Figure 2.13: Physical and virtual packet links

Physical-packet links: Discovery can be performed by the control plane using test-packets. With the use of a mechanism to send packets selectively out of packet-interfaces, the UCP can send specially generated test-packets out of all known ports on switches that it has control over. When such a test-packet is received back by the UCP from a switch other than the one it was sent out of, the UCP can determine which port on the receiving switch is connected to which port on the sending switch (Fig. 2.14a).

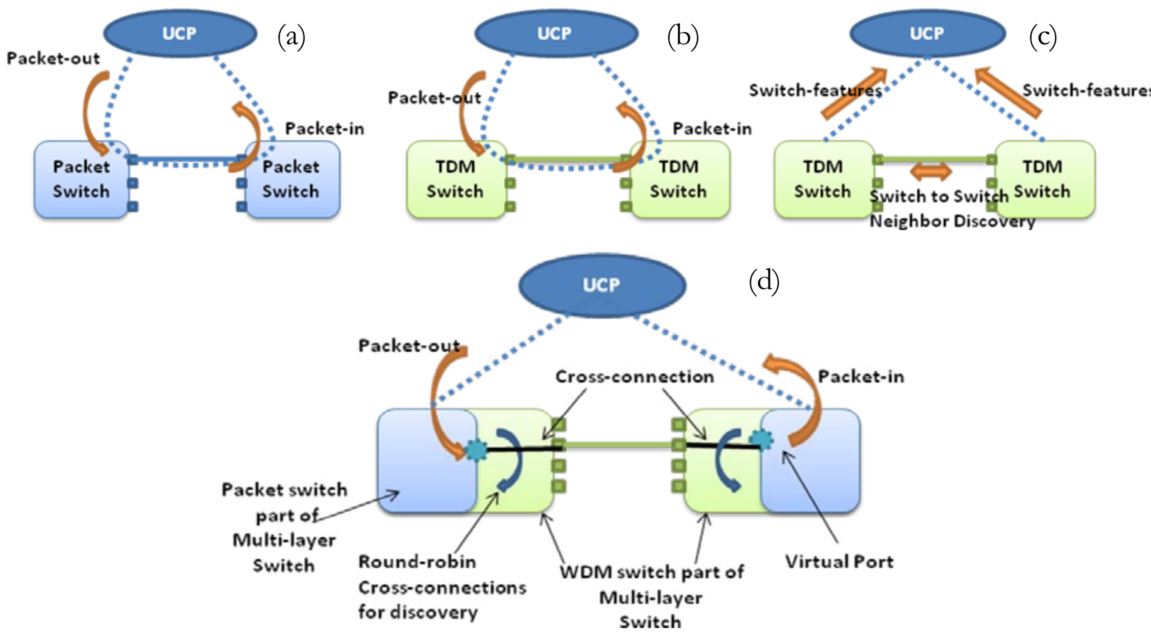


Figure 2.14: Discovery of packet and circuit links

TDM Circuit Links: On circuit switches such a mechanism is possible, but slightly more involved. Consider first a TDM switch based on SONET/SDH (but applicable to TDM switches based on OTN as well). SONET switches periodically send out SONET frames on all interfaces (eg. 8000 frames/sec). Each frame consists of a payload and a header, and there exists special header bytes (the DCC bytes [5]) reserved for packet-communications. These bytes can be used to carry the packets sent by the packet-out mechanism, with the understanding that software on the receiving end must put together the packet (spread over the DCC bytes in multiple consecutive frames), then add its receiving port and switch-id and make it available to the UCP. This scenario is depicted in Fig. 2.14b and is a viable way to support discovery of physical circuit-links.

Another way to achieve the same result for both packet and circuit *physical* links is to have the switches themselves perform this discovery using software/hardware *outside* the purview of the UCP. Since a lot of existing equipment already uses these DCC bytes to do neighbor discovery, all that is required is to report the *results* of the discovery process to the UCP as part of the switch-API (Fig. 2.14c).

One advantage of doing some link-layer tasks in the switches is that very fast keep-alives can be sent switch-to-switch to monitor the health of the link (link up-or-down) and save the UCP from processing the keep-alive. However, the disadvantage is that these protocols are vendor proprietary today making them hard to interoperate for discovery or keep-alives. We feel that both cases should be allowed, together with the use of standardized neighbor discovery (like LLDP [31]) and fault-detection protocols (like BFD [32]).

WDM Circuit Links: In the case of wavelength and fiber based circuit switches, where there is no visibility into packets, neighbor discovery and therefore link-discovery is hard to achieve. However we find that supporting the trend towards multi-layer switches, a lot of wavelength-switches are supporting packet interfaces with limited packet switching capability [33, 34]. We can then support the discovery of circuit links in wavelength/fiber switches via mechanisms shown in Fig. 2.14d. A packet sent via a packet-out message is periodically sent out of a virtual-port which is cross-connected to one of the interfaces. A virtual port on a receiving switch is cross-connected in round-robin order with all circuit ports. When one of the test-packets periodically sent out is eventually received successfully on a particular connection, the corresponding-link can be discovered.

However, this procedure has two drawbacks – a) it is time consuming as the round-robin nature needs to be repeated for all ports in all switches; b) it is disruptive to service as while this discovery procedure is going on, live traffic cannot be carried over the switches. For these two reasons, in this case, instead of doing live discovery it may be better to 'configure' the switches with their neighbor switch and port ids, so that the switch can report it to the UCP via the switch-API.

Given the above argument and keeping Fig.2.14c in mind, we included the ability to report peer switch-id and port-id per switch circuit-port in the *switch-API* as neighbor-discovery support; as well as the packet-in/out mechanism for UCP based discovery in the switch-API. Note that use of this API feature means that the UCP is not directly involved in the discovery process. Thus the reporting of neighbor information requires the UCP to have a verification methodology that ensures that different switches which report each other as peers on a certain port, report their peer's switch and port id correctly. A circuit or packet link reported this way is deemed discovered only when both sides of the link report the other end correctly.

Constructing Layers: In discussing the common-map abstraction we said that we can choose whether to treat packet and circuit switches as part of the same topology or as part of different topologies (and thus different layers).

Creating network applications across packets and circuits is simple when we treat both kinds of switches as part of the same topology or layer (a sea-of-switches). On one hand, the right-side of Fig. 2.15 shows how *physical* packet links can be discovered by the controller (using test-packet-in and outs) and physical circuit links can be reported by the circuit (or multi-layer) switches and verified by the UCP, using any of the mechanisms shown in Fig. 2.14. This allows the creation of a single *physical* topology comprising of a sea of packet and circuit switches. We show an example of a network application on top of such a topology in Chapter 3.

On the other hand, the left-side of Fig. 2.14 shows a mechanism for creating separate packet and circuit topologies, where the network application treats them as separate *layers*. The packet layer (or topology) can consist of physical packet links when they link together packet switches that do not go over the wide-area. These links can be discovered by the packet-in-and-out mechanism provided by the switch-API. Similarly physical circuit links can be discovered or verified by the methods mentioned in the previous section. These links then form a separate circuit-topology (or layer). Importantly the circuit layer also includes the packet-switches that are at the border of the packet-network and connect physically to circuit-switches. These physical connections (either packet or circuit links) are also part of the circuit-topology as they contain vital information for mapping packet flows to circuit flows.

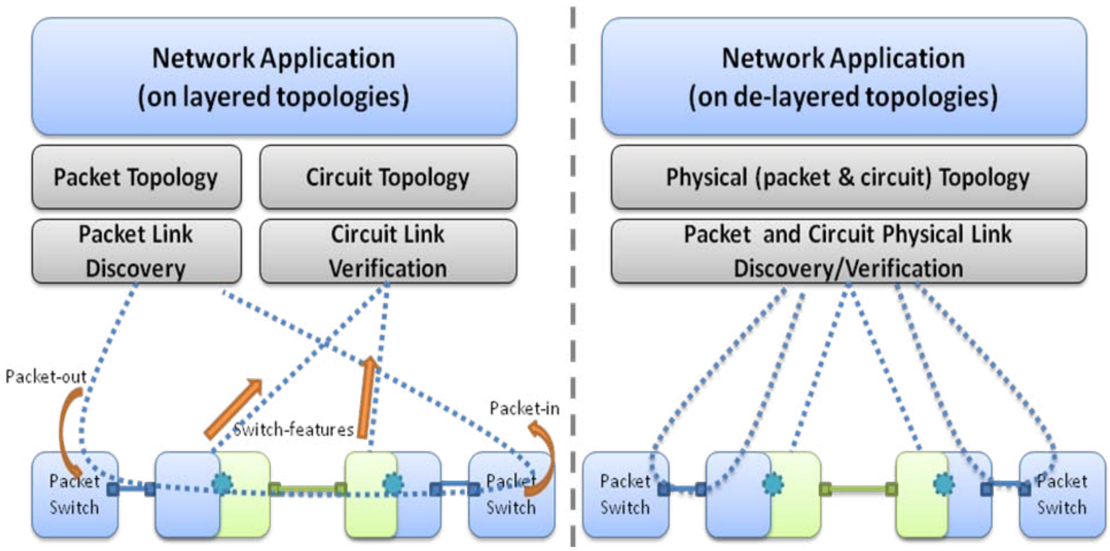


Figure 2.15: Layering and the Common-Map Abstraction

Discovery of virtual-packet links over the wide area can be performed by tunneling test-packets from packet switches over circuits created to support the virtual-packet-link. At the other end of the tunnel, the test-packets are received by the UCP completing the discovery of the virtual packet-link across the wide area.

Common-Map Maintenance: Entities such as ports and nodes report their status to the UCP via the switch-API. Additionally the status of links can be discovered, implied (from port status) and verified by the UCP via the switch-API. The common-map is updated and maintained by reacting to these status updates by creating/removing/or updating the relevant data-structures for the entities.

2.2.3 Network-API

A network-API for the common-map can have two parts: one that allows switch-level control and the other allows network-wide control. We consider them separately below. In discussing the network-API we do not distinguish between API calls for packet flows and circuit flows. The set of calls described, apply to both packet switched networks and circuit switched networks. The only difference is in the switching technology. Thus the implementation of these API calls is typically technology specific. For example, a function-call to setup a flow can involve installation of only packet-header fields in packet flow tables, while a circuit flow setup can include creating virtual ports at the end-points and setting up switch-fabric-specific (WDM, TDM) cross-connects for the circuit.

Switch-Level Control: A network application may require access to (internal) switch level detail for a variety of reasons depending on where the switch is located in the network topology and the kind of network function being targeted – examples include access-control, specialized/advanced forwarding (multipath-selection), monitoring queue depths etc. In such cases, the network-API essentially provides a wrapper around the switch-API calls. Thus all (or most of) the switch-API calls discussed in Secs. 2.1.1 and 2.1.2 could in principle be wrapped into a network-API call as (network-node-id, *switch-API*-call). Such calls include commands that the application can use to change/configure attributes in the switches and control the forwarding path within a switch. Also almost all network-wide monitoring is done on a per-switch basis. The application can register to be notified when such monitoring-events happen. Behind the scenes, the network API wrapper sets the switches using the 'trap' functionality of the switch-API.

Network-Level Control: Network applications may require the following min-set of capabilities from the network-API:

- Topology-choice: Depending on requirements the common-map abstraction may provide a single-topology or multiple topologies – two common examples are a) virtual-topology of packet-links on a physical-topology of circuit links, and b) a virtual topology of tunnels (eg. MPLS) on a physical-topology of packet-links. The application needs to choose which topology it works with for routing etc.

- Routing: In general routing can itself be considered as an application on top of the network-map. The user can define a routing-algorithm to support the following API calls, or the UCP could include a built-in generic routing-algorithm that supports the API. One example of a generic routing-API uses a Constrained Shortest Path First (CSPF) algorithm that finds the shortest path in the network given certain constraints. Such constraints can be bandwidth, delay, or any other user-defined link - attribute.
  - Methods to set attributes for links in the chosen topology before routing can be performed. Simple attributes are link-costs and maximum reservable bandwidth.
  - Method to get a route (get_route) on the chosen topology between source and destination nodes, depending on a specified set of constraints– this is essentially a check to see if a route can be found that meets all the constraints (applies well to MPLS-TE and circuit networks). Note that get_route can be run with multiple constraints simultaneously, or with no constraints at all. In the latter case the caller would get the path with the shortest cost. If all link costs are the same, the route would have the shortest number of hops.
  - Method to check an explicit route - the application can specify an explicit route by detailing the nodes along a path from source to destination. This call will verify the validity of the explicit route, given the current state of the network and the specified constraints.
  - Method to check an existing route – the application can specify an existing route explicitly and check for the feasibility of a change in a constraint along that route. For example, applications can check if a flow (with reserved-bandwidth) can increase its bandwidth-reservation along the path it currently takes.
  - Method to detail a loose-route – loose routes are partially specified routes. This call fills in the details of the nodes along the loose route by checking for all the specified constraints along the loose route.

- Flow-setup:
  - Method to setup a flow (packet or circuit flow) in the network – given a route, and a set of <flow_definitions, associated actions> for each node along the route, a flow is created (packet-flow, circuit –flow, virtual-circuit-flow). Under the hood, the UCP sets up the flow-table entries in each switch along the route using the switch-API. If such a call changes the link-attributes then the common-map is updated. For example when a circuit is created with a certain bandwidth, it consumes that bandwidth along the links the route traverses. Thus the links in the common-map have their attributes updated.
  - Method to delete a flow (which updates the map if necessary)
  - Method to re-route a flow by changing the set of actions associate with the flow or changing the route.
  - Method by which an application can register for a type-of flow so that if matching packets arrive then they can be routed.
- Recovery:
  - Methods to inform applications of changes in network topology (link or node failures) so that the application can figure out new routes and install them.
  - Method to configure the routing engine to automatically respond to change in network topology by re-routing flows without waiting for the application to explicitly re-route them.
  - Methods to set advanced re-routing state, so that the switches have pre-installed state for re-routing when network failures occur. Such pre-installed state could be backup paths (like MPLS Fast Reroute or circuit protection paths)
- Network-wide Monitoring:
  - Methods for the application to be notified of network-state along a route or on specific links eg. congestion
  - Methods for sampling packets along installed packet-flows in the network.

2.3 pac.c Prototypes

We discuss three prototypes we built to validate our architectural and control plane constructs [35]. The Unified Control Plane (UCP) from Fig. 2.16 consists of:

- OpenFlow: An interface/protocol that instantiates the common-flow abstraction by enabling the switch-API into packet and circuit switches. Our work extended version 1.0 of the protocol [28, 36] for circuit switches (Sec. 2.1.2).
- A Controller running a network-wide Operating System called NOX [15]. We instantiated our common-map abstraction by building and maintaining the common-map and network API on top of NOX, with ideas discussed in Sec 2.2
- A Slicing Plane (not implemented) which is crucial to the practical deployment of the common-map, as we show in Chapter 3.

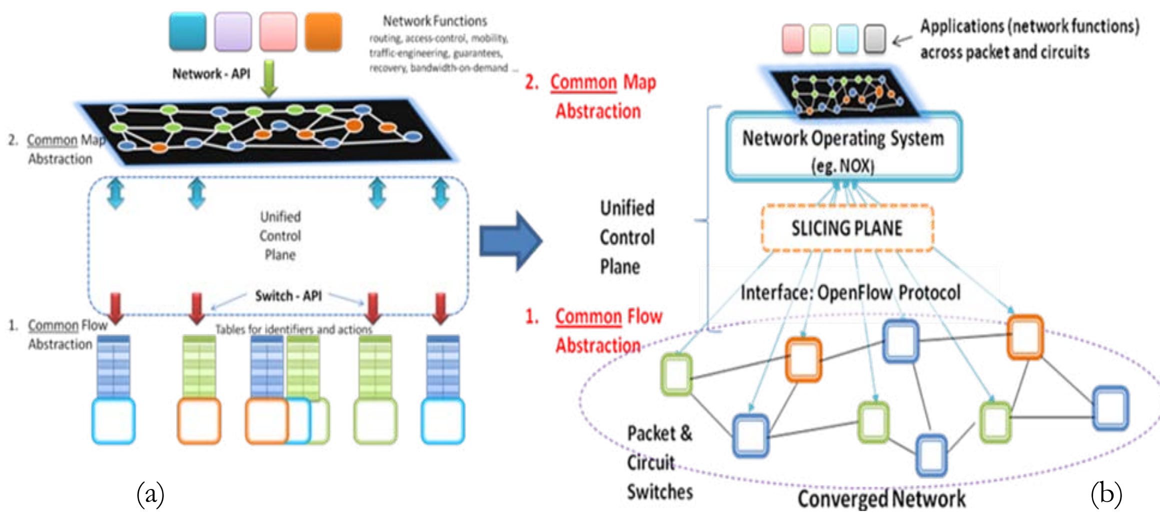


Figure 2.16: Implementation of our Architectural Approach

We first present two early proof-of-concept prototypes we built with two different kinds of circuit switches - wavelength & TDM. These prototypes focused on validating the flow-abstraction. Our third and more complete pac.c prototype validates the common-map abstraction.

2.3.1 Prototype #1: Packet & Wavelength Switch Control

Goal: Validation of the common-flow abstraction by demonstration [37, 38] of unified control over OpenFlow enabled packet switches and an OpenFlow enabled optical wavelength switch.

Data-Plane: The main circuit switching element is a Wavelength Selective Switch (WSS) from Fujitsu which forms the basis of the Flashwave ROADM systems [34]). The WSS is an all-optical wavelength- switch in a 1X9 configuration. It has the ability to independently switch any of 40 incoming wavelengths at the single input port, to any of 9 output ports (Fig. 2.17). The incoming wavelengths (100 GHz spaced, ITU C-band) are de-multiplexed and directed to their respective MEMS mirrors, which are rotated to direct the wavelength to any of the 9 output ports, where they are multiplexed back into the outgoing fiber.

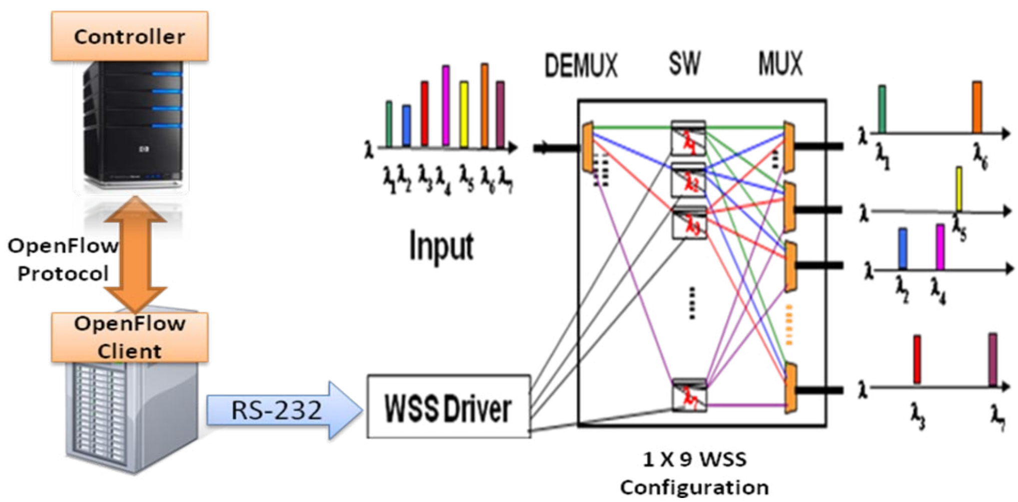


Figure 2.17: OpenFlow enabled Wavelength Selective Switch(WSS)

The WSS mirrors are controlled with a voltage-driver, which is sent commands over RS232 from a PC. The PC runs a modified version of the OpenFlow reference switch [39]. We used the OpenFlow client part of the code which interacts with the Controller and modified according to our changes to the OpenFlow protocol [36]. The rest of the code that implements packet-switching in software is eliminated. The modified client was integrated with an RS-232 driver to send commands to the voltage-driver that directs the mirrors. Together the OpenFlow-client PC, voltage-driver and the WSS can be regarded as an OpenFlow-enabled circuit switch.

The data plane also consists of two OpenFlow enabled packet-switches implemented in software with the OpenFlow reference switch [39]. The switches can be hosted in any PC with multiple Ethernet interfaces. In our testbed we used PCs with NetFPGAs* as 4-port Network Interface Cards (NIC) [40].

Control-Plane: Our controller was implemented by making changes to the simple reference-controller that is distributed with the reference switch [39]. The focus in this prototype was on implementing the common-flow abstraction correctly. As a result the simple controller used in this prototype does not really create the common-map. Instead it treats each packet-switch as independent Ethernet learning-switches.

Experimental Setup: Fig. 2.18 shows our experimental setup. Two packet-switches NF1 and NF2 are connected via an optical link. Each packet-switch has four Gigabit-Ethernet (GE) electrical interfaces. One of the electrical ports (on each switch) was converted to an optical interface via a GE-to-SFP electrical-to-optical converter from TrendNet [43]. We used 2.5 Gbps SFP transceiver modules from Fujitsu in the converter, which transmitted DWDM ITU grid wavelengths 553.3 nm and 1554.1 nm. These wavelengths travel in opposite directions to form the bidirectional optical link.

The optical link comprised of 25km of single-mode fiber separating the two packet switches. The two wavelengths that form the bidirectional link were multiplexed/de-multiplexed into the fiber at the output/input of NF1 by a wavelength mux/demux (AWG). At the other end of the link, the wavelengths are again multiplexed/de-multiplexed into NF2 by the wavelength switch (as the switch has mux/demux capabilities as well). Initially however the wavelength *switch* is 'open' and so the wavelength coming out of NF2's transmitter (1554.1nm) does not reach NF1, and the wavelength transmitted by NF1 (1553.3nm) does not reach NF2. In other words the

* The NetFPGA is a programmable hardware platform [41]. Implementations are available now that can be installed in the NetFPGA to make it behave like an OpenFlow enabled hardware packet-switch [42].

optical- link is broken, and so is the Ethernet link it supports between NF1 and NF2. The optical link is monitored by tapping-off a small percentage of the light (2%) before the receivers in both directions, and feeding the tapped-light into an Optical Spectrum Analyzer (OSA). Finally we connected client PCs (end-hosts) to the other GE interfaces on NF1 and a video-server PC to NF2.

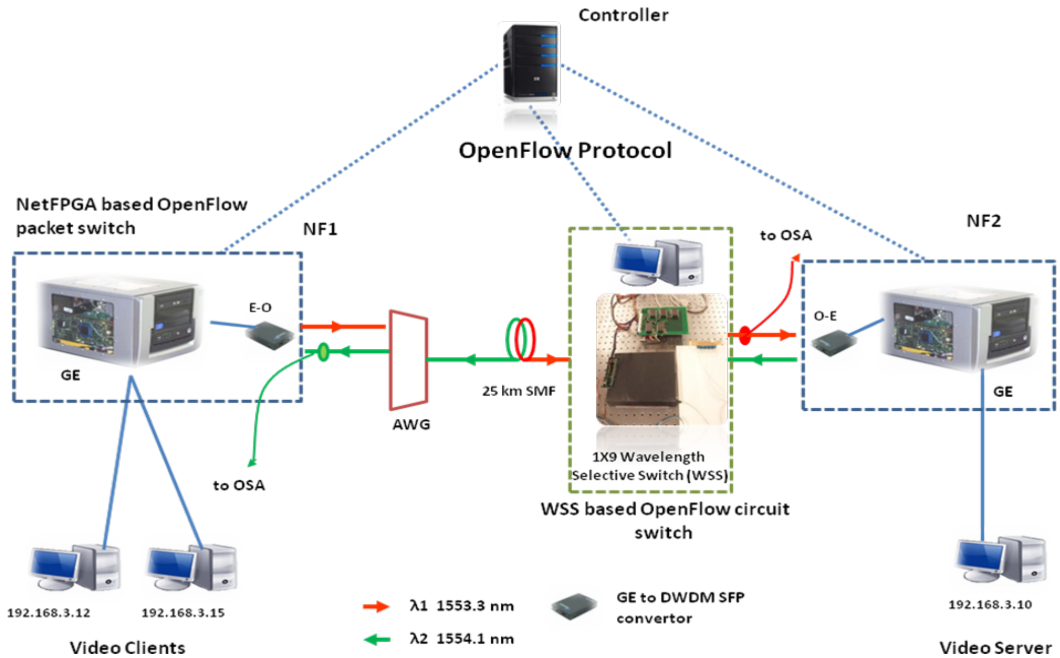


Figure 2.18: Prototype #1 - Packet and Wavelength switches

Experiments & Results: The basic idea is that the video-clients make requests for videos from the video-server. The video request is transported over TCP. But initially the client PC is not aware of the MAC address corresponding to the IP address of the video-server. It therefore sends out an ARP request, which is received by NF1. Since the ARP packet does not match any flow entries in NF1†, it gets forwarded to the OpenFlow controller as a packet-in message.

Upon receiving the ARP packet, the controller decides that in order to reach the server, a circuit needs to be created between NF1 and NF2. The controller uses the OpenFlow protocol to insert rules into the WSS cross-connect table, thereby making the

† Actually all flow-tables (in NF1, NF2 and the WSS) are empty at the start of the experiment.

cross-connection for each wavelength in the WSS. Once the bi-directional optical link is up, the Ethernet link also comes up between NF1 and NF2. The controller also inserts flow table entries in the packet-switches to broadcast the ARP request to all interfaces other than the one in which it received the packet. This results in the ARP request reaching the server PC via the WSS and NF2. The server PC sends the ARP reply, following which TCP handshaking takes place and the video request is transported to the server. To serve the video request, the server streams the video data packets using RTP over UDP. These packets are transported over the same bidirectional circuit created by the controller, and are received and displayed by the video-client.

We measured the time taken by various steps of the process (Table 2.1). The measurements were made using Wireshark running on NF1, NF2 and the WSS driver PC. Details of the configurations and the measurement procedures can be found in [37].

Steps	GE-Optical-GE link with WSS initially not cross-connected	GE-Optical-GE link with WSS already cross-connected	GE-GE link (no optics)
WSS connect command received	1 ms	-	-
Cross-connection confirmation	1.3 s	-	-
TrendNet GE-SFP module link-up	4.74 s	4.10 s	-
NetFPGA GE-GE link-up	1 ms	1 ms	1 ms

Table 2.1: Prototype #1: Time taken for connection set up

Conclusions: We make the following observations from our first prototype:

- The most important lesson from this prototype is the feasibility of our common-flow abstraction; that it was possible to treat OpenFlow-enabled packet and circuit switches as flow-tables that switch at different granularities (packet and lambda) and can be commonly-controlled by an external controller using a common switch-API.
- We found that the time taken to create a wavelength cross-connection is high (1.3sec). However this has nothing to do with our implementation of the OpenFlow protocol.

Instead the time is actually taken by the RS-232 communication between the driver PC and the WSS voltage-driver, which is not optimized for fast setup of connections. For example, while the mirrors can be rotated simultaneously, the cross-connect commands (over RS-232) are sent one at a time, and the second command cannot be sent until detailed feedback is read off from the serial port for the first command. We believe that optimizing this interface can reduce this number to tens of ms.

- The time taken by the GE-SFP convertor to recognize a link-up is high (4.10s), most likely because the internal mechanisms of the converter box have not been optimized for rapid link-up. We believe that a dedicated ASIC/FPGA optimized for fast setup will help in this regard.

2.3.2 Prototype #2: VLAN & TDM Switch Control

Goals: To build on our implementation of the common-flow abstraction by abstracting a different kind of circuit switch – a SONET/SDH based TDM switch; To explore the implementation of a simple-application across packet-and-circuit switches.

Data-Plane: The data-plane consists of carrier-class Ciena CoreDirector (CD/CI) switches [44].The CDs are hybrid switches – they have both Layer 2 (GE) interfaces with a packet switching fabric, as well as Layer 1 (SONET/SDH) interfaces with a TDM switching fabric (Fig. 2.19a). The packet switching capabilities on the CD are limited to switching on the basis of VLANs (and incoming- port). Nevertheless, the prototype has both packet and circuit switches (albeit housed in the same box).

Working with Ciena's development team, the OpenFlow protocol was extended to serve as a switch-API into the CD's VLAN and TDM (SONET) switching fabrics. Ciena's development team added native support in their switches for the OpenFlow protocol (and its circuit-extensions [36]) to serve as a switch-API. Simultaneously, we worked on a controller to communicate with the CD and run unit-tests to debug the development of the OpenFlow client in the CD.

Control Plane: The control plane featured a controller running NOX [15] over an out-of-band Ethernet network. NOX was modified to include our changes to the OpenFlow protocol for circuit switching. We retained only the basic NOX event-engine and connection-handler part of NOX (known as nox-core – see Fig. 2.19b). This early prototype did not include any of the discovery and layering features presented in Section 2.2. Neither did it use the built in features in NOX for packet link discovery and topology, as there aren't any standalone packet switches in this prototype (like the software packet-switches in Prototype #1). The map abstraction for this topology only applies to the circuit-topology which was statically defined (i.e. the topology was hard-coded; not discovered). The circuit-API shown in NOX was basically a wrapper around the switch-API commands in OpenFlow. The controller also interfaces with a GUI that shows network state in real-time (Fig. 2.20b, created using ENVI [47]).

Experimental Setup: We used three CDs in our prototype connected to each other via OC-48 (2.5 Gbps) SONET/SDH links (Fig. 2.20a). Similar to prototype #1, we connected video clients and server PCs to our switches, except this time instead of standalone software packet switches, we connected the PCs directly to the Ethernet (GE) interfaces on the CDs. The three CDs together form a small demo-network.

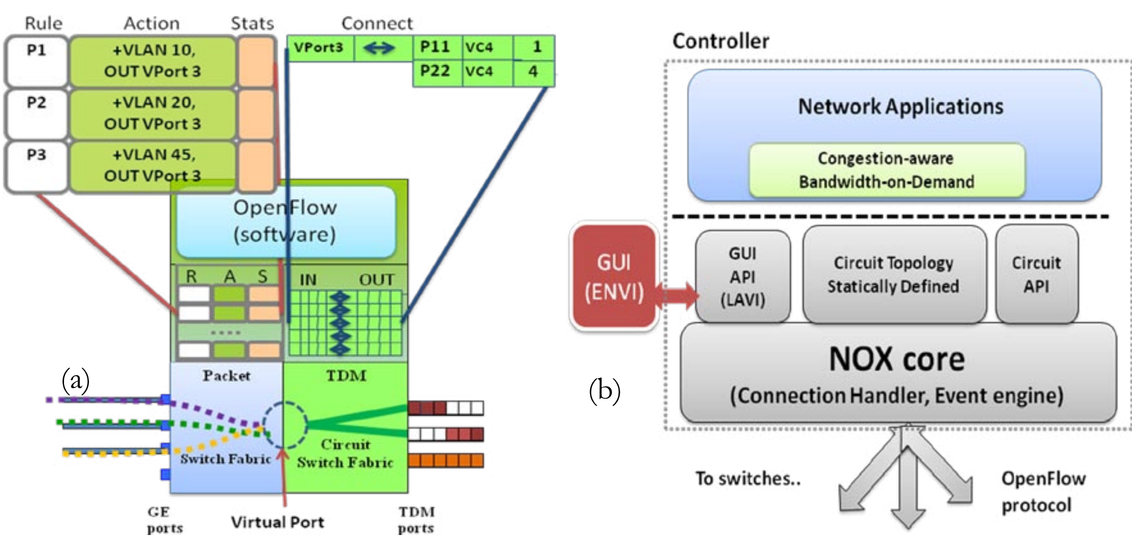


Figure 2.19: (a) CD internals and Flow Tables (b) Controller internals

Experiments: The focus was still on validating the switch-API (OpenFlow) for TDM switching, and developing the common-flow abstraction by interfacing TDM flows with VLAN based packet-flows. The best way to show this was by demonstrating the capability in NOX to set-up, modify and tear-down both L1 (SONET) and L2 (VLAN) flows on–demand. Once we had the ability to dynamically create and map packet-flows to circuit-flows, we wished to explore a simple network-application across them. And so, we created an application that dynamically responds to and relieves network congestion by adding bandwidth on-demand (Fig. 2.16b) [45, 46]. We discuss each of these experiments next.

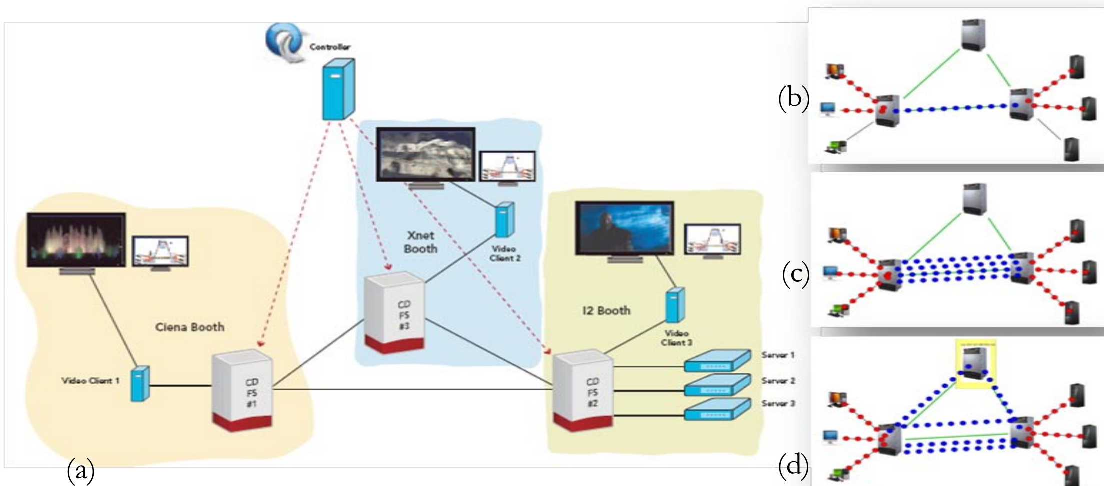


Figure 2.20: Prototype #2 Experimental Setup and Demo GUI

<u>L1 and L2 control</u>: The objective is similar to the one in the previous prototype where video-clients make requests for videos from remote streaming-video servers. When such packets (TCP SYN) from the first client arrive at the GE interfaces of the CDs, the CD tries to match the packet to rules in its packet-flow-table. Since the flow-table does not have entries for such packets, they are redirected to the OpenFlow controller (as packet-ins). The controller decides that for the packets to reach the IP address of the server, there needs to be a circuit-flow between CDs #1 and #2. Thus our application:

- Creates virtual-ports in the CDs to serve as end-points for circuit-flows, where packet-flows can be mapped into them – eg. VPort3 in Fig, 2.19a ;
- Inserts rules in the packet-flow tables that dictate that all packets coming in from client GE-port (eg. P1 in CD #1) and the video server GE-port (in CD #2) are tagged with a particular VLAN-id (eg. VLAN10 for P1 in CD#1) and forwarded to the virtual ports in the respective CDs (as shown in Fig. 2.19a);
- Inserts rules in the packet-flow tables for the *opposite* direction, which stipulate that all packets matching the <virtual port and VLAN-id> combination, have the VLAN tag stripped and then forwarded to the GE port corresponding to the client-PC (not shown in Fig. 2.19a);
- Inserts rules in the cross-connect tables that connect the virtual port to SONET signals (timeslots) that connect the two CDs bi-directionally (eg. in Fig. 2.19a, VPort3 is cross-connected to two VC4s – 150Mbps each – which start on timeslots 1 and 4 on ports P11 and P22 respectively). The cumulative circuit-bandwidth is then 300Mbps[^].

All subsequent packets (in both directions) for this client-server pair match the existing flow definitions and get directly forwarded in hardware. For other client-server pairs the application chooses a different VLAN tag, so the CD's packet-switch fabric can distinguish between packets coming in from the virtual-port and destined to different client/server pairs. As video data is received from the server, the packets are tagged with the internal VLAN id and mapped to the virtual-port. At the client side, the packets received from the virtual-port are switched to the client port based on the VLAN tag, which is then stripped off before the packets are forwarded to the client PCs, where the video is displayed on the screen. Packet flows are shown in the GUI (Fig. 2.20b) between the PCs and the CDs, and circuit flows are shown between the CDs.

Congestion-aware Bandwidth-on-Demand: Initially, the cumulative data-rate of two video streams is less that the bandwidth of the circuit flow they are multiplexed into (Fig. 2.20b), and the videos play smoothly on the client PC displays. However, when a third

[^] The virtual-port is created using SONET Virtual Concatenation (VCAT) and Link Capacity Adjustment Scheme (LCAS) features [5].

video stream is multiplexed into the same circuit-flow, the bandwidth is exceeded, packets start getting dropped, congestion develops in the network, and the video displays stall. However our application monitors network performance by acquiring circuit-flow statistics from the virtual ports in the CDs. It becomes aware of the packet drops, makes sure that the congestion is due to long-lived flows, and then responds by increasing the circuit bandwidth by adding more TDM signals to the virtual-port (Fig. 2.20c); thereby relieving congestion & restoring the video streams which start displaying smoothly again.

Conclusions: From this second prototype, we arrived at the following conclusions:

- Validation of our common-flow abstraction; that it was possible to treat a hybrid packet-circuit switch as a combination of flow-tables that switch at different granularities (packet and time-slot) and can be commonly-controlled by an external controller using a common switch-API.
- Validation of the flexibility of virtual-ports as the mapping-functions between packet and circuit domains. Prototype #1 had the virtual-port represented by the physical GE-wavelength (SFP) convertor-ports in the packet switches, and Prototype #2 had the virtual port represented by a GE-TDM mapper in the hybrid switches.
- We learnt a number of lessons about how the API can be improved. For example, our TDM port bandwidth representation in v0.2 of the extensions (which we used in this prototype [35]) is cumbersome - and so in v0.3, we use a much simpler representation [36]. Another example relates to the use of a common structure to represent packet & circuit ports. In v3 we use separate structures, as it helps implementation. For example, vendors of packet-only-switches need not implement circuit-related features of the OpenFlow protocol (switch-API) and vice-versa.
- With the common-flow abstraction feasibly instantiated, we made our first foray into developing network-applications that worked across packets and circuits. But we did so without the important features of discovery and fault-detection, which help create the common-map and keep it updated. This was fixed in the next prototype.

2.3.3 Prototype #3: Full pac.c Prototype

The early prototypes discussed in the previous sections were useful as proof-of-concept demonstrations of our architectural constructs. They were also useful for demonstrating simple applications in the controller across packets and circuits. However they were limited in the following ways – the prototype in Sec. 2.3.1 had standalone packet switches and a wavelength switch, but it was only a single-link demonstration; the prototype in Section 2.3.2 was more involved but had limited packet switching capability (only VLAN); finally both prototypes lacked a more evolved implementation of the common-map abstraction as described in Sec. 2.2 (no discovery, recovery etc.) In this section and the next Chapter, we describe our evolved and complete pac.c prototype.

Goals: The initial goal of this prototype was to instantiate the common-map abstraction by creating and maintaining a common-map and network-API. Then building on the initial goal: we wished to create a full pac.c prototype network with standalone packet and circuit switches that emulated WAN *structure*; we wished to provide the choice of treating packet and circuits in different layers; and use different data-plane packet-flow to circuit-flow mappings than what had been previously demonstrated. Finally, the ultimate goal of this prototype was to enable the creation of a fairly involved network application across packets and circuits; and then using this experience to validate our architectural claims of simplicity and extensibility.

Data-Plane: In part, the data-plane consists of the same three Ciena CoreDirector (CD) hybrid packet-circuit switches from the previous prototype (Fig, 2.21). The OpenFlow client firmware inside the CDs was upgraded to support all the features of version 0.3 of the circuit switching extensions to the OpenFlow protocol [36]. Discovery and recovery features were added and other deficiencies (mentioned in previous section's conclusions) were corrected in this version of the extensions. The CD firmware was also upgraded to support version1.0 of the packet-switching part of the OpenFlow protocol (previous prototype was based on v0.8.9).

In addition to the hybrid switches, in this prototype we added standalone packet-switches with full capability to support packet-switching based on the OpenFlow spec v1.0 [28] (Fig.2.21). We used a single 48 port GE switch from Pronto (earlier Quanta [48]) supporting the Indigo firmware [49]. The single switch was 'sliced' to behave like *seven independent* packet-switches each with 6 GE ports. The slicing plane was based on a modified version of the FlowVisor [50]. The basic idea of the FlowVisior is that a switch can be sliced to behave like multiple switches with each slice of the switch under the control of a different controller. But if we give the control of each slice to the same controller, then the sliced switch appears to that controller as multiple switches.

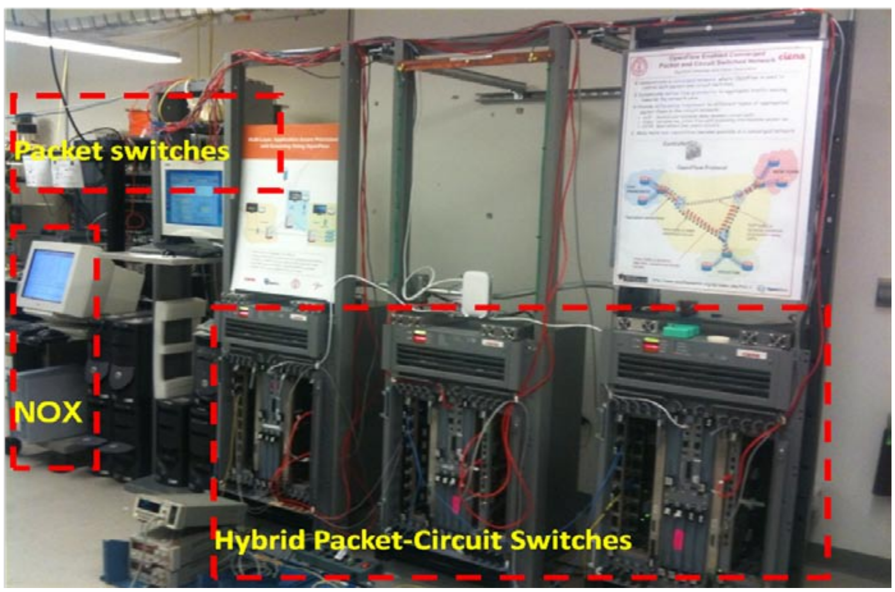


Figure 2.21: Full pac.c Prototype

Control-Plane: In this prototype we focused initially on developing the common-map abstraction. Similar to Prototype #2 we again used NOX as our controller. NOX was written mainly for enterprise networks, which have host-machines and middleware directly connected to the network†. In our work, to develop the common-map abstraction for carrier networks, we ignored most of the LAN-related-functionality that NOX provides (eg. authenticator, host-namespace, DHCP/DNS modules etc.).

† The NOX paper [15] briefly discusses the internal architecture of the network-OS.

The parts of NOX we did use are the basic event-engine and connection handlers that are collectively referred to as nox-core (similar to Prototype #2). Additionally some modules we used, that come built-in [51], include packet-link discovery and packet-topology, together with library-support and GUI API's (Fig. 2.22). Network applications (shown above the horizontal dashed-line) such as routing, can use these modules to implement network-functions; together the built-in modules create a network map for packet networks and keep it updated with network state.

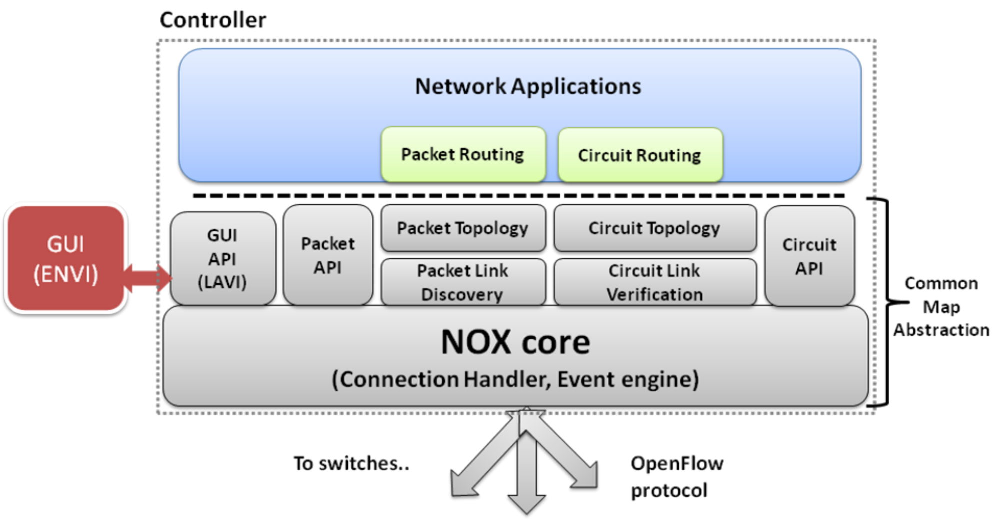


Figure 2.22: Common-Map Abstraction instantiated in NOX

In NOX-core we changed the definition file for the OpenFlow protocol, in order to include the changes we made to the protocol to support the common-flow abstraction (Sec. 2.1). The switch-features message allows us to build up the map representation for nodes, ports, flow-tables, and queues. We discover physical packet-links via the NOX's discovery module. For reasons mentioned in Sec. 2.2.1, we have the TDM switches perform neighbor-discovery and report on their peers as part of switch-features. We added a circuit-link verification module (Fig. 2.22) to ensure that the peers reported by the switches on either end of a circuit-link matched-up, and only then were the links included in the topology.

At this point, both physical packet and circuit links had been discovered. However we wished to enable the treatment of packet and circuit switches as part of different *layers* (as discussed with respect to Fig. 2.15). The reason relates to our ultimate goal of using this prototype to validate our architectural claims. In order to do that we need to compare our work with the way industry 'sees' these networks today – as part of different layers. And so we followed the procedure briefly mentioned in Sec. 2.2.2 to create these layers with separate topologies. Importantly, since we already had all physical links, we needed a way to discover virtual-packet-links, which are supported by underlying circuits over the wide-area.

We detail the procedure followed in the context of NOX. We quote from [15] (with small clarifications): "To cope with network change events, NOX applications use a set of (event) handlers that are registered to execute whenever a particular event happens. Event handlers are executed in order of their *priority* (which is specified during event registration). A handlers' return value indicates to NOX whether to stop execution of this event, or to continue by passing the event along to the next registered handler."

Using NOX's event priorities, discovery of virtual packet-links can be performed as follows: Consider the situation where two standalone packet switches are connected over the wide area through multi-layer transport switches like the CDs.

- Since there are two discovery modules in NOX, we give higher priority to the circuit-discovery module to received discovery events. The module can ignore or pass-on the discovery packet to the packet-discovery module (depending on the layering choice).
- The discovery-packet sent out through a standalone packet-switch is reported back to NOX by the packet part of the CD. The circuit discovery module receives it and discovers the physical packet link connecting the standalone packet-switch to the CD. But now it chooses to stop the execution of the event. So the packet-discovery module never learns about this physical packet-link.
- The circuit discovery module is already aware of the circuit-links via the CDs SONET discovery mechanisms. It then has all the information it needs to complete

the circuit topology – the CDs, the circuit links that connect them, the packet switches that are physically connected to the CDs and the physical-packet links being used for those connections.

- The controller then enters rules in the packet-switch part of the CD to tunnel all subsequent discovery test-packets from the packet-switch through the circuit-flow to the packet-switch at the other end of the circuit flow.
- At the other end of the circuit-flow, the discovery packets are reported back to the NOX by the other packet-switch; this time the test packet is *ignored* and passed on by the circuit-link discovery module; and received by the packet discovery module. The latter can therefore receive this message and determine the (virtual) packet-link, which inter-connects the standalone packet-switch interfaces via the circuit-flow over the wide area.

In this way the packet, the packet topology (i.e the packet-layer) can be completed with standalone packet switches and physical as well as virtual packet links, and kept separate from the circuit-layer (which includes the CDs and circuit-links).

We kept the common-map updated via two different mechanisms. The packet-layer uses the packet-discovery process as keep-alives to discover link-downs. The circuit-layer uses SONET's ability to quickly discovery link-down and report it to the controller as a port-status message; at which point the circuit-link goes down in the circuit topology, and the virtual-link it supports in the packet-topology may also go down (depending on the type of recovery mechanism selected[†]). Finally we also implemented the beginnings of a network-API (including routing and recovery from Sec. 2.2.3) to manipulate the common-map.

Experimental Setup: With this prototype we had most of the elements necessary to *emulate* a wide-area network. Importantly, we emulated WAN *structure*[*] i.e. packet-switches (access and core routers) clustered in a city's Point-of-Presence (PoP); with the core routers connected over the wide area (inter-city) by optical transport equipment.

† For example if all recovery is deemed to be handled by the circuit-layer, then the virtual-packet-link does not go down. It simply gets supported by a different recovery-circuit.

* as opposed to WAN latencies that come from propagation over thousands of miles in the wide-area. All our switches are housed in the same lab with only a few meters of fiber connecting them.

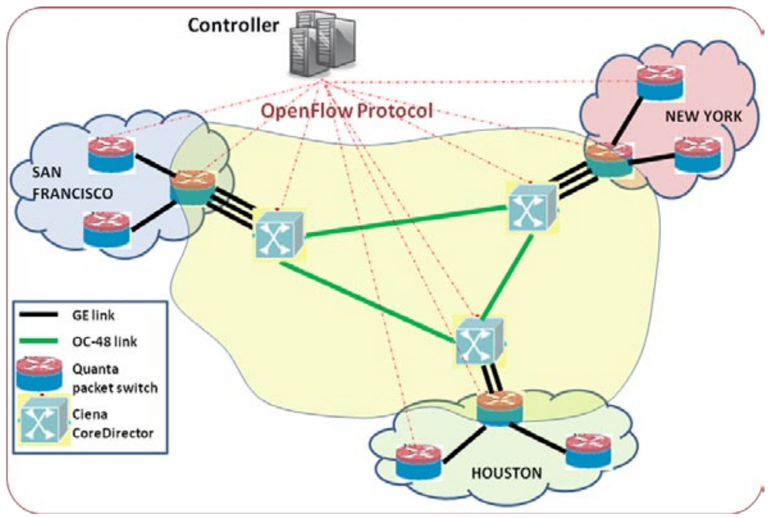


Figure 2.23: Prototype #3: WAN Emulation

The links shown *within* the cities in Fig. 2.23 correspond to physical GE links between the independent Quanta packet switches; and also between the packet switches and the CoreDirector (CD)'s Ethernet ports. For example, in the San Francisco (SFO) cloud, we see two (access) switches connected by GE links to a single core switch. The SFO core switch in turn is connected to the SFO CD via GE links. The CDs in turn are connected by OC-48 optical-fiber-links over the wide-area. As before, all the switches are OpenFlow enabled and communicate with the controller over an out-of-band Ethernet network (non-OpenFlow controlled).

Experiment: We have implemented a fairly complex network-function in a network-application on our controller, that makes use of both packets and circuits at the same time, to achieve the functionality. We defer the discussion of the entire application, and the conclusions we drew from the implementation, to the next chapter. Here we demonstrate some of the underlying flow-table manipulations our application drives with special emphasis on how packet-flows get mapped into different circuit flows. With this experiment we show how we can write an application that simultaneously controls flows on the basis of identifiers that belong to Layer 4 (TCP/UDP), L3 (IP), L2 (VLAN) and L1 (SONET).

Fig. 2.24 presents a snapshot of the forwarding tables in the SFO core packet-switch and the SFO transport-switch (CD) physically connected to it. The objective of this experiment is to show that packet-flows for different *types* of traffic (voice, video and web) can be identified and *mapped* into different circuit-flows.

In this snapshot, rules have been inserted into the SFO core packet-switch's flow-table that differentiate between three different types of traffic from the same customer – in other words, three different packet-flows have been created. The customer is identified by the IP source-address range of 10.44.64.0/18. The different traffic-types are differentiated on the basis of well-known (eg, HTTP traffic on TCP port 80) or registered (eg. VLC media player RTP/UDP stream) transport port numbers [55]. Thus each rule is a combination of the IP src-address (L3) and a different TCP/UDP destination port (L4).

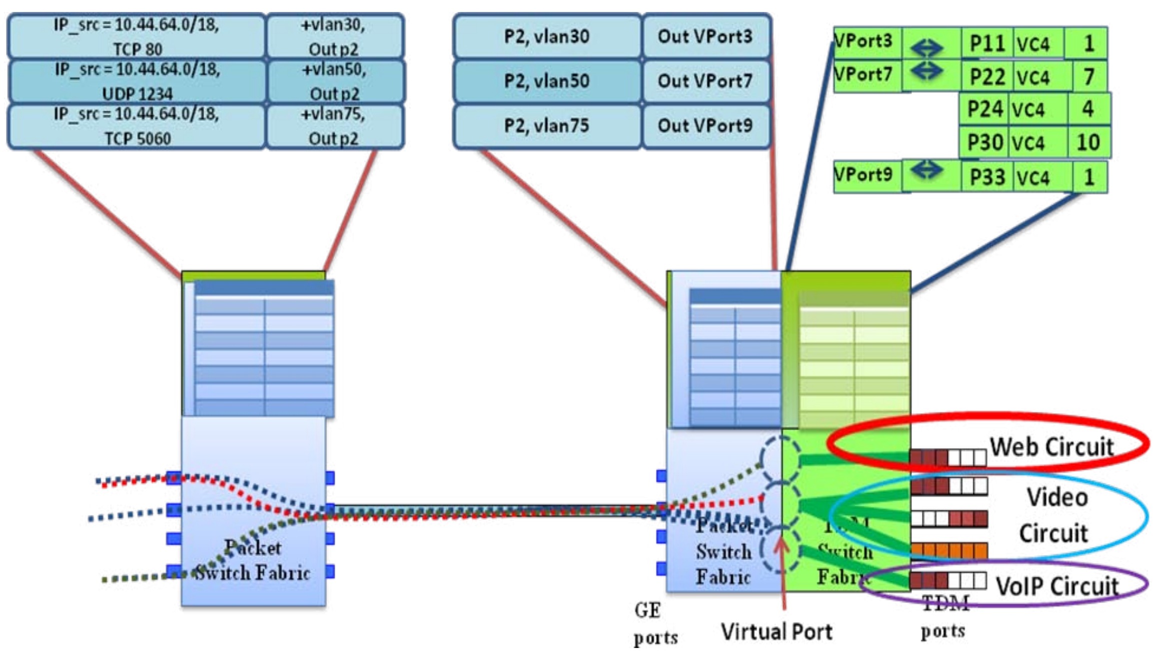


Figure 2.24: Programming the Flow-Tables

The basic idea is to map the three different packet-flows into three independent circuit-flows. One way to achieve this could be to use three different physical-ports between the core-packet switch and the CD. But this is an expensive approach, especially since the application learns from the common-map that the transport switch's (the CDs) packet-forwarding table is capable of matching on <input-port, VLAN tags>; and is therefore capable of demultiplexing packet-flows coming in from the *same* physical port on the basis of different VLAN-ids. Thus the application performs the following actions on all packets that match the rules in the core-packet-switch. It adds VLAN tags (L2) with different VLAN ids (30, 50 and 75) and forwards the flows out of a *single* port (port#2) on the core-packet-switch†.

The application also adds rules in the CD's packet-flow-table to match on incoming-port P2 and the different VLAN ids, with the action defined to forward all packets that match the rules to different *virtual ports* (end-points of circuit flows). For example, HTTP packets which had VLAN30 inserted into them by the core-packet switch, match the rule in the CD with the corresponding action that forwards them out of virtual-port 3. The virtual ports have been created beforehand by the application and they contain the mapping actions necessary to map Ethernet frames to SONET frames.

Finally the snapshot shows that the application has inserted rules that cross-connect the virtual-ports to time-slots on the physical SONET ports. For example, HTTP traffic (VPort3) is assigned 150 Mbps of bandwidth (VC4) on SONET port P11 (starting at time-slot1), while the video-traffic aggregate (VPort 7) gets 450 Mbps bandwidth (three VC4s) spread over 3 different SONET ports (using VCAT).

In this way packet-flows for three different traffic types from the same customer have been mapped into separate circuit-flows with different bandwidths. The application has the power to dynamically change any of these mappings as it sees fit. For example:

- The packet-flows could be bundled into the same circuit-flow by changing the VLAN ids in the core-packet-switch. Or by forwarding to the same virtual-port in the CD.
- Packet-flows from different customers can be mapped into the same circuit-flow by matching on the different customer IP-src address but adding the same VLAN tag.

† Note that if the packet-flow-table in the CD had full matching capabilities, we wouldn't need the VLAN tag insertion. The same flow-table rules based on IPsrc/TCP/UDP could have been used in the transport switch as well. However the use of a tag/label has another advantage. In a flow-table implemented with TCAMs, rules based just on tags or labels take less space.

- Bandwidth allocations for the circuit-flows can be changed on the fly by cross-connecting the virtual-ports to more (or less) time-slots. Time-slots can be re-assigned by cross-connecting them to other virtual ports.
- New circuit-flows can be created by adding new virtual-ports, cross-connecting them to time-slots, and re-directing packet-flows by forwarding them to the newly created virtual port

Conclusions: To summarize, we draw the following conclusions:

- We achieved our initial goal of instantiating the common-map abstraction, by using features built-in to NOX for packet-switching, and by adding a set of modules for discovery, topology and recovery for circuit-switching. Compare Fig. 2.22 to the usage of NOX in Prototype #2 (Fig. 2.19b).
- We were able to create the choice of viewing packet-switches and circuit-switches in different layers using an innovative procedure for discovering virtual-packet-links. In the next Chapter we will explore an application that treats the switches as different layers but still under common-control.
- In support of our larger goals to converge packet-and-circuit network operation in WANs, this prototype was sufficiently evolved to emulate WAN structure (unlike the previous two prototypes).
- Finally we were able to demonstrate control over a wide variety of data-plane flow-identifiers (L1-L4) and mappings that showed the power of this approach as a multi-layer control plane.

Prototype #3 implements most of the features of our architectural approach – the common-flow and common-map abstractions via a unified-control plane. It therefore gives us an opportunity to compare and contrast our work, to industry solutions for common-operation of packet and circuit switched networks. In the next chapter, we validate the simplicity and extensibility claims we made for our unified-control-architecture in Ch. 1 by comparing our work these industry solutions.

2.4 Summary

In this chapter, we gave details of the design of the two abstractions we introduced in Ch.1, as part of our unified-control architecture for packet and circuit network control.

We first discussed the common-flow abstraction. We showed what the flow-abstraction means in the context of packet-switching, and then showed how it can apply equally well to circuit-switching and different kinds of circuit switches. We then gave design details of a common switch-API to manipulate the data-abstraction of flow-tables in both packet and circuit switches. Importantly, we showed how the data-plane construct of a virtual-port can flexibly map packet and circuit flows to each other.

Next, we gave design-details of the common-map abstraction. We briefly discussed the representation of the common-map and how it can be built and maintained. We detailed the discovery of packet and circuit links which is an important part of the puzzle when building a common-map. And we outlined procedures via which the common-map can be represented as single or dual layers/topologies. Finally we discussed a set of network-API calls that are needed to manipulate the common-map.

We then delved into instantiations of the common-flow and common-map abstractions in the OpenFlow protocol and a network-operating-system called NOX. We presented three prototypes that helped us progressively validate our design constructs. Two early prototypes focused on the common-flow abstraction for packet-switches working with different kinds of circuit-switches (wavelength and time-slot based). The third, more complete prototype focused on the common-map abstraction and building a testbed that emulates WAN structure. We will use the latter in Ch. 3 to validate the simplicity and extensibility claims of our control architecture.