

Managing future networks: a case study with Fibbing and Segment Routing

Thesis authors

Florent LEJOLY , Cedric VANDEN BULCKE

A thesis submitted in fulfillment of the requirements
for the degree of
Computer science engineer

Supervisor

Olivier BONAVENTURE

Readers

Olivier TILMANS, Marco CHIESA

2016-2017

Future networks will have to cope with the ever growing mega trend toward Social medias (e.g. Facebook, Twitter, etc), online video streaming services (e.g. Netflix), or even Voice over IP (e.g. Skype, Facetime). However, these trends reveal different traffic patterns that network administrators would want to support in order to provide better services to their customers. Besides that, the emergence of more and more lucrative businesses on the Web coincides with an increase of massive DDoS attacks carried over the network. As a network administrator, you would want to quickly react to such attacks by deviating the suspicious traffic toward a Deep Pack Inspection tool.

For all those reasons, we developed a management solution that works with Fibbing and Segment Routing for next generation networks, that network administrators can use to enforce network requirements. Our solution enables to specify requirements, on a per destination basis, that can be triggered by predefined conditions. This allows the management solution to automatically react to certain traffic patterns.

First, we would like to thank our thesis supervisor Pr. Olivier Bonaventure of the EPL at the UCL, for the weekly meetings that greatly helped us along the realizing of this work. We would also like to thank him for the interest he showed for what we were doing and how this enabled him to direct us on promising paths.

Secondly, many thanks to Olivier Tilmans who's office was always open whenever we ran into a trouble spot or had a question about our work or writing. We also thank him for attending the weekly meetings.

We would also like to thank David Lebrun for his explanations about Segment Routing and for the VM he provided us and finally François Aurby for his help and explanation concerning the Algorithm used in the Simplifier.

Contents

1	Introduction	3
2	State of the Art	7
2.1	Distributed control	7
2.2	Centralized control	8
2.2.1	Presentation of SDN	8
2.2.2	Benefits of SDN	8
2.2.3	SDN Three Layers Architecture	9
2.2.4	Trade-offs	10
2.3	Fibbing	10
2.4	Segment Routing	11
2.5	Traffic Engineering	11
2.5.1	RSVP TE	12
2.5.2	Traffic Engineering with Fibbing	12
2.5.3	Traffic Engineering with Segment Routing	13
2.6	Management Tools	13
2.6.1	SNMP	14
2.6.2	NETCONF & YANG	14
2.6.3	ConfD	15
2.7	Conclusion	16
3	The YANG model	19
3.1	Introduction	19
3.2	Network Data model	19
3.2.1	Generic Network Data model	20
3.2.2	Features specific to the Fibbing Solution	22
3.2.3	Features specific to the Segment Routing Solution	24
3.3	Requirements YANG Data model	24
3.3.1	Principal attributes	25
3.3.2	Scheduled Conditions	26
3.4	Constraints in the YANG models	26
3.4.1	Uniqueness constraint	27
3.4.2	Reference constraint	27
3.4.3	loop-free path constraint	27
3.5	Motivation of the YANG data model	27
3.5.1	Model of the network	28
3.5.2	Model of the requirement	28
3.6	YANG model for Segment Routing Routes	29
3.6.1	Motivation for the YANG data model for Segment Routing Routes	29
3.7	Interpretation of the YANG data models	30
3.8	Conclusion	30

4	Architecture	31
4.1	Introduction	31
4.2	General architecture of the solutions	31
4.2.1	Network Manager CLI	32
4.2.2	the Autotopo framework	32
4.2.3	ConfD	34
4.2.4	Network Manager Core	35
4.2.5	Requirement deployment checks	39
4.2.6	Southbound Interface	39
4.2.7	Network Emulator	40
4.2.8	Network Elements	40
4.3	Solution with Fibbing	40
4.3.1	Autotopo for fibbing	40
4.3.2	Southbound Interface for Fibbing	41
4.3.3	Network Emulator for Fibbing	41
4.3.4	Network Elements for Fibbing	42
4.4	Solution with Segment routing	42
4.4.1	Autotopo for Segment Routing	42
4.4.2	Southbound Interface for Segment routing	43
4.4.3	Network Emulator for Segment Routing	48
4.4.4	Network Elements for Segment Routing	48
4.5	Toward a real life solution	48
4.5.1	First case	48
4.5.2	Second case	49
4.5.3	Controller replica	50
4.5.4	Support for new ideas	50
4.6	Implementation	50
4.7	Conclusion	50
5	Practical example of the working and a configuration	53
5.1	Introduction	53
5.2	Applications	53
5.2.1	Regular network	53
5.2.2	Partitioned network	54
5.2.3	Failure with Segment Routing	55
5.2.4	Dealing with Main Controller failure	56
5.3	A simple example network	57
5.3.1	The network with Fibbing	57
5.3.2	The network with segment routing	58
5.3.3	Requirement	59
5.4	Conclusion	61
6	Performance	63
6.1	Network size scaling	63
6.2	Number of Requirements scaling	64
6.3	Performance of Fibbing	65
6.4	Dealing with failures with Fibbing	66
6.5	Southbound Interface with Segment Routing	66
6.5.1	Simplifier	66
6.6	Failure with Segment Routing	68
6.7	Performance of Segment Routing	69
6.8	Conclusion	70

7 Conclusion	71
Bibliography	73

In today's networks, the ever popular mega trends toward social medias (e.g. Facebook, twitter, YouTube,...) has revealed new traffic patterns like flash crowd whenever a YouTube video or tweet goes viral. Which is by essence unpredictable and hard to determine for how long it will last. As for the last few decades, we have also seen some other forms of communication migrating over the Internet like telephone with Voice over IP (e.g. Skype, Facetime), radio with online music services (e.g. Spotify), and of course television with online streaming services (e.g. Netflix). Those new technologies add new constraints on the Internet that the infrastructure wasn't originally designed for. Indeed, most of the Internet protocols were designed to perform Best Effort, while those services require Real Time delivery.

Because the Internet has become the medium for lots of lucrative services (e.g. advertisement or premium like Netflix), this last decade has also seen number of massive DDoS attacks carried over the network. Since the Internet wasn't designed with security in mind, the hassle of dealing with such attacks has fallen onto network administrators, that may need to be able to deviate the suspicious traffic toward a Deep Pack Inspection tool.

These are the challenges that next generation networks have to cope with. However, traditional solutions based on manual configuration of various proprietary hardware revealed to be often error-prone and sometimes underutilize network resources. A promising technology in this area is Software Defined Network (SDN) which allows fined grained control over your network. Although most existing SDN implementations work with OpenFlow, we developed a SDN-like solution to work with two state of the art technologies: Fibbing, and IPv6 Segment Routing. The idea behind Fibbing is essentially to have a controller trick the routers into seeing a fake topology that is matching user defined requirements. IPv6 Segment Routing is an implementation of the classic Segment Routing scheme, based on the IPv6 data plane instead of MPLS.

More specifically, we developed a network controller that closely follows the SDN architecture. Through this controller users can express network requirements via an expressive interface. There are different types of requirements. The simple requirement simply specifies the path that must be followed for a certain destination. Then, users have the possibility to define some conditions associated with the simple requirement. These conditions can express a time during which the requirement must be valid, that the requirement should work as a backup path in case of link failure or if the bandwidth utilization of link is above a predefined threshold.

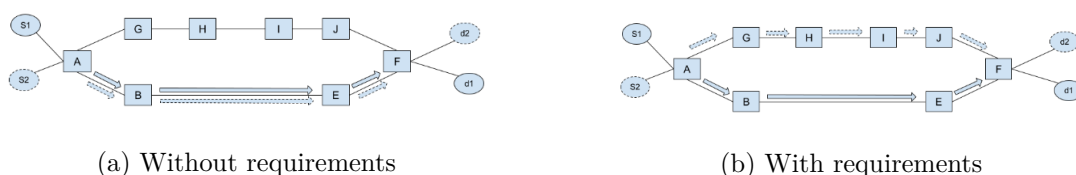


Figure 1.1: Time to deal with failures

For example, let's imagine that you are a network administrator of the abstract network depicted in figure 1.1. Originally, the network is in the configuration on figure 1.1a, where the two sources ($S1$, $S2$) are sending traffic destined for two destinations ($S1$ for $d1$, and $S2$ for $d2$) along the same path in the network: $[A\ B\ E\ F]$. This situation may not be desirable since the two flows compete for the same resources while there exists another disjoint path. From the network administrator point of view, you want to specify that traffic toward destination $d2$ must follow the path $[A\ G\ H\ I\ J\ F]$ (figure 1.1b), while the traffic for $d1$ remains unchanged. With our solution, this could be done simply using the following syntax:

```
requirement
  name toD2
  dest  d2
  Path  [ A G H I J F ]
  state running
!
```

As can be seen specifying requirements is very simple. We chose to rely on router names to express requirements instead of IP address or router-ID because router names are more intuitive to manipulate. Indeed, it is more likely that you will detect a typing mistake if you wrote *OLSA* instead *LOSA*, than *64.57.33.23* instead *64.57.99.23*.

Imagine now, that the destination $d2$ is for a video streaming website (e.g. Netflix) popular among teenagers. You know from previous analysis that the traffic load is particularly heavy between 15:30 and 23:30, when school is over. Thus you want to deploy the requirement $[A\ G\ H\ I\ J\ F]$ only if we are in this period and if the load can't be handled by the situation figure 1.1a. This can be translated using our syntax:

```
requirement
  name toD2
  dest  d2
  Path  [ A G H I J F ]
  state scheduled
  schedule Type bandwidth
  schedule start-hour 15:30
  schedule end-hour 23:30
  schedule days [ Monday Tuesday Wednesday Thursday Friday]
  schedule link from A to B
  schedule link bw-percentage 50
!
```

This representation of the requirement indicates that the requirement $[A\ G\ H\ I\ J\ F]$ should only be deployed between 15:30 and 23:30, if we are in one the week day specified, and if the bandwidth utilization of the link **A-B** is above 50%.

Concerning the front-end, the interface exposed to users is a Command Line Interface (CLI) generated by ConfD, based on the structure of the requirements described using the YANG data model. This interface can also be used to describe the network that the requirements should work with.

Concerning the back-end, we designed our controller to be adaptable to different technologies with only few "tweaks". In order to demonstrate the modularity of our SDN-like architecture, we implemented solutions that work with Fibbing and IPv6 Segment Routing. The solution with Fibbing can serve to show how the core of the application can work in pair with an existing controller (i.e. Fibbing controller), while the solution with Segment Routing can serve to show

how the core of the application can be extended to implement a centralized controller.

Chapter 2 presents the State of the Art concerning this area of networking, and will emphasize on the technologies used to develop our solution. Chapter 3 then provides a complete overview of the YANG data model used to render the front-end interface via which users can specify both network configurations and network requirements. After this, the Chapter 4 presents into detail our SDN-like Architecture, and provides insights on how to adapt this architecture to work with Fibbing and Segment Routing. Then Chapter 5 provides some practical applications of the solution. Finally, Chapter 6 shows some performance and evaluation of our solution.

This chapter first presents some of the different techniques that exist to manage and overall control a network in section 2.1, 2.2,2.3, and 2.4. Then, section 2.5 motivates why there is an increasing need for network management with the Traffic Engineering use case. After this section 2.6 presents some of the management tools that exist, and that can help network administrators to better manage their networks. This state of the art should give an insight of where our solution is situated among existing work.

2.1 Distributed control

A simple way to manage a network is to rely on a series of protocols to learn, distribute, and organize the routes information among the different routers composing the network. To do so, as a network administrator you can rely on a combination of protocols, for example ISIS [rfc, 1990a] or OSPF [Moy, 1998] for the interior gateway protocols (IGPs), and BGP as exterior gateway protocol [Rekhter et al., 2006].

These protocols are distributed, which means that they run on all the nodes of your network and will work together to learn, distribute, and organize routing information. Typically, IGP protocols work by letting nodes flood link-state advertisement (LSA) among all other nodes in the network. These LSAs contains reachability information (what prefix can be reached via the link) as well as the cost associated with links. The costs are usually set during configuration of the network, and are thus fixed. All nodes participating in the protocol will eventually converge to a shared view of the network, and discovering the shortest paths to reach each nodes according the received LSAs. Each nodes install forwarding entries in their respective routing tables, to direct packets to the next hop according to the shortest path for the destination.

In the other hand, BGP can be used to exchange routing and reachability information with other networks. Typically used to have access to the global Internet, by advertising known destination and discovering new destination prefixes.

These protocols are old for the most part but have stood the test of time. Indeed, these protocols are very robust, thus can scale to very large networks. Also from an administrator point of view, there is a big community that has worked on the standardization of these protocols, thus they are well defined and well documented. As well, most vendors provide devices already implementing these solutions. This has lead to many heterogeneous networks with many devices from different vendors which may have different trade-off between cost and performance.

Although these protocols are robust and can scale to large networks, they lack flexibility. Indeed, routers will work together to find the shortest paths, but once these paths are found these will be the unique paths taken all the time, unless there is a change in the topology (e.g. failures, adding new nodes, or modifying the costs). As an administrator, you can carefully choose the costs to guide traffic on desirable paths. However, this does not give fine grained

control, since each time the costs are changed the routers will have to recompute the shortest paths thus leaving the network in an uncertain state in the meantime. Also, as a network administrator if you have a heterogeneous network, configuration can be tedious. Indeed, while most vendor devices implement these protocols, the configuration interfaces are often proprietary and thus can vary from vendor to vendor. This means that in order to configure your heterogeneous network, you will need to know the commands for all vendor specific devices in your network.

Hence the trade-off with this technique is to have robust and scalable protocols in favor of fine grained control over the paths taken by the traffic, and flexibility for configuration.

2.2 Centralized control

As an opposition to distributed protocols, Software Defined Networking (SDN) aims at providing finer grained control over the network via a centralized controller. More specifically, SDN is a Networking paradigm that can enable dynamic and centralized control over a network. This is a very promising method especially for data centers and ISPs, but it can be useful for enterprise networks as well.

2.2.1 Presentation of SDN

Software-Defined Network (SDN) is an emerging network architecture where network control is decoupled from forwarding and is directly programmable ([ONF, 2012]). This definition was proposed by the Open Networking Foundation (ONF) [ONF,] which is a non-profit consortium dedicated to development, standardization, and commercialization of SDN. Two characteristics dominate this definition, namely the decoupling of control and data plane, and programmability of the control plane.

The idea of decoupling control and data plane has been proliferated during the last decade. Caesar *et al.* first introduced a Routing Control Platform (RCP) in 2004 [Feamster et al., 2004], where BGP inter-domain routing is replaced by a centralized routing control. The goal being to reduce the complexity of fully distributed path computation. The IETF released the Forwarding and Control Element Separation (ForCES) during the same year. This framework separates control and packet forwarding element [Yang et al., 2004, Doria et al., 2010].

The way SDN stands out is in its ability to provide programmability through decoupling of control and data plane. Because SDN proposes the control and data plane decoupling at the architectural level, this design allows network control to be done separately on the control plane without affecting data flows. Overall, the network "intelligence" can be moved from the switching devices to external controllers. This provides not only a simpler programmable environment but also a greater flexibility for external software to define the behavior of the network as well.

2.2.2 Benefits of SDN

Thanks to the decoupling of data and control plane, SDN brings potential benefits for easier configuration, better performance, network innovation and network virtualization.

Because SDN provides programmability of the network, it can tackle some of the inflexibilities and complexities of traditional networks. SDN simplifies network management and brings virtualization to the network, by removing the complexity of the Infrastructure Layer and adding visibility for applications and services. Administrators can better define network flows that meet user needs, on a network-wide manner, without needing to implement custom policies and protocols on each device separately. Moreover, with the ability to gather live network status, SDN

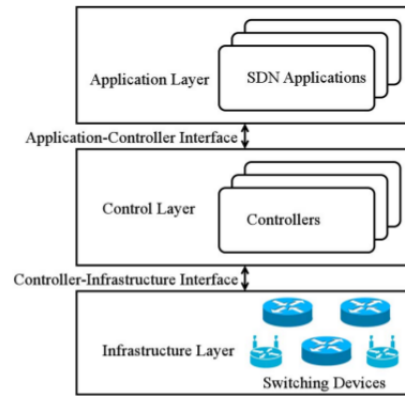


Figure 2.1: Three layers architecture of SDN [Xia et al., 2015]

allows a centralized control in real-time based both on live network status and user defined policies.

Configuration is an important part of network management but due the heterogeneity of network devices (i.e. different vendors) this can be a tedious (i.e. manual configuration) and error-prone (i.e. incoherence in the network) task. SDN aims at unifying the control plane over all kind of devices [Gember et al., 2012] (e.g. routers, switches, NATs, firewalls, etc). Hence, networks can be programmatically configured in a centralized place.

The expectation of SDN is also to improve network performance. In traditional networks, because of the heterogeneity of technologies present, optimizing performance can be a difficult undertaking. SDN provides the opportunity to develop centralized algorithms based on a global network view and global information obtained across different layers (in the ISO network architecture). Such algorithms would provide solution for general Traffic Engineering (TE) problems : Quality of Service (QoS) [Ferguson et al., 2013], end-to-end congestion control [Ghobadi et al., 2012] and traffic scheduling [Al-Fares et al., 2010].

2.2.3 SDN Three Layers Architecture

Along with the definition (section 2.2.1) ONF also provided a reference model which consists of three layers (figure 2.1) : Infrastructure Layer, Control Layer, and Application Layer (in a bottom-up manner).

The Infrastructure Layer mainly consists of switching devices (e.g. switches, routers, etc) interconnected to form a network that operate in the data plane. Those devices are primarily responsible for two jobs : collecting and reporting network status to the controller, and processing packets according to forwarding rules.

The Control Layer closes the gap between the infrastructure and Application Layer. It interacts with the Infrastructure Layer via its *south-bound interface*, which is used to expose tools for controllers to access the functions of the switching devices (e.g. accessing network status and installing forwarding rules). Via its *north-bound interface*, the Control Layer can interact with the Application Layer to which it can expose an Application Programming Interface (API). Through this API the Application Layer can obtain the network status gathered by the switching devices and the Control Layer can translate high-level requirement received from the Application Layer into forwarding rules for the Infrastructure Layer. The control Layer can perform *east-west* (i.e. controller-to-controller) communication in the situation where multiple controllers are

present in the network for redundancy and/or increased performance purposes [Xia et al., 2015].

Finally, the application Layer provides the interface through which the users can interact with the SDN controller.

2.2.4 Trade-offs

SDN aims at placing the control of the network onto centralized controllers. This has the advantage of making network management much easier, since it enables to have a fine grained control over the network flows. However, this loses the advantages of distributed control that are robustness and scalability. Indeed, by having the network managed by centralized controllers, this introduces new complex challenges. If the solution needs to be robust to failures, there needs to be multiple controller replicas. Those replicas must synchronize their information which can be achieved but with complex algorithms (e.g. Paxos). In the other hand, the controllers need to maintain a view of the network which can struggle to scale to large networks.

2.3 Fibbing

Since there are advantages of using distributed methods for scalability and robustness, and using centralized methods for easier management and finer grained control over the network, Fibbing is a recent technique that aims at combining the distributed and centralized approaches.

Conceptually, distributed routing protocols (e.g. OSPF) can be seen as black boxes that take routing messages as input and produces forwarding paths as output [Vanbever and Vissicchio, 2014a]. The goal of Fibbing is to exploit this black box behaviour to carefully manipulate the input routing messages so as to achieve desirable forwarding paths as output. Indeed, by sending "fake" routing messages (or lies) on behalf of existing routers, Fibbing coaxes routers into computing desirable forwarding entries.

More specifically, the Fibbing relies on centralized controllers (denoted as Fibbing controllers) that take as input one DAG per destination, and the IGP topology. From there it will compute the corresponding *Augmented Topology*. The *Augmented Topology* being the topology containing the *fake* nodes so that do achieve the Directed Acyclic Graph (DAG) requirement for each destination. Once the *Augmented Topology* is computed, a series of algorithm will aim at reducing the number of *fake* nodes used in the augmented topology, resulting into a *Reduced Topology*. Finally, from this *Reduced Topology* the controller will be able to inject the *fake* nodes as OSPF messages into the network, so that the targeted routers install this new routes. The controller will also monitor the network and recompute the *Augmented Topology* according to eventual updates (e.g. failures). The current implementation of the Fibbing controller works with the OSPF protocol.

Fibbing demonstrates how both centralized and distributed approaches can be combined. Network management is made easier by expressing DAG requirements on centralized controllers. On the other hand, these requirements are enforced by routers running distributed algorithms which makes this solution scalable as well. Also, Fibbing can work with traditional IP routers, so there is a clear backward compatibility with legacy networks.

2.4 Segment Routing

Segment Routing [Filsfils et al., 2014] is a source routing paradigm that can be implemented in the MPLS [Srinivasan et al., 2004] or IPv6 data plane.

Conceptually, Segment Routing (SR) will express on a source router the path that needs to be taken for a given destination as a list of segments. There are two types of *segments*: node segment and adjacency segment. Node segment forwards packets to a specific network node. Conversely, the presence of an adjacency segment forwards packets along a specific link. When the network administrator has decided which segments he/she wants to install in his/her network, the segments will be advertised to corresponding routers that will install the segments in their data plane.

For the implementation in the MPLS data plane, segments are encoded as MPLS labels. And nodes learn these labels by flooding the information in the network.

The other implementation [Previdi et al., 2017] relies on the IPv6 data plane with the addition of a new type of Routing Extension Header. With this implementation, the list of segments forming a path is encoded as IPv6 addresses corresponding to routers in the SR domain. The SR domain being the set of nodes participating into the source based routing model. In the general case the list of segments will be represented as the list of IPv6 loopback addresses assigned to the nodes forming the path. The new header extension both contains the segments list and a special field indicating the *active segment*. Since it follows the source based routing model, it will be the ingress routers of the SR domain that will push the extension header on top the packets for a given destination. From there, the packet will be routed based on the IPv6 address indicated by the active segment of the extension header. Once the packet arrives to the current active segment, the active segment field is updated to point to the next address in the segments list. In order to represent an adjacency segment, instead of the loopback address of the node, the interface address of node corresponding to the adjacency is used.

Segment Routing routes A Segment Routing (SR) route, on a router working with the SR-IPv6 linux kernel implementation¹ (this types of routers will be denoted as SR routers), is expressed as a *destination prefix*, an ordered list of *segments*, and an *interface*. The list of *segments* contains the list of IP addresses (most often the loopback address) for which the traffic must pass for the given *destination prefix*. The *interface* is the source router interface via which the traffic following the SR route should be forwarded.

It should be noted that the implementation of Segment Routing relying on the IPv6 data plane is the one we used for the implementation of the solution with Segment Routing (see chapter4).

This solution can also be viewed as a combination of centralized and distributed approaches. Indeed, the shortest paths between segments can be derived by an underlying IGP protocols running in the network, while the Segment Routing routes can be installed on the source routers by a SDN-like centralized controller.

2.5 Traffic Engineering

The different techniques presented so far all enable to guide traffic along a certain path. This section presents a typical use case why being able to express specific paths is a useful property.

¹<http://www.segment-routing.org>

The principal purpose of a network is to deliver packets from a source to a destination (e.g. ingress to egress routers). In traditional IP network, this functionality is performed by routers that search the lowest cost path in a lookup table to determine where to forward the packet next. The major drawback of this technique is that once a optimal path is found between two routers, the traffic will always follow that path, provided that there are no node or link failures. This behaviour that tends to route traffic along a common path may cause underutilization of network resources on other possible paths and even sometimes congestion on the optimal path. With the ever increasing load upon which today's networks are under, came the need for Traffic Engineering (TE). TE allows network administrators to have more and better control over the paths taken by specific types of traffic. For example, Voice over IP (e.g. Skype call) needs strict bound on the transmission delay, while in the other hand a video downloading application wouldn't necessarily need such strict bound, but a large bandwidth.

In section 2.5.1, will be briefly presented a popular technique to perform Traffic Engineering with MPLS RSVP-TE. Section 2.5.2 and 2.5.3 will show a practical example of how we can do Traffic Engineering with Fibbing and Segment Routing respectively.

2.5.1 RSVP TE

One of the traditional TE technique is Resource Reservation Protocol [Zhang et al., 1993] (RSVP). MPLS utilizes RSVP TE to construct custom paths. RSVP TE works in two phases. In the first phase, the ingress router will send a RSVP PATH message to see if the requested resources are indeed available along the requested path. If the resources are available, the last router in the path will send a RSVP RESERVATION message back to the ingress router, along the same requested path so that the routers can install the MPLS labels in their local memory. Once the path is created, the network resources can be better utilized. The drawback of this approach is that the path creation requires signaling with RSVP-TE, and thus the path creation process is not very reactive.

2.5.2 Traffic Engineering with Fibbing

This section will provide a practical explanation of how Fibbing can achieve the Traffic Engineering requirement described in the introduction (figure 1.1). As a reminder the goal was to force the traffic destined to destination $d2$ to follow the path $[A\ G\ H\ I\ J\ F]$. Figure 2.2 depicts the Augmented Topology computed by the Fibbing controller, that corresponds to the requirement. As can be seen the controller will require the addition of two fake nodes $f1$ and $f2$. These two fake nodes will steer the traffic for destination $d2$ along the links indicated by their respective arrows. Once the traffic for destination $d2$ reaches node H , it will follow the normal shortest path along $[I\ J\ F]$ thus completing the requirement.

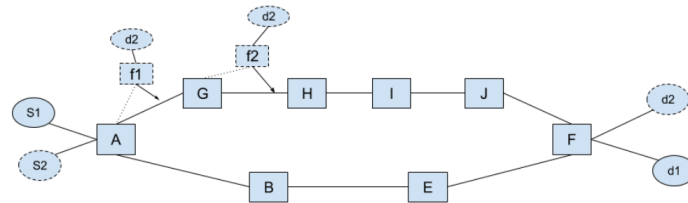


Figure 2.2: Augmented topology. Fibbing can steer traffic for $d2$ by using two fake nodes $f1$ and $f2$.

From a pure routing point of view, the fake node *f1* corresponds to router **G** advertising to router **A** that it can reach destination *d2* directly. Router **A** will thus update its shortest path for destination *d2* to go via **G**. This happens also for fake node *f2* between router **H** and router **G**. It should be noted that the actual routing messages will be sent by the controller but on behalf of router **G** and router **H**.

2.5.3 Traffic Engineering with Segment Routing

Similarly to section 2.5.2, this section provides a practical explanation of how the Traffic Engineering requirement of the introduction can be achieved with Segment Routing. Figure 2.3 depicts how Segment Routing can be used to steer the traffic for destination *d2*. As can be seen from the picture, the SR route, indicating the segment stack [**G H I J F**] must be pushed on top of packets destined to *d2*, is installed on source router **A**. Then the encapsulated packet is routed according to the active segment in the segment stack (indicated by the dashed border). Once the packet arrives to a router that corresponds to the active segment in the stack, this router updates the active segment to point to the next segment in the stack. Finally, once the active segment reaches the last segment in the stack, the corresponding router (**F** in our example) pops the stack from the packet and forwards it according to the shortest path. In our example, **F** directly forwards to destination *d2*.

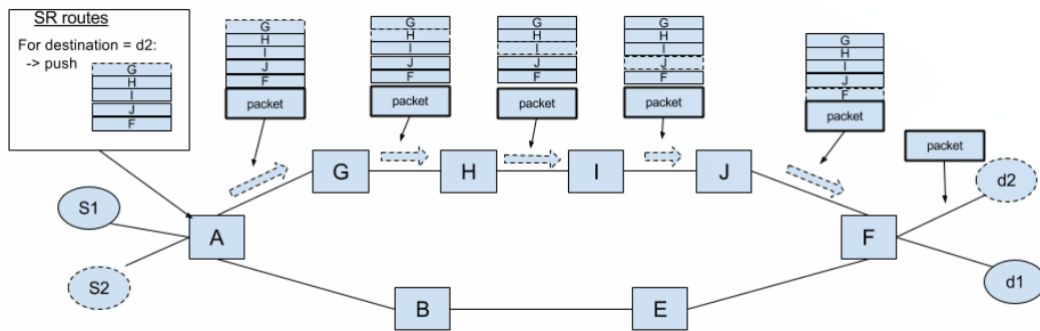


Figure 2.3: Router **A** pushed the SR segment stack [**G H I J F**] to steer the traffic for *d2*.

Traffic Engineering is thus a practical example of what can be achieved with Fibbing and Segment Routing. It should be noted that Traffic Engineering has become a more and more desirable property to have in networks. Indeed, Today's networks need to transit different types of traffic which have different constraints (e.g. large bandwidth, or small delay). A traditional way to accommodate such different behavior is to over-provision the network resources (e.g. adding more fiber). However, this technique comes with a significant cost, and thus may not be able to efficiently scale for financial reasons.

2.6 Management Tools

Up until now, we have seen different methods that can steer traffic among desirable paths, and seen how it can be used for Traffic Engineering purposes. This section will then presents some possible tools that can help network administrators to express their requirements and overall manage their networks.

Different network management tools are presented here, that network administrators can use to facilitate network management. They are organized in their chronological order of apparition. We will present the existing solutions as a basis of comparison, and go into more details for the technologies we used to build our own solutions.

2.6.1 SNMP

With the growing popularity of the Internet and the increased complexity in TCP/IP based networks, came the need for standardized management solutions to facilitate network administrators' jobs [rfc, 1988]. One proposition that came out was the Simple Network Management Protocol [rfc, 1990b] (SNMP). The key features provided by SNMP are information sharing by devices about their current state, and predefined values modification by an administrator queries.

In a network where SNMP is deployed, devices can contain SNMP agent or manager. An SNMP agent is a program running on the devices that is responsible for gathering, storing and potentially updating information in an organised data store called the *Management Information Base* (MIB). It is also responsible for replying to queries sent using the SNMP protocol. SNMP managers usually maintain information about all the SNMP agents in the network and can issue requests to retrieve or set values on SNMP agents devices.

While SNMP is nowadays only still used for monitoring, it failed to provide strong properties for global network configuration. This lead to the development of NETCONF by the IETF. Concerning network monitoring, other popular solutions are Netflow [Claise, 2004] or Ipfix [Claise et al., 2013].

2.6.2 NETCONF & YANG

The Network Configuration Protocol (NETCONF) can be used to retrieve, update, and manipulate configuration information, and overall manage network devices [Enns et al., 2011]. NETCONF exposes an API that devices can use to send and retrieve configuration data. It relies on a Remote Procedure Call paradigm (RPC), where both requests and responses are encoded in XML format.

NETCONF makes a clear distinction between configuration and state data (this is something that SNMP failed to do). *Configuration* data is the set of writable data required to change the system from one state to another. *State* data is all the additional data that are not configuration related. This can be status information and network statistics for instance.

While NETCONF is a protocol, YANG [Bjorklund, 2010] is a data-model describing the structure of the data that can eventually be manipulated via NETCONF. YANG is a human readable data model, that can describe the structure of data in a hierarchical way. This is particularly well suited to work with XML, which is also human readable and can easily express hierarchical data.

Where NETCONF really stands out from SNMP is in its ability to provide network wide transactions. This means that configuration changes impacting multiple devices will done in a *all or nothing* way: either all devices update their configuration or none of them does, in the even of failures during the transaction (e.g. configuration errors, devices failures). This transactional abstraction gives a powerful tool to network administrators, as they now have the ability to manage their network in a global way instead of managing each devices individually. Service activation is another capability of NETCONF that is not easily performed with SNMP.

Overall, NETCONF is a easy to use network management protocol that was specially designed to support remote service activation and provisioning, by relying on network wide transactions.

2.6.2.1 Architecture of the YANG model

YANG models the structure of the data in a hierarchical way, using a tree structure where each node has a name and a either a value or a set of child nodes.

YANG structures data models into modules and submodules. Such a module can import data from other external modules, and include data from submodules. The hierarchy can be augmented, allowing one module to add data nodes to the hierarchy defined in another module. This augmentation can be conditional, with new nodes appearing only if certain conditions are met.

It is also possible to add constraints to be enforced on the data, restricting the appearance or value of nodes based on the presence or value of other nodes in the hierarchy. YANG has a set of built-in types, and has a type mechanism through which additional types may be defined.

Modules and Submodules A module contains three types of statements:

- module header statement: describes the module and gives information about the module itself
- revision statement: gives information about the history of the module
- definition statement: is the body of the module where the data model is defined.

A module may be divided into submodules, based on the needs of the module owner. One can use the "include" statement to reference material in submodules and the "import" statement to references material in other modules.

Data nodes YANG[Bjorklund, 2010] defines four types of nodes for data modeling:

- Leaf Nodes: A leaf node contains simple data like an integer or a string. It has exactly one value of a particular type and no child nodes.
- Leaf-List Nodes: A leaf-list is a sequence of leaf nodes with exactly one value of a particular type per leaf.
- Container Nodes: A container node is used to group related nodes in a subtree. A container has only child nodes and no value. A container may contain any number of child nodes of any type (including leafs, lists, containers, and leaf-lists).
- List Nodes: A list defines a sequence of list entries. Each entry is like a structure or a record instance, and is uniquely identified by the values of its key leafs. A list can define multiple key leafs and may contain any number of child nodes of any type (including leafs, lists, containers etc.).

2.6.3 ConfD

ConfD [Tail-f, 2016] is an on-device management framework which provides a set of interfaces to manage the device.

Figure 2.4 depicts the general structure of the ConfD framework. On the top side of the figure, the EMS/OSS and terminal icons represents high-level management systems and user operators that can interact with ConfD via the large set of interfaces supported by ConfD:

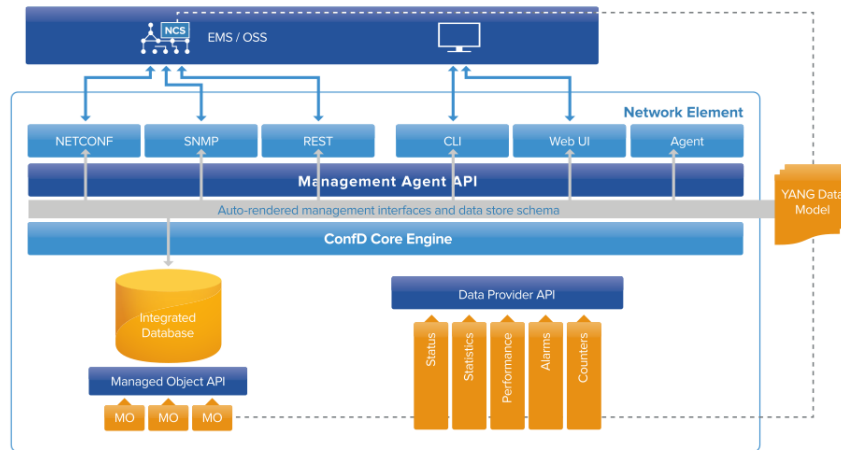


Figure 2.4: ConfD Architecture diagram

NETCONF, SNMP, REST, CLI, WebUI. The *Agent* block represents the applications that users can build to interact with the ConfD framework. ConfD agents can easily utilize others ConfD interfaces via the Management Agent API. And at the bottom of the diagram, we find the build-in data store along with an API to interact with it.

ConfD is model-driven, which means that it will automatically render all the management interfaces based on a YANG data-model describing/constraining input by the user. This is handled by ConfD core engine which "glues" all components together. The ConfD core engine is composed of different sub-systems.

The *Data Model Manager* organizes and keeps track of all YANG data models. *CDB Database* is the build-in database, and can be used to store configuration data. The structure of the database is directly derived from the YANG data models. CDB is a transactional ACID compliant database, which guarantees always to be in a consistent state.

Globally, ConfD regroups various management solutions into a single framework. Network operators can rapidly build their management solutions by only focusing on writing their personal ConfD agent (API in C, Python, Java, Erlang) and providing a YANG data model, while ConfD handles the rest.

To conclude this section, management solutions increased in complexity of implementation over time, with the aim to provide more and easier to use functionalities. Because the needs of network administrators are continuously evolving, new implementations improved over existing solutions to better suit those needs. Thanks to this management tools, this complex area of ICT that is networking, is made more accessible to new comers. Also, others alternative are Gen.it, Netomata, Solarwinds and Autonetkit.

2.7 Conclusion

The networking community is split among two approaches: centralized or decentralized control of the network. On the one hand, the decentralized approach is supported by set of robust distributed protocols along with a large number of legacy networks. In the other hand, the centralized approach has seen a growing support with the community behind the development of SDN. Although very promising SDNs have shown some limitation in deployment and scaling to large networks. There are some innovative solutions like Fibbing that aims at combining the

advantages of the two approaches or Segment Routing which, although not being a centralized solution, is particularly well suited to work with a SDN-like controller.

In the mean time, we saw that leaving the network solely under the control of distributed protocols can fail to provide some of the properties required by specific types of traffic (e.g. large bandwidth for video downloading, low delay for Voice over IP). Hence, this further motivates the needs for finer grained network control that can enables Traffic Engineering requirements in the network.

Finally, network administrators need some tools to efficiently specify those network requirements and perform appropriate network monitoring. This should give an insight of the landscape in which we developed our solution, which will be presented in the following chapters.

3.1 Introduction

The YANG models presented in this chapter aim at structuring the information concerning the network description and the requirements. The structure of the network for the two solutions (with Fibbing, and with Segment Routing) will be explained in section 3.2 and the differences between the two models will be explained in the same section. Section 3.3 will then cover the structure of the requirements. The motivation behind the chosen model for both the network and the requirements will be discussed in section 3.5. A short section (3.6), will go over the representation of Segment Routing routes. Finally, the last section will cover how these YANG models are used by the application.

3.2 Network Data model

This section covers the YANG models used to represent the structure of a network. In figure 3.1, a general representation on how we represent a network can be found. We will start by detailing this general representation and then the following subsection will explain the differences in the model for Fibbing and for Segment routing and motivate their differences.

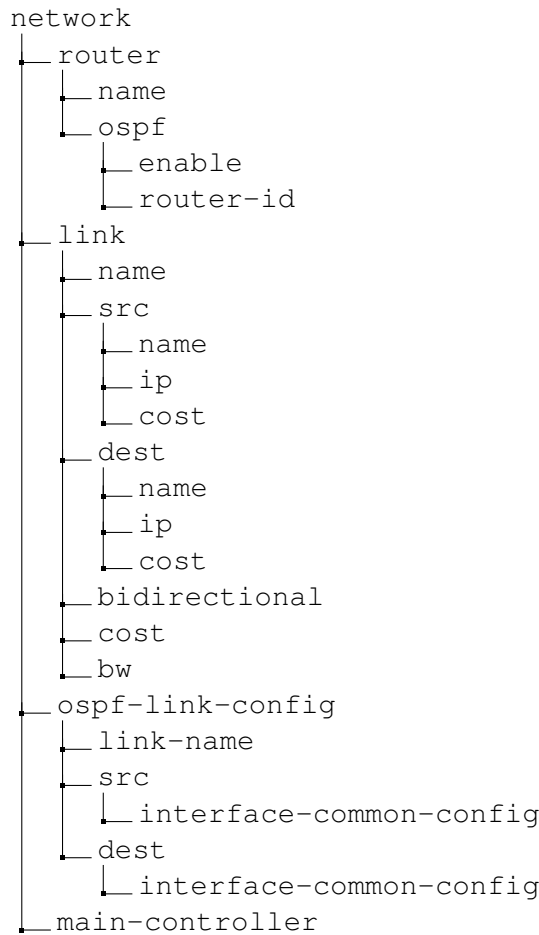


Figure 3.1: General YANG model structure of the network.

3.2.1 Generic Network Data model

The Generic model of the YANG model can be found on figure 3.1. This structure is the base of both solutions, but there still are differences specific to the implementation of the two different solutions. This section will first present the similar structure, then explain the specificities of both solutions.

At the top level, it can be seen that a *network* is globally composed of a set of *routers*, a set of *links*, a set of *ospf configurations* for those links, and a *main controller*. All these components will be presented in details in the following sections. The generic representation of the network topology by a set of links and set of routers is inspired by [Medved et al., 2017] that presents a Data Model for network topologies in YANG.

3.2.1.1 router

For the model, a network is composed of a set of `router`. Each of those routers must have a `name` (which uniqueness will be enforced by ConfD). Optionally, a given router can have `ospf` enabled, in which case the router must be assigned a unique `router-id`.

3.2.1.2 link

Along with its set of routers, the network is also composed of a set of `link`. Those links are connecting two routers (`src` and `dest`). Those routers must be routers of the network (previously declared) and are referenced by their respective `name`. For each of the routers composing

the link an `ip` address can be specified. This IP being the IP address (along with the prefix size) corresponding to the router interface for the link (e.g. 192.168.10.1/24 or fc00:42::1/64). It should also be noted that the `ip` attributes have a different type for the Fibbing solution than for the Segment Routing solution. Indeed, the Fibbing solution only supports IPv4 addresses while the Segment Routing solution supports IPv6 addresses. Along with this, some extra information about the link can be specified such as the `cost` or the maximum bandwidth (`bw`) of the link. The `bidirectional` attribute has a Boolean value indicating whether the link is bidirectional (i.e. `bidirectional=true`) or not.

Algorithm 3.1: Determining link cost

```

1  begin
2    cost_linkFromSrcToDest = DEFAULTVALUE
3    cost_linkFromDestToSrc = DEFAULTVALUE
4
5    if bidirectional == true
6      if hasValue(link_cost)
7        cost_linkFromSrcToDest = link_cost
8        cost_linkFromDestToSrc = link_cost
9      end
10   else
11     if hasValue(link_src_cost)
12       cost_linkFromSrcToDest = link_src_cost
13     end
14
15     if hasValue(link_dest_cost)
16       cost_linkFromDestToSrc = link_dest_cost
17     end
18
19   end
20 end

```

An overview of how the link costs are derived can be seen on Algorithm 3.1. In the case where the link is bidirectional only the `cost` attribute of the link is used to indicate that the links in both direction from `src` to `dest` and from `dest` to `src` have the same cost. If the `bidirectional` attribute is set to `false`, the link is considered asymmetric. The cost of the link in the direction from `src` to `dest` can be specified with the `src cost` attribute. Similarly the cost in the direction from `dest` to `src` can be specified with the `dest cost` attribute. It should be noted that we specify a default cost in the case users don't explicitly specified them. For the Segment Routing solution the default cost considered is 1. However, for the Fibbing solution the default cost is 2 in order to assure Fibbing compliance [Vissicchio et al., 2014, Vissicchio et al., 2015].

The cost value will be used by the application to derived shortest path information, and the maximum bandwidth is used for link monitoring. The last field of information needed, is a name that must be stipulated for the link. The name uniqueness will be enforced by ConfD and this will be used to reference the link. It should be noted that the name attribute, if carefully chosen, can convey meaning on its own. For example, for a link between router *LOSA* and router *HOUS* the name could be *LOSAtoHOUS*. So only by looking at the name of the link, you would know that the link has a `src` name *LOSA* and a `dest` name *HOUS*.

3.2.1.3 ospf-link-config

In order to configure OSPF for the network, the routers must first have OSPF enabled. Then to enable OSPF for some interfaces, `ospf-link-config` feature can be used. The `ospf-link-config` must have a `link-name` value which references the name attribute of a link. This will be used to map the `src` and `dest` attributes of the `ospf-link-config` with the `src` and `dest` attributes of a link. Meaning that, for example, the interface corresponding to the `src` attribute of a link named *LOSAtoHOUS*, will be the same interface referenced by the `src` attribute of

a `ospf-link-config` with `link-name` equals to *LOSAtoHOUS*.

For each of the referenced interfaces (`src` and `dest`), some OSPF *interface-common-config* values can be specified (see figure 3.2). The `interface-common-config` attributes that gather some of OSPF common configurations for an interface comes from the IETF draft [Yeung et al., 2017]. The attributes presented on figure 3.2 are the feature currently supported by the solution but the complete model of OSPF can be found in [Yeung et al., 2017].

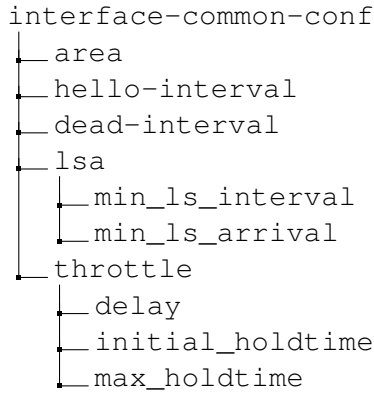


Figure 3.2: Structure of OSPF interface-common-config

3.2.1.4 main-controller

The `main-controller` referenced the node of the network that will act as the main controller of the network. The main controller is the controller that implements the architecture presented in chapter 4.

3.2.2 Features specific to the Fibbing Solution

We will now explain the features added to the generic model presented in figure 3.1 to support Fibbing. Those are the `fibbing-controller` and the `hosts-link`. These two features are presented in the following sections.

3.2.2.1 fibbing-controller

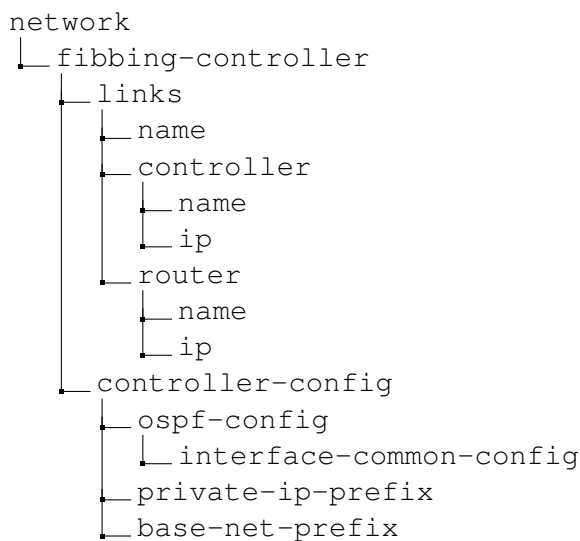


Figure 3.3: Structure of the Fibbing controller

The branch of the hierarchical tree that gathers the configurations specific to the `fibbing-controller` can be seen on figure 3.3. The declaration of a new controller is done by connecting the controller to a router of the network. This is specified via the `links` attribute. A controller/router link has two main attributes (`controller` and `router`) along with a `name` (being used to reference the link). Similarly to the `link` feature of the network, the two endpoints (`controller` and `router`) of the link have a `name` and an `ip` attributes. In the case of the `controller` the `name` is used to declare a new controller, while the `router` name attribute is used to reference a router. The `ip` attribute works in a similar fashion as for the network links.

Some Fibbing Controller specific configurations can be specified via the `controller-config` branch. Because the Fibbing Controller must be able to send OSPF messages to routers of the network, it needs to know some of the OSPF configurations. This can be stipulated via the `ospf-config` attribute, that follows the same structure as the `interface-common-config` (see figure 3.2). The Fibbing Controller needs to have two prefixes `private-ip-prefix` and `base-net-prefix` for its implementation. Therefore the `ip` addresses of the routers must not belong to those prefixes.

3.2.2.2 hosts-link

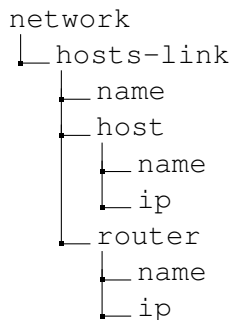


Figure 3.4: Structure of a hostlink

The `hosts-link` branch is used to represent the connection between a host and a router of the network and is depicted on figure 3.4. This feature is mainly there for simulation purposes and for easily prototyping with the simulation framework. The `hosts-link` branch has two main attributes (`host` and `router`) and a `name` used to reference the link. Similarly to the other link representation, each link endpoint (`host` and `router`) has two attributes `name` and `ip` that have the same purpose as explained before.

3.2.2.3 Motivation

The `fibbing-controller` feature is motivated trivially since the Fibbing Solution requires to have a Fibbing Controller. The representation of hosts as `hosts-link` is a design choice to represent destination of requirements. As explained, this is relevant in the context of a simulation. In reality, those "hosts" could, for example, be destinations prefixes advertised to edge routers via a protocol such as BGP.

Also, it should be noted that the current implementation considers that the main controller and Fibbing controller are running on the same node (see chapter 4). The main controller could have the extra attributes corresponding to the Fibbing controller specified explicitly under the `main-controller` branch instead of relying on a separate `fibbing-controller` feature. However, it should be noted that the current description of the `fibbing-controller` feature is relevant when the main controller and the Fibbing controller run on separate nodes.

3.2.3 Features specific to the Segment Routing Solution

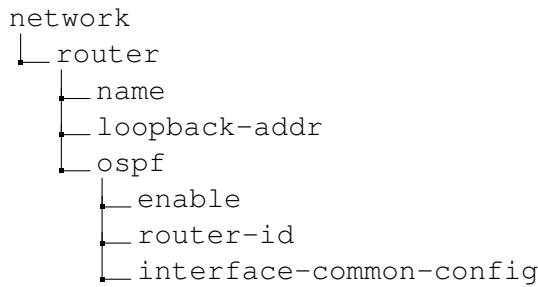


Figure 3.5: YANG model of a router specific to Segment Routing

As can be seen on figure 3.5, the router attribute of the model for the Segment Routing solution has some extra attributes with respect to the model for the Fibbing solution. This extra attributes are namely `loopback-addr` and `interface-common-config`.

The `loopback-addr` represents the loopback address of the router. This loopback address is used by the Segment Routing solution to reference the router when computing the minimum label stack (see chapter 4). Since all other routers need to know the loopback address of the other routers, it needs to be advertised with OSPF. Hence, the need for the `interface-common-config` corresponding of the OSPF configuration for this loopback address interface. It should be noted that although this `loopback-addr` field is mandatory for Segment Routing, a `loopback-addr` could be specified for the Fibbing solution as well. In which case the YANG structure of a router would be the same as for Segment Routing.

The last difference with the Fibbing solution is in the way to specify destinations for simulation purposes. For the sake of simplicity the Segment Routing solution represents destinations of requirements as a router. The `destinations name` attribute thus references a router name. The application will map the router name corresponding to a destination to its loopback address (see chapter 4).

3.3 Requirements YANG Data model

The figure 3.6 depicts the structure of the YANG data model for the requirements.

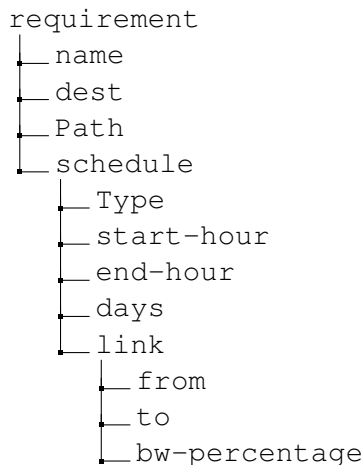


Figure 3.6: YANG Structure of Requirement

3.3.1 Principal attributes

A requirement must have the following principal (and mandatory) attributes: `name`, `dest`, `Path`, and `state`.

The `name` attribute will be used to reference the requirement (its uniqueness is enforced by ConfD).

The `dest` attribute indicates the given destination for which the requirement must hold. In the context of simulation, this `dest` attribute references the name of a destination (or a host depending if it is the Fibbing or Segment Routing solution). This indirection is convenient for simulations but in a real life scenario, this `dest` field could be assigned directly to a given destination prefix.

The `Path` attribute represents the list of routers that constitutes the hop-by-hop path that traffic must follow for the given destination. This actually references the name attributes of the routers of the network. The application will infer the indirection (see chapter 4) based on the network model.

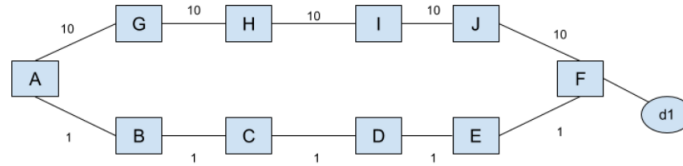


Figure 3.7: Abstract network

If the user does not wish to encode a strict path, he/she has the possibility to use a specific marker `*` that we denote as the **-router*. A path using this **-router* must have the marker `*` in between two other routers, because this **-router* symbol will indicate the application that this part of the path can be completed by the shortest path according to IGP cost between the two routers preceding and following the marker `*`. If there are multiple shortest paths, chapter 4 details how the application chooses only one of them. For instance in the network in figure 3.7, a path such as [**A G * F**], will force the traffic for a given destination prefix to go through **A-G**, then will follow the shortest path between router **B** and **F**. It should be noted that although the actual shortest path between **G** and **F** is [**G A B C D E F**], this would result into the complete requirement being [**A G A B C D E F**]. In order to avoid this scenario, and only have loop-free paths, the shortest path is actually computed on a subset of the original network where all edges already present in the requirement are removed (i.e. edge **A-G** in our example). The resulting requirement computed is thus [**A G H I J F**]. Therefore, a valid path must be loop-free, start with the first router for which the requirement must hold (typically an ingress routers), and end with the egress router for the specified destination (i.e. a router advertising the destination).

The `state` attribute can indicate one of three values: `running`, `not-running`, or `scheduled`. The two firsts are used to stipulate whether the requirement should be applied or not. The `scheduled` specifies that the requirement should be applied if certain conditions are met. These conditions can be specified in the `schedule` branch, and are explained in the next section.

3.3.2 Scheduled Conditions

There can be three types of scheduled conditions: `time`, `backup`, and `bandwidth`. One of these can be specified with the `Type` attribute.

requirement conditions Here is a summary of the scheduled conditions:

- **temporal conditions :**
 - **start_time** and **end_time**: the requirement will run only if **start_time** < *current_time* < **end_time**
 - **days**: a list of week days. The requirement will only run if the **current_day** ∈ **days**
- **bandwidth conditions:**
 - **link src**, and **link dest** indicating the link to be monitored
 - **link bw-percentage**: a percentage of bandwidth utilization. The requirement will run only if the **link bw-percentage** < **current_bw**
- **backup conditions:**
 - **link src**, and **link dest** indicating the link to be monitored. The requirement will run when the link from **src** to **dest** is down.

The time conditions can be enforced using the `start-hour`, `end-hour`, and `days` attributes. The `start-hour` and `end-hour` are time values, and `days` is a list of week days. Those time conditions stipulate that the requirement should only be applied between `start-hour` and `end-hour` if the current day belongs to the list of `days`. It should be noted that the time conditions must be specified for the other type of schedule conditions (`backup`, and `bandwidth`).

The backup conditions can be indicated with the `link` attribute. This attribute needs to reference a source (`from`) and destination (`to`) router corresponding to the two endpoints of a link. These backup conditions stipulate that the requirement should only be applied if the time conditions are met and if the link between the `from` and `to` routers is down.

The bandwidth conditions can be specified via the `link from` and `to`, and the `bw-percentage` attributes. The `from` and `to` are used to indicate a link of the network, and the `bw-percentage` indicates a percentage threshold of bandwidth utilization of the link. The bandwidth conditions stipulate that the requirement should only be applied if the time conditions are met and if the monitored bandwidth utilization of the link is above the specified threshold. It should be noted that, unlike requirements with backup conditions where the requirement is removed when the link comes back up, a requirement with bandwidth conditions will stay applied when the bandwidth utilization goes below the threshold. The reason is that the purpose of a requirement with bandwidth conditions is to alleviate some traffic from an overused link. So one of the intended results is to reduce the bandwidth utilization of the link.

In order to add new scheduled requirement types, one would need to add a new values for the `Type` attributes, and then add features corresponding to the newly added type.

3.4 Constraints in the YANG models

Along with the representation of the data, some constraints are also described in the YANG models. This section presents those constraints.

3.4.1 Uniqueness constraint

Let L be a list of tuples $(x_1, \dots, x_r)$ with arity r . Uniqueness or key constraint, $key_L = x_i$ indicates that

$$\forall (x_1, \dots, x_r), (y_1, \dots, y_r) \in L, x_i \neq y_i$$

This constraint is expressed by specifying a key attribute for a list. This attribute's value must be unique for each list entries, and thus can also serve as a unique reference for the given list entry. Here are the following lists from the YANG model that must satisfy this constraint:

- **router** (key=name): this will force routers to have each a different name. This is a desirable property since the router name is used, for example, in the requirement to indicate the path. The name attribute must thus uniquely identify a given router.
- **link** (key=name): this will force each link to have different names. As previously mentioned, this attribute will be used to uniquely identify a link by `ospf-link-config`.
- **requirement** (key=name): Requirements also have as key the attribute name. This attribute will serve to uniquely identify a given requirement (see chapter 4).

3.4.2 Reference constraint

Let L be a list of tuples $(x_1, \dots, x_r)$ and an attribute a , a references $L[x_i]$ indicates that if $a = r$

$$\exists (x_1, \dots, x_r) \in L, x_i = r$$

This constraint indicates that attribute a can only be assigned to an existing value of attribute x_i in the list L . Here are the following attributes that must satisfy this constraint:

- **link src name, link dest name** references **router[name]**: Those constraints assert that the `src name` and `dest name` attribute of a link must be assigned to an existing router name.
- **requirement Path** references **router[name]**: This constraint indicates that each element of the `Path` attribute must be an existing router name.

3.4.3 loop-free path constraint

Let P be path $[p_1, \dots, p_n]$, P is *loop-free* if

$$\forall p_i, p_j \in P \wedge i \neq j, p_i \neq p_j$$

As a reminder the `Path` attribute of requirement is assigned to an ordered list of router names. In order to avoid loops in the path, ConfD will check that the `Path` has not multiple times a router name in the list.

3.5 Motivation of the YANG data model

This section will discuss the motivation for the YANG data models of both network and the requirements.

3.5.1 Model of the network

An interesting point to retain from the network data model is its layered structure. Indeed, the approach taken here is to conceptualize the network representation from a high-level to low-level fashion. Meaning that a user can start with the generic representation of the network, which is a set of routers and a set of links between those routers. Thus thinking first about the global topology without going into the technical details of assigning IP addresses or router-IDs. Once the user's topology is defined, the user can start assigning the IP addresses and router-IDs and then, configuring the protocols.

This idea of conceptualizing the network in a layered and link oriented way, seems a good choice to reduce configuration incoherences. Another approach could be to provide the full configuration for each routers independently. While this approach would work, in our opinion, it is more likely to cause network configuration incoherences in practice. Our link oriented approach captures better the fact that those routers are intended to work together, as the network configurations are specified for each endpoints of a link.

The second interesting point to retain is the customizability of the model. Indeed, since the minimum information describing the network is the network connectivity (in other words the routers and the link between those routers), extending the model with new features would simply require to add new branches to the structure. For instance, if you want to add support for ISIS you would simply need to add a `isis-link-config` acting similarly to the `ospf-link-config`. The chapter covering the architecture (chapter 4 will cover into much details how to adapt the existing implementation into a real life solution). As an illustration of this customizability, we can see that there are little differences between the model for the Segment Routing and the Fibbing solutions.

3.5.2 Model of the requirement

First of all, it should be noted that this model for the requirement is meant to work in pair with the generic model of the network. This is because the application will at some point perform a mapping between a router name and either an IP address or a router-ID. This mapping is explained into details in chapter 4 covering the architecture of the solutions. The motivation behind this, is that router names are much more intuitive to use than router-ID or IP addresses where it can be difficult to keep track of which IP address is assigned to which router. So the cumbersome task of assigning the IP addresses and router-ID is done only when building the network model. So that when configuring the requirements users can only manipulate the router names, which is much more comfortable.

A second design choice for the requirement is to rely on hop-by-hop path to express a requirement instead of a Directed Acyclic Graph (DAG) for instance. While DAGs are more easy to reason with when graphically displayed (e.g. identify bottlenecks), they are not intuitive to encode in a CLI. This is the reason why we chose to express path requirement as a simple list of routers. Nevertheless, because of how convenient it is to work with DAG's to reason with requirements, we added a command in our CLI that graphically displays the DAG for a given destination. Meaning that if you configure multiple path requirements for a same destination, they will be represented a common DAG.

Another strong point of this model is its ability to specify scheduled conditions. Indeed, it seems judicious to propose such functionality as today's networks have a tendency to evolve over the time of the day. As example a network located in one timezone may be expected to have less traffic during the night, than during the day. Knowing this, a network administrator would want

to have different requirements at different times, so that network resources are better utilized. With our solution, he could schedule his requirements to meet this goal, instead of having to manually update the requirements. In a similar fashion upon failures or over-utilization of a link, the administrator could have scheduled requirements to directly cope with this event. This could leave him the time to either solve the failure or to find a potentially better suited requirements for the current needs of the network.

Having time conditions for *backup* and *bandwidth* conditions may seem strange at first, since one would expect those requirements to hold at all times. This is why you can configure a permanent scheduled requirements by setting the time conditions to be scheduled all days, during 24 hours. However, you should be aware that this kind of requirements require that the critical link (*from*, *to*) is explicitly monitored. This can add some overhead on the controller, and produce some unwanted traffic with status messages. This is a behaviour that a user may not want to have all the time. Thus the time condition might also be used to specify when a link should be monitored. If you know that there is little probability for link to fail or be overused during a certain period: it may be a waste of resource to monitor it during that period.

3.6 YANG model for Segment Routing Routes

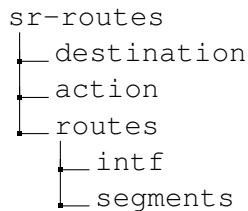


Figure 3.8: YANG Structure of Segment Routing Routes

Figure 3.8 presents the YANG structure for the Segment Routing routes. This model is used by the controller to represent the routes that the Segment Routing routers need to install. More specifically the controller will rely on this structure to convey the SR routes via NETCONF, in order to instruct the SR routers of the new SR routes.

A generic SR route is composed of *destination* which is the destination prefix for which the route must hold and a set of *routes*. Those routes are composed of *intf*, a default interface via which the route will start, and *segments* the set of segments through which the route must pass for the given destination. Specifying multiple *routes* for a same destination will be interpreted as being a set of ECMP routes. This means that for the corresponding SR router, there will be multiple routes possible for the destination and the traffic will be spread among those routes.

An *action* is also associated with the SR route. The *action* attribute is used by the controller to indicate to the SR router what action to perform with the corresponding route. The actions are *add* when the SR router needs to add a new routes to its set of SR routes, *delete* when it needs to remove the route from its set of SR routes, and *modify* when it needs to modify an existing route from its set of SR routes. This could happen when, for instance, the set of segments has changed but not the corresponding destination.

3.6.1 Motivation for the YANG data model for Segment Routing Routes

The YANG structure of the SR routes is a convenient way to organize the routes that will be transported via NETCONF and will need to be interpreted by the SR routers. This structure

follows the command used to install the route via the SR-IPv6¹ Linux Kernel implementation that we use. A more detailed explanation of how the route is installed into the IP routes of the SR router can be found in the chapter 4 covering the architecture of the Segment Routing solutions.

3.7 Interpretation of the YANG data models

The YANG data models described the previous section will serve as input to ConfD. Indeed, ConfD [Tail-f, 2016] is a model-driven framework that automatically renders its management interfaces from a YANG data model. The compilation and interpretation of the models are thus handle by ConfD. An example of the CLI interface rendered by ConfD based on the YANG data models presented in this chapter, can be found in the chapter 4 covering the architecture of the solutions.

3.8 Conclusion

The YANG model is a core component of our architecture as it describes the structure of the data and the constraints that network administrators will need to follow to specify both the network configuration and the requirements. It was designed to be structured and intuitive to ease this work. Our YANG model for the network is straightforward: first declaring the routers, than creating the links, then configure them and add the controller. The YANG model for the requirements is logical as the basis of all requirements is the same. An example of the CLI based on those YANG models can be find in chapter 5. It is also really easy to extend the models with new technologies (other than Fibbing and Segment routing), and with new types of requirements.

It should be noted that one of the main reasons why the Fibbing and Segment Routing solutions have differences between their respective YANG model is due to implementation specific features. Indeed, we first started by implementing the Fibbing solution and at the time the corresponding YANG model was tailored for this specific implementation. However, when implementing the Segment Routing solution we aimed at designing a unified YANG model. Nonetheless, some specificities were left in order to be compatible with the previous iteration of the implementation without having to make major changes in the code.

¹<http://www.segment-routing.org/>

4.1 Introduction

In this chapter, we present the architecture of our solutions. First we start by presenting the generic architecture that is common for both solutions in section 4.2. Then we present the specific features for the Solutions with Fibbing (section 4.3) and Segment Routing (section 4.4). Then section 4.5 gives some insight about how this generic architecture can be adapted into a real life solution.

4.2 General architecture of the solutions

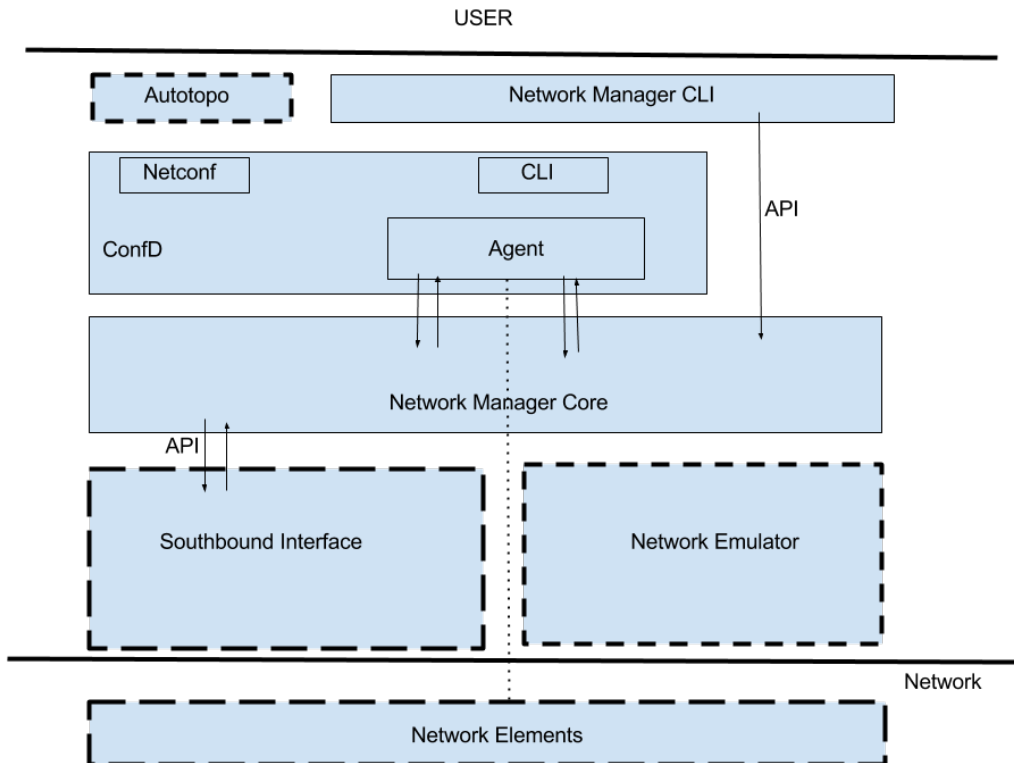


Figure 4.1: General Architecture

The general architecture of our solution is represented on figure 4.1. The blocks drawn in full lines are the part of the architecture that is the same for the two solutions. Those blocks will be explained next in details. The blocks in dotted lines are those who differ for the two solutions. We will explain their functions in this section and then the different implementations in the following sections.

It should be noted that this generic architecture was inspired by the Three-layers Architecture of SDN presented in the chapter 2, covering the State of the Art. Indeed, the *Network Elements* block would correspond to the *Infrastructure layer*, the *Southbound Interface* would correspond to the *Control layer*, and finally all blocks above the *Southbound Interface* would correspond to the *Application layer*, as will be seen in the following sections.

4.2.1 Network Manager CLI

The Network Manager CLI is the interface exposed, through which the user can interact with all the components of the application. The Network Manager CLI also gives access to the ConfD CLI that allows to enter the network and requirement configurations. The CLI is one of the interfaces automatically rendered by ConfD based on a YANG model. While chapter 3 covers in detail the structure of the YANG models, the chapter 5 shows a practical example of how the CLI is rendered.

The Network Manager CLI mainly presents a series of commands to the user. Those commands will trigger callback functions from the Network Manager Core API. As an example, these commands can be used to start the network, start the controller, view the set of current requirements, etc.

The principal purpose of the Network Manager CLI is to provide a centralized interface to the user, that enables easy access to all the functionalities that the application has to offer. In the following sections, those functionalities will be explained.

4.2.2 the Autotopo framework

Autotopo can be used to generate network typologies and the corresponding requirements by writing Python code. This framework allows to easily generate configuration files destined to be used with NETCONF. This *Autotopo* framework, although being very simple, is inspired by the Network Programming Language paradigm because it enables to declare a network topology using a programming language, which enables to write scripts to easily generate specific topologies. This framework is mainly used in the context of simulations as it enables the user to only express the high-level topology of the network (only the links between the network elements). The framework will then automatically assign the IP addresses for the corresponding topology and generate the corresponding NETCONF configuration files. In the context of a simulation, this configuration files will be sent to the NETCONF interface of ConfD so that the network configurations can be loaded into the application. Although the structure of the network varies slightly for the two solutions (see chapter 4 describing the structure of the YANG models), the *Autotopo* framework has a unified API to declare a network:

- `add_edge_by_type('A', 'B', OSPF_LINK, cost, bandwidth)` to create an OSPF link between router A and router B with a certain cost and a certain bandwidth.
- `add_edge_by_type('DEST', 'A', DEST_LINK)` to create the link between destination DEST and router A.
- `set_controller('Main-Ctrl')` to indicate which one of the nodes will act as the main controller (the main controller being the network element implementing the architecture described in this chapter).
- `add_edge_by_type('Ctrl', 'B', CTRL_LINK)` to create the link between the Fibbing controller Ctrl and the router B. Obviously the type CTRL_LINK is only valid for the solution with Fibbing.

Practical examples of how to use the *Autotopo* framework for the two solutions will be presented in the sections corresponding to the two solutions.

In a similar fashion, the *Autotopo* framework can also be used to generate NETCONF configuration files to express some requirements. Since the requirement structure is the same for both solutions, the Python syntax responsible for representing the requirement is the same as well.

4.2.2.1 Expressing requirement with Autotopo

Listing 4.1 shows an example of how to use the *Autotopo* framework to express some requirements. In order to use the framework, the user can extend the `SimulationSchedule` class and override the `build_requirement` function. The API to build a requirement is relatively straightforward as it closely follows the CLI syntax to configure requirements. The `next_timestamp()` function can be used to indicate that the next added requirements should be dumped in a new configuration files for each new timestamp. Because requirements in a same configuration file will be loaded into the application at the same time, building a schedule of requirements containing different files can be used to simulate the action of a user entering new requirements or modifying existing ones, at different moment in time.

For example, if you want to simulate the action of a user configuring a requirement :

```
requirement
name toDst2
dest Dst2
Path [ A C D ]
state running
!
```

Then some time later, the user wants to modify the path of the requirement in question to :

```
requirement
name toDst2
dest Dst2
Path [ A B D ]
state running
!
```

You would need to generate two separate configuration files because these two configurations for the same requirement cannot appear at the same time. So this is the motivation behind the `next_timestamp()` feature.

Because it is often easier to start from an example, a template file of how to express the requirement schedule is provided with the *Autotopo* framework.

```
class Simulation(SimulationSchedule) :
    def build_requirement(self):
        # configuring a simple requirement
        #                               Dest
        Req1 = Requirement('Hawaii',
        #                               Path
        #                               ['NEWY' , 'CHIC' , 'KANS' , 'HOUS' , 'LOSA'])
        #                               state
        Req1.set_state(RUNNING)
        # Adding requirement to the schedule
```

```

self.add_requirement(Req1)

# finish the set of requirement for this timestamp
self.next_timestamp()

# configuring a path with *
Req4 = Requirement('Sidney',
['NEWY' , '*' , 'LOSA'])
Req4.set_state(SCHED)
Req4.set_type(SCHEDBW)
Req4.set_start_time('10:00')
Req4.set_end_time('23:30')
Req4.set_days([MONDAY, TUESDAY, WEDNESDAY,
               THURSDAY, FRIDAY, SATURDAY, SUNDAY])

# configuring link with bw at 50%
link = ReqLink('ATLA', 'HOUS')
link.set_bw(50)
Req4.set_link(link)
self.add_requirement(Req4)

Req5 = Requirement('China',
['NEWY' , '*' , 'SEAT'])
Req5.set_state(SCHED)
Req5.set_type(SCHEDBACKUP)
Req5.set_start_time('10:00')
Req5.set_end_time('23:30')
Req5.set_days([MONDAY, TUESDAY, WEDNESDAY,
               THURSDAY, FRIDAY, SATURDAY, SUNDAY])

# configuring link
link = ReqLink('ATLA', 'HOUS')
Req5.set_link(link)
self.add_requirement(Req5)

```

Listing 4.1: Python syntax of Autopopo framework for the requirements

4.2.3 ConfD

In our architecture, ConfD is in charge of handling the configuration of both the network and the requirements. More specifically, ConfD provides the ConfD CLI to manually enter configuration data but as well the NETCONF interface that can also be used to send configuration data to ConfD (usually the configuration files being generated via *Autotopo*). While the NETCONF interface is not explicitly exposed via the Network Manager CLI, this interface is used when running automated simulations (see section 4.2.2). An example of how to use the ConfD CLI will be illustrated in section 5.3.1, and 5.3.2.

4.2.3.1 Yang model of ConfD

As explained in the State of the Art, it is primordial to provide a YANG model to ConfD, because ConfD will derive all the other components based on this YANG model. The YANG provided in this case, describes the structure of both the network configuration and the requirements.

ConfD will enforce this structure via its CLI, and via NETCONF. From the YANG model, ConfD will create and manage a database (called CDB) to store and update the configuration data. ConfD also provides the necessary API to interact with this database. Such API is used by the Agent which is explained in the next section. A complete overview of the YANG model structure is detailed in chapter 3.

4.2.3.2 ConfD Agent

Since ConfD stores the configurations in its own database, a way to access it is also provided through an API. This API is used by the ConfD Agent. From an abstract perspective, the ConfD Agent makes the bridge between ConfD and the Network Manager Core. More specifically, the ConfD Agent subscribes (via the ConfD API) to changes that may occur in the database. This means that whenever configuration data stored in the database changes, ConfD notifies the Agent which in turn can read the new data and send the updated configurations to the Network Manager Core. More specifically, the Agent has two subscription points: one for the network configurations and one for the requirement configurations. This is because while the network configurations are not expected to vary much over time, the requirements are. So these two subscription points will trigger two different callback functions that will read the changes for the corresponding subscription (either network configurations or requirements), and forward the changes to the Network Manager Core.

4.2.4 Network Manager Core

The Network Manager Core (NMC) is constituted of all the core code of the application, that bounds all the components together. As can be seen on figure 4.1, there are two major data flows in this architecture: the data flow of the network configurations and the data flow of the requirements (respectively on the right and left part of the vertical dashed line). The role of the NMC is to process from end to end those data flows and to deliver them to the correct end points: the Network Emulator for network configurations and the Southbound Interface for requirements.

The following subsection will present the main functionalities offered by the NMC.

4.2.4.1 Entry point

The entry points of the NMC are where the configurations sent by the ConfD Agent are received.

The format used to send and process the configurations is JSON. This format allows to easily express the structure of the network (i.e. list of routers, list of links, etc). Since most of the time the user will enter the configuration via the ConfD CLI, once he/she triggers a commit the ConfD Agent will be notified of the change via its subscription point. The ConfD Agent will read corresponding changes, organize the information following the JSON format, and forward the changes to the NMC corresponding entry point (one entry point for network configuration data, and one for requirement configurations). From there, the first thing to do is to check the validity of the received configurations.

4.2.4.2 Network configuration data flow

The NMC relies on a checker function to verify the network configurations. First, this function starts by checking that all the routers are connected. Second, it checks that all routers have an IP address assigned for their interfaces and that two routers do not use the same router-ID. Each time an error is detected, a visual feedback is displayed for the user. The user can consequently adapt the configuration until it is correct. It should be noted that these checks can be

easily extended. Indeed, the NMC calls the API function `checkNetworkConfig` that returns a Boolean value stating whether or not the checks succeeded. It should also be noted that ConfD will enforce some rules provided within the YANG model (e.g. links must be between two existing routers), preventing the commit of incorrect configurations. More information about the YANG model constraints are provided in chapter 3.

Once the configuration passes the checks, its content will be passed to the Network Emulator (see section 4.2.7). This will serve to build the network when the user initiates a `start_network` command. The Network Emulator will also be responsible to start the different Network Elements (see section 4.2.8).

The Network Emulator block will be explained in more detailed in the section covering the specificities of the two solutions (section 4.3 and 4.4) .

4.2.4.3 Requirement configuration data flow

The data flow of the requirements is a bit more complex than the network configurations data flow. Figure 4.2 depicts the data flow diagram corresponding to the requirements.

Similarly to the network configurations, the requirements will first go through a series of checks to verify their correctness. Such checks will verify that the path expressed by the requirement is indeed a valid path. A path is valid, if there is indeed an existing corresponding path in the network. As explained in chapter 3 describing the structure of the requirement, if the user relies on the `*` symbol when expressing a path, the checker function will verify that the usage of this `*` symbol is correct. As a reminder, the `*` symbol indicates that the symbol can be replaced by the shortest path between the two routers preceding and following the `*`. For instance, a path such as [**A B * C**], will force the traffic for a given destination prefix to go through edge **A-B**, then will follow the shortest path between **B** and **C**. It should be noted that it might be that there are multiple equal cost shortest paths for the `*` in the requirement. In this case, the algorithm computes first the shortest paths DAG from the destination toward the source. Then it builds the path from this DAG starting at the destination and always choosing the router with lowest router names according to the alphabetical order. By doing so, we ensure that the path computed is always deterministic. The alphabetical order is thus an arbitrary condition acting as a tie-breaker. Finally, if the checks fail, the user will have to retry entering the requirements via the ConfD CLI, similarly to the network configurations. Like for network configurations, the NMC relies on another API function `checkRequirementConfig` that can be extended to add new checks on the requirements.

processing the new requirements Once the requirements configuration succeeds the checks, the NMC will need to process this configuration. From the configuration changes received, the NMC will determine what actions must be made. Such actions can be for example, halting a running requirement, adding a new requirement, or just store new scheduled requirement into the scheduled store. Flags for each corresponding actions can be set during this step.

processing the flags After having some of its flags set up during the *process requirement* phase, a requirement will pass through a next step that will process the flags, and apply the appropriate actions. Such action may need to remove obsolete requirements from the scheduled store, or call the Southbound API.

Southbound API This Southbound API is a generic API that is similar for the Fibbing and for the Segment Routing solutions. Mainly composed of `add_requirement`, `remove_requirement`,

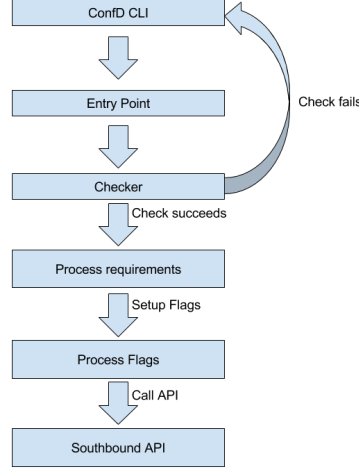


Figure 4.2: Data flow Diagram of requirements

and `commit_change` functions. The `add_requirement` (resp. `remove_requirement`) function is used to indicate a requirement needs to be added (resp. removed), and the `commit_change` to make the actions happen. The exact actions corresponding to this API are different for the two solutions, and thus will be explained in details in the section 4.3 and 4.4 covering the specifics of the solutions.

Thanks to this layered structure for the requirements data flow, adding new or modifying existing functionalities is relatively straightforward. One could add a new function in the *process requirement* phase that sets up new flags, then a specific handler for those flags can be added in the *process flags* step. Or even just relying on existing flags but modifying the corresponding handlers.

4.2.4.4 Scheduled requirement

The NMC is also in charge of processing the *scheduled requirements*. Indeed, the user has the possibility when configuring requirements, to specify requirements as type *scheduled* (see section 3 for more information). Those *scheduled* requirements are intended to be deployed when specified conditions are met. Such conditions can be for instance when the time is between a starting and ending point, or when the bandwidth of a link is above a specified threshold.

Upon processing of new requirements with type *scheduled*, the NMC will store those requirements in a specific store. The user can then choose to enable/disable the `auto_scheduled` feature. This feature will indicate the NMC to periodically go through the store of *scheduled* requirements and test the condition of the requirements (e.g. time, bandwidth, etc). Whenever all the conditions for a requirement are met, the `auto_scheduled` feature will automatically add the corresponding requirement to the Southbound Interface via the API. Similarly, whenever the conditions are not met anymore (e.g. the ending time specified for the requirement is reached), the `auto_scheduled` will automatically remove the requirement from the Southbound Interface.

Testing the conditions Since the NMC `auto_scheduled` is responsible for enforcing the aforementioned conditions, it is also responsible for gathering the related data. To do so, the NMC relies on the current time of the controller for the **temporal conditions**, and on SNMP for the **bandwidth** and **backup data**. As a reminder, SNMP (Simple Network Management

Protocol, see chapter 2 covering the State of the Art) is mainly used for network monitoring, and provides a query model on a set of running SNMP agents. Hence, the routers present in the network have each a SNMP agent running, and the controller can query these agents to obtain the information of the link status, and bandwidth. The NMC will thus, when periodically checking the conditions, query (*pull mode*) the different SNMP agents running on the routers. The period can be dynamically modified via the Network Manager CLI.

More specifically, the computation of the bandwidth utilization is performed according to a Cisco technical support post "how to calculate bandwidth use with Simple Network Management Protocol (SNMP)"¹, which states that the estimate bandwidth utilization can be obtained using:

$$InputUtilization = \frac{\Delta IfInOctets * 8 * 100}{\Delta t * BW}$$

$$OutputUtilization = \frac{\Delta IfOutOctets * 8 * 100}{\Delta t * BW}$$

where *IfInOctets* (resp. *IfOutOctets*) is the total number of octets received (resp. transmitted) on the interface, and *BW* is the interface's configured bandwidth in bits per second. The *BW* value being entered with the network configurations. Concerning the link status we rely on the SNMP *ifOperStatus* MIB variable, which indicates if the corresponding interface is *Up* or *Down*. It should be noted that this failure detection method works for point-to-point links. However, if for example the link between router **A** and **B** goes through switch **S** (as depicted in figure 4.3), and there is a link failure between the switch and router **B**, in this case the *ifOperStatus* of the interface of router **B** will be *Down* while the *ifOperStatus* of router **A** will still be *Up*.

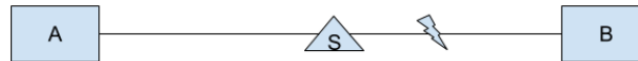


Figure 4.3: Link between router **A** and **B** goes through switch **S**, and there is a link failure between **S** and **B**.

Another solution could have been to make the routers notify the controller (*push mode*) whenever an event occurs (e.g. bandwidth above a threshold or a link is down). This would make the detection faster, but this has a series of drawbacks as well. Because scheduled requirements could change over time (i.e. old requirements could be removed, new requirements could be added), the controller would need a way to indicate the routers what links to monitor and what links to stop monitoring. One other solution could be to let the routers advertise all of their link status to the controller, but this would require the controller to be highly available to cope with all the incoming status updates, and would generate some unwanted traffic inside the network (e.g. updates from routers that don't need to be monitored). It should be noted that techniques such as OSPF-TE or IS-IS-TE can achieve that goal.

Our solution performs active monitoring on the links specified in scheduled requirements (i.e. type bandwidth or backup) by relying on SNMP. This was solely implemented to easily evaluate that the bandwidth and backup requirement work correctly. However, although this solution works, the detection time of failures can be high if the number of scheduled requirements is

¹<http://www.cisco.com/c/en/us/support/docs/ip/simple-network-management-protocol-snmp/8141-calculate-bandwidth-snmp.html>

important. This, because the NMC checks the status of the scheduled requirements sequentially. Nevertheless, the solution using SNMP is easy to adapt, as the NMC relies on two API function (`check_bw_status` and `check_link_status`) to monitor the links. One would only need to modify those functions to use other monitoring techniques.

4.2.5 Requirement deployment checks

The NMC will also preform a verification once the requirements are deployed in the network, this to ensure that they have the intended effect. To do so, the NMC last task will be to verify the path taken by some probe traffic toward the destinations specified by the requirements. The tool utilised by the NMC to that effect is *paris-traceroute*²³ for the Fibbing solution which is a variant of the classical network measurement and diagnosis tool, *traceroute*. The way *paris-traceroute* improved over its predecessor, is by its ability to support load balancing. We use *paris-traceroute* on the source router of a requirement with as destination the prefix of the destination of the requirement using the 'exhaustive algorithm' as probing algorithm with the default values. This causes the 'exhaustive algorithm' to return all the possible "load balanced" paths to the destination. At this point, if a requirement fails the *paris-traceroute* test, an error message will be displayed so that the user can investigate further the reason of the issue. The *paris-traceroute* test consists on parsing the output of the *paris-traceroute* between the first router of the requirement and the destination, and then compare the result with the path specified by the requirement.

For the Segment Routing solution, since Segment Routing encapsulates packets with the SR header, and due to the fact that the implementation of Segment Routing we used (SR-IPv6) was very recent, the traditional traceroute implementation did not fully support the processing of the SR header. Thus we chose another approach to test the correct deployment of the requirements. The test will consist on verifying that the path taken by the packets between each segments matches the requirement path. Indeed, we take the hypothesis that Segment Routing processes correctly the SR routes, and thus we compute the traceroutes between each pair of segments in the segment stack to form an aggregate path. We then compare the aggregate path with the path described by the requirement.

4.2.6 Southbound Interface

The Southbound Interface receives requirements from the NMC via the API. The final step to deploy the requirement is then to translate the requirements into a format that the network will understand and to send the appropriate messages in the network. As already mentioned, the way the Southbound Interface translate the requirements greatly depends on the solution. It will be discuss in more details in section 4.3 and 4.4 covering the specificities of the solutions. Nonetheless, this is the part of the application that will infer network understandable information (e.g. router-ID, IP address) from the router names present in the requirements received through the Southbound API. The requirements only rely on router names to express the path of the requirements. So at this point in the requirements data flow, the NMC has only manipulated requirements in high-level format (i.e. with router names). One of the main purpose of the Southbound Interface is then to, from the network configurations, derive the relevant information to map the router names into network understandable data. By making this mapping at the Southbound Interface, it allows the application to only manipulate requirements in high-level format.

²<https://paris-traceroute.net/>

³<http://manpages.ubuntu.com/manpages/trusty/man8/paris-traceroute.8.html>

4.2.7 Network Emulator

The Network Emulator will, in the context of simulation, be responsible to start the virtual routers and eventually create corresponding name spaces. It should be noted that this block of the architecture can either be omitted or modified in a real life scenarios. Indeed, if you want to integrate this solution with an already (or externally) configured network, this block can be omitted altogether. And if you want to rely on this solution to also configure your network from a centralized place, you would need to adapt the Network Emulator to that effect. Section 4.5 will cover into much details how to adapt the solution for real life scenarios.

4.2.8 Network Elements

The Network Elements represent all the elements running in the network that the main controller can interact with. Some of these elements will be different for the two solutions and thus will be detailed in the section 4.3 and 4.4 covering the specificities of the solutions.

4.3 Solution with Fibbing

We will now describe the dotted elements of the architecture in the case of the solution with Fibbing. The architecture of the Fibbing solution with the corresponding dotted elements is shown in figure 4.4.

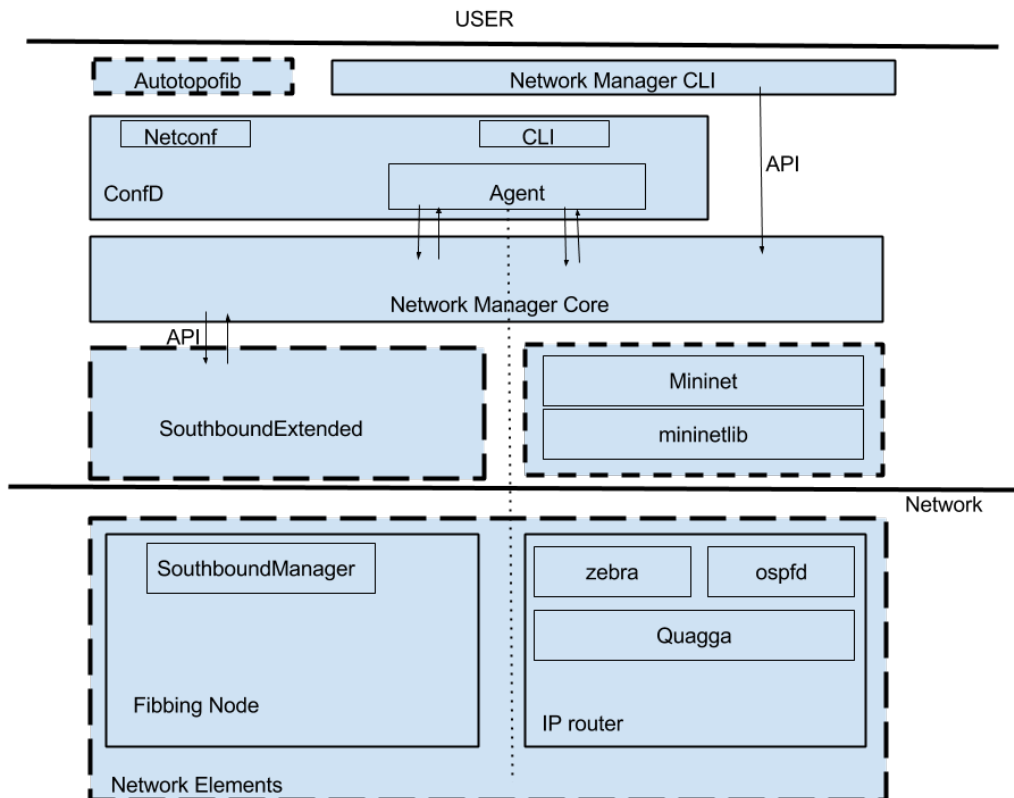


Figure 4.4: Architecture of Fibbing

4.3.1 Autotopo for fibbing

The *Autotopo* block is replaced by the *Autotopofib* block in the Fibbing solution. As mentioned before the *Autotopofib* is used to generate network topologies and requirements by writing Python

code. The way to describe requirements is the same for both solutions and was described in section 4.2.2.

4.3.1.1 Example of Autotopofib

```
class Test(Topo) :
    def build(self) :
        self.add_edge_by_type('A', 'B', OSPF_LINK, cost=10)
        self.add_edge_by_type('A', 'C', OSPF_LINK, cost=1)
        self.add_edge_by_type('C', 'D', OSPF_LINK, cost=1)
        self.add_edge_by_type('B', 'D', OSPF_LINK, cost=1)

        self.add_edge_by_type('S', 'A', DEST_LINK)
        self.add_edge_by_type('Dest', 'D', DEST_LINK)
        self.add_edge_by_type('Ctrl', 'D', DEST_LINK)

        self.add_edge_by_type('C1', 'C', CTRL_LINK)
        self.add_edge_by_type('C2', 'B', CTRL_LINK)

        self.set_controller('Ctrl')
```

Listing 4.2: Python syntax of Autotopofib framework for the network

Listing 4.2 shows an example of how a user can describe a network using the *Autotopofib* framework. As a reminder, the framework will, based on the description of the topology, generate the corresponding configuration file destined for the NETCONF interface of the Fibbing solution. The *Autotopofib* framework is mainly used in the context of simulation, as users don't have to bother choosing IP addresses or router-ID, the framework will automatically assign these fields.

4.3.2 Southbound Interface for Fibbing

The Southbound Interface for Fibbing is replaced by the Southbound Extended block. The Southbound Extended is mainly responsible for translating the *path requirements* into *DAG requirements*. A *path requirement* is expressed as a continuous path toward a destination, and a *DAG requirement* is expressed as a Directed Acyclic Graph towards a destination. A *DAG requirement* can be obtained from a set of *path requirements* by merging all paths toward a same destination. The reason being that users configure path requirements via the ConfD CLI while the Fibbing Node processes DAG requirements via its own Southbound interface.

The second responsibility of the Southbound Interface is to perform the mapping between the router names and some network understandable information (see section 4.2.6). In the case of the Fibbing solution, the Southbound Interface of the FibbingNode needs to receive the DAG requirements expressed with router-IDs. The Southbound Interface of the Fibbing Solution (i.e. Southbound Extended) will thus map the router names present in the paths received via its Southbound API, into the corresponding router-IDs. This router name/router-ID mapping is directly derived from the network configurations, hence it is primordial for the application to have the correct configurations.

4.3.3 Network Emulator for Fibbing

The block containing the Network Emulator is composed of Mininet and *mininetlib* in the case of the Fibbing solution. Mininet is used for representing the network and mininetlib is used

to configure and start the different Quagga routers (see section 4.3.4). As mentioned before in section 4.2.7, the Network Emulator is mainly used in the context of simulation. In a real life scenario, this block could be modified. The only additional purpose provided by Mininet is to easily provide a view of the network for the main controller. Indeed, especially for when the Southbound Interface needs to perform the router names/router-IDs mapping Mininet facilitates the process. Additionally, it should be noted that in the context of a real life scenario, any framework representing a network could work as well.

4.3.4 Network Elements for Fibbing

The network elements are composed of Fibbing controllers and IP routers.

Fibbing controllers will push Fibbing lies⁴. A Fibbing controller takes one DAG per destination as input via its SouthboundManager, computes the set of OSPF messages ("Fibbing lies") needed to be sent to the routers so that they enforce the requirements, and then actually injects those messages. As mentioned in [Vanbever and Vissicchio, 2014b], Fibbing was conceptualized to work with traditional IP routers. So it is expected to mostly find traditional IP routers as main network elements. When using the Mininet framework, there will be another type of network element present: a host. As Mininet makes an explicit difference between the hosts and the routers, we chose to represent the destinations as hosts. This is just a convenient way to simulate destinations, for which we can perform some connectivity tests and traceroutes.

Although the current implementation implicitly considers that the Fibbing controller and main controller are running on the same node since the main controller directly calls the API of the Southbound Interface of the Fibbing controller, we could imagine a decentralized solution as well. Similarly to the solution with Segment Routing there could be a ConfD Agent running on the Fibbing Controllers. NETCONF could then be used to carry the DAG requirements from the main controller to the Fibbing Controllers. The ConfD Agent would thus be in charge to process the DAG requirements and to call the Southbound API of the Fibbing Controllers.

4.4 Solution with Segment routing

We will now describe the dotted elements of the architecture of the solution with Segment Routing. The architecture of the Segment Routing solution with the corresponding dotted elements is shown in figure 4.5.

4.4.1 Autotopo for Segment Routing

The *Autotopo* block is replaced by the *AutotopoSR* block. Like for the Fibbing solution, the AutotopoSR allows us to generate topologies and requirements by describing them in Python code. The way to describe requirements is the same for both solutions and was described in section 4.2.2.

4.4.1.1 Example of AutotopoSR

Listing 4.3 shows an example of how a user can describe a network using the *AutotopoSR* framework.

Similarly to the *Autotopofib* framework, users don't need to bother with IP addresses and router-IDs, these fields will be assigned automatically by *AutotopoSR*. But it should be noted

⁴the controller announce additional nodes and links, which we call fake nodes and fake links, in the network in order to respect the current active requirements(see chapter 2)

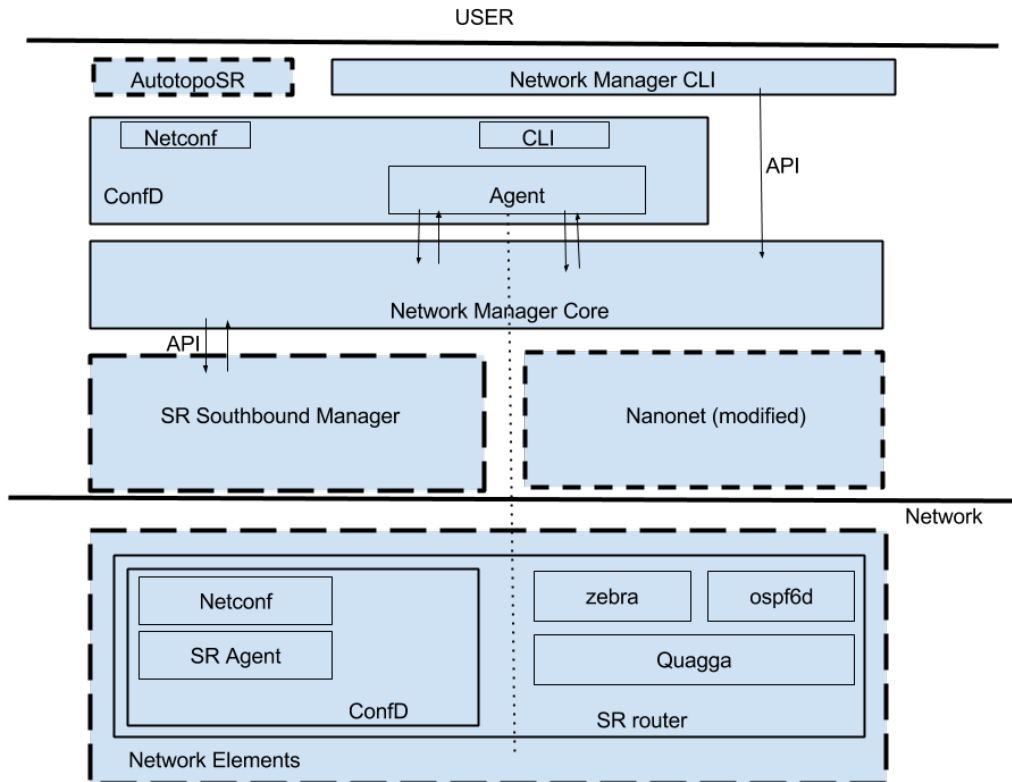


Figure 4.5: Architecture of Segement routing

nonetheless that the Fibbing solution works with IPv4 addresses, while the Segment Routing Solutions works with IPv6 addresses.

```
class Test(Topo) :
    def build(self) :
        self.add_edge_by_type('A', 'B', OSPF_LINK, cost=10)
        self.add_edge_by_type('A', 'C', OSPF_LINK, cost=1)
        self.add_edge_by_type('C', 'D', OSPF_LINK, cost=1)
        self.add_edge_by_type('B', 'D', OSPF_LINK, cost=1)

        self.add_edge_by_type('S', 'A', DEST_LINK)
        self.add_edge_by_type('Dest', 'D', DEST_LINK)
        self.add_edge_by_type('Ctrl', 'D', DEST_LINK)

        self.set_controller('Ctrl')
```

Listing 4.3: Python syntax of AutopopoSR framework for the network

4.4.2 Southbound Interface for Segment routing

In the architecture of the SR solution, the Southbound Interface is called the SR Southbound Manger and has an identical interface (API) to the Southbound Extended of the Fibbing solution. The SR Southbound Manger also takes as input *path requirements* (see section 4.3.2), but since the Segment Routing solution does not rely on an existing controller such as the Fibbing Controller, the Southbound Interface of the Segment Routing solution will also be in charge of

installing the SR routes on the network elements composing the network. For this purpose, the Southbound Interface will need to first translate the path requirements received by its Southbound API into SR routes, instruct the routers of these SR routes and finally assure that the routers correctly installed (or removed) the SR routes.

The following sections will explain how the Southbound Interface achieves those objectives. More specifically, section 4.4.2.1 presents how the Southbound Interface derives the SR routes from the path requirements, section 4.4.2.2 explains how to generate the corresponding minimum SR segments stack to achieve the SR routes, section 4.4.2.3 details how to carry the instruction to install (or remove) the SR routes to the routers, and finally section 4.4.2.4 covers the way to ensure correctness among the requirements viewed by the user and the actual SR routes on the routers.

4.4.2.1 SRSouthboundManager

The responsibility of the SR Southbound Manager is to translate the path requirements entered by users into Segment Routing routes. For Segment Routing, these SR routes must be installed on an initial source router (see chapter 2 covering the State of the Art). Hence the source router will always be the first router in the requirement path. For example, if we have $[A\ B\ C\ D\ E]$ as requirement path, the source router on which we need to install the SR route is **A**, and the list of segments corresponding to that route must correspond to $[B\ C\ D]$.

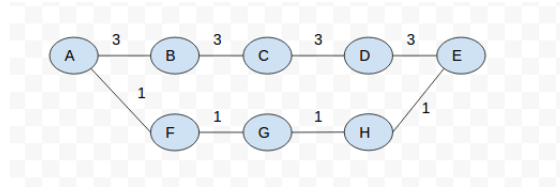


Figure 4.6: Abstract Segment Routing network

An important role of the SR Southbound Manager is to perform the mapping between the router names present in the requirements and network understandable information. In the general case, a SR route's list of segments is expressed with the list of loopback addresses of the routers that compose the path. But since the packet is routed between active segments, it is not always necessary to specify each loopback address of the routers of the path. For example, if we consider the network at figure 4.6, and we have as requirement path $[A\ B\ C\ D\ E]$, we don't necessarily need to express the segment stack as $[< loopbackB >, < loopbackC >, < loopbackD >, < loopbackE >]$. Indeed, according to the IGP cost, the shortest path between **C** and **E** already passes by **D**. So instead, we could have just expressed the segment stack as $[< loopbackB >, < loopbackC >, < loopbackE >]$. There is a benefit to reducing the size of the segment stack since each segment is a IPv6 address that is 128 bits long. The section 4.4.2.2 will explain how to optimize the size of the segment stack.

Finally, in our example ($[A\ B\ C\ D]$) the *interface* corresponding to the SR route on router **A**, is simply the interface of **A** of the corresponding to the link **A-B**. This information is directly derived from the network configurations.

4.4.2.2 Simplifier

Simplifier is used to create the minimum segment stack given a SR route. The idea is that the algorithm ([Aubry et al., 2016]) will only add a segment to the segment stack if the shortest path between the source and the destination is not passing through the right routers. Once a router is added to the stack, the shortest path is computed between the source and the newly

added router and between the newly added router and the destination. This is repeated until the right path is obtained.

Some other complications have to be taken into account. First, in the case of equal cost paths between 2 routers, a segment must be added to the stack to force the traffic on the right path. Secondly, if the shortest path between 2 adjacent routers is not going through the link between the 2 routers, we add an adjacency segment. As example, lets look at figure 4.7. If we have as requested path $[S, A, B, D]$ we will need to force the use of the link between **A** and **B** using an adjacency segment (**A-B**). An adjacency segment is expressed by placing in the segment stack the loopback address of the first router of the adjacency segment (in our example that would be **A**), then adding the address of the interface of second router of the adjacency segment that corresponds to the link that the adjacency wants to force (in our example that would be the interface address of the router **B** corresponding to the link **A-B**).

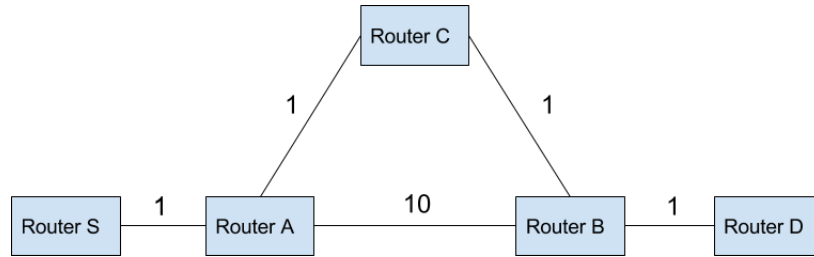


Figure 4.7: Case where we will need an adjacency segment

Implementation Our algorithm to compute the minimum segment stack for a corresponding requirement path is composed of two principal functions: `Simplify`, and `find_segment` whose pseudo-code can be found on Algorithm 4.1 and 4.2 respectively.

Algorithm 4.1: Simplify

```

1  input: Req (Requirement Path)
2  output: segment stack
3  begin
4      FinalDest  $\leftarrow$  last(Req)
5      src  $\leftarrow$  first(Req)
6      dest  $\leftarrow$  FinalDest
7      CurrentReq  $\leftarrow$  Req
8      SegmentStack  $\leftarrow$   $\emptyset$ 
9      finished  $\leftarrow$  false
10
11     while  $\neg$  finished
12         node  $\leftarrow$  find_segment(CurrentReq, src, dst)
13         if node  $\neq \emptyset$ 
14             if node.value  $\neq$  FinalDest or node.type == adjacency
15                 SegmentStack.append(node)
16                 src  $\leftarrow$  node.value
17                 CurrentReq  $\leftarrow$  Req[index(src)  $\rightarrow$  index(FinalDest)]
18                 dst  $\leftarrow$  FinalDest
19             end
20
21             if node.value == FinalDest
22                 finished  $\leftarrow$  true
23             end
24         else
25             remove_last(CurrentReq)
26             dst  $\leftarrow$  last(CurrentReq)
27         end
28     end
29     return SegmentStack
30 end

```

Algorithm 4.2 presents the naive implementation of the function `find_segment`. For the comprehension of the Algorithm, we present some the helper functions used:

- `Dijkstra(src)`: computes the shortest distance of all nodes in the graph with respect to a source node.
- `BuildPaths(src, dst, DijkstraOut)`: computes, based on the output of the *Dijkstra* function, all shortest paths between a source and a destination in a DAG format.
- `isIncluded(CurrentReq, ShortestPathDag)`: computes whether the current requirement is included in the shortest paths DAG computed by the *BuildPaths* function.
- `hasNodeDegreeToBig(ShortestPathDag)`: returns the first node (starting from the current source) that has a in-degree > 1 . For example, if the DAG computed by the *BuildPaths* function is the one depicted in figure 4.8, node **C** and **G** have an in-degree > 1 , and the first node computed by *hasNodeDegreeToBig* will be **C**.

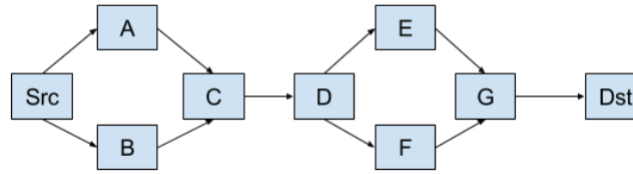


Figure 4.8: example of DAG computed by *BuildPaths* function

Algorithm 4.2: `find_segment` (Naive implementation)

```

1  input: CurrentReq
2  input: src
3  input: dst
4  output: segment
5  begin
6      DijkstraOut  $\leftarrow$  Dijkstra(src)
7      ShortestPathDag  $\leftarrow$  BuildPaths(src, dst, DijkstraOut)
8
9      if isIncluded(CurrentReq, ShortestPathDag)
10         node  $\leftarrow$  hasNodeDegreeToBig(ShortestPathDag)
11         if node  $\neq \emptyset$ 
12             return Node(value=dst, type=normal)
13         else
14             if size(CurrentReq) == 2
15                 return Node(value=dst, type=adjacency)
16             end
17
18             prevNode  $\leftarrow$  previous(CurrentReq, node)
19             return Node(value=prevNode, type=normal)
20         end
21     else
22         if size(CurrentReq) == 2
23             return Node(value=dst, type=adjacency)
24         else
25             return  $\emptyset$ 
26         end
27     end
28
29 end

```

As can be seen on line 6 of Algorithm 4.2, every time there is a call to `find_segment` function, the *Dijkstra* for the input source will be computed. However, this may require to compute the *Dijkstra* for sources that were already computed. This is very ineffective, especially since

Dijkstra is quite expensive for large graphs.

An optimization could be to pre-compute the output of the *Dijkstra* function for all nodes in the graph and store the result in a Dictionary. While this would make the `find_segment` function faster (since a dictionary access can be done in $\mathcal{O}(1)$). However, for large graphs, this would make the initialization very long and potential generate a huge storage cost. Moreover, it may be that you would have computed the *Dijkstra* and stored the output for source nodes that were never required by the `find_segment` function.

We believe that a good compromise between the two ideas, is to lazily compute the *Dijkstra* and store the output for the nodes as they are required by `find_segment`. Algorithm 4.3 shows what are the modifications needed to be brought to the naive implementation of `find_segment` (Algorithm 4.2) aiming at achieving the lazy implementation. More specifically, the line 6 of the Algorithm 4.2 must be replaced by the lines in Algorithm 4.3.

Algorithm 4.3: `find_segment` (Lazy implementation)

```

1   if  $\neg$  cacheMap.exist(src)
2       DijkstraOut  $\leftarrow$  Dijkstra(src)
3       cacheMap.put(src, DijkstraOut)
4   else
5       DijkstraOut  $\leftarrow$  cacheMap.get(src)
6   end

```

4.4.2.3 Netconf Manager

The principal role of the *Netconf Manager* is to transport and remotely instruct the SR routers to add or remove SR routes. To that effect a combination of NETCONF and a specific ConfD Agent is used.

In order to transport the SR routes and the instructions of how to handle these SR routes, the NETCONF protocol is used. Since ConfD natively exposes a NETCONF interface, at the endpoint we rely on another ConfD Agent (referred as `sagent`) that runs on each SR routers. The YANG model representing both the SR routes and the actions to perform with these SR routes is presented in chapter 3 covering the YANG data models.

As a reminder, the possible actions are **add** to install a new SR route, **delete** to remove an already installed SR route, and **modify** to modify an already installed SR routes. The ConfD Agent will read the SR routes received and perform the corresponding actions, using the SR-IPv6 kernel commands.

4.4.2.4 Transaction-like updates

One of the main advantages of using the NETCONF protocol is that it natively notifies whether the update was successful or not. This enables the Netconf Manager to know whether a given requirement has successfully been installed on a remote SR router. This information is useful to enable the application to tell users which requirements is currently running. So, for example, if a user wishes to remove a requirement **[A B C]** but the SR router **A** is not reachable, the remove operation of the user will be aborted and the user will be notified. This also means that the high-level requirements seen by the user are only updated when the application knows that the underlying SR routes have successfully been updated. This allows the main controller to keep an up-to-date view of the active requirements translated into active SR routes.

4.4.3 Network Emulator for Segment Routing

The *Network Emulator* used for the Segment Routing solution, is a modified version of Nanonet⁵. Nanonet enables the build of a virtual network using only static routes, pre-computed based on input topology costs via the Dijkstra algorithm. We modified Nanonet to just create the virtual network and we rely on OSPF to advertise the routes. In our case, it is Quagga that will implement the OSPF protocol.

4.4.4 Network Elements for Segment Routing

The Network Elements present for the Segment Routing Solution are mainly SR routers. As can be seen on the figure 4.5, the SR routers have two main functionalities.

The first functionality that a SR routers must support is to advertise its own IPv6 loopback address and learn the loopback address of the other routers, this is handled by the OSPF protocol. In this case, Quagga and more specifically its daemon, `ospf6d`, will be in charge of handling this feature. On the other hand, a SR router also runs an instance of ConfD and its Agent (`sragent`) to handle the NETCONF messages. Those messages give instruction about SR routes to add or to remove. Since the routers must support Segment Routing they need to have the SR-IPv6 linux kernel, so that to install SR routes and to process the SR headers for forwarding.

4.5 Toward a real life solution

This section will explain how to adapt the architecture presented in this chapter into a real life scenario. More specifically, we will present the changes to be implemented based on the general architecture (figure 4.1 in section 4.2) to stay generic with respect to the two solutions.

In our opinion, two cases need to be distinguished: either the architecture is intended to be used to centrally configure a network as well as to specify its requirement, or the architecture is intended to be used with an already configured network.

4.5.1 First case

It case corresponds to the controller architecture also used to centrally configure the network.

First, as already mentioned in the chapter 3 covering the YANG models, users would need to enter the full configurations for the underlying network. As a reminder, the provided YANG model of the network allows to specify some generic configurations of the network (i.e. what are the routers present in the network, and what are the links between those routers) but as well some protocol related configurations. For the time being, only the OSPF protocol is present in the YANG model, but thanks to the modular structure of the YANG model of the network, supporting other protocols should not be too complicated. Indeed, since the general structure of the network would not change (i.e. routers and links between routers), one would only need to add a new branch to the YANG model (under the `network` attribute) to represent the new protocol's configurations for the links. So, for example, if we want to add support for the ISIS protocol, we could add a `isis-link-config` branch and similarly to the `ospf-link-config`, we could reference the `link` name attribute to indicate that the given configurations should be mapped to this particular link. For the destinations, since it is likely that the destination prefixes will be learned via a network protocol such as BGP, one could just build a script that automatically load the prefixes from the BGP routing table into ConfD so that the prefixes can be referenced

⁵It is based on Mininet for the concept as originally developed by David Lebrun for segment-routing.org.

by the `dest` field of the requirements. It should be noted that YANG is popular data model in networking and that most of the protocols have some existing YANG models. Thus in order to support a new protocol, there is *a priori* no need to write a new YANG model on your own. The ConfD Agent would obviously need to be updated, as well as the checker functions to cope with the changes of the YANG models.

Secondly, the *Network Emulator* block should be replaced by a configuration interface that can remotely instruct the network elements of the configurations. As an example, the NET-CONF protocol is particularly well suited for this task, as it enables both to carry configuration information and to remotely activate services on the network elements. Nonetheless, the Network Manager Core would still need to have access to a "Mininet-like" framework that enables easy representation of the network and easy access to configuration information such as IP addresses and router-IDs.

Third, for monitoring of the network (required for bandwidth and backup conditions of the scheduled requirements), if one wishes to rely on some other technique than SNMP, only the `check_bw_status` and `check_link_status` functions from the Network Manager Core monitoring API, would need to be modified.

Finally, since in this section it is intended that the architecture works with real life routers, it is expected that the network elements would change as well. The only constraint would be to support the core features. In the case of Fibbing, since the current version of the controller support only OSPF, the network elements should have OSPF running. In the case of the Segment Routing solution, while any protocol should *a priori* work to learn the loopback addresses of the other routers present in the network, the network elements should still run an instance of ConfD and the corresponding `sagent`. In any case the network elements must support the SR-IPV6 Linux kernel implementation to work with this version of Segment Routing.

It should be noted that while the *Autotopo* framework was designed to be used in the context of simulation, it could be updated to cope with the modified YANG models, and to enable to specify all the configurations via the Python programming language.

4.5.2 Second case

The second case corresponds to the controller architecture only used to specify, represent and enforce requirements.

First, on the contrary to the first case, the controller would not need the full protocol configuration information. Indeed, in order for the controller to enforce the requirements it must have some general view of the network but no protocol specific information. For this general view of the network the main controller would need to have access to the general structure of the network (i.e. what are the routers present in the network, and what are the links between those routers) and the IP addresses and router-IDs of the routers are also required in order to perform the mapping between the router names and the network understandable information.

Concerning the *Network Emulator* block, this could be either omitted or only replaced by a "Mininet-like" framework that enables easy representation of the network, and easy access to configuration information such as IP addresses and router-IDs.

For the changes that need to be brought to the network monitoring, network elements, and eventually to the *Autotopo* framework, they are the same as described in the first case (section 4.5.1).

4.5.3 Controller replica

At this point, this chapter presented the generic Architecture for the main controller, but as with this kind of centralized controller managing the whole network, it is crucial to have some additional blocks preventing the main controller of being a single point of failure.

First, the controller should be replicated, and the replicas located at strategic locations across the network. There should be some *controller synchronization* block added to the original architecture that handles synchronization of the requirements among the replicas via controller-to-controller communication. It should be noted that there are numerous distributed system algorithms that exist to achieve this functionality.

Second, when a replica fails it should restore its previous state upon reboot. Concerning the requirements and network configurations, the state can be directly retrieve from CDB (ConfD internal database) that is resilient to failures. Concerning the information specific to the state of the network (e.g. what are there any down links, what are the running requirements), the rebooted replica should be able to get these information from the other active controllers. Usually this feature is included in distributed system algorithms for synchronization. In case of all replicas failure, the information concerning which are the running requirements would need to be retrieved from the network elements themselves.

4.5.4 Support for new ideas

Up until now, we only explained how this controller architecture can work with Fibbing and Segment Routing, but the generic architecture presented in figure 4.1 could theoretically work with other protocols.

Indeed, thanks to this layered architecture, the Configuration layer (ConfD) and the Network Manager Core layer only manipulate the requirements in a high-level format (i.e. only router names are used). As a reminder it is the job of the Southbound Interface to actually translate the high-level requirements into network understandable information, then to concretely enforce the requirements. This is done either via an existing controller like in the case of Fibbing or explicitly like in the case of Segment Routing. In any case in order to support other Traffic Engineering methods (or even OpenFlow), one would basically need to replace the Southbound Interface block, and implement the corresponding handlers for the new protocol. The only constraint would be to implement the Southbound API so that to be compatible with the Network Manager Core.

4.6 Implementation

The current implementation of the two solutions can be found on *Github* at (github.com/NetworkManagerCLI/NetworkManagerCLI). The solution with Fibbing is constituted of around 3850 lines of Python (resp. 810 lines of C) for the Network Manager (resp. the ConfD Agent). And the solution with Segment Routing is composed of around 4660 lines of Python (resp. 1080 lines of C) for the Network Manager (resp. the ConfD Agent and SR Agent).

4.7 Conclusion

Our architecture is really generic and easily adaptable as each module on figure 4.1 could be extended or replaced with some other technology having the same function. In addition to the general workflow, our solutions offers validity checks that verify that the input configuration

respects some constraints and a way to generate network topologies and requirements using Python (Autotopo).

Besides introducing this Architecture, the principal purpose of the current implementation is to demonstrate how our model of requirements can be practically integrated into a controller. This chapter shows what are the guidelines to follow to achieve that goal, but as well indicates what are the features that can be adapted to different implementations (e.g. how to extend the checks on the configurations, or how to use a different monitoring technology than SNMP). Overall, the implementation was designed to provide the basic features needed for the different types of requirements to work (e.g. link monitoring, requirement deployment). But it was also designed to be easily extended. Hence, although the current version works only on simulated environment, as explained in this chapter, we believe that it has the potential to be adapted for real life networks.

5

Practical example of the working and a configuration

5.1 Introduction

In this chapter we will first explain some differences in behaviour between Fibbing and Segment Routing, by presenting different scenarios. Finally, this chapter ends by providing a small example of how to use the solutions in section 5.3.

5.2 Applications

This section explains how the solutions react to different scenarios. First, we will look at a basic network configuration, and explain how the two solutions (Fibbing and Segment Routing) handle a simple requirement (section 5.2.1). Second, will look at what happens in case of network partition in the event of link failure (section 5.2.2). Third, we will explain an issue that can happen when there is a link failure with Segment Routing, and we will see how the main controller can react to cope with the problematic behaviour. Finally, section 5.2.4 will cover what can happen in the event of main controller failure.

5.2.1 Regular network

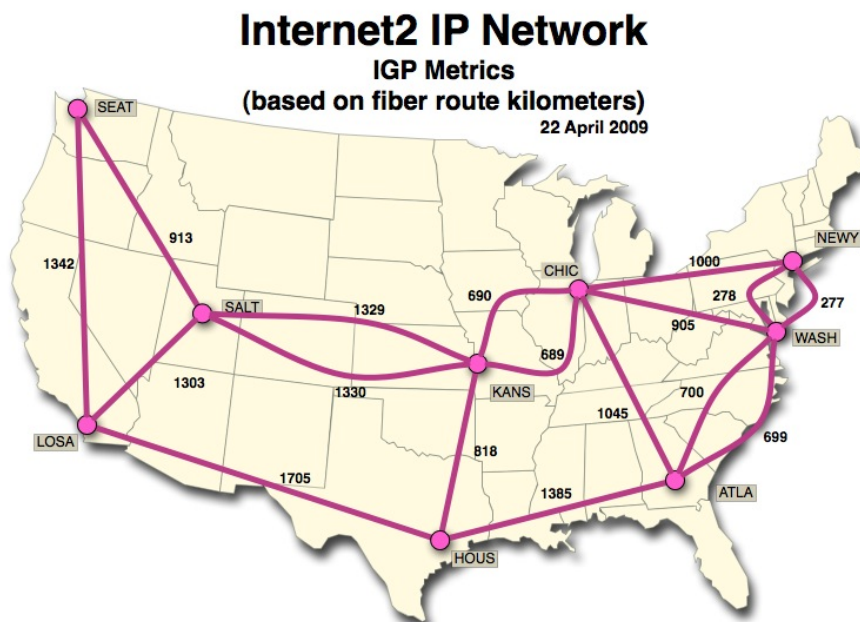


Figure 5.1: topology of Abilene¹

The first scenario has as purpose to show how Fibbing and Segment Routing handle the same problems and to show the trade-offs on a regular network. To illustrate this, we chose to take an existing topology: the Abilene network (figure 5.1). Suppose that two different sources (Source1 and Source2) are connected to SALT and two different destination hosts (Dest1 and Dest2) are connected to KANS. Let's suppose Source1 is sending traffic to Dest1, and Source2 to Dest2. By default, when no requirements are applied, all the traffic follows the shortest IGP path and flows through the link between SALT and KANS with the lowest cost 1329. We then apply a requirement [SALT, LOSA, HOUS, KANS] for Dest1. By doing so, both the path with cost 1329 and the path of the requirement will be used, and the load on the link between SALT and KANS (with cost 1329) will be lightened. To resolve the requirement, Fibbing and Segment Routing will use 2 different methods.

- *Fibbing*: The Fibbing controller will 'lie' to Router SALT by installing a fake node between LOSA and SALT. This fake node is advertising the prefix of the destination Dest1 to SALT with a smaller cost than the cost of the shortest IGP path. Hence forcing the traffic for Dest1 to go through the link between LOSA and SALT. After that, when a packet for Dest1 has reached LOSA, it will flow to HOUS using the Shortest IGP path, passing by the link between LOSA and HOUS.
- *Segment Routing*: The Southbound Interface of the Segment Routing solution will configure SALT by pushing a single segment (corresponding to the loopback address of LOSA) in the segment stack when trying to reach Dest1. After that, when a packet has reached LOSA, the segment is popped and the packet will flow to HOUS using the Shortest IGP path, passing by the link between LOSA and HOUS.

We then look at another case: we have two different sources (Source1 and Source2) that are connected to SALT and two different destination hosts (Dest1 and Dest2) that are connected to CHIC. Let's suppose Source1 is sending traffic to Dest1, and Source2 to Dest2. By default, when no requirements are applied, all the traffic follows the shortest IGP path and flows through the link between SALT and KANS. We then apply a requirement [SALT, LOSA, HOUS, ATLA, CHIC] for Dest1.

- *Fibbing*: The fibbing controller will potentially (depending on the algorithm computing the *Reduced Augmented Topology*) 'lie' to the routers on the path by adding fake nodes to steer traffic for Dest1.
- *Segment Routing*: The Southbound Interface of the Segment Routing solution will configure SALT by pushing segments in the segment stack when trying to reach Dest1.

Therefore we can see that the load is put on different parts of the network using Fibbing or Segment Routing. With Fibbing, the load is put on the controller that will have to advertise messages to create the fake nodes. For Segment Routing, the major load is put on the first router of each requirement, that has to push the right segments stack according to the destination. The other routers in the path of the requirement will have a small overhead to correctly process the SR headers.

5.2.2 Partitioned network

In this example (figure 5.2) Source1 is generating traffic towards Dest1 and Source2 to Dest2. There are also 2 requirements. For Dest1 the requirement is [RouterA, RouterB, RouterD] and for Dest2 the requirement is [RouterG, RouterH, RouterF]. The two requirements are running when suddenly the link between RouterC and RouterE goes down, resulting into the network being partitioned. It should be noted that the *Ctr* node represents the main controller, and

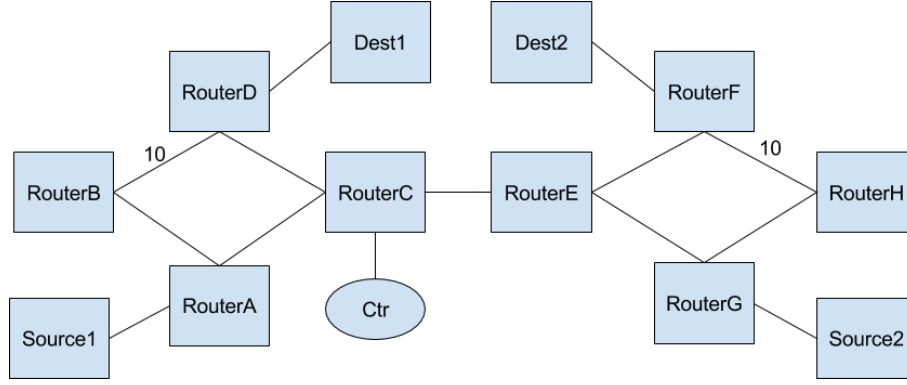


Figure 5.2: topology of the second scenario

in the case of the Fibbing solution we also consider that the main controller and the Fibbing controller are running on the same node *Ctr*. We now compare how both implementations will react:

- *Fibbing*: The controller can't reach some routers that may have some fake routes installed. These would be router E, F, G, H. Eventually, if the fake routes are not maintained by the Fibbing controller they will expire. Fibbing can implement both the fail-open or fail-close semantics, on a per-destination basis. For non-critical requirements the corresponding destinations can be injected in the IGP, so that connectivity can be preserved as long as the partition leaves at least one source-destination path. For stringent requirements, Fibbing can implement fail-close semantics by not announcing the corresponding destinations in the IGP [Vissicchio et al., 2015].
- *Segment Routing*: When using Segment Routing, the segment stacks that are already configured on the routers will stay active, even for the non reachable routers. The part of the graph that is still connected to the controller will stay configurable, and we can add or modify requirements if wanted. However, the part that is isolated from the controller will be unreachable and we can't add or remove SR routes. This includes scheduled requirement as well. We could think about adding a 'keep alive' timer to remove the paths if they are not maintained by the controller.

Both solution have a way to deal with network partitioning according to how the network administrator want them to be handled. It should be noted that in the event where part of the network is unreachable, also reflect the behaviour in case of controller failure.

5.2.3 Failure with Segment Routing

This scenario will analyze how Segment Routing behaves in the presence of a link failure. We evaluated this behaviour on a poorly connected network. The network in question is depicted on figure 5.3. We consider that the high-level requirement path imposed in this network is **[A B C D E]**, for a destination advertised by **E**. For this example let's call it *dest*. The corresponding minimum segment stack computed by the *Simplifier* is thus **[C]** (as a normal node segment), and this SR route must be installed on router **A**. The problem occurs when there is a link failure between **C-D**. In this situation router **A**, unaware of the failure, will still push the segment stack **[C]** for packets destined to *dest*. Following the normal protocol usage, the packet will be routed toward **C**, passing by **A-B-C**. **C** being the current active segment and the last segment in the segment stack, will pop the SR header and let the packet be routed according to the original destination (*dest*). The link **C-D** being down, the packet will be routed along **C-B-A**, which is now the new shortest path. At this point, router **A** will see a packet with destination

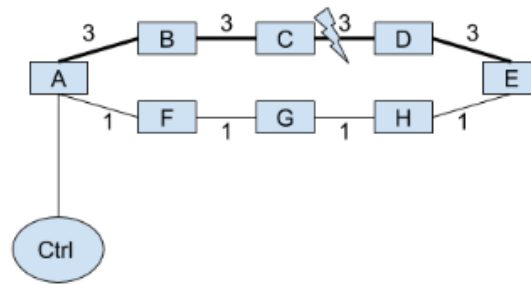


Figure 5.3: Segment Routing dealing with network with failure

address *dest*, matching the address of the SR route installed, and will push the segment stack [C]. And thus the packet will loop between A and C. However, if you have strong indications that when the network is in a error-free configuration, router A should never receive traffic for this destination (*dest*) on its interface corresponding to A-B but only on an interface for example A-F. You could match the SR route to this interface A-F. Thus preventing the looping behaviour.

Nonetheless, to limit this looping behaviour in general cases, it is important for the controller to detect the failure and to react accordingly. While the initial implementation of the controller was only designed to install the SR routes on the SR routers, and not to perform active monitoring of the whole network, we slightly modified the *backup* functionality (old behaviour was described in chapter 4) of the requirement to cope with the failure. Now, to handle this link failure, the user can specify a *backup* requirement for the link C-D. We modified the NetworkManagerCore to remove all running requirements that contain the down link in their path, when detecting that a link has gone down. The NetworkManagerCore will also delay the addition of any new requirements that contain a down link. Obviously, when detecting that a link is up again, the NetworkManagerCore will re-deployed the requirements, provided that they don't have any other down links.

In our example, specifying [A F G H E] as *backup* requirement will have as effect that the SR routes [C] on router A will be removed. Indeed, since [A F G H E] is the default shortest path for destination *dest*, it will be simplified as an empty segment stack. There is thus no need to install a SR route on router A to satisfy this requirement [A F G H E]. Hence, resulting in the controller simply removing the SR route corresponding to the previous requirement [A B C D E], and letting the IGP enforcing the *backup* requirement [A F G H E].

5.2.4 Dealing with Main Controller failure

This section will cover the unfortunate event of simultaneous failures of all main-controller replicas (see section 4.5.3).

In this case, no requirements could be added or removed from the network (this includes the scheduled requirements). However it does not necessarily means that the network will be left unmanaged. Indeed, the network would still be under the control of the underlying managers. This would be the Fibbing controller or the Segment Routing routes already installed on the SR routers, depending on the solutions. Hence most of the requirements wouldn't be impacted by a main controller failure, the only issue would be for the scheduled requirements. Indeed, the principal functionality of the main controller for the solution with Fibbing is to translate user defined requirements into DAGs requirements for the Fibbing controller. And for the solution with Segment Routing it is to install the user defined requirements on the SR routers as SR

routes. The second functionality is to handle the scheduled requirements which, as previously mentioned, wouldn't be able to be assured in the event of all main controller failures.

5.3 A simple example network

In this section we will illustrate with an example how to configure a network (shown on image 5.4) using the Fibbing solution and the Segment Routing solution and how to create a requirement.

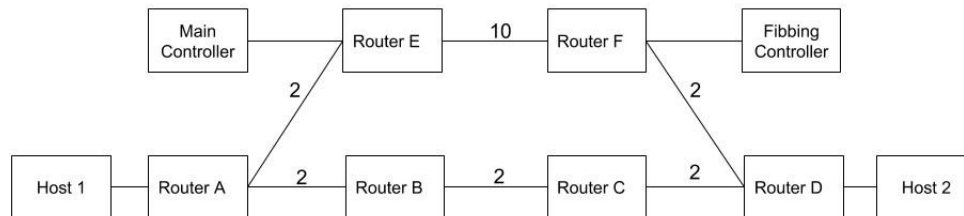


Figure 5.4: Example of a simple network

5.3.1 The network with Fibbing

In order to configure the network, we will need to start the ConfD CLI. Once the CLI is started, use `config` to enter the network configuration mode. To create a network, one must first declare all the routers present in the network.

5.3.1.1 Create the routers

To create a router, use the following command in config mode:

```
jessie(config)# fibbing network router <router-name>
```

where `<router-name>` is the name that you want to give to a router. In our example this command needs to be entered six different time, with each time a different router-name ranging from RouterA to RouterF. Now that the routers have been declared, one can declare and configure the network links.

5.3.1.2 Create the links

To create a new link, we first create a link by giving it a unique name. Then the user has to specify the source and destination of his link using previously declared routers and by assigning them an ip. The user can then change the bandwidth and cost of the link (otherwise the default values will be used). As example to create the link between router A and router B that we will call linkA-B:

```

jessie(config)# fibbing network link linkA-B
jessie(config-link-linkA-B)# src name RouterA ip 192.168.1.2/24
jessie(config-link-linkA-B)# dest name RouterB ip 192.168.1.1/24
jessie(config-link-linkA-B)# cost 2
jessie(config-link-linkA-B)# bw 100

```

Once all the links are created, we will need to enable OSPF on the links but before that, the user also needs to enable OSPF on each router and to give each router a router-id. For linkA-B this is done by using:

```

jessie(config)# fibbing network router A ospf enable true
jessie(config)# fibbing network router A router-id 1.1.1.1

```

```
jessie(config)# fibbing network router B ospf enable true
jessie(config)# fibbing network router B router-id 2.2.2.2
jessie(config)# fibbing network ospf-link-config linkA-B
```

The user can also modify the OSPF configs of the interfaces if wanted. For more information about the OSPF configs see the chapter about the Yang model (chapter 3).

5.3.1.3 Adding the hosts

Adding a host is almost similar to creating the link between two routers, the user just needs to create the link between the host and the previously declared router. As example for Host1 that is connected to RouterA:

```
jessie(config)# fibbing network hosts-link Host1-A
jessie(config-hosts-link-Host1-A)# router name RouterA ip 111.111.1.2/24
jessie(config-hosts-link-Host1-A)# host name Host1 ip 111.111.1.1/24
jessie(config-hosts-link-Host1-A)# bw 100
```

Once the hosts are added, the last thing to do for the Fibbing solution is to add the controllers to the network.

5.3.1.4 The controllers

There are 2 types of controllers: one unique Main Controller and Fibbing Controllers. The main controller is added by specifying a host as the main controller. Therefore to add the Main controller, a new host must first be created as done previously:

```
jessie(config)# fibbing network hosts-link MainCtrl-A
jessie(config-...)# router name RouterA ip 222.222.2.1/24
jessie(config-...)# host name MainController ip 222.222.2.2/24
jessie(config-...)# bw 100
jessie(config)# fibbing network main-controller MainController
```

Then we can add the Fibbing Controller, by creating the link between the Fibbing controller and a router:

```
jessie(config)# fibbing network fibbing-controller links FibCtrl-F
jessie(config-links-...)# router name RouterF ip 67.45.78.1/24
jessie(config-links-...)# controller name FibCtrl ip 67.45.78.2/24
```

The configurations specific for the Fibbing controller can be found in the chapter 3 covering the YANG data models.

5.3.2 The network with segment routing

In order to configure the network, we will need to start the ConfD CLI. Once the CLI is started, use *config* to enter the network configuration mode.

5.3.2.1 Create the routers

Like for Fibbing, we will first declare all the routers. The difference being that the Segment Routing routers need a loopback address and that the hosts need to be created as routers too. As example, here is how to create router A:

```
sr6(config)# segment-routing network router RouterA
sr6(config-router-RouterA)# loopback-addr fe12::01/64
sr6(config-router-RouterA)# ospf enable true
sr6(config-router-RouterA)# router-id 1.1.1.1
```

5.3.2.2 Create the links

The links are created in the same way as for Fibbing (except for the fact that ipv6 is used):

```
sr6(config)# segment-routing network link linkA-B
sr6(config-link-linkA-B)# dest name RouterB ip fe13::01/64
sr6(config-link-linkA-B)# src name RouterA ip fe13::02/64
sr6(config-link-linkA-B)# bw 100
sr6(config-link-linkA-B)# cost 2
```

Once the links are created OSPF needs to be enabled on the links. This is done like for Fibbing. For linkA-B we have:

```
sr6(config)# segment-routing network ospf-link-config linkA-B
```

Once again the user can modify the OSPF configurations of the interfaces if wanted. For more information about the OSPF configs see the chapter about the YANG (chapter 3).

```
sr6(config)# segment-routing network ospf-link-config linkA-B src
Possible completions:
    area dead-interval hello-interval lsa throttle
sr6(config)# segment-routing network ospf-link-config linkA-B dest
Possible completions:
    area dead-interval hello-interval lsa throttle
```

5.3.2.3 Adding the destinations

In Segment Routing, destinations are declared as routers and are then given the role of destinations. Therefore to create Host1, Host1 must have first be created as a router. Next the following command is used:

```
sr6(config)# segment-routing network destinations Host1
```

5.3.2.4 The controller

Using Segment Routing, there is only one controller: the main controller. The main controller is added by specifying a router into the main controller. Therefore to add the Main controller, a main-controller must have been created as a router previously:

```
sr6(config)# segment-routing network main-controller Main-controller
```

5.3.3 Requirement

This section is valid as well for Fibbing as for Segment Routing since both solutions have exactly the same syntax, except for the first two keywords of each commands. For Fibbing those will be:

```
jessie(config)# fibbing fibbing-requirement
```

and for segment routing those will be:

```
sr6(config)# segment-routing sr-requirement
```

Therefore for the rest of this section we will limit us, when showing a command, to how it is represented for the Fibbing solution. To obtain the same command for segment routing the user has to replace the 'fibbing fibbing-requirement' of each command by 'segment-routing sr-requirement'.

As seen in the Yang model there are various types of requirements (Chapter 3). We will start by presenting how to create a simple requirement and then the fields to add and modify to make Scheduled requirements.

5.3.3.1 Simple requirements

Let's say we want to configure a requirement that forces the traffic for a destination Host2 to go via the path < RouterA RouterE RouterF RouterD >. We will give this Requirement the name 'Requirement1'. The command to use when working with fibbing will be:

```
jessie(config)# fibbing fibbing-requirement requirement Requirement1
jessie(config-...)# dest Host2
jessie(config-...)# Path [ RouterA RouterE RouterF RouterD ]
```

Once the requirement is created, the user has to change the state of the requirement to 'running' to push the requirement into the network:

```
jessie(config-...)# state running
```

5.3.3.2 Time requirements

Time requirements are created in the same way as simple requirements. The only difference is that the state has to be set to scheduled. One must also configure the schedule during the creation of the requirement. Lets create the same requirement as previously, but this time the requirement will only be running between 12:00 and 15:00, during the weekend. We will also give our new requirement the name 'requirement2'.

```
jessie(config)# fibbing fibbing-requirement requirement Requirement2
jessie(config-...)# dest Host2
jessie(config-...)# Path [ RouterA RouterE RouterF RouterD ]
jessie(config-...)# state scheduled
jessie(config-...)# schedule Type time
jessie(config-...)# schedule days [ Saturday Sunday ]
jessie(config-...)# schedule start-hour 12:00 end-hour 15:00
```

5.3.3.3 Bandwidth requirements

To specify a Bandwidth requirement the schedule field needs to be changed as well. The bandwidth requirement will become running when the bandwidth percentage of the specified link exceeds the bandwidth percentage that was specified. As example we will create the same requirement as before but this time it will only be running if the traffic between RouterB and RouterC exceeds 60% of the bandwidth, every time:

```
jessie(config)# fibbing fibbing-requirement requirement Requirement3
jessie(config-...)# dest Host2
jessie(config-...)# Path [ RouterA RouterE RouterF RouterD ]
jessie(config-...)# state scheduled
jessie(config-...)# schedule Type bandwidth
jessie(config-...)# schedule days [ Monday Tuesday ... Saturday Sunday ]
jessie(config-...)# schedule start-hour 00:00 end-hour 23:59
jessie(config-...)# scheduled link from RouterB to RouterC
jessie(config-...)# scheduled link bw-percentage 60
```

5.3.3.4 Backup requirements

The last type of requirements is the Backup requirements. As for the two previous ones, the schedule field must be changed. The Backup requirement will become running if the link for which it is a backup goes down. As example we will once again create the same requirement as before, but this time the requirement will become running if the link between RouterB and RouterC goes down, every time:

```
jessie(config)# fibbing fibbing-requirement requirement Requirement4
jessie(config-...)# dest Host2
jessie(config-...)# Path [ RouterA RouterE RouterF RouterD ]
jessie(config-...)# state scheduled
jessie(config-...)# schedule Type backup
jessie(config-...)# schedule days [ Monday Tuesday ... Saturday Sunday ]
jessie(config-...)# schedule start-hour 00:00 end-hour 23:59
jessie(config-...)# schedule link from RouterB to RouterC
```

5.4 Conclusion

As long as the network is not poorly designed, Fibbing and Segment Routing should work in all scenarios. In the case of a poorly designed network or massive failures, the network can still rely on underlying protocols. We also showed with a basic example how easy, intuitive and fast it is to create and configure a network and to add requirements.

This chapter covers the performance of the different architecture components. We will first look at how the application reacts to increasing network size (section 6.1), and then to increasing number of requirements (section 6.2). Then section 6.3, and 6.4 will remind the performance specific to the Fibbing controller. Then section 6.5 will cover the performance of the *Southbound Interface* of the Segment Routing solution, and section 6.6 how the solution reacts to failures. Finally, section 6.7 presents some performance specific of the IPv6 Segment Routing implementation. It should be noted that all measurements reported in this section were obtained on an Intel Core i3 1.7GHz CPU, and if not mentioned otherwise the reported results are average over 10 trials.

6.1 Network size scaling

Figure 6.1 evaluates how ConfD scales to large networks. More specifically, it depicts the time required by ConfD and its Agent to process new network configurations with respect to the size of the network. In this case the processing time is computed as the time difference between the moment we send the network configuration via NETCONF and the time the ConfD Agent notifies the Network Manager Core's entry point (see chapter 4). This thus corresponds to the time required by ConfD to enforce the YANG structure and constraints on the configuration, time to store the valid configuration in its database, and the time for the ConfD Agent to read the configuration from the database.

For this experiment, we used networks in squared mesh topology of increasing sizes, with maximum node degree of 4. On figure 6.1 is reported the time (in second) taken by ConfD and its Agent to process these network configurations according to the number of nodes in the network. It should be noted that a $n \times n$ squared mesh graph has n^2 nodes and $2n^2 - 2 * n$ links. For this experiment we used 15 values of sizes n ($n \in \{5, 6, 7, \dots, 20\}$) with for each is reported the average processing time according to the corresponding number of nodes n^2 . As can be seen from the plot, the processing time grows linearly according to increasing network sizes, but for very large networks the processing time can be high (more than 20 seconds for a network with 400 nodes). However, grid networks don't really reflect the reality. So we evaluated the processing time for 20 randomly selected real topologies from the Topology-Zoo. Since for the topologies from Topology-Zoo there is no predefined relation between the number of nodes and the number of edges, we computed the ratio $\alpha = \frac{nodesNbr}{edgesNbr}$. From the 20 selected topologies, we defined two groups: the topologies with $\alpha < 0.7$ (denoted on figure 6.1, 6.2a, and 6.2b by triangles) and the topologies with $\alpha > 0.7$ (denoted on figure 6.1, 6.2a, and 6.2b by crosses). As can be seen from figure 6.1, there exist very large topologies (e.g. 754 nodes and $\alpha > 0.7$) for which the processing time can be above 30 seconds.

However, the number of nodes (along with the ratio α) on its own does not give a good *a priori* indication of the processing time of ConfD and its Agent. In the general case, we can consider that ConfD will read all attributes of the YANG model (see chapter 3) corresponding

to the *router*, *link* and *ospf-link-config*. Since the number of edges will only determine the size of *link* and *ospf-link-config*, and the number of nodes will determine only the size of *router*, we can determine a ratio corresponding to the overhead specific to the number of nodes with respect to the number of edges. We will consider the YANG model for the Segment Routing solution since it has the most attributes. The number of YANG attributes for the *router* is 12, while the number of attributes for *link* and *ospf-link-config* is 27. Figure 6.2b plots the processing time according to $ratio * nbrNodes + nbrEdges$ where $ratio = (12/27) = 0.44$, while figure 6.2a plots a Zoom on the figure 6.1. From figure 6.2 that provides the side by side comparison between the two previously mentioned figures, it can be seen that the method using $ratio * nbrNodes + nbrEdges$ seems a better predictor of the processing time than just the number of nodes. This is motivated by the fact that the measurement points on figure 6.2b more closely follows a line than the points on figure 6.2a. So the method using $ratio * nbrNodes + nbrEdges$ should give a good *a priori* estimate of the processing time. It should be noted that if the YANG model changes the *ratio* should be updated accordingly.

We also measured the memory (RAM) consumption of ConfD during the experiment, and for the 20 by 20 grid network (i.e. 400 nodes) the maximum RAM consumption measured was 49 MiB (in comparison ConfD uses 29.7 MiB without any configurations). Hence the extra load is negligible. Concerning the Network Manager, the memory overhead of the 400 nodes network was about 10 MiB, which is reasonable.

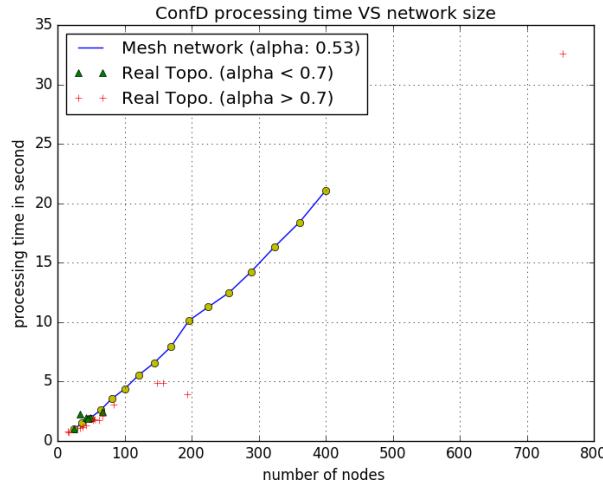
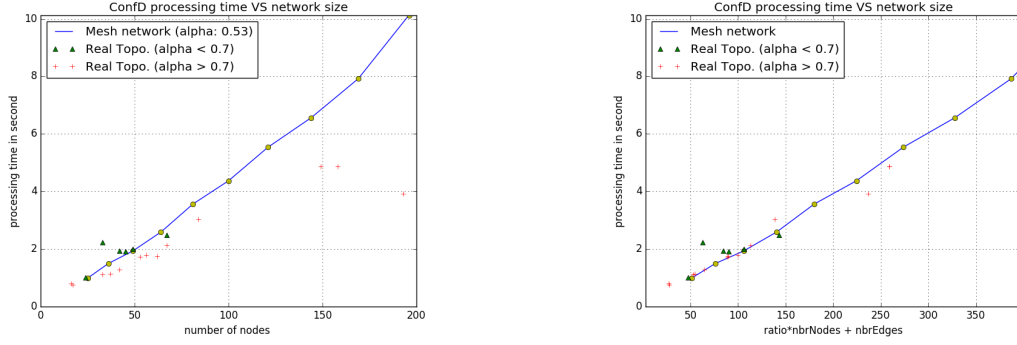


Figure 6.1: ConfD and Agent processing time according to increasing network sizes. Note that the full lines, and dots correspond to the mesh network described in section 6.1 while the triangles and crosses are the value obtained on real topology from Topology-Zoo¹.

6.2 Number of Requirements scaling

Figure 6.3 evaluates how the application scales to increasing number of requirements. More specifically, figure 6.3a reports the processing time of ConfD and its Agent according to an increasing number of requirements, while figure 6.3b outlines the processing time of the Network Manager according to the same increased number of requirements. For this experiment we relied on 3 sizes of requirements (Path of 5, 10, 20 routers, and type *bandwidth*) so that to evaluate small, medium and long requirements. The type *bandwidth* is the type that has the most field with respect to the YANG model (see chapter 3), and thus serves as an upper bound. As can be seen on both figure 6.3a and 6.3b, the processing time is linear according to the increased



(a) Time for ConfD and Agent to process new network configurations according to number of nodes. (b) Time for ConfD and Agent to process new network configurations according to both number of nodes and edges.

Figure 6.2: ConfD and Agent processing time according to increasing network sizes.

number of requirements. However, it clearly appears that ConfD and its Agent have a processing time way higher than the Network Manager. This may be due to the fact that ConfD will keep the configurations persistent by writing them into its own database CDB (see chapter 2), and that in general, database accesses are expensive operations. This also explain the high processing time for large networks (section 6.1). As can be seen on both plots (figure 6.3a and 6.3b), the length of the path of the requirement has an impact on the processing time. While this impact is small when the number of requirements is small, it becomes greater for large number of requirements. However, most of the time the requirements are expected to be used for Traffic Engineering purposes, and thus should have a relatively small length.

It should also be noted that the measurements for the Network Manager were taken from the *Entry Point* to the *Southbound Interface*. As the *Southbound Interface* is different for the two solutions they will be evaluated separately, in section 6.5, and 6.3. Nonetheless, when comparing, for example a network with size 200 with 200 requirements, the processing time of the network configuration is higher than for the requirements. This is a desirable result since the network configuration should only be entered once, while requirements may vary over time.

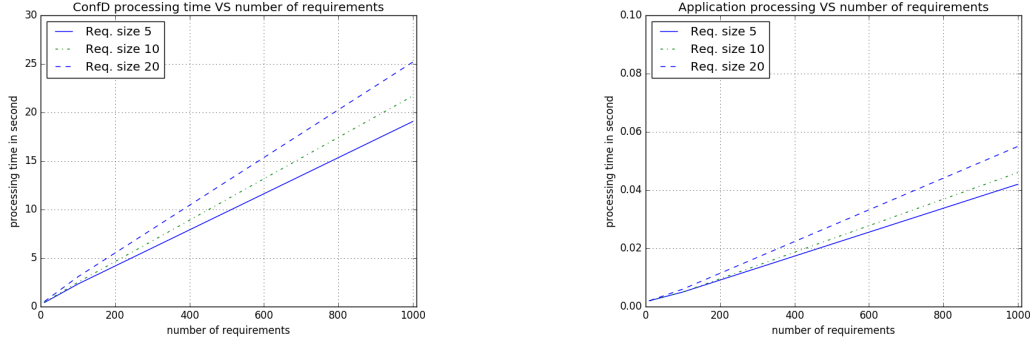
Similarly to the experiment with increasing network size, we also evaluated the memory (RAM) consumption of ConfD during the experiment. The maximum RAM measured during this experiment was 29.8 MiB for 1000 requirements, but ConfD without any configuration still used 29.7 MiB. Hence the extra load is negligible. And the Network Manager RAM overhead was about 5 MiB for the 1000 requirements, which is reasonable.

6.3 Performance of Fibbing

This section will remind the key performance aspects of the Fibbing controller² from [Vissicchio et al., 2015]. For more specific information about the experiments, and measurements we refer to the corresponding paper [Vissicchio et al., 2015].

Concerning the CPU and memory overhead, the conclusion of the measurements was that "Fibbing induces very little CPU and memory overhead on routers". Indeed, their experiment on Cisco ASR9K running IOS XR v5.2.2 equipped with 12GB of DRAM assigned to the routing engine, showed that two processes were impacted by the *fake nodes*: the RIB process (maintaining

²Version of the controller dating from 2015



(a) Time for ConfD and Agent to process new requirements
(b) Time for the NetworkManagerCore to process new requirements

Figure 6.3: ConfD and NetworkManagerCore processing time according to increasing number of requirements.

information about all the routes known to each destination), and the OSPF process (maintaining the entire OSPF topology). The result was that even for large number of *fake nodes* (100 000), the memory overhead for the two processes was 154MB, and the CPU utilization was at most 4%. It should be noted that "the Fibbing controller uses Type-5 LSAs which do not cause routers to recompute their shortest paths for each other".

6.4 Dealing with failures with Fibbing

This section reminds the key aspects of how Fibbing reacts to failure from [Vissicchio et al., 2015]. For more specific information about the experiments, and measurements we refer to the corresponding paper [Vissicchio et al., 2015].

The main result is that "Fibbing quickly reacts to non-partitioning failures". Indeed, upon detection of the failure, the Fibbing controller will have to re-compute the *Augmented Topology* to determine the actions to perform so that the requirements are still valid. And thus some requirements may need the injection of new *fake nodes*. It should be noted that the re-computation of the *Augmented Topology* is negligible (see [Vissicchio et al., 2015] for more information). Hence, in the worst case, the time required to deal with failures is twice the IGP convergence time. In the case where the controller injects the new *fake nodes* when the network has already converged. But most of the time the controller manages to inject the *fake nodes* during IGP convergence, hence the overall reaction to failure is slightly higher to the IGP convergence time.

6.5 Southbound Interface with Segment Routing

This section will evaluate the performance of the *Southbound Interface* for the Segment Routing solution. As a reminder, the *SR Southbound Manager* need to ensure two main functionalities: compute the minimum segment stack corresponding to the requirement path, and inform the SR routers about the SR routes.

6.5.1 Simplifier

Since most of the complexity comes from the *Simplifier* algorithm, we evaluated the time needed to solve the minimum segment stack for both a fixed size requirement in a network of increasing

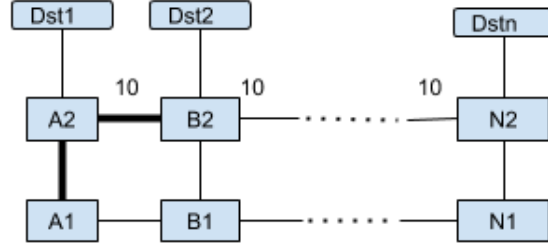


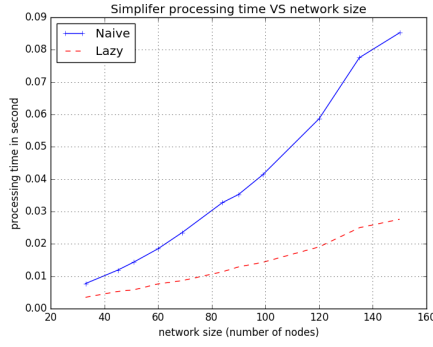
Figure 6.4: Test network for the Simplifier. (NB: unmentioned costs are considered to be 1).

size, and a fixed size network and a requirement of increasing length.

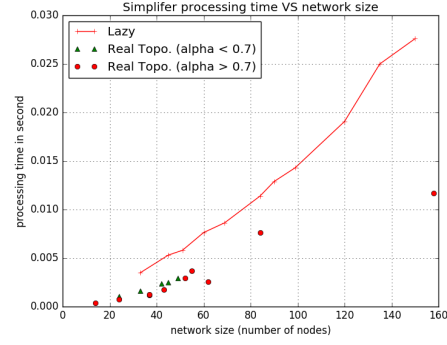
For these experiments we used the test network depicted on figure 6.4. Which consists of a two by N grid with a destination node for each router of the top row. As a metric, a network with size N results in $3 * N$ nodes, $3 * N$ links (including the destinations). Concerning the requirements, they always start with **A1-A2**, then the length will specify how many routers to go through on the top row of the grid. So for example a requirement of length 5 will be [**A1 A2 B2 C2 D2 E2**]. For this experiment, we considered two scenarios. In the first one, we used a fixed size requirement of length 7 (so requirement being [**A1 A2 B2 C2 D2 E2 F2 G2**]) while increasing the network size (number of nodes tested ranged from 33 to 150). In the second scenario in the other hand, we used a fixed size network of size 33 (i.e. 99 nodes) while increasing the requirement length (lengths tested: 5 to 15).

Figure 6.5 plots the time required by the *Simplifier* to solve the fixed length requirement with respect to an increasing network size (first scenario). More specifically, figure 6.5a shows the results of using the Naive versus the Lazy implementation of the *Simplifier* (see Chapter 4) while figure 6.5b details more the results of using the only Lazy implementation of the *Simplifier*. Again, in order to more closely relates to reality, we also perform measurements on 15 randomly selected topologies from Topology-Zoo, with an average requirements length of 7 to be compared with the first scenario. Those results are plotted on figure 6.5b and were obtained with the Lazy implementation as well. As can be seen, all of the points corresponding to Topology-Zoo are below the points corresponding to our first scenario. This is because the requirement used on the grid network (figure 6.4) constitutes a worse case scenario. Indeed, since the test requirement is following the links that have the costs of 10 (with respect to cost of 1 for the other links), the *Simplifier* will be forced to use adjacency segments for all the edges of the requirement. Which, for the Algorithm 4.2 that the *Simplifier* implements, is the worst case. Hence, the results of the first scenario should give a good estimated upper bound of the processing time of the *Simplifier* with respect to the number of nodes.

In the other hand, figure 6.6 depicts the time required to solve a requirement with increased length on a fixed size network (second scenario). And similarly, figure 6.6a (resp. figure 6.6b) shows the result when using the Naive versus lazy (resp. Lazy only) implementation of the *Simplifier*. From these results, it clearly appears that the Lazy implementation greatly improves the computation time of the *Simplifier* with respect to the Naive implementation by a factor of 10. This is explained by the fact that the Lazy implementation only computes the Dijkstra for sources not already seen, and that the Dijkstra algorithm has a time complexity of $\mathcal{O}(E * V)$ where E is the number of edges and V the number of vertices in the graph. Similarly to the first experiment, this second experiment also constitutes a worse case scenario because the requirements also follow the same behaviour explained previously. Thus the *Simplifier* will be forced to use adjacency segments for all the edges of the requirement. Hence, the results of the second

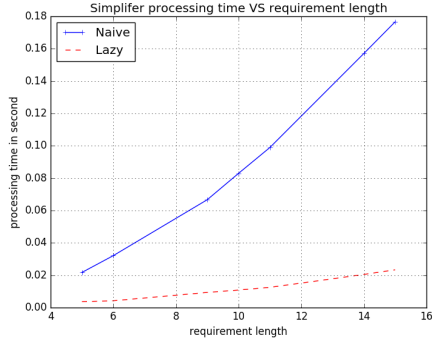


(a) Simplifier processing time with respect to increasing network size, using naive and lazy algorithms.

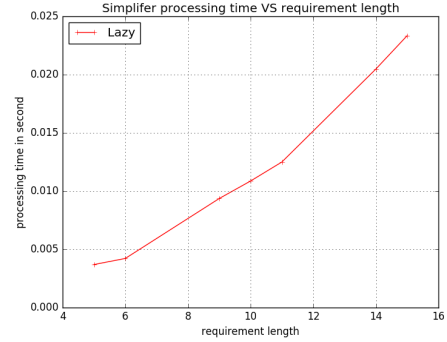


(b) Simplifier processing time with respect to increasing network size using a lazy cache of shortest paths. The triangles and crosses are the value obtained on real topology from Topology-Zoo with an average requirements length of 7.

Figure 6.5: Simplifier processing time according to network size



(a) Simplifier processing time with respect to increasing requirement length using naive and lazy algorithms.



(b) Simplifier processing time with respect to increasing requirement length using a lazy cache of shortest paths.

Figure 6.6: Simplifier processing time according to network requirement length.

scenario should give a good estimated upper bound of the processing time of the *Simplifier* with respect to the requirement length.

So, the Lazy implementation of the *Simplifier* adds a small overhead on the processing time of requirements. However, there is still great benefit to gain from the use of the minimum segment stack. Indeed, since each segment is a 128 bits IPv6 address, reducing the segment stack can result in a smaller Segment Routing header. This would reduce the overall size of the packets transmitted, and thus could save some precious bandwidth. Moreover, the processing time is expected to decrease over time. Because the Lazy implementation will store the result of the Dijkstra for each new sources, it is expected that over time more and more calls to the *Dijkstra* function will be able to fetch the result directly from the store.

6.6 Failure with Segment Routing

As seen in section 5.2.3 failures for Segment Routing can have terrible effects (e.g. black hole). This section evaluates the reaction to failures on the same scenario that creates the black hole

for Segment Routing (network for this scenario is depicted on figure 5.3).

For this experiment, we evaluated two scenarios: first, the time for the *Southbound Interface* to detect the failure, and second, the time for the *Southbound Interface* to detect and send the *backup* requirement. The send operation denotes the moment when the *Southbound Interface* initiates the NETCONF send call. The results for this scenario were obtained using different link delay (3, 6, and 9 ms) for all links in the network, and the failure occurs on the link **C-D** which is 3 hops away from the controller.

The results of the two scenarios are plotted on figure 6.7a. More specifically, this figure shows both the time to detect and the time to react to the link failure (**C-D**) with respect to the link delay. As can be seen from this figure, once the failure is detected the reaction time is small, with the average reaction time measured of 0.008 seconds. It should be noted this reaction time also includes the time to *Simplify* the requirement. A possible improvement could be to pre-compute the minimum segment stack for the backup requirement so that to reduce the reaction time to failure.

The other event that matters in the presence of failure is the time for the ConfD SR Agent to process the SR routes sent by the *SR Southbound Manager*. For this experiment we evaluated the time required to process N destinations with for each 5, 10, or 20 segments in the segment stack. Indeed, in case of failure not only it is important to quickly detect the failure and send the update, but also for the SR routers to rapidly install the updated routes.

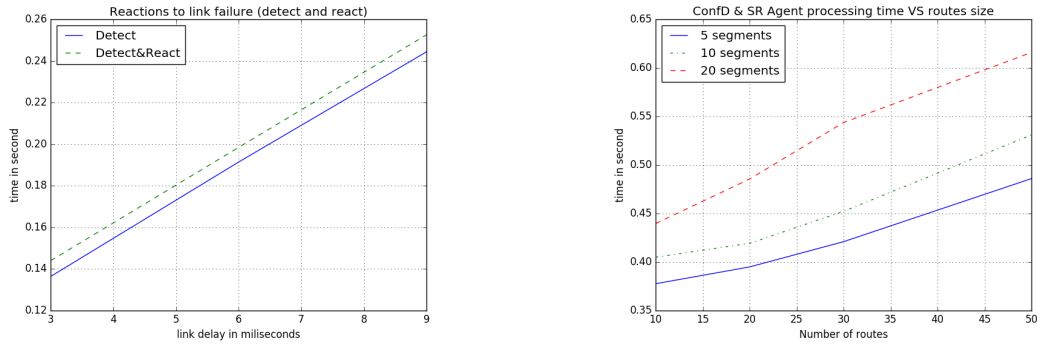
Figure 6.7b depicts the results obtained for the different number of routes tested ($N \in \{10, 20, 30, 50\}$). From this figure we can see that for a large number of routes (50) and with for each 20 segments in the segment stack, the insertion of the routes can take up to 0.62 seconds. Although, this is quite high in case of failure, it should be noted that this is considered an extreme scenario. Indeed, especially when using the *Simplifier*, the number of segments in the segments stack is expected to be lower than 20, which reduces the processing time. So this is another reason why the *Simplifier* should be used.

Overall, the reaction time for a failure is the time to detect the failure (depending on the link delay and the monitoring method used) + the reaction time to simplify and send the updated route (average of 0.008 seconds for our experiment) + time to reach the target router (depending on the network delay and the NETCONF transport protocol) + time for the SR Agent to install the SR routes (depends on the number of routes and the size of the segments stack).

6.7 Performance of Segment Routing

This section mentioned the overhead of processing the Segment Routing header and the storage cost associated to a Segment Routing route. These results come from [Lebrun David, 2017].

First we will consider the storage cost (or memory cost) of a SR route. A Segment Routing route is composed of 24 bytes of metadata, the Segment Routing header (8 bytes), and the segments (each 128 bits or 16 bytes). So for example for a 4 segments stack, the total storage cost is $24 + 8 + 16 * 4 = 96$ bytes. Concerning the processing overhead, the experimental results showed that the "Segment Routing header processing reaches about 90% of the plain IPv6 forwarding performance", where *plain* denotes regular IPv6 forwarding without any SR headers.



(a) Time to react to the failure with respect to link delay (b) Time for ConfD and the SR Agent to install the SR routes according to increasing routes.

Figure 6.7: Time to deal with failures

6.8 Conclusion

The main bottleneck of our solution seems to be ConfD as its processing time is way higher than the Network Manager. However, it should be noted that the interactions with ConfD are done via the configuration CLI. So even if 30 seconds latency for ConfD to process large network configuration may seem high, the time required to write the CLI command to exit the configuration mode and exit the CLI should make this 30 seconds latency seamless. The Simplifier algorithm also runs in a reasonable time and even if it introduces a small overhead on the processing time of requirements. But it brings a great advantage as it will reduce the overall size of the packets transmitted, and thus saves bandwidth. Also the time to detect and react to a failure is too high to be competitive or attractive. But as previously mentioned (see chapter 4) the monitoring solution using SNMP was just implemented to evaluate the correct deployment of backup requirements upon failure detection, but not necessarily to have fast failure detection. Other alternatives could be IPfix or Netflow. However, for both solutions, once the failure is detected the reaction time is acceptable.

Configuring and overall managing networks are complex operations that require a lot of expertise to get it right. The reason is that first the comprehension of the underlying protocols is not trivial and second the proper configurations of large networks can be time consuming to do manually and may be a tedious task. Indeed, most of vendor devices come with their own proprietary interfaces that vary from vendors to vendors. This means that for a same operation, human operators will need to learn the different commands specific for each different vendor devices in their network. In this context, we can understand the need for centralized management solutions.

Network administrators also want their network more programmable as the advent of SDN demonstrates. Indeed, today's networks need Traffic Engineering solutions to better utilize resources. And manually configuring routers to achieve requirements is needlessly time consuming, and lack dynamic control. However, SDN deployment comes at a cost. Existing networks have a huge installed base of devices not necessarily compatible with SDN, and SDN has limitation to scale to large networks.

Our solution is an attempt to provide the features expected for future networks: an intuitive and expressive interface to specify requirements, and a modular architecture that can adapt to different technologies. Indeed, thanks to its modularity the solution can be adapted to various networks that rely on different technologies (e.g. Fibbing or Segment Routing). So the architecture is not protocol dependent like other SDN solutions are with OpenFlow. Nonetheless, Fibbing and Segment Routing seems both to be good compromise solutions between centralized and distributed approaches and thus very promising for future networks.

Also, similarly to DNS that had a undeniably important role in the popularization of the Web, we believe that using a naming system for network management can make this technology more accessible. Indeed, while IP addresses are expressive in essence and particularly well suited to be processed by a machine, they are cumbersome when manipulated by humans. Especially 128 bits IPv6 addresses, which are expected to be seen more and more in future networks. Our solution seems a good compromise between the use of IP addresses and router names. Similarly to DNS, our controller still needs to know the mapping between router names and IP addresses. Hence, with our solution users would need to enter IP addresses along with the representation of the network, but this operation can be done only once. Then in order to specify requirements, users would be able to only rely the router names. This adds a great flexibility as the interface used to specify requirements can completely be independent of the underlying technology enforcing those requirements.

Although the naming system adds flexibility in the process of specifying requirement, the YANG model describing the structure and the constraints lays down a formal framework to encode those requirements. First this prevents users to enter requirements with errors. Second this enables the underlying solution (i.e. Fibbing or Segment Routing) to directly receive sound requirement configurations, which in turn reduces the risk of potential errors.

Overall, our solution introduces an intuitive but nonetheless formal way of specifying a variety of network requirement types, and a modular architecture to enforce those requirements working with both Fibbing and Segment Routing.

Bibliography

- [ONF,] Open Networking Foundation website. <https://www.opennetworking.org/>.
- [rfc, 1988] (1988). IAB recommendations for the development of Internet network management standards. RFC 1052.
- [rfc, 1990a] (1990a). OSI IS-IS Intra-domain Routing Protocol. RFC 1142.
- [rfc, 1990b] (1990b). Simple Network Management Protocol (SNMP). RFC 1157.
- [Al-Fares et al., 2010] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. (2010). Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA. USENIX Association.
- [Aubry et al., 2016] Aubry, F., Lebrun, D., Vissicchio, S., Khong, M. T., Deville, Y., and Bonaventure, O. (2016). Scmon: Leveraging segment routing to improve network monitoring. In *INFOCOM*.
- [Bjorklund, 2010] Bjorklund, M. (2010). YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020.
- [Claise, 2004] Claise, B. (2004). Cisco Systems NetFlow Services Export Version 9. RFC 3954.
- [Claise et al., 2013] Claise, B., Trammell, B., and Aitken, P. (2013). Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011.
- [Doria et al., 2010] Doria, A., Dong, L., Wang, W., Khosravi, H. M., Salim, J. H., and Gopal, R. (2010). Forwarding and Control Element Separation (ForCES) Protocol Specification. RFC 5810.
- [Enns et al., 2011] Enns, R., Bjorklund, M., Bierman, A., and Schönwälder, J. (2011). Network Configuration Protocol (NETCONF). RFC 6241.
- [Feamster et al., 2004] Feamster, N., Balakrishnan, H., Rexford, J., Shaikh, A., and van der Merwe, J. (2004). The case for separating routing from routers. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, FDNA '04, pages 5–12, New York, NY, USA. ACM.
- [Ferguson et al., 2013] Ferguson, A. D., Guha, A., Liang, C., Fonseca, R., and Krishnamurthi, S. (2013). Participatory networking: An api for application control of sdns. *SIGCOMM Comput. Commun. Rev.*, 43(4):327–338.
- [Filsfils et al., 2014] Filsfils, C., Previdi, S., Bashandy, A., Decraene, B., Litkowski, S., Horneffer, M., Milojevic, I., Shakir, R., Ytti, S., Henderickx, W., Tantsura, J., and Crabbe, E. (2014). Segment Routing Architecture. Internet-Draft draft-filsfils-spring-segment-routing-04, Internet Engineering Task Force. Work in Progress.

- [Gember et al., 2012] Gember, A., Prabhu, P., Ghadiyali, Z., and Akella, A. (2012). Toward software-defined middlebox networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 7–12, New York, NY, USA. ACM.
- [Ghobadi et al., 2012] Ghobadi, M., Yeganeh, S. H., and Ganjali, Y. (2012). Rethinking end-to-end congestion control in software-defined networks. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 61–66, New York, NY, USA. ACM.
- [Lebrun David, 2017] Lebrun David, B. O. (2017). Implementing IPv6 Segment Routing in the Linux Kernel. In *ACM SIGCOMM*.
- [Medved et al., 2017] Medved, J., Bahadur, N., Ananthakrishnan, H., Liu, X., Varga, R., and Clemm, A. (2017). A Data Model for Network Topologies. Internet-Draft draft-ietf-i2rs-yang-network-topo-12, Internet Engineering Task Force. Work in Progress.
- [Moy, 1998] Moy, J. T. (1998). OSPF Version 2. RFC 2328.
- [ONF, 2012] ONF (2012). Software-defined networking: The new norm for networks. Technical report, Open Networking Foundation.
- [Previdi et al., 2017] Previdi, S., Filsfils, C., Field, B., Leung, I., Linkova, J., Aries, E., Kosugi, T., Vyncke, E., Lebrun, D., Leddy, J., Raza, K., daniel.voyer@bell.ca, daniel.bernier@bell.ca, Matsushima, S., Steinberg, D., and Raszuk, R. (2017). IPv6 Segment Routing Header (SRH). Internet-Draft draft-ietf-6man-segment-routing-header-06, Internet Engineering Task Force. Work in Progress.
- [Rekhter et al., 2006] Rekhter, Y., Hares, S., and Li, D. T. (2006). A Border Gateway Protocol 4 (BGP-4). RFC 4271.
- [Srinivasan et al., 2004] Srinivasan, C., Nadeau, T., and Viswanathan, A. (2004). Multiprotocol Label Switching (MPLS) Label Switching Router (LSR) Management Information Base (MIB). RFC 3813.
- [Tail-f, 2016] Tail-f (2016). ConfD tail-f’s management agent. <http://www.tail-f.com/management-agent/>.
- [Vanbever and Vissicchio, 2014a] Vanbever, L. and Vissicchio, S. (2014a). Enabling sdn in old school networks with software-controlled routing protocols. Open Networking Summit (ONS) research track.
- [Vanbever and Vissicchio, 2014b] Vanbever, L. and Vissicchio, S. (2014b). Enabling sdn in old school networks with software-controlled routing protocols. In *Presented as part of the Open Networking Summit 2014 (ONS 2014)*, Santa Clara, CA. USENIX.
- [Vissicchio et al., 2015] Vissicchio, S., Tilman, O., Vanbever, L., and Rexford, J. (2015). Central Control Over Distributed Routing. In *ACM SIGCOMM*, London, UK.
- [Vissicchio et al., 2014] Vissicchio, S., Vanbever, L., and Rexford, J. (2014). Sweet Little Lies: Fake Topologies for Flexible Routing. In *ACM HotNets*, Los Angeles, California.
- [Xia et al., 2015] Xia, W., Wen, Y., Foh, C. H., Niyato, D., and Xie, H. (2015). A survey on software-defined networking. *IEEE Communications Surveys Tutorials*, 17(1):27–51.
- [Yang et al., 2004] Yang, L., Dantu, R., Anderson, T., and Gopal, R. (2004). Forwarding and control element separation (forces) framework.

- [Yeung et al., 2017] Yeung, D. M., Zhang, Z. J., Lindem, A., Qu, Y., and Chen, I.-W. (2017). Yang Data Model for OSPF Protocol. Internet-Draft draft-ietf-ospf-yang-07, Internet Engineering Task Force. Work in Progress.
- [Zhang et al., 1993] Zhang, L., Deering, S., Estrin, D., Shenker, S., and Zappala, D. (1993). Rsvp: A new resource reservation protocol. *Netwrk. Mag. of Global Internetwkg.*, 7(5):8–18.

