Programming Recursive Software-Defined Networks

Jonathan Frankle '14

Abstract

Over the past few years, the networks community has devoted a tremendous amount of energy to the area of software-defined networking (SDN), where traditional control-planes are supplanted by a single, centralized controller that supplies packet-handling decisions to the network. Some researchers have proposed more complex distributed systems of controllers to compensate for some of the shortcomings of this single-controller approach. One concept, called recursive software-defined networking (rSDN), arranges controllers in the shape of the tree, with controllers at higher levels responsible for managing their children. Programming languages researchers have rushed to develop programming languages for single-controller SDN topologies, but no such model exists for rSDN. This paper describes a set of primitives inspired by concepts from functional-programming for writing network algorithms in rSDN and explores the underlying incremental data-model upon which these primitives rely. The result is a network model that distributes computation evenly across all controllers while minimizing network traffic and recomputation when network conditions change.

1. Introduction

In recent years, *software-defined networking* (SDN) has become a tremendously popular topic in both academia and industry. Researchers from a variety of fields, from networks to programming languages, have flocked to the rich array of new challenges that the concept introduces. Many prominent players in industry, most notably Google, have disposed of their older networks

of proprietary switches running proprietary firmware in favor of the enticing programmability that SDN promises. In a software-defined network, a single, central *controller* makes packet-handling decisions for *switches*, which form the fabric of the network itself. These controllers are end-user programmable, meaning that researchers and network administrators can experiment with and tailor the behavior of networks to fit their individual needs. The lavish research attention devoted to SDN has hardly been limited to networks experts; the programming languages community has eagerly grappled with the challenge of codifying this programmability into usable languages. SDN controllers communicate in a low-level dialect called OpenFlow, which can be aptly compared to assembly language in terms of its ease of use. A variety of high-level languages have been developed to make programming SDN more convenient.

These single-controller networks are hardly the end of the story for SDN research. The computational and geographic burdens inherent in connecting every switch in massive networks to a single controller have led researchers to propose a variety of alternate, multi-controller topologies. Specifically, one concept, known as *recursive software-defined networking* (rSDN) [2], describes an architecture with a tree of controllers sharing responsibility for administering sub-networks. A root controller might have multiple child controllers, each of which would recursively manage other controllers until the lowest-level controllers administer switches directly. From a systems perspective, this idea resolves many of the inherent flaws of single-controller SDN topologies. The key benefit of using SDN, however, is the ability to easily write programs for networks. By widening the set of permissible network topologies, rSDN dispenses with some of the central assumptions underlying many existing SDN programming paradigms while adding new constraints.

In this paper, I propose a programming model designed specifically for the goals and limitations of recursive software-defined networks (Section 3). This model is rooted in the principles

of functional programming in both their traditional form and a newer reimagining known as incremental computation. To demonstrate the feasibility and usability of this paradigm, I have developed two software models of a recursive software-defined network, one in F# (Section 5) and one in C# (Section 6), and implemented several fundamental network tasks that form the basis for addressing, routing, and packet forwarding (Section 4). I aim to leverage the potent combination of recursion, abstraction, and an incremental data model to develop a programming model for rSDN that makes it possible to:

1. Manage large networks in a high-level, portable manner.
2. Distribute computational work evenly across all controllers.
3. Abstract subnetworks, reducing the latency for response to network events.
4. Respond to network with the minimum necessary communication and computation.
5. Develop scalable network algorithms that run efficiently over wide geographic distances.

2. Background

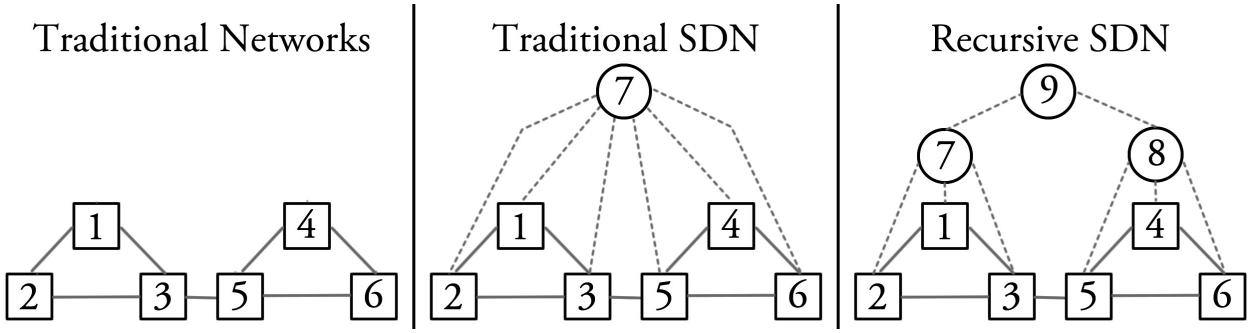

Figure 1: Kinds of networks described in this section. Switches are squares; circles, controllers; solid lines, data links; dashed lines, control links. Numbers represent each entity's unique identifier.

2.1. Traditional Networks

The fundamental purpose of any computer network is to transmit data between end-hosts in the most efficient (bandwidth, latency etc.) manner possible. In a traditional network, a collection of interconnected *switches* forms this end-to-end connection. An *end-host* is connected to one or more switches, which pass on the host's traffic to other switches in a chain that eventually terminates with a final switch forwarding messages to a recipient host.

Each switch consists of two components, which together allow it to forward packets to the proper destinations. A *control plane* examines all information available to a switch, including routing metadata shared among switches and the sources of packets that the switch receives. It uses this data to make *routing* decisions – those about where to send packets destined for various recipients. These decisions are installed an implemented in the switch's *data plane* in the form of forwarding rules. The data plane is responsible for sending the packets themselves to the proper destinations.

Although flexible and extensible, this decentralized configuration is not all that efficient. Switches organize themselves into a spanning tree, which rarely equates to shortest-paths routing and fails to utilize every available link. Each switch must monitor packet traffic to learn the location of end-hosts, prior to which it simply broadcasts all packets, inducing a significant quantity of unnecessary traffic. These distributed networks are slow to detect and respond to failure, potentially forwarding many packets into a "black hole" in the interim. Finally – and most critically for this paper – these switches run hard-coded, proprietary firmware installed by their manufacturers. An enterprising network manager who wishes to modify the routing algorithm will see his or her efforts thwarted by a network of locked-down switches.

2.2. Software-Defined Networking

In response to the challenges and frustrations of traditional networking, a large body of research has developed around the topic of *software-defined networking* (SDN). SDN introduces a new network entity, the *controller*. This central unit provides a single, shared control plane for the entire network to which every switch is connected. Switches are reduced to data planes, receiving all rule installations from the network's controller. In contrast to traditional networks described in the previous section, the controllers themselves are typically fully programmable general-purpose computers. This fundamental premise of SDN – that we can write our own network algorithms and program our network hardware freely – is the key idea behind the entirely new area of research that SDN has spawned. Under these conditions, switches become general network "boxes," capable of filling the role of traditional switches, firewalls, or more exotic devices.

SDN, at least in this basic, single-controller configuration, is not without its own weaknesses. As networks grow in numerical size, the controller requires steadily more computational power to keep up with the load, a form of unsustainable vertical scaling. In addition, the geographic size of many networks makes linking every switch to a single controller inconvenient if not infeasible. In turn, an increasingly distant and overworked controller responds to network events with greater latency, reducing the network's resiliency to failure.

2.3. Recursive Software-Defined Networking

In response to these deficiencies, researchers have proposed a variety of multi-controller SDN topologies. Specifically, McCauley et al.[2] describe a distributed system of controllers arranged in a tree. They refer to this topology as a *recursive software-defined network* (rSDN). At

the lowest level, each switch might have its own controller. Groups of these switch-local controllers would contain further control links to a "regional" controller. These regional controllers would respond to still-higher controllers, forming a recursive tree of controllers culminating in a single root. With carefully-written algorithms, computational work could be divided evenly to limit the burden on any one controller. As the numerical and geographic scale of a network increases, the quantity and span controllers would grow linearly to accommodate this expansion. These regional controllers would be able to respond quickly to network events, keeping latency low even in massive networks. They could even abstract entire subnetworks into virtual switches, hiding their inner complexity from higher levels to seamlessly respond to failures without notifying the entire network.

More broadly, this hierarchical approach closely mirrors the actual organization of large, production networks. A modern datacenter, for example, consists of small networks of racks that are aggregated at higher and higher levels until all nodes can access one another. These datacenters are then connected together by a backbone network. Each of these sub-networks forms the appropriate domain for a higher-level controller in an rSDN topology.

3. Functional Primitives for Recursive Software-Defined Networks

3.1. Overview

This section details a set of functional primitives that, together, comprise a programming model for writing network algorithms for rSDN topologies. These primitives are inspired by traditional higher-order functional combinators, including map (Subsection 3.2), reduce (Subsection 3.3), and a hybrid of the two (Subsection 3.4). All of these operations can be described as specializations of a recursive fold over the network (Subsection 3.5). In this programming model,

we will represent each switch and controller as kinds of a subtype *box*, which, for our purposes, is simply a key-value store with named *properties* that comprise its state. The fundamental difference between switches and controllers is that the latter has *children*, boxes connected via *control links*, while the former has *neighbors*, other switches connected via *data links*. With the exception of the root, all boxes have a control link to a *parent*. Each switch has a series of *ports*, identified by switch-unique integers, into which data-links can be attached. For each of its ports, a switch knows the identity and port of the connected neighbor and the cost of the attached link.

For the purposes of this programming model, both data links and control links can carry control traffic and controllers must have either controller or switch children but not both. A primitive may be invoked at any controller in the network and is applied recursively from parent to child until it is installed at all boxes that descend from the invoker.

3.2. Mapping Primitives

In a functional *map* operation, a function of type `'a → 'b` is applied to each value in a collection of items with type `'a`, resulting in a collection of items with type `'b`. The distinguishing quality of a map operation is that it statelessly applies the same operation to every item, producing a one-to-one correspondence between items in its domain and range. In this programming model, the signature of this map operation is as below:

```
map (start : box) (f : box → box)
```

The function `f` is recursively applied to `start` and each of its descendants. It examines the initial state of the box and produces a "new" box updated as necessary. A more specific instance of map is the initialize operation, which sets a property in every box to a particular value:

```
initialize (start : box) (p : property) (v : value)
```

3.3. Reducing Primitives

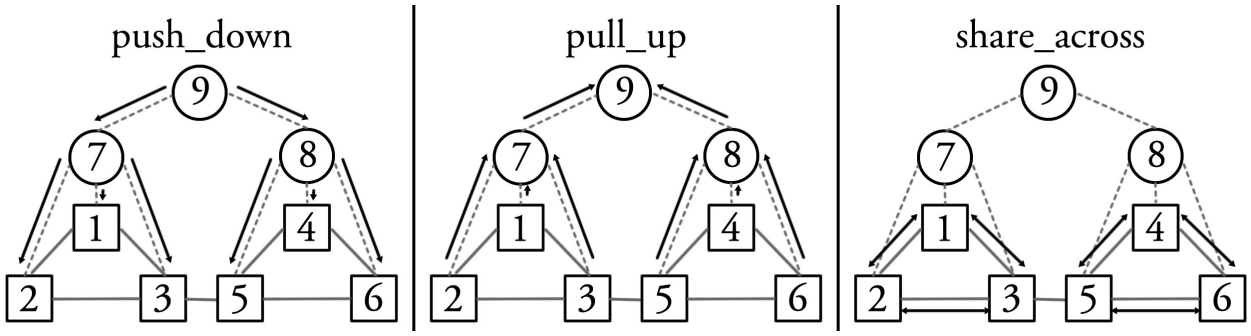

Figure 2: The flow of data in the reducing and hybrid primitives.

In a functional reduce operation, a function of type `'a → 'b → 'a` is applied to each value in a collection of items with type `'b` in sequence. Given a base value of type `'a`, the function is applied to the first value in the sequence, producing a new base of type `'a` that serves as the base for the next iteration. The return value of a reduce is the value of the base after every item in the collection has been visited. A simple example is taking the sum of a list of integers by reducing with the (+) function and a base of 0. Unlike maps, reduces are stateful, with the value of the base after an iteration dependent on its value beforehand. In the context of rSDN, the statefulness of reduce operations provides a convenient way to link together the properties of connected boxes. Within the confines of the tree-like structure of rSDN, reduces can be applied in two directions: in a top-down fashion from an ancestor controller to descendent switches (push_down) or the reverse (pull_up):

```
push_down (start :  box) (f :  base     → box → box * base) (b :  base)
pull_up   (end   :  box) (f :  base set → box → box * base) (b :  base)
```

In a push_down operation, `f` receives a base and a box, producing an updated value of the box and a new base to be used for the next iteration. `f` is initially applied to `start` and `b`, updating `start` and creating a new value that serves as the base for the push_down operation as applied

to `start`'s children. This operation terminates when it reaches the switch level, at which point the final bases are discarded. A push_down operation links the state of a box to the state of its parent, which is convenient for topology discovery and the propagation of routing decisions from higher-level controllers down to switches.

A pull_up operation works in reverse. `f` is first applied to every switch that descends from `end`, provided with the switch and a singleton set consisting of `b`. It produces an updated version of the switch and a new base. This operation is repeated at the next level up, where the base set for `f` consists of each of the bases produced by the controller's children. The recursion terminates at `end`, with the final base discarded. Much like a push_down operation, a pull_up links the state of a parent box to that of its children, which is useful for gathering the necessary information to make routing decisions.

Although these two primitives are able to support a variety of important network tasks, they are insufficient to support cases where it is convenient for a parent to handle information differently depending on the child with which it is interacting. For example, it would be desirable for a parent using a push_down operation to install routing decisions to be able to send different information to each of its children. Although it could hypothetically send all routing decisions to all children and force each child to determine rules applicable to itself, this method hardly makes efficient use of network resources. Instead, we can provide alternate, asymmetric versions of pull_up and push_down called push_down_asym and pull_up_asym with slightly modified signatures of `f`.

```
push_down_asym (start : box) (f : base → box → box * dict) (b : base)
pull_up_asym   (end   : box) (f : dict → box → box * base) (b : base)
```

In push_down_asym, `f` returns a box and a dictionary from the box identifiers of its children to bases. This dictionary is automatically demultiplexed, sending each child its corresponding base

as specified by the dictionary. Likewise, `f` in pull_up_asym receives a dictionary from the box identifiers of its children to the bases they supply rather than an anonymous collection of bases as in push_down.

3.4. Hybrid Primitives

The final piece of the programming model is the ability to link a switch's state to that of its neighbors. This functionality is critical for allowing switches to compare ancestors, an operation central to several algorithms described in Section 4. This functionality is implemented in a single primitive:

```
share_across (start : box) (pull : box → base)
             (aggregate : base set → box → box)
```

In a share_across operation, all switches that descend from `start` apply `pull` to each of their neighbors, producing a set of bases. These bases provide the input for the `aggregate` function, which each switch applies to itself. This operation can be viewed as both a map and a reduce; the `pull` phase statelessly applies the same operation on each neighbor, but the `aggregate` function collects this information into a single result.

3.5. The Fold Primitive

All of the previous, high-level primitives can be expressed as specializations of a single function: a recursive, two-way reduce over a tree of boxes:

```
fold (start : box) (next : 'c → box → box set)
(down : 'a → box → box * 'c) (demux : 'c → identifier → 'a)
(up : dict → box → box * 'b) (down_base : 'a) (up_base : 'b)
```

Fold requires four higher-order functions and seven total arguments. They are:

start: The box from which the operation is initiated.

next: A function that takes the current box and the output from its downward recursion step, providing the set of boxes to visit next.

down: Given a multiplexed downward base and the identifier of a box, produces the unique base corresponding the specified box.

demux: The function used for the downward recursion step. It takes a downward base (type `'a`) and a box, producing an updated box and a multiplexed downward base (type `'c`).

up: The function for the upward recursion step. It takes a box and bases from the previous upward recursion step in the form of a dictionary from box identifier to upward base (type `'b`), producing an updated version of the box and a new upward base for the next step in the recursion.

down_base: The initial base for the downward recursion.

up_base: The initial base for the upward recursion.

Fold works as follows:

1. The downward recursion begins at `start` with a downward base of `down_base`.
2. `down` is applied to current box and downward base, producing an updated version of the current box and a multiplexed downward base.
3. `next` is applied to the current box and the multiplexed downward base produced in step 2. This produces a set of boxes on which down will be applied in the next recursive iteration.
4. Steps 2-3 are applied recursively to the boxes indicated in step 3 until `next` returns an empty set. The base for each subsequent recursive step is determined by applying `demux` to the multiplexed downward base produced in step 2 and the identifier of the box to which `down` will be applied.
5. The upward recursion begins at each box at which the downward recursion terminated. The

initial base is a singleton dictionary consisting of `up_base`.

6. `up` is applied to the current box and dictionary of upward bases, producing an updated version of the box and a new upward base.
7. This upward base is passed to the box's predecessor in the downward recursion, providing an entry in its predecessor's dictionary of upward bases.
8. Steps 6-7 are repeated until they are applied to `start`.

We can express each of the preceding seven primitives using this fold function. For each of the primitives, the `next` function simply returns the children of the provided box. The `demux` function is the identity function for all primitives except for pull_up_asym and push_down_asym, in which it does a dictionary lookup instead. pull_up uses the identity function in its downward recursion step; push_down and share_across do the same for upward recursion. map and initialize ignore all bases, obviating any statefulness.

The importance of fold is its ability to generate new primitives. The flexibility of the `next` function, for example, opens the possibilities of breadth-first-search across switches in the network or selectively cutting off recursion as an optimization to reduce network utilization and recomputation. Other operations could be developed to link the state of a single switch to its predecessors or to initiate operations within other operations. The set of primitives in this section are effective for general-purpose operations, but they certainly do not comprise a complete set of all possible operations on a tree.

4. An rSDN Network Model

4.1. Topology Discovery

A recursive software-defined network is organized into subnetworks of switches represented by the descendants of controllers. The network as a whole can be viewed as the recursive composition of subnetworks into larger and larger units until finally the root controller's domain encompasses the entire network. For many of the operations that follow, it is essential that a switch know the subnetworks of which it is a member, each of which is uniquely identified by the identifier of the subnetwork's root. We define a box's *hierarchy* as a list of its ancestors' identifiers from nearest to furthest. We can use a push_down operation to allow each switch to learn its hierarchy:

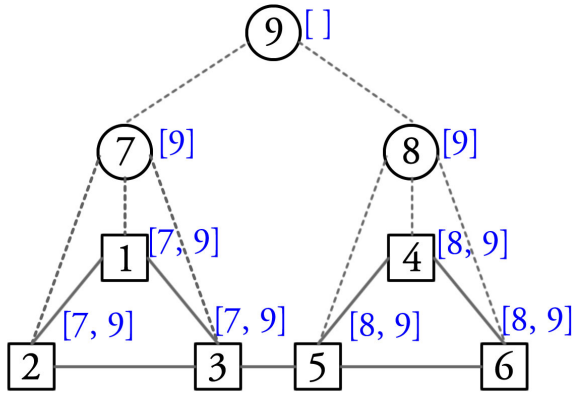

Figure 3: An example rSDN topology. Each box's hierarchy is noted in blue.

```
let root = {network root} in
let f base box = box.hierarchy := base; (box, box.id ::  base) in
push_down root f []
```

This push_down operation starts at the network root with a base of the empty list. Each box sets its hierarchy to the base provided by its parent before prepending its unique identifier to its hierarchy and passing this value down as the base for its children. After this operation is complete, each box will have the complete list of its subnetwork membership ordered from smallest to largest.

Each controller must also learn some basic information about its topology, namely the membership of its subnetwork as a set of switch identifiers. This process can be accomplished in a similar pull_up_asym operation. Initially, switches pass their parents their identifiers. Each controller stores the sets of its children's descendants indexed by child, passing to its parent the

union of these sets. Thus, each controller knows both its children and its grandchildren, a property that will also prove useful for routing.

4.2. Addressing

We now need to consider a third kind of network entity – an *end-host*. Although switches form the fabric of the network, they are not typically the origins or destinations of packets. Instead, end-hosts connected to switches send packets to one another, using the network merely to facilitate this communication. In rSDN, we can think of each switch as a name server, able to assign end-hosts leases on network names. Each name takes the form of a globally-unique list, structured as below:

```
{switch port number} :: {switch unique identifier} :: {switch hierarchy}
```

For example, if switch 3's ancestors are controllers 2 and 1 and an end-host is connected to its fifth port, the host's name would be [5, 3, 2, 1]. Although each end-host could be assigned a globally unique name solely by the switch identifier and the port number to which it is connected, the above scheme embeds in each name the host's subnetwork membership. As we will see later, giving names this quality makes packet forwarding significantly easier and routing far more efficient.

4.3. Virtual Links

The key insight for performing distributed routing is that a recursive software-defined network is really a series of composed, interconnected subnetworks. As noted before, each controller has a particular subnetwork comprising its descendant switches. These switches have internal links to other switches in the subnetwork and external links connecting to other subnetworks. If we imagine a world where controllers can be linked to one another, we can express *virtual links* as these connections between subnetworks – namely we create a virtual link between the *furthest*

unshared ancestor of two switches for every physical link between them. This action converts the network from a graph of interconnected switches into a multi-level hierarchy of networks that, when flattened, returns to its original form. By re-envisioning the network in this fashion, we also divide the computational work necessary to performing routing and the administrative responsibility for network links.

Establishing virtual ports is a relatively straightforward operation using the primitives from Section 3. First, we perform topology discovery as discussed in Subsection 3.1. In a share_across operation, each switch then learns the hierarchies of its neighbors. Using this information, it determines the furthest unshared ancestor with each of its neighbors:

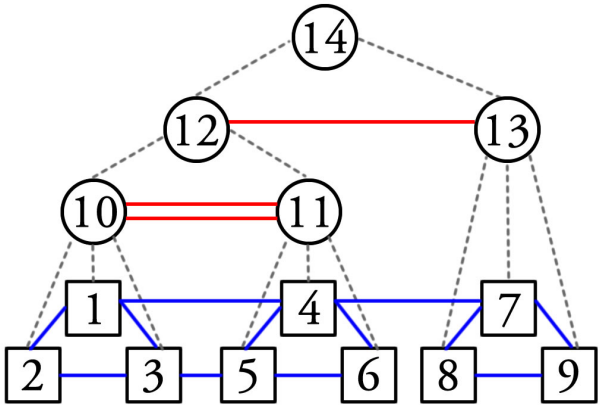

Figure 4: An example rSDN topology with physical links in blue and virtual links in red.

when comparing the reversed hierarchies of two switches, the first pair of controllers that are not identical are their furthest unshared ancestors. For example, suppose switches 1 and 2 have unshared parents 3 and 4 respectively and a shared grandparent 5. The network itself is rooted at controller 8, meaning that 1's hierarchy is [3, 5, 8] and 2's is [4, 5, 8]. By examining their hierarchies in reverse, we find that 8 and 5 are shared but 3 and 4 are not, making 3 and 4 their furthest unshared ancestors. Using this information, each switch adds an entry in a list of *requested virtual ports* with the pair of unshared ancestors, its port number and identifier, its neighbor's port number and identifier, and the cost of the link.

Using a subsequent pull_up operation, these requested virtual ports are propagated up the network. When a controller sees a request directed at itself, it creates a new virtual port mapped to

the physical port of its descendant switch. In another pull_up operation, each controller informs its parent about these *virtual port mappings*. Finally, a push_down operation allows each controller to learn the virtual port mappings of its siblings, revealing the virtual destinations to which their virtual ports are connected, completing the process of establishing virtual links.

4.4. Routing Overview

Using these virtual links, we can perform routing over the network. In conventional routing, each switch would learn the outgoing port from which it should forward all traffic destined for each end-host. If it lacks an entry for a destination, it should simply broadcast this packet until it learns the end-hosts's location. Leveraging the global network picture and more nuanced addressing system we have established for rSDN, we can do better. At each level in the network, a controller will receive routing decisions from its parent, run Dijkstra's algorithm over the virtual network comprising its children, and coalesce these decisions into a forwarding table to be shared with its children. Rather than learn to route to end-hosts, however, each switch will forward packets to subnetworks, namely to the *virtual neighbors* of itself and its ancestors. These subnetworks are non-overlapping and contain every switch in the network, meaning that, even with this added abstraction, every switch in the network can forward packets to every destination.

Combined with the address system described in Section 4.2, this method guarantees that switches always know how to forward to end-hosts, with no learning or broadcasting required. When it receives a packet, a switch will examine the entries in the destination address in order of highest scope (reverse order). When it finds a matching forwarding entry, it sends the packet out the specified port. This form of forwarding will gradually move each packet into scopes nearer and nearer to its destination until it finally arrives at a switch whose hierarchy is a prefix of the

destination address. The switch may then examine the port number in the destination address and forward the packet to the appropriate end-host.

There are several advantages to this form of abstracted routing. The simplest benefit is a significant reduction in the number of forwarding entries that a switch would need to store. In a traditional network, a switch must store an entry for each end-host. Even in a more naïve implementation of this routing algorithm, a switch would need state for every other switch in the network. With this hierarchical approach, a switch only requires knowledge of address "neighborhoods," greatly reducing the amount of state it needs to store. In addition, knowledge of network disturbances can remain confined to the smallest containing subnetwork where changes have occurred. Switches and controllers outside of a subnetwork see it as a single forwarding destination and are unconcerned with its inner workings; when its internal circumstances change, only boxes within the subnetwork need to be notified, producing far less routing entropy than if boxes had knowledge of all switches or end-hosts.

That is not to say that this routing method is without drawbacks. In the process of distributing computation evenly over the network, it sacrifices providing globally shortest-paths routing. Each controller only sees the subnetworks comprising its children, meaning that it may make routing decisions that make sense if each of its children were a switch but are inefficient given that there is a non-zero cost for sending packets through a subnetwork. Certain unlucky switches may possess high-bandwidth direct links to other networks that go unused, meaning they send packets to neighboring destinations through longer, less efficient paths. In addition, since each subnetwork will be connected to others by a single link, some number of switches must inherently lose connectivity with one another if any subnetwork becomes partitioned. Many of these disadvantages can be softened or alleviated entirely by more nuanced sharing of path information

that is beyond the scope of this paper.

4.5. Routing Mechanics

Once virtual ports have been established, routing amounts to repeatedly performing Dijkstra's algorithm with complicated bookkeeping. After each box pulls up the virtual ports of its children and converts them into a graph, the routing process is accomplished in a single push_down_asym operation. As a base, each box receives two items: a forwarding table for the virtual network it occupies and a list of "external destinations" with the switch and port numbers to reach them. These external destinations are the virtual neighbors of a box's ancestors to which some switch among the box's descendants is directly connected; the virtual ports used to reach them have been translated into physical ports. The routing operation works as follows:

1. The box runs Dijkstra's algorithm on the virtual network of its children, compiling a preliminary, internal forwarding table for each of its children.
2. The box examines its own forwarding table and translates each of its entries into physical switch and port identifiers. For each entry, it uses knowledge of its children's descendants to determine which of its children "owns" the physical switch. It adds the switch, port, destination entry to the owner's list of external destinations. It subsequently adds an entry to all other children's forwarding tables for the destination using the same virtual port number that the child would use to forward to the owner. In the special case where the owner of the switch is the switch itself, the entry is added to the owner's forwarding table rather than its list of external destinations.
3. It performs the same process as in step 2 for its external destinations.

If the box is a switch, it merely installs the forwarding table it has received. Steps 2 and 3 ensure that no switch should ever receive any entries in its list of external destinations.

4.6. Broadcast Spanning Trees

Even without learning-switches, many networks require the ability to broadcast messages. The process of establishing a broadcast spanning tree is largely identical to that of the routing algorithm described in the previous section. The key difference is that building a correct spanning tree is possible within the confines of the abstraction that this network model imposes. In order to broadcast a message, an end-host would simply send packets to a special address, perhaps the empty list. Although the full method is not explained in depth here, it is a simple extension of the routing algorithm.

5. A Static Model of rSDN in F#

I have written three software models of recursive software-defined networks in order to experiment with usability and develop new algorithms. I used the first model, written in a few hundred lines of OCaml, as an initial test of what would eventually evolve into the primitives that comprise the current rSDN programming model. The second model, consisting of about 1000 lines of F#, was the first end-to-end test of rSDN, the programming model, and the network model as described in this paper. I discuss the third, C# implementation in detail in Section 6.

5.1. Data Model

In the F# model, each box is a simple key-value store that retains no information about the dependence relationships between its properties. Running a primitive on such a network is very similar to performing a map operation on a list of integers: the operation statically produces a result that has no knowledge of its previous state or the operations that were run to produce its current state. At first glance, this seems like a straightforward representation about which it is easy to

reason, but, as we will see in the next two sections, it makes handling network changes exceedingly difficult.

5.2. Functionality

The F# model provides the full range of functionality specified by the rSDN programming model. It supports all seven primitives written as specializations of the generalized fold and performs routing as described in Section 4. Most significantly, the model supports basic event-handling. When a link or switch comes up or goes down, the network brings all routing metadata up to date. This functionality can be implemented naively by re-running the entire routing process starting at the root of the network, which is effective but preposterously inefficient.

As an optimization, the algorithm can be rerun from the nearest shared parent of the switches involved in the event. For example, if a link goes down, only the virtual ports and forwarding tables within the smallest subnetwork that encompasses the switches at either end of the link might need to be updated. At any higher level in the rSDN tree, the subnetwork is abstracted into a single virtual switch, meaning its inner workings do not affect virtual ports or routing beyond its subnetwork. This concept is the basis for the failure-masking property of the rSDN programming model: failures are hidden away from large sections of the network, obviating the need for most controllers to ever know that a failure occurred. This property improves response time to network events by reducing the number of controllers that must be involved in any network updated. For the same reason, it reduces load on control-links and additional, unnecessary controller computation. Ideally, failures should be hidden away from all boxes that they do not affect, but the F# model does not make a guarantee that strong.

5.3. Weaknesses

Localized event-handling is certainly a good start toward improving the efficiency of the rSDN network model, but it might still induce a tremendous amount of unnecessary computational work and network load. There are many scenarios where changes to network state are unlikely to require altering any forwarding rules, for example if an unused link goes down. These events will still force recomputation starting at the nearest shared parent of the switches involved, even when the results will be the same as before. Further, the F# model's optimizations are applicable only to the specific algorithms described in Section 4. A different routing methodology or entirely new algorithms might induce different dependence requirements that necessitate different event-handling behavior. As more operations are added, the event-handling behavior would have to become increasingly complex to deal with the interaction between multiple concurrent operations and more nuanced dependence relationships.

Generally speaking, leaving event-handling behavior to the user undermines the overall usability of the programming model. Even with only a few operations installed, fully and correctly implementing failure handling for the entire network model specified in Section 4 was an intricate and bug-riddled process. Managing upward and downward dependencies during partial recomputation required a variety of special cases, undermining the readability, maintainability, and modularity of the network programs. Fundamentally, the F# model assumes that the network primitives are static, one-time operations that are run over the network and subsequently forgotten. Change propagation is merely an afterthought. The C# model, described in the following section, views the primitives from a fundamentally different perspective, allowing the network to automatically rerun operations without user intervention and only to the extent that state has changed.

6. An Incremental Model of rSDN in C#

6.1. Background: Functional Programming and Incremental Computation

In pure functional programming, all memory is immutable, meaning that, once assigned, a variable retains its value and may not be altered. Whenever a data-structure is "modified," new memory is allocated for anything that has changed, leaving old versions of the data-structure intact. Computation is performed by compositionally applying functions to values, producing new quantities based on *transformations* applied to inputs. Since these inputs are immutable, the results of transformations also retain this quality. We can imagine a functional computation as a *dependence graph* composed of values and transformations. Inputs are the graph's leaves, with outgoing edges to transformations that are performed on them. These transformations, in turn, contain edges to the outputs they produce, values that can serve as the inputs for further transformations.

Incremental computation is an extension of this pure functional programming model. Suppose we make the leaf-inputs of the graph modifiable. We could then propagate these changes through the graph, recalculating transformations and their outputs recursively until all values have been updated appropriately. We can refer to these leaf-inputs as *modifiables*, which may be read or written, and the outputs of transformations as *readables*, which may only be read. As a simple optimization,

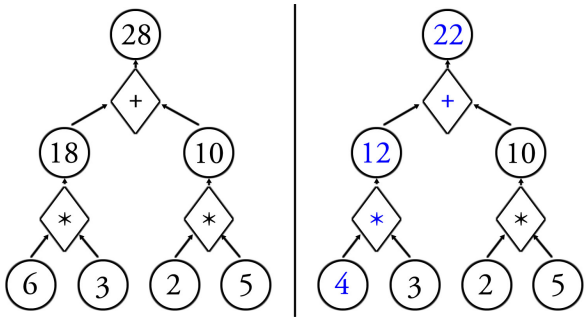

Figure 5: An example incremental computation. Circles represent readables/modifiables and diamonds represent transformations. The original computation is on the left with the computation after a modifiable is changed on the right. The blue nodes are those that must be recomputed.

we can also introduce *cutoff* functions for each readable and modifiable. These functions return `false` if a change to a value is small enough that recursive recomputation is unnecessary and `true` otherwise. Using these mechanisms and topological sorting on the graph, we can utilize our dependence graph such that, upon every update to a modifiable, the minimum possible quantity of recomputation is performed.

6.2. Incremental Computation and rSDN

The fatal flaw of the F# model was that it viewed the seven-primitives as one-time operations on a network represented as a series of static values. We can reimagine a recursive software-defined network as a single incremental dependence-graph. Each box has certain modifiable properties – its unique identifier and, for switches, port information. Mapping primitives create new readable properties from existing ones within boxes, while reducing primitives build dependence relationships across boxes. When, for example, the identifier of a controller changes, this information would automatically be propagated to the hierarchies of its children, which could spawn changes to certain boxes' virtual port information and finally trigger routing updates. In essence, the incremental picture of rSDN makes the implied dependence relationships of the programming model explicit and persistent. Programmers are thereby freed from considering recomputation or change propagation; the framework automatically updates the entire network – but only as much as necessary – when circumstances change.

The third and current iteration of the rSDN model, implemented in approximately 2500 lines of C#, is intended to serve as a proof of concept for incremental rSDN. The network algorithms in Section 4 are not yet fully implemented, but enough test code has been written to verify that the model behaves as expected. In the following sections, I will discuss a simple incremental

computation framework (Subsection 6.3) and its specialization as the key-value store underlying a box (Subsection 6.4). Afterward, I will explain inter-box dependence relationships manifested in *messages* (Subsection 6.5) before describing the primitives re-implemented in an incremental fashion (Subsection 6.6). Finally, I present a brief example of topology discovery implemented in the incremental framework (Subsection 6.7) and conclude with an explanation of bringing new or failed boxes up to date (Subsection 6.8).

6.3. An Incremental Computation Framework

Fundamentally, incremental computation *context* is nothing more than a dependence graph with auxiliary methods for reading, writing, and propagating updates. We can implement incremental computation with an explicit dependence graph, built on top of a simple directed graph that stores a value to accompany each node and edge. Each node contains either a *data cell*, which holds an incremental value, or a *transformation cell*, which contains metadata about every transformation.

Users have access to *modifiables* and *readables*, wrapper classes with pointers to the underlying data cells present on the graph. As their names suggest, readables provide read-only access to their underlying values, while modifiables allow a user to change their values. When a modifiable is altered, it notifies its incremental context about the change, triggering the change propagation process. This process performs a topological sort to place the successors of the modified node in dependence order, after which it recalculates each value and checks a cutoff function to determine whether any further recomputation is necessary. A user may optionally supply a *callback* function for each readable or modifiable that will be called whenever its underlying value changes.

A user adds additional readables by creating a *transformation* object, supplying a function

from an input dictionary of values to an output dictionary of values and specifying the number of outputs to be created. The indexing of these dictionaries is user-specified, but must be kept consistent to ensure that the provided function is able to properly distinguish its inputs. The choice of dictionaries rather than arrays opens the possibility of varying a transformation's number of inputs. The transformation object itself contains functions for adding or removing a readable input (providing the input dictionary key to which it corresponds) or obtaining an output (by output dictionary key). Since readable is a subtype of modifiable, modifiables can also serve as the inputs to transformations.

By allowing a user to dynamically change the inputs to a transformation, this framework provides additional flexibility similar to that referred to by Umut Acar et al. as *adaptive functional programming*[1]. In the context of a recursive software-defined network where topology changes may dynamically generate new dependence relationships, this functionality is mandatory. The danger in providing such an interface, however, is that a transformation must be able to handle any intermediate state of inputs as a user makes modifications. In initial testing, this property greatly complicated the task of writing incremental algorithms, making it far more difficult to reason about the state of the dependence graph.

To overcome this deficiency, the transformation interface provides two additional methods: *pause* and *resume*. Upon creation, a transformation is in the `paused` state – no changes are propagated through it and its outputs have no values. As soon as a user has connected the proper inputs, a transformation may be resumed, triggering an initial round of updates and opening it to future change propagation. Pause and resume combine to provide a transaction-like interface where a user can atomically modify a transformation's inputs without needing to reason about intermediate states, greatly simplifying the programming model.

6.4. Boxes as Incremental Key-Value Stores

This incremental framework serves as the basis for the key-value store that represents a box. No longer a simple dictionary, the key-value store contains many of the same primitives found in the previous section. As before, user can set named values, a manifestation of modifiables in this higher level of abstraction. The key-value store supports two kinds of names – one for properties and one for bases. Property names are strings, providing a mnemonic, user-friendly way to refer to items stored in the box. Bases, which are created and managed automatically, need predictable, regular, and unique names; base names are tuples of the unique identifier of the operation for which the base exists (discussed in Subsection 6.5) and the identifier of the box responsible for creating and modifying the base's value. A user can retrieve values by name and attach callbacks to values as in Subsection 6.3.

The most drastic new abstraction is the process of establishing transformations. Transformation functions receive two dictionaries of values, indexed by base and property name respectively, providing a convenient way to read box state. The key-value store handles the underlying conversion from the indexing used by the incremental framework and the higher-level constructs it exports. Transformation dependencies are still added dynamically, with pause and resume functionality visible to this higher level. Again, these dependencies are expressed in the language of base and property names. Critically, the cutoff functionality described in the previous section is also maintained in the key-value store, ensuring that recomputation ceases as early as possible.

6.5. Messages

Within a box, the incremental framework provides an effective basis for change-propagation. Between boxes, however, *messages* must be passed over control links to install operations and relay changes with ramifications beyond the local box. Messages provide the essential connection that allows the network to behave as one monolithic incremental graph. Individual message types exist for installing each kind of primitive, carrying the requisite metadata from box to box. Each *install message* receives a globally unique identifier represented as the tuple of the identifier of the box that triggered the installation and the next value of a serial number that the box maintains. A second kind of message, *notify messages*, facilitate incremental change-propagation. Each contains the name of the base being updated and its new value. These messages are passed between pairs of boxes whenever the network-wide change-propagation process steps across an inter-box dependency.

6.6. rSDN Primitives

All primitives described in Section 3 are fully implemented in the incremental rSDN model, but their signatures have been modified. In Section 3, each primitive was considered to be a one-time operation that performed an arbitrary operation on a box; the primitives were unconcerned with the content of the operation itself. In contrast, the incremental rSDN model is deeply concerned with this content: the set of properties and bases being read and the writes that follow. In this model, each call to a primitive must declare its read and write sets; it will only be allowed to touch the properties it specifies. Although it is possible to eliminate this restriction, the model would have to assume that every operation reads from and writes to every property in the box, causing all transformations to rerun upon any change to the box and undermining the entire purpose of the incremental model.

6.7. Example: Topology Discovery

Putting together the pieces described in this section, we will examine how the incremental rSDN model functions on a high-level using the topology discovery process explained in Section 4. We assume that a network springs into existence, complete with a tree of controllers and interconnected switches. Each box is created with a unique identifier, which is a modifiable property in its key-value store. The root controller receives a request for a new push_down operation with a read set consisting of the box's id and a default base value of an empty list. It increments its serial number and creates a new install message with the appropriate identifier. Within its key-value store, it creates a modifiable base to record the operation's default base. It then establishes a new transformation, creating dependence edges from this default base and its identifier to the transformation; out of this transformation, it adds dependence edges to a property named "hierarchy" and the base it will pass to its children. After unpausing the transformation, it attaches a callback to the readable base that sends a notify message to the controller's children whenever the base changes. Finally, it sends the install message to its children along with the updated base that it has produced. Each box receives this message and performs the same key-value store operations until the command has been installed throughout the network.

Now suppose that the root controller modifies its identifier. The incremental framework underlying the controller's key-value store will propagate the change, detecting that the root's "hierarchy" property and the updated base for the topology discovery operation have changed. This sets off the callback attached to this base, which sends a notify message with the new base value to each of the controller's children. These children receive the message and update the input bases for their topology discovery operations accordingly, in turn triggering change propagation in

their key-value stores. This process also repeats until it has been distributed to the entire network, updating every box's hierarchy appropriately. The key benefit of this incremental model is that the entire update process took place without any user-intervention. A user can continue to think about the network in a static fashion, and the framework manipulates the incremental model such that this illusion becomes reality.

6.8. Bringing Boxes up to Date

Suppose a controller or switch is added to network, or even that a box goes down, misses a series of messages, and comes back online. These boxes must be brought back up to date by the network. Since all rSDN operations are entirely deterministic, a box can return to the proper state by replaying all messages it would have received in order. This process may, however, produce unnecessary network turbulence as a box goes through all the past states of the network, producing erroneous notifications that temporarily disrupt proper network functionality. As an optimization, boxes might transmit only install messages before supplying notifications for the current values of the corresponding bases. This procedure also reduces the amount of state that boxes have to retain; storing all notifications that have ever been received would require unbounded storage.

7. Future Work

7.1. Compiling for Real Networks

The natural next step for this framework is to bring it to life in a recursive software-defined network. The current primitives and data-model are exceedingly high-level and have been designed to run only in a simplified model of a network. This programming model is an excellent candidate for a domain-specific language, providing an opportunity to smooth out many of C#'s semantic

rough-edges and simplify the network programmer's task. The read and write sets of an operation, for example, could be inferred by a compiler in a way that is presently impossible. Designing, interpreting, and translating such a network language into the lower-level constructs that SDN understands opens an entirely new set of research avenues.

7.2. Better Network Algorithms

Section 4.4 details a number of shortcomings of the current abstracted shortest-paths routing algorithm. Since the network algorithms themselves were considered less central to this project than the programming and data models, there is likely significant room for improvement in the area of abstracted routing. Subnetworks could export limited guarantees about the path lengths between their ports, allowing their parents to make more informed routing decisions. Parents could also select multiple outgoing paths from subnetworks, allowing the network to maintain connectivity even if a subnetwork becomes partitioned. There is enormous space for more research in this area.

7.3. Optimizing Incremental Code

When performing incremental recomputation, a transformation is a single, atomic unit – it must either be left alone or performed in its entirety. Many transformations, however, have several sub-steps that could potentially be broken down into separate sub-transformations. If a transformation were decomposed into these intermediate steps, it is possible that it could be cut off after only a few of these steps have been performed, reducing the total computation necessary. The key insight of this observation is that, in the context of incremental computation, transformations should be made as small as possible to reduce the effort that the incremental context must put into recomputation.

In the context of the network algorithms described in Section 4, several instances of reducing primitives could be expressed as smaller reducing calls followed by many uses of map. Programming in this fashion is much more efficient for the incremental framework but is difficult to understand from a programmer's perspective. When a similar conflict between functionality and programmability arose in the F# model, the solution was to make the framework do the work to simplify the programmer's experience. In this situation, the scenario is little different. Although a slightly more complex problem, it should be possible to automatically "incrementalize" large functions, breaking them into smaller constituent pieces that can be expressed as separate incremental units. It is likely that this process will involve both language and compiler extensions.

8. Conclusions

Evaluating a programming model is a difficult endeavor. Although many properties can be proved or measured, usability is a qualitative and subjective criterion that lacks these luxuries. Looking back at the five goals for this network model mentioned in Section 1, however, we can begin to gauge the efficacy of the model by the needs it sought to satisfy.

8.1. Manage large networks in a high-level, portable manner.

The central purpose of this paper was to develop a programming model that makes administering recursive software-defined networks straightforward. Without this ease-of-use criterion, an otherwise theoretically sound programming model would be unusable. Although it is impossible to quantitatively measure the convenience of this model, the anecdotal experience of implementing routing suggests that this model meets the expected standards. The primitives work at an exceedingly high-level, abstracting switches and controllers into key-value stores and the network itself into a

basic functional data-structure. Although these primitives do not yet compile into any language that a software-defined network would understand, they are sufficiently high-level to ensure that they can be freely compiled into a number of different target languages.

8.2. Distribute computational work evenly across all controllers.

This property is not statically enforced, but the programming model is "distributed first" – it heavily favors writing distributed algorithms. That is to say, it is possible but highly inconvenient to write operations where all information is passed through lower-level controllers to the root, which performs all computation and passes decisions back down to switches. Doing so requires intricate special-casing that is more difficult to write than the accompanying distributed algorithm, in which a single, repeated operation is performed at every level. This repetition, in turn, protects the notion that computation is relatively – if not perfectly – evenly distributed.

8.3. Abstract subnetworks, reducing the latency for response to network events.

For the same reason that the programming model is "distributed first," it is also "abstraction first." The programming model allows direct dependence relationships between parents and children but statically bans explicit inter-generational dependencies. Within the confines of the programming model, it is easiest to write programs in which a box only sees its parent and children, encouraging programmers to delegate responsibility for subnetworks to their nearest shared parent. The network model described in Section 4 embodies this principle, with virtual links and abstracted routing giving each subnetwork responsibility for its own administration. This abstraction opens the possibility of hiding changes in network conditions within their smallest containing subnetwork, facilitating local event-handling with lower latency.

8.4. Respond to changes with the minimum necessary communication and computation.

This condition was the most difficult to satisfy of the five. Even with a programming model that encouraged dividing work evenly and making use of abstraction, recomputation still proved difficult to tame. The F# model, with its static picture of the network, forced the programmer to reason about the recomputation necessary when particular network conditions changed. This approach was challenging to manage and highly error-prone. The C# model's incremental basis resolved this issue while requiring no additional intervention on the part of the programmer. The network model inherits the same promise made by the underlying incremental computation architecture: assuming that cutoff functions are properly implemented, changes will be propagated only as far as is necessary. Combined with the appropriate use of abstraction, the incremental approach minimizes the work spent on recomputation while ensuring that event-handling stays as local as possible.

8.5. Develop scalable network algorithms that run efficiently over wide geographic distances.

Satisfying this larger, overarching goal is a natural consequence of meeting the preceding four. Through a combination of a functional programming model and an incremental data model, it is now possible to write network programs that leverage the many benefits of recursive software-defined networking without falling victim to its underlying complexity.

References

[1] U. A. Acar, G. E. Blelloch, and R. Harper, "Adaptive functional programming," Nov. 2006.
[2] M. McCauley *et al.*, "Scaling sdn through recursion," Apr. 2013.