Software Resolved Networks: Rethinking Enterprise Networks with IPv6 Segment Routing

David Lebrun
UCLouvain, Belgium
david.lebrun@uclouvain.be

Mathieu Jadin
UCLouvain, Belgium
mathieu.jadin@uclouvain.be

François Clad
Cisco Systems, Inc.
fclad@cisco.com

Clarence Filsfils
Cisco Systems, Inc.
cfilsfil@cisco.com

Olivier Bonaventure
UCLouvain, Belgium
olivier.bonaventure@uclouvain.be

ABSTRACT

Enterprise networks often need to implement complex policies that match business objectives. They will embrace IPv6 like ISP networks in the coming years. Among the benefits of IPv6, the recently proposed IPv6 Segment Routing (SRv6) architecture supports richer policies in a clean manner. This matches very well the requirements of enterprise networks.

In this paper, we propose Software Resolved Networks (SRNs), a new architecture for IPv6 enterprise networks. We apply the fundamental principles of Software Defined Networks, *i.e.*, the ability to control the operation of the network through software, but in a different manner that also involves the endhosts. We leverage SRv6 to enforce and control network paths according to the network policies. Those paths are computed by a centralized controller that interacts with the endhosts through the DNS protocol. We implement a Software Resolved Network on Linux endhosts, routers and controllers. Through benchmarks and simulations, we analyze the performance of those SRNs, and demonstrate that they meet the expectations of enterprise networks.

ACM Reference Format:
David Lebrun, Mathieu Jadin, François Clad, Clarence Filsfils, and Olivier Bonaventure. 2018. Software Resolved Networks: Rethinking Enterprise Networks with IPv6 Segment Routing. In *SOSR '18: SOSR '18: Symposium on SDN Research, March 26–27, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3185467.3185471

1 INTRODUCTION

IPv6 adoption in the global Internet has grown in a spectacular fashion. Pushed by the increasing pressure of the IPv4 addressing space exhaustion, Content Delivery Networks (CDN) and Internet Service Providers (ISP) have deployed IPv6 at a large scale [45]. Today, a growing fraction of mobile and home users rely on IPv6 to access web-based services [12, 13, 64] and some mobile providers have deployed IPv6-only networks. This IPv6 wave has not yet reached the majority of enterprise networks but the most advanced ones have already pledged to move to an IPv6-only architecture [3, 27, 33, 41]. With significantly fewer users than large providers, small and middle-sized enterprises do not feel the same pressure to move to IPv6 as ISPs. Many consider IPv6 as simply a variant of IPv4 with more addresses and have difficulties in justifying the cost of an IPv6 deployment. This incorrect assumption plays a key role in the current *status quo* of IPv6 deployment in enterprise networks.

In parallel, many entreprises are seduced by Software Defined Networks (SDN) [10, 35, 42] that promise to simplify the management of their networks. These are often more complex than ISP networks, given the need to support a variety of business policies [40, 58]. These policies correspond to various business objectives that need to be met by the network. A first example is Quality of Service. Many entreprise networks have deployed Voice over IP and video services that require special QoS in particular on low bandwidth wide area links. A second example is the need for fine grained access control. It is frequent in enterprises to restrict access to parts of the network for some classes of users. These restrictions can be implemented by using firewalls, routing policies and other techniques. A third example is the need to support specific paths for specific applications, e.g. to use dedicated links or support extranet services. A fourth example is the large number of middleboxes that are deployed in many enterprise networks [55]. Several types of SDN networks have been proposed (see [35] for a detailed survey). They typically rely on a logically centralized controller that interacts with the network devices (routers, switches and sometimes middleboxes) to support the network policies defined by the operator. OpenFlow is a popular protocol that enables SDN controllers to interact with network devices [42].

In this paper, we propose a new architecture to instantiate the SDN vision in IPv6 enterprise networks. To achieve this, we leverage IPv6 Segment Routing (SRv6) [20, 47], a recent feature of IPv6 enabled by the flexibility of the protocol. SRv6 is a modern source routing paradigm that allows to steer packets through an ordered list of instructions, called segments. Those segments are encoded in each packet as IPv6 addresses, inside a Segment Routing Header (SRH) [47]. Numerous use cases have been documented [8] and network vendors have demonstrated implementations [22]. On endhosts, we have recently added SRv6 support to the mainline Linux kernel [38]. However, to the extent of our knowledge, enterprise networks have not yet deployed this technology.

We propose a variant of the SDN architecture that we call **Software Resolved Networks** (SRN). An SRN is a network that is managed by a logically centralized controller. The name SRN originates from the fact that our architecture colocates its controller with a DNS resolver and uses extensions of the DNS protocol to interact with endhosts. In developing Software Resolved Networks, we make the following contributions.

An architecture implementing an application-centric SDN paradigm suitable for enterprise networks (§2, §3). This architecture supports *conversations* between applications, regulated by their interactions with the controller through the DNS protocol.

DNS extensions enabling (*i*) applications to embed traffic and/or path requirements in their DNS requests and (*ii*) the controller to return the appropriate path to the applications (§3).

A prototype implementation running on Linux that demonstrates the feasibility of our approach and enables other researchers to replicate and expand our results. We extend the SRv6 implementation included in the mainline Linux kernel [38] and provide a complete implementation of a modular controller, as well as extensions to DNS libraries (§5).

Measurements demonstrating the flexibility of our prototype and its performance through various microbenchmarks and experiments in an emulated network (§6).

2 ENTERPRISE NETWORKS

Most entreprises, notably within a campus, have a dense network, *i.e.*, there are many possible paths between a given pair of hosts through the network. From a high level viewpoint, a mathematician could represent the network engineer's job as a function $\mathcal{M}$ that maps the end-to-end communication flows on specific network paths, or the empty path when a flow is prohibited by the network policies. The output of this mapping function typically depends on a mix of configuration parameters (link weights, IP addresses, VLANs, access lists, etc.), the state of the network links and nodes, and the characteristics (*e.g.*, source and destination addresses and ports) of the packets exchanged for each flow.

In practice, function $\mathcal{M}$ can be realised by using a variety of technologies. A very simple approach in a small network is to use the Spanning Tree protocol. In this case, the active network topology is restricted to a tree and there is only one path for each end-to-end flow. Several variants of this approach have been deployed. IP routing protocols such as OSPF are other popular examples. In this case, function $\mathcal{M}$ becomes a shortest path algorithm whose main parameters are the link weights and the status of the links and nodes. Packets follow the shortest path, possibly with per-flow load balancing when there are equal cost paths. Enterprise network operators have deployed a variety of solutions to leverage paths other than the shortest ones. Some have used several routing protocols [40], configured VLANs or access lists [58] or used BGP inside the enterprise [36]. Others have relied on policy routing to provide finer control on the selection of the end-to-end paths. Integrated Services [7] was proposed as an architecture to add resource reservations for end-to-end flows. With this architecture, packets are still forwarded along the shortest path and the RSVP protocol [66] is used to maintain per-flow state on the intermediate nodes.

Various realisations of the Software Defined Networks (SDN) paradigm have revisited this problem. Instead of using a distributed protocol with configuration parameters that sometimes indirectly influence the end-to-end paths, SDN relies on a logically centralised controller that implements function $\mathcal{M}$ and programs the intermediate nodes to realise the chosen network path for a given end-to-end flow. A simple realisation is to intercept the first packet of each flow on the edge node, forward it to the controller that selects the path and programs the intermediate nodes by using the OpenFlow protocol [42]. A wide range of solutions have been proposed under the generic umbrella of a logically centralised controller [35].

Any realisation of function $\mathcal{M}$ is always a tradeoff between the number of parameters that the network operators can tune and the amount of state that needs to be maintained on the network nodes. Some enterprises require solutions that provide a very fine-grained control on the mapping of traffic flows onto network paths, possibly on a per connection or a per source/destination pair basis. On the other hand, network operators need to minimize the amount of state that must be installed and maintained on each node for obvious scalability reasons.

Enterprise networks are composed of three main types of network devices: layer-2 switches, layer-3 routers and middleboxes [55]. We distinguish three types of layer-3 routers. At the edge are *access* and *border* routers. Hosts are connected to access routers. Border routers are connected to upstream providers. *Core* routers are only connected to other enterprise routers. We use Router Advertisements [56] (RAs) and StateLess Address AutoConfiguration (SLAAC) to assign one or more IPv6 addresses to the hosts. The DNS configuration of the hosts can be distributed through RAs or DHCPv6 [39].

Routers exchange routing information by using a link state routing protocol such as OSPFv3 [19]. In particular, we assume that the link state routing protocol used in the enterprise supports traffic engineering extensions that distribute unidirectional link metrics such as bandwidth utilization and link delay [1].

2.1 IPv6 Segment Routing

Segment Routing (SR) is a modern instantiation of the source routing paradigm, currently being standardized within the IETF [20]. SR can be used on top of an MPLS or IPv6 dataplane to steer packets through an ordered list of *segments*. The MPLS variant of SR was the first to be defined. It is now supported by most router vendors [22] and deployed by several major ISPs [60], most often for traffic engineering [5, 26] and fast-reroute [4] purposes. However, since more and more service providers and enterprises are now transitioning from IPv4 to IPv6, running Segment Routing over the plain IPv6 dataplane (without MPLS) is a good opportunity for further network simplification.

The IPv6 flavor of Segment Routing (SRv6) relies on a dedicated IPv6 routing extension header, called Segment Routing Header (SRH) [47]. Each SRv6 segment is an IPv6 address representing an intermediate function to be executed at a specific location in the network. The *active segment* is the destination address in the IPv6 header, while the complete list of segments is carried in the SRH. Transit nodes on the way to the active segment are thus doing plain IPv6 forwarding. SRv6 support is only required at the intermediate destinations, also called *segment routing endpoints*. This enables incremental deployment of SRv6 where only a subset of the network devices are capable of processing the SRH. Each segment belongs to an IPv6 prefix that can be advertised by the *parent node* in the routing protocol. As such, SRv6-enabled traffic can leverage existing ECMP paths computed by the underlying IGP. When a packet reaches the active segment parent node, the SR endpoint function encoded in the host part of the IPv6 address is triggered. Such functions include moving forward to the next segment, forwarding the packet on a given link, as well as any arbitrary user-defined operation. For interoperability purposes, a small set of basic SRv6 functions is being defined at the IETF [21].

An SRH is added to a packet when it is steered into an *SRv6 policy*. This steering can be achieved by the source, an SR endpoint or any transit node along the path of the packet. At the source, a socket option can be configured by the application to send out packets with a specific SRH. Alternatively, a routing entry can be mapped to the SRH insertion function. All packets matching the route will then receive an SRH with the configured list of segments. This list may contain a segment associated with a special endpoint function, called a *binding segment*. Such a segment maps to another SRv6 policy, *i.e.*, another list of segments. When a packet arrives at the binding segment parent node, it is steered into the SR policy associated with the binding segment, and is augmented with the corresponding SRH. Depending on the policy configuration, the new SRH can either be added through *encapsulation* within an outer IPv6 header, or *inserted* between the existing IPv6 and SR headers [62]. Finally, an SRv6 steering policy may be installed on any SR-capable node to encapsulate or insert an SRH into transiting packets, based on their destination address or any other classification mechanism. Encapsulation is the preferred mode for tenant isolation (VPN), persistent traffic engineering (*e.g.*, latency optimization) or service chaining within a segment routed domain. The traffic is encapsulated at the ingress edge of the domain and the last segment in the SRH is associated with a decapsulation function, which removes the outer IPv6 and SR headers at the egress edge.

3 SOFTWARE RESOLVED NETWORKS

A key principle of Software Resolved Networks (SRN) is to let applications have an active role in the management of their flows.

When an application running on a client host communicates with a server, the packets carried from one endpoint to the other are usually called a unidirectional flow. In this paper, we always associate this flow with a return flow. In other terms, we consider that two applications always communicate in a bi-directional fashion. We call this pair of flows a *conversation*. We distinguish the two endpoints of a conversation. The application that initiates a conversation is called a *client application*. The process of initiating a conversation is referred to as *establishing a conversation*. Conversely, an application accepting conversations is called a *server application*. We also distinguish applications with respect to their location. An application whose endpoints are located inside the enterprise is called an *internal application*. Likewise, an application whose endpoints are located outside the enterprise is called an *external application*.

We make two reasonable assumptions about the enterprise network: (*i*) all endpoints are reachable over IPv6 and (*ii*) all endpoints are identified by DNS names. Each device in the network is properly named according to a DNS naming plan. Furthermore, we leverage the large number of IPv6 addresses [11]. For example, each server may receive several IPv6 addresses, and each address is only used by a particular application. Furthermore, we assume that server applications are never referred to by their IPv6 address at the application layer, but rather by their DNS name[1]. The applications use the DNS protocol to interact with the controller. We prefer to use DNS instead of a signalling protocol such as RSVP or NSIS for two main reasons. First, our architecture does not create state on core routers. Second, DNS is easy to extend, widely implemented and well-known by network operators. Furthermore, using the DNS minimises the additional delay caused by the creation of the network path. To facilitate this interaction, the default resolver configured for these applications is the controller itself (the *SDN Resolver*), acting transparently as a regular DNS resolver.

When a client application initiates a conversation to a server application in a Software Resolved Network, it performs the following operations. First, it issues a DNS request to resolve the name of the server application into an IPv6 address. Then, it sends data to the resolved address. As the DNS resolver is actually the controller, it may automatically perform appropriate actions, such as allocating a network path. Furthermore, the client can embed in the DNS request requirements about the conversation, using existing DNS extensions [61]. For example, those requirements can list the expected bandwidth and latency. We name such a DNS request a *conversation request*. Along with the resolved IPv6 address, the controller returns a *Path ID* in the DNS reply. This *Path ID* is an opaque string that maps to the allocated network path and is the key to enabling the application to use this selected path. It is important to note that a *Path ID* uniquely identifies *one half* of a conversation, *i.e.*, the packet flow starting from the client application that issued the conversation request and going to the other endpoint. An application may re-issue a conversation request at any time during the lifetime of a conversation. This enables the application to request a new network path corresponding to updated requirements.

As such, the *SDN Resolver* provides a mechanism for the dynamic registration of server applications, namely *server registration*. A server registration is very similar to a conversation request. Instead of resolving the name of an application, the server attempts to resolve a name that is pre-configured

[1]The only exception is the DNS resolver whose IP address is distributed by DHCPv6 or Router Advertisements. We also assume that entreprise applications will be written in a high-level programming language that provides a `connect-by name` API instead of the `connect-by address` of the C socket API.

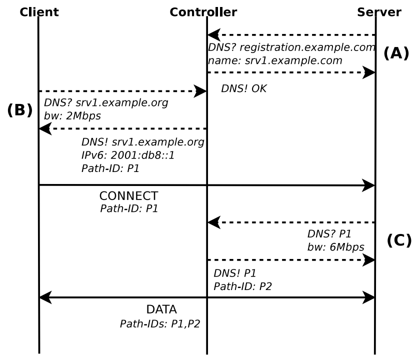


Figure 1: Workflow for server registration and connection establishment.

by the operator. This name is not attached to any particular application. Rather, its resolution signals a registration request to the controller. If the server has the credentials to register this name, the request is translated into a DNS update message [51, 63] that updates the corresponding entry.

When the server receives a connection from a client, half of the conversation is established. To establish the other half, the server issues a conversation request to the controller in order to fetch a Path ID. This is realized upon reception of the first packet, and before returning any packet to the client. The server must have some way to identify the other end of the conversation. Using the classical IP 5-tuple is not sufficient as the controller does not have protocol-level information such as source and destination ports. The only simple, unique identifier of that particular conversation is the Path ID used by the client. To enable the server to use it as reference, the client's Path ID is embedded in the connection request. This is realized using the Segment Routing Header (SRH), further discussed in Section 3. Instead of resolving an application name, the server issues a conversation request for the Path ID. This request may also include traffic requirements. The controller allocates a network path for the other direction of the conversation. A new Path ID is mapped to this network path and returned to the server.

Figure 1 shows an illustration of the conversation request and server registration workflows. In exchange `(A)`, the server issues a server registration request to the controller. In exchange `(B)`, the client issues a conversation request towards that same server, with a requested bandwidth of 2 Mbps. The controller replies with the IPv6 address of the server, and with a Path ID. In exchange `(C)`, the server has received a connection request from the client. The client's Path ID is embedded in the connection request. The server then issues a conversation request to the controller, stating the original Path ID and requesting the corresponding reverse Path ID, with a

```
1: allow from LAN1 to LAN2
         via FIREWALL maxidle 60s

2: allow from LAN1
         to STREAMSERVER1
         bw 5Mbps delay 10ms

3: allow from SERVER1
         to EXTERNAL_BACKUP
         bw 100Mbps
```

(a) Examples of controller rules.

Keyword	Argument	Type	Role
allow	∅	Action	Accept the conv. req.
deny	∅	Action	Reject the conv. req.
from	*name*	Matching	Specify source app.
to	*name*	Matching	Specify destination app.
via	*list of nodes*	Property	Set loose path
last	*node*	Property	Set last node of path
bw	*integer*	Property	Bandwidth to reserve
delay	*integer*	Property	Maximum one-way delay
lifetime	*integer*	Property	Max conv. life time
maxidle	*integer*	Property	Max conv. idle time

(b) List of available keywords in rules syntax.

Figure 2: Illustration of the controller rules through examples and keyword descriptions.

bandwidth requirement of 6 Mbps. The controller replies with the relevant Path ID and the exchange of packets continues.

The enterprise network can be connected to one or more upstream providers. As such, client applications may initiate connections towards external servers. Conversely, external applications may initiate connections towards internal servers. We consider three types of conversations with respect to the application locations.

Internal client communicating with internal server. This is the main kind of conversation we focus on. When a client initiates the connection, it sends a conversation request to the controller. The request states the server application and optional traffic requirements. The controller allocates a path in the network for the client-server direction of the conversation and returns a corresponding Path ID `P1` to the client. This Path ID is embedded in the subsequent connection request packet sent to the server using Segment Routing features that are independent of the transport protocol. Upon reception of the connection request packet, the server issues a conversation request. This request includes the received Path ID, `P1`, and asks for the reverse one. The controller replies with a Path ID `P2`, corresponding to the server-client direction of the conversation. Packets can now be exchanged and each application can re-issue a conversation request at any time to update the requirements of their direction of the conversation.

Internal client communicating with external server. The controller can only control one direction of the conversation because the server resides outside the enterprise network. The connection establishment procedure does not change. However, the other direction of the conversation, *i.e.*, the return traffic, will have to follow some default policies set up at the edge of the network. Such policies might include a detour via a firewall to ensure the traffic is legit. The operator defines those default policies in the controller, which configures the border routers to implement them.

External client communicating with internal server. In this case, the connection establishment originates from outside the enterprise network. As the external client uses its own DNS resolver rather than the enterprise resolver, the controller cannot handle this part of the conversation. The ingress traffic follows default network policies configured at the border routers (*e.g.*, firewall traversal). The server has the opportunity to issue a conversation request, retrieving a Path ID for the server-client direction of the conversation.

Many SDN solutions allow the network operators to configure or program the controller with rules or specific languages [50, 57]. SRNs also support such operator-defined policies. We define those policies in a per-rule fashion. Each rule matches a set of conversation requests and defines the actions to apply and the properties to implement. Figure 2a illustrates these rules. Table 2b details the main keywords supported by our current rules syntax. The rules match conversation requests based on the source and destination application. When the controller receives a conversation request with a Path ID as name (*i.e.*, to request a Path ID for the opposite direction of a conversation), it simply performs a rule lookup by inverting the source and destination applications of the initial half-conversation.

4 THE SDN RESOLVER

The *SDN Resolver* is the logical controller that manages an SRN. It must accept, process and maintain conversation requests issued by applications. To realize this, it exposes the network state and the conversation requests to externally pluggable path selection algorithms. Those algorithms then select a path that matches the requested conversation properties. This path is transformed into Segment Routing (SR) instructions by the *SDN Resolver* and enforced in the network.

We leverage the *binding segments* [20] to implement the Path IDs presented in Section 2. Indeed, a binding segment is the unique identifier of an SR policy, which encodes a path in the network. Each host is configured to send its packets with an SRH containing two segments: the first one is a binding segment (*i.e.*, its Path ID) mapped to an SR policy on the access router and the second one is the final destination of the packet (*e.g.*, a server or client application). The packets sent by the host thus follow the shortest path up to the access router, using regular IPv6 forwarding. Then, they are encapsulated within an outer IPv6 header and the SRH computed by the controller. Hence, the conversation state, implemented with SR policies, is only maintained by edge routers. The core routers only need to be configured with stateless segments that can then be used by the controller to enforce specific paths in the network. These segments can be shared by any number of SR policies. Such a stateless core has the advantage of reducing memory requirements and avoiding conversation state synchronization between routers.

4.1 Path segmentation

Network paths implementing application policies are enforced using Segment Routing. Once an appropriate path is selected, it must be transformed into a list of segments. To realize this, we leverage in our prototype the MinSegECMP path segmentation algorithm proposed in [2]. We implement the segmented path construction function buildSegpath. From a graph G, a source node s, a destination node t, and a set of policies $\mathcal{P}$, this algorithm computes (i) the full path from the source to the destination that matches the policies (*path computation*), and (ii) the minimal list of segments that implements the previously computed path (*path segmentation*). The path computation is realized through a generic selectPath function. This function must be extended by any external algorithm that provides path selection features. Then, path segmentation is achieved using MinSegECMP. This algorithm computes the minimal ECMP-free segmentation of a path.

4.2 SRN Control plane

The control plane of an SRN consists of several components. At the core is the logically centralized controller (the *SDN Resolver*). It does not communicate directly with applications. Instead, the exchanges between the applications and the controller are mediated by a DNS proxy. The proxy receives DNS requests from applications. It performs the actual DNS resolution, using the enterprise's resolver, and forwards the conversation requests to the controller. The controller processes the request, then instructs the access router to insert the resulting SR policy into its Forwarding Information Base (FIB). The controller explicitly waits for the SR policy to be inserted. This synchronization is necessary to ensure that the router does not receive legitimate packets with a binding segment that is not yet recognized in the FIB. Afterwards, the controller returns the generated binding segment (*i.e.*, Path ID) to the DNS proxy. Ultimately, the proxy crafts the corresponding DNS reply, using the resolved address and the binding segment. This reply is transferred to the application. Assuming that the controller, proxy, and resolver are in the same network vicinity, this setup allows to group most of the transactions within a low-delay network radius.

The last component of an SRN is the Network State Daemon (NSD) which gathers the network state as exposed by OSPF-TE and forwards it to the controller. The rate of network state updates depends on the OSPF refresh timer defined by the operator. The value should be a trade-off between control traffic overhead and up-to-date metrics. A high-level illustration of an SRN architecture is shown in Figure 3.

4.2.1 Interfaces. The controller's northbound interface (*i.e.*, facing the applications) leverages the DNS protocol. We extend DNS with a new type of *Resource Record* called `BSID`. This record carries a binding segment implementing a Path ID encoded as an IPv6 address. We also use EDNS0 [61] to carry metadata in DNS messages. We define three new option codes to carry the requested bandwidth, requested latency and application identifier. The flexibility of EDNS0 enables to easily define new options in the future.

In our prototype, the communications between the controller and its components (*i.e.*, southbound interface) are realized through the OVSDB protocol [46]. Originally designed for Open vSwitch, OVSDB is a generic, JSON-RPC based protocol, supporting transactional queries on NoSQL-like databases. However, any per-flow configuration protocol can be used. One can make a different choice to match the protocols available on his access routers (*e.g.*, BGP-LS [24, 48] or XMPP [52]).

We define five database tables for *SDN Resolver*. We call this set of tables the *Segment Routing Database* (SRDB). `ConvReq` stores the conversation requests generated by the applications and translated by the DNS proxy. It contains the source and destination of the request, the resolved destination address, traffic requirements, and a status. The status is set to `PENDING` when the entry is inserted by the DNS proxy. If the request is accepted by the controller and the conversation is created, the controller changes the status to `ALLOWED`. Otherwise, the status is set to a value that reflects the reason why the conversation was not created (*e.g.*, administrative deny, impossible to satisfy the traffic requirements, etc.). This table is written and read by both the DNS proxy and the controller. `ConvState` stores the state for half-conversations. Each entry contains the source and destination applications, traffic requirements, Path ID and mapped segments, expiration timers, etc. It is written by the controller and read and written by the routing daemon. The write access of the routing daemon is required to enable the removal of expired conversations (*e.g.*, due to idle timeout). `ServReg` stores the server registration requests. It contains the name and address of the requesting servers, and a status field that has the same semantics as for the `ConvReq` table. It is written and read by both the DNS proxy and the controller. `LinkState` and

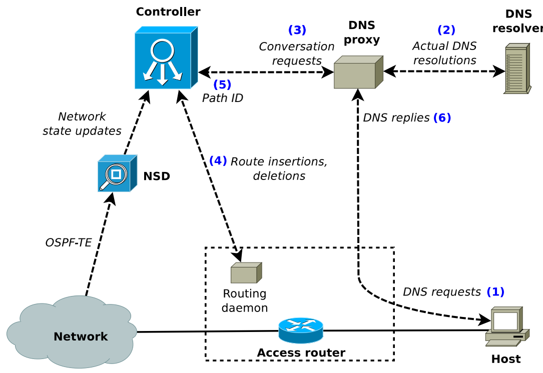


Figure 3: Illustration of the components of a SRN. The figure shows the exchanges involved in a conversation request.

`NodeState` store topology data gathered through OSPF-TE, such as announced prefixes, link utilization, etc. They are written by the NSD and read by the controller.

4.2.2 Operations. To ensure the correct operation of the network according to the principles described in Section 2, our controller includes at least two processes: conversation requests and server registrations (see Figure 1). We also support a third operation: reactions to network events. Those processes operate as follows.

Conversation request. The controller matches each conversation request against the rules defined by the operator. The last matching rule wins. When a rule matches, the controller applies the rule's main decision: accept or deny. In the latter case, an error is returned to the application and the process stops. In the former case, the controller combines the policies defined in the rule and the path constraints provided by the conversation request into a final set of policies. Once the final set of policies is defined, it is translated into a list of segments. Then, the controller generates a binding segment and creates a route that implements the SR policy. This route matches packets for that particular binding segment and encapsulates them with the previously computed list of segments. This route is immediately inserted using the chosen per-flow configuration protocol (*e.g.*, OVSDB, BGP-LS, XMPP,. . .) into the access router of the initial requesting application. Note that if the application is susceptible to use more than one access router (*e.g.*, with VRRP [44]), then the route would be inserted in all concerned routers. Finally, the binding segment is returned to the application. The controller keeps the resulting state in memory until expiration. For resiliency purposes, the controller may leverage a dedicated algorithm for backup path computation. Those backup paths should be configured as such in the access router and associated to the same binding segment.

Server registration. When the controller receives a server registration request, it uses the DNS update mechanism [51, 63]. If there is no pre-existing DNS record for the server name, then a new record is created with a given TTL. If the server is already part of an existing record, then its TTL is refreshed. Otherwise, the server is added to the list of entries associated with this name. The server receives a DNS reply with an associated TTL. The server must refresh its registration before the expiration of the TTL.

Network event. We consider as a *network event* any link or node failure that affects active conversations. We also consider sudden increases in link utilization or delay that would break conversation requirements. The controller does not have the same reaction time for these events. A link failure is quickly propagated by OSPF [23]. A node failure is detected when all its neighbors have reported the loss of the adjacency, which can take some time. The link bandwidth utilization and delays are updated by using adaptive timers and thresholds by OSPF-TE implementations [53]. When the controller detects a network event, it scans its conversation state database and builds a list of affected conversations. In case of link or node failure, all conversations that traverse the failed link or node are affected. In case of congestion or delay increase, the affected conversations are those whose path no longer matches the traffic requirements. For each affected conversation, the controller looks for pre-computed backup paths. If such a path exists and satisfies the requirements, it is selected as replacement. Otherwise, the controller recomputes a new path that suits the conversation policies. The controller updates the conversation and pushes the new state to the affected routers. In turn, they change their FIB. Note that the controller can change a path without even interacting with the application, as Path IDs provide a level of indirection to the actual segmented path implemented on the access router. This is the key benefit of such indirection. We described a very basic algorithm for online path recomputation. More advanced approaches (*e.g.*, recompute non affected paths to reach a more optimal state) discussed in the literature [15, 49] could also be included in the controller.

If the controller fails, then many features become unavailable. However, active conversations are not affected. New conversations can be handled by default SR policies, configured on access routers for packets lacking a binding segment. To avoid switching too quickly to degraded mode, more controllers may be added to the network. Two or more controllers can act in a master-slave fashion. A detailed discussion of *SDN Resolver* fault tolerance may be found in [37].

4.3 Security

From a security viewpoint, an *SDN Resolver* is exposed to the same security risks as the enterprise's internal DNS resolver.

As such, packets from untrusted sources should not be able to reach the controller, as it is already protected by the enterprise firewall.

The introduction of SRv6 in the enterprise network implies the support of the IPv6 Segment Routing header extension. As such, it is crucial that SR-enabled IPv6 packets originating from outside the enterprise network are filtered by the border routers. Otherwise, an external attacker could gain unauthorized access to the enterprise network resources. The mitigation for this threat is simply to configure the border routers to drop all SR-enabled packets originating from an external interface [21].

Internal threats come from, *e.g.*, malicious employees or applications. They want to get more services from the SDN Resolver than allowed. Two actions can counter such abuses. First, to prevent misuses of network resources, only authorized equipment should be allowed to emit arbitrary SRHs on the network. Network administrators can leverage existing ACLs to prevent unauthorized usage of segments. Access control schemes such as 802.1x can be deployed to control the access to the layer-2 network. Furthermore, the interactions between the clients and the SDN Resolver can be protected by using techniques such as DNS over TLS [28].

4.4 Supporting legacy devices

To fully leverage the capabilities of our *SDN Resolver*, applications need to explicitly interact with the controller. There are legacy applications that do not understand our DNS extensions and/or cannot install the Path ID. In this case, instead of using a binding segment, the Path ID can be implemented as special destination address returned in the DNS reply. Packets using this particular destination are mapped to an SR policy on the access router. Before applying the policy, the destination address would be translated to the real destination address, using NAT rules dynamically inserted by the controller.

5 PROTOTYPE IMPLEMENTATION

To assess the performance of our proposed architecture, we developed a fully functional prototype implementation. It runs on Linux clients, routers, servers and controllers. Overall, our prototype comprises about 10,000 lines of C code. We describe the main components of this prototype in this section.

5.1 Kernel modifications

The Linux kernel, since version 4.10, already includes basic support for SRv6 [38]. However, this release does not explicitly support binding segments, which are required for the Path IDs that are used in our architecture. To use Path IDs, routers must be able to encapsulate a packet when its active segment matches a given address. The active segment is defined as the destination address of the packet.

The recent SRv6 network programming specifications [21] define several functions on SR-enabled packets that a segment endpoint must support. One of these functions is the binding segment. We extended the SRv6 kernel implementation to support these specifications. Our extension is available in the mainline Linux kernel for the 4.14 release. [2] It consists in a new type of lightweight tunnel (named `seg6local`) specified by an action type, which defines the function to apply, and optional parameters. Packets can be steered through a given function by creating a route for an arbitrary prefix, associated with an ad-hoc `seg6local` lightweight tunnel. Such an architecture enables to easily add new functions to the existing ones.

5.2 Path ID propagation

In Figure 1, after the client has established the connection, the server requests a binding segment for the server-client direction of the conversation. If the controller is able to infer the full requirements of the conversation from the client request, then the server-side request is superfluous. This can speed up the connection establishment process. To realize this, we implement the following solution.

When the client issues its conversation request, the controller immediately computes a network path for both directions of the conversation and maps them to two Path IDs (resp. P_{c-s} and P_{s-c} for the client and the server). The controller then inserts a particular SR policy into the client's access router. This SR policy maps P_{c-s} to the corresponding encapsulation, but also instructs the router to *overwrite* P_{c-s} with P_{s-c} in the SRH before the encapsulation. The server then receives a packet with its own Path ID present in the SRH, instead of the client's Path ID. Then, it simply needs to echo the binding segment. Once the conversation is fully established, the server is free to request an update of the conversation with its Path ID, *e.g.*, to reflect changes in traffic requirements. This technique has obvious security implications. Blindly echoing a binding segment is a process that must be strictly controlled. Allowing only authorized network equipment to emit arbitrary SRHs as explained in Section 4.3 should prevent misuses of the feature. Additionally, the HMAC feature of SRv6 can be leveraged to ensure the authenticity and integrity of the SRH.

5.3 Controller implementation

The controller is implemented as a heavily multi-threaded program. It consists of three main subsystems.

First, the *graph* subsystem implements all the structures, operations and algorithms that deal with the network topology represented as graphs. The graph nodes and edges contain a generic `data` pointer that can provide additional information (*e.g.*, metric, link latency, bandwidth, etc.). Three helper

[2]See https://lkml.org/lkml/2017/9/6/11 for the list of commits.

hashmaps are precomputed to speed up the path computations. The first one (`neighs`) maps each node to its neighbors. The second one (`min_edges`) maps each pair of neighbors to the lowest cost edge that connects them. The last one (`dcache`) maps each node to its precomputed shortest-path Directed Acyclic Graph (SP-DAG). The first two hashmaps speed up the Dijkstra algorithm. The third one speeds up the MinSegECMP algorithm, which requires switching between multiple SP-DAGs. To make graphs friendly to multi-threaded environments, each graph is protected by a read-write lock. Moreover, each node and edge structure contains a reference counter. This enables to hold references to them outside critical sections (*i.e.*, when the per-graph lock is not taken).

Second, the *Segment Routing Database* (*SRDB*) subsystem implements an abstraction layer to interact with the OVSDB server. It provides an API for two classes of operations: watching for OVSDB table updates (*monitors*) and performing insertions, updates, and deletions on OVSDB tables (*transactions*). A dedicated thread monitors each table and calls a user-defined callback function whenever a change happens. Transactions are performed by a pool of threads, each maintaining a permanent TCP connection with the OVSDB server. A shared thread-safe buffer is used to pass the transactions to the thread pool. Each transaction consists of the data to transmit as well as a one-element thread-safe result buffer. When the result of the transaction is available, it is pushed into the buffer by the processing thread. Such a producer-consumer architecture enables to (*i*) scale the number of transaction threads with the database load and (*ii*) support asynchronous transactions. This subsystem is also used by other components of the *SDN Resolver*, such as the DNS proxy.

The third subsystem is the *core* of the controller. Using the *graph* and *SRDB* subsystems, the *core* implements the necessary features to run a *Software Resolved Network*. It maintains a *global state* that consists of three sets of data: (*i*) the operator-defined policies, (*ii*) the current network state, and (*iii*) the active conversations. The network state consists of two graphs. A *production* graph is used to perform the path computations, and a *staging* graph is used as a buffer to store the link-state changes. This global state is processed and updated by three major components. The first component is the set of monitoring threads, each of them watching an OVSDB table and calling an associated callback function when necessary. The callbacks for the `NodeState` and `LinkState` tables update the staging graph accordingly. The callback for the `ConvReq` table extracts the conversation request and stores it in a shared thread-safe buffer. This buffer is consumed by the second component, which is a pool of *worker threads*. The worker threads handle most of the controller's workload. Concurrently, they match the requests against the operator policies, compute a path between the source and destination, generate an associated binding segment, create an internal conversation state and commit the newly created conversation to the `ConvState` table. The third component is the network monitoring thread. At configurable intervals, it sets the staging graph as the production graph if the network state changed, it recomputes potentially affected conversations, and garbage collects expired conversations.

5.4 Application API

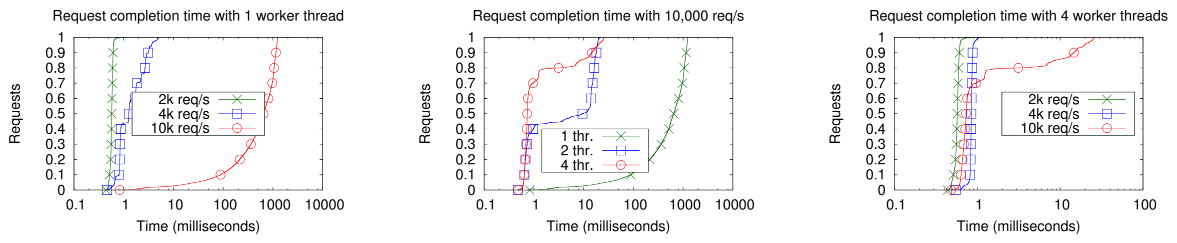
To facilitate the deployment of SR-aware applications, we implement a user API that abstracts the utilization of the DNS extensions and the interactions with the kernel. To support our DNS extensions, we modify the `c-ares` DNS library [9]. We implement a custom library, `libsrdns`, that provides an API to handle SR-enabled sockets. For example, the library exposes the `sr_socket()` function that takes as input the socket type (*e.g.*, TCP or UDP), the destination name and port, the local application name, and optional bandwidth and latency parameters. Using this input, the function (*i*) resolves the destination name and fetches any associated binding segment, (*ii*) creates a socket with the corresponding type and (*iii*) attaches an SRH with the binding segment to the socket.

6 EVALUATION

In this section, we evaluate our *SDN Resolver* through two angles. First, we use microbenchmarks to measure several aspects of its performance. Then, we evaluate how the *SDN Resolver* behaves when all its components are integrated in an emulated network. Those evaluations are performed on a four years old laptop using a Core i7-3740QM running at 2.7 GHz with 8 GB of RAM.

6.1 Microbenchmarks

6.1.1 Conversation requests. The fundamental operations that the controller needs to support is handling conversation requests and generating the corresponding conversation state. We measure how the controller handles conversation requests under a high load. We proceed as follows. First, we initialize an empty OVSDB database with the SRDB schema. We populate the `NodeState` and `LinkState` tables using the Abilene topology [29], composed of 11 nodes and 15 links. This topology can represent the core of a middle-sized enterprise network. With a benchmarking tool that leverages our SRDB subsystem, we generate batches of conversation requests, at a configurable uniform rate, and without path constraints. The source and destination of each request are selected at random. We measure the delay between the insertion of the request in the `ConvReq` table and the insertion of the conversation state in the `ConvState` table The request completion time is defined as the time elapsed between those two events. As such, it measures (*i*) the insertion of the conversation request in OVSDB, (*ii*) the reception of the



(a) A single worker can handle loads of several thousand requests per second

(b) Various worker threads for 10,000 reqs/s.

(c) Four worker threads for various loads.

Figure 4: Request completion times under different conditions.

`ConvReq` entry by the controller's monitor, (*iii*) the processing of the request and the generation of the conversation state, (*iv*) its insertion in OVSDB, and (*v*) the reception of the `ConvState` entry by the benchmarking tool's monitor.

To estimate the load that an *SDN Resolver* would need to sustain, we looked at the main DNS resolver of our university campus. This campus gathers about 5,000 employees and 28,000 students. Three DNS resolvers serve the campus and the load is well balanced between them. Their statistics show that the DNS resolvers load never exceeds 1,000 requests per second. We consider this as a baseline for our benchmarks.

In a first batch of measurements, we evaluate how the controller performs over various request rates, using a single worker. The results are shown in Figure 4a. At 2,000 requests per second, the requests are consistently processed within less than one millisecond. At 4,000 reqs/s, the completion time noticeably increases for more than half of the requests but stays below 10 milliseconds. At 10,000 reqs/s, the controller is unable to cope with the load and the completion time increases by three orders of magnitude.

In a second batch of measurements, we measure the horizontal scaling of the controller with additional workers. We load this controller with 10,000 reqs/s. The results are shown in Figure 4b. As previously shown, a single worker is unable to cope with this load and the completion time quickly reaches about one second. Doubling the number of workers (*i.e.*, 2 workers) consistently improves the completion time. About 40% of the requests are completed within less than a millisecond. The remaining 60% require between 10 and 25 milliseconds. By doubling again the number of workers (*i.e.*, 4 workers), about 70% of the requests complete within less than a millisecond. The remaining 30% take between 10 and 25 milliseconds.

In Figure 4c, we show the evolution of the request completion time for various rates using 4 workers. At 2,000 and 4,000 reqs/s, the request completion time remains below one millisecond. As previously shown, 70% of the requests at 10,000 reqs/s are also completed within a millisecond.

Existing OpenFlow controllers such as Beacon and NOX have been shown to return responses at a higher rate [16, 54, 59]. However, our *SDN Resolver* will receive several orders of magnitude less requests than a standard OpenFlow controller. Indeed, only conversation requests (*i.e.*, DNS requests) are handled by our *SDN Resolver*. In contrast with an OpenFlow controller, an *SDN Resolver* does not need to act on a per flow basis. Furthermore, DNS caching is known to perform very well [32] and directly applies to an *SDN Resolver*.

Those benchmarks are significant for two reasons. First, they show that the controller is able to cope with the typical load of a DNS resolver in a large enterprise network. Furthermore, it efficiently scales with respect to the available CPU. The benchmarks also show that the reference implementation of the OVSDB protocol can sustain such a load. This result is important as it shows that, while OVSDB was not originally designed as a signalling protocol, it can still be used as such without major performance issues, as proposed in [14].

6.1.2 Reaction to link failures. Link failures are inevitable in a network. When a link fails, the *SDN Resolver* potentially needs to recompute all the paths that were using the failed link. This recomputation is not required to preserve the connectivity since IGP convergence and SR-based fast reroute techniques cover this part of the recovery. We evaluate how quickly the controller handles the recomputation of conversations in case of link failures. Using the Abilene topology, we generated various numbers of conversation states with random sources and destinations, without path constraints. Then, we simulated random link flaps by updating entries in the `LinkState` table. When such an event happens, the controller sweeps the conversation states to decide which one must be recomputed. Without path constraints, the controller simply recomputes a new shortest path for conversations that were using the faulty link. We observe that there is a constant delay of approximately 5 milliseconds. This corresponds to the controller's recomputation timer, which enables to absorb bursts of link state changes. In average, the time spent to process each conversation is approximately 0.6 microseconds. With a database of 100,000 active conversations, the average full sweep is completed within 65 milliseconds. Due to space limitations, we do not provide a figure for these results but it can be found [37].

After the sweep, the controller recomputes the affected conversations. It takes about 0.2 milliseconds to recompute a single conversation. Considering 100,000 active conversations, in the extremely unlikely case where all the conversations are affected by a link failure, it would take 20 seconds to sequentially recompute all of them. This time can be reduced by distributing the recomputations over multiple threads.

6.2 Emulated network

To evaluate our SRN architecture in an end-to-end setting, we instantiated a virtual network in a Mininet-like environment. The topology is shown in Figure 5a. Each node is assigned an IPv6 prefix and contains pre-computed shortest-path routes towards all other nodes. The `Controller` node contains the main components of an *SDN Resolver*, *i.e.*, the controller, DNS proxy, and OVSDB server. It also hosts a regular DNS server. The DNS server maps the `server.test.sr` domain name to the main IPv6 address of the server node. Nodes `A` and `F` are access routers and each of them contains a routing daemon.

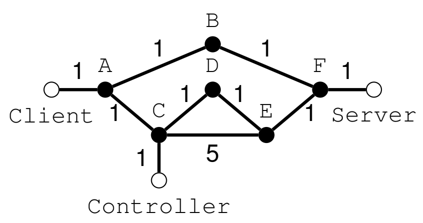
In a first experiment, we generate conversation requests for `server.test.sr` from the client. We measure the time needed to complete the request, as seen by the client. Then, we compare this delay against the time needed to complete a regular DNS request. In a first batch of measurements, we use the network topology as shown in Figure 5a. The round-trip time between the client and the controller is 6 milliseconds. In a second batch of measurements, we increase the delay of each link between the client and the controller to 5 milliseconds. As a result, the round-trip time between the client and the controller increases to 30 milliseconds. The results are shown in Figure 5b. We observe that the conversation request completion time consists of one RTT between the client and the controller (*i.e.*, the DNS conversation), and one RTT between the access router and the controller (*i.e.*, the route installation), plus, on average, a constant overhead of 5 milliseconds. This overhead can be explained by the fluctuations of the virtualized environment. Note that 30ms for the RTT between the controller and the client is not realistic in an enterprise network but this is useful to identify the causes of the delay.

In a second experiment, we observe the effects of link failures on active conversations. To realize this, we use the network topology as shown in Figure 5a. First, we request a conversation between the client and the server, with a minimum-latency path constraint. The controller then configures router `A` with a binding segment B_1 mapping to the computed list of segments. In the initial network state, this list contains only the segment for node `F`. Indeed, the minimum latency path is also the shortest IGP path. Similarly, router `F` is configured with another binding segment B_2 mapping to a list of segments that consists of node `A`. The client node is configured to use the binding segment B_1 for all packets sent to the server node, and the server node is configured to use the binding segment B_2 for all packets sent to the client node. Then, from the client node, we run ICMPv6 echo-request measurements every 10 milliseconds. This precision was the best we could obtain with the `ping6` tool in our virtualized environment. Then, we repeatedly shut down the $(\mathsf{A}-\mathsf{B})$ link, wait for the controller reaction, then switch the link up again.

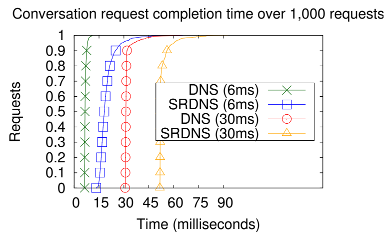
Figure 5c shows one cycle of this experiment. Between $t = 0$ and $t = 80$ ms, the path is the best possible one, *i.e.*, $(\mathsf{A}-\mathsf{B}-\mathsf{F})$ with a round-trip time of roughly 8 ms. At $t = 80$ ms, link $(\mathsf{A}-\mathsf{B})$ is shut down. It takes about 30 milliseconds for the routes to change and the new path to be visible. The path converges to the now-shortest path, *i.e.*, $(\mathsf{A}-\mathsf{C}-\mathsf{E}-\mathsf{F})$. However, this path is also the longest-delay path with a round-trip time of about 18 ms. The controller is notified of the link failure and recomputes the affected paths. At $t = 170$ ms, the recomputed path is visible. During that time interval, (*i*) the controller was notified of the link failure, (*ii*) recomputed the new paths, (*iii*) updated the `ConvState` table with the updated list of segments, (*iv*) the routing daemons on nodes `A` and `F` were notified of the update, and (*v*) they updated their routing table to reflect the new list of segments. The new lists of segments are now resp. $\langle\mathsf{D},\mathsf{F}\rangle$ and $\langle\mathsf{D},\mathsf{A}\rangle$ for nodes `A` and `F`, which correspond to the current minimum latency path. The measured round-trip time is about 12 milliseconds. At $t = 420$ ms, the link $(\mathsf{A}-\mathsf{B})$ is brought back up and the new IGP routes converge at $t = 450$ ms. Note that the path does not change after the convergence. Indeed, the current lists of segments force the packets to transit through node `D`. However, the link state change triggers path recomputations and the controller's updates are visible at $t = 480$ ms. The packets then resume their original shortest and minimum latency path.

7 RELATED WORK

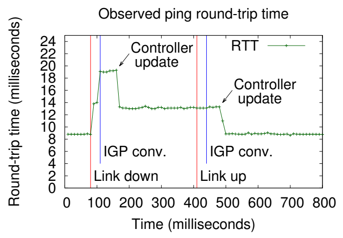
Software Defined Networks have been a hot topic in the research community since the publication of [43]. Several survey papers have analyzed this vast literature in details [17, 35, 65]. In this section, we compare Software Resolved Networks (SRN) with several of the key related work. We



(a) Emulated network topology. Unit link weights. Numbers are link delays (ms).



(b) Conversation request completion time for various RTTs.



(c) Controller reaction to link failure.

Figure 5: Emulated network experiments.

structure the comparison according to the bottom-up approach adopted in section IV of [35].

The dataplane layer of SRNs differs from many proposed SDN solutions. SRNs operate in IPv6 networks. To fully benefit from SRNs, the ingress routers need to support the binding segments that are part of SRv6 [21, 47]. Given that all IPv6 routers can forward packets with an SRH, it is possible to incrementally deploy SRNs starting with some upgraded hosts and ingress routers.

A second difference between SRNs and OpenFlow-based SDNs is the southbound interface. SDNs rely on the OpenFlow protocol to configure flow tables on the switches. Future deployments could leverage more programmable switches [6]. In SRNs, the controller interacts with the routers through OVSDB tables. Note that while traditional SDN networks require the installation of flow tables on all network devices, in SRNs the controller only interacts with the edge routers. The controller does not need to interact with the other routers which improves the scalability of SRNs. SRNs leverage Segment Routing to select and enforce network paths. Several SDN solutions also encode paths inside packets. Hari et al. propose in [25] to encode network paths inside the layer-2 MAC addresses of the packets on the first switch of the path. Jeyakumar et al. propose in [31] to leverage the 20-bit flow label field in the IPv6 header and the 6-bit DS field in the IPv4 header to dynamically parametrize middleboxes.

Another important difference is that the endhosts participate actively in SRNs with their DNS requests. This implies that our SDN resolvers can use policies based on DNS names and not only addresses and ports. Other SDN solutions such as PANE [18] use a related approach. PANE proposes an API enabling the applications to interact with the controller. This API is implemented with a new protocol, while SRNs extend the DNS protocol. Beyond SDN and enterprise networks, researchers have proposed several architectures where content is retrieved directly with names [30, 34].

8 CONCLUSION

In this paper, we proposed Software Resolved Networks, an SDN-like architecture for enterprise networks. Like OpenFlow-based SDN solutions, SRNs enable the network operators to specify policies that control the network paths that are used by applications. For this, Software Resolved Networks build upon the IPv6 Segment Routing architecture (SRv6) and the DNS protocol. Applications use the DNS as a signalling protocol to request the creation of end-to-end network paths. Those paths are computed by a controller that interacts with the DNS resolver and returns the selected path to the requesting application as a segmented SRv6 path.

There are two important differences between SRNs and traditional SDNs. First, SRNs enable the applications to directly interact with the controller to specify path requirements like delay or bandwidth. Second, SRNs do not require per-flow state in all network nodes. The controller installs state on the access routers but not on the core routers. This improves the scalability of SRNs.

We implement a complete Software Resolved Network on Linux hosts, routers and servers. Our performance evaluation demonstrates that our prototype meets the performance expectations of enterprise networks.

Software Artifacts

All the code for our prototype implementation of *SDN Resolver* are publicly available from http://segment-routing.org/index.php/SRN. We also provide measurement tools enabling other researchers to reproduce our results.

ACKNOWLEDGEMENTS

This work was partially supported by a Cisco grant and by the ARC grant 13/18-054 (ARC-SDN) funded by Communaute francaise de Belgique. M. Jadin is supported by a grant from F.R.S.-FNRS FRIA.

REFERENCES

[1] Alia Atlas et al. 2015. OSPF Traffic Engineering (TE) Metric Extensions. RFC 7471. (14 Oct. 2015). https://doi.org/10.17487/rfc7471

[2] Francois Aubry, David Lebrun, Stefano Vissicchio, Minh Thanh Khong, Yves Deville, and Olivier Bonaventure. 2016. SCMon: Leveraging segment routing to improve network monitoring. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524410

[3] Haythum Babiker, Irena Nikolova, and Kiran Kumar Chittimaneni. 2011. Deploying IPv6 in the Google Enterprise Network. Lessons learned.. In *Proceedings of the 25th international conference on Large Installation System Administration, LISA*. 10.

[4] Ahmed Bashandy, Clarence Filsfils, Bruno Decraene, Stephane Litkowski, and Pierre Francois. 2017. *Abstract*. Internet-Draft draft-bashandy-rtgwg-segment-routing-ti-lfa-00. Internet Engineering Task Force. Work in Progress.

[5] R. Bhatia et al. 2015. Optimized network traffic engineering using segment routing. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. 657–665. https://doi.org/10.1109/INFOCOM.2015.7218434

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[7] Robert T. Braden, Dr. David D. Clark, and Scott Shenker. 2013. Integrated Services in the Internet Architecture: an Overview. RFC 1633. (2 March 2013). https://doi.org/10.17487/rfc1633

[8] J. Brzozowski et al. 2017. IPv6 SPRING Use Cases. (April 2017). Internet draft, draft-ietf-spring-ipv6-use-cases-10, work in progress.

[9] C-Ares. [n. d.]. https://c-ares.haxx.se. ([n. d.]).

[10] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. 2007. Ethane: Taking Control of the Enterprise. In *SIGCOMM '07*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1282380.1282382

[11] Lorenzo Colitti, Dr. Vinton G. Cerf, Stuart Cheshire, and David Schinazi. 2016. Host Address Availability Recommendations. RFC 7934. (July 2016). https://doi.org/10.17487/rfc7934

[12] Lorenzo Colitti, Steinar H Gunderson, Erik Kline, and Tiziana Refice. 2010. Evaluating IPv6 adoption in the Internet. In *Passive and active measurement*. Springer, 141–150.

[13] Jakub Czyz, Mark Allman, Jing Zhang, Scott Iekel-Johnson, Eric Osterweil, and Michael Bailey. 2015. Measuring IPv6 adoption. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 87–98.

[14] Bruce Davie, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Natasha Gude, Amar Padmanabhan, Tim Petty, Kenneth Duda, and Anupam Chanda. 2017. A Database Approach to SDN Control Plane Design. *SIGCOMM Comput. Commun. Rev.* 47, 1 (Jan. 2017), 15–26. https://doi.org/10.1145/3041027.3041030

[15] Hilmi E Egilmez, S Tahsin Dane, K Tolga Bagci, and A Murat Tekalp. 2012. OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks. In *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific*. IEEE, 1–8.

[16] David Erickson. 2013. The beacon openflow controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 13–18.

[17] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2013. The road to SDN. *Queue* 11, 12 (2013), 20.

[18] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. 2013. Participatory Networking: An API for Application Control of SDNs. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 327–338. https://doi.org/10.1145/2534169.2486003

[19] Dennis Ferguson et al. 2015. OSPF for IPv6. RFC 5340. (14 Oct. 2015). https://doi.org/10.17487/rfc5340

[20] Clarence Filsfils et al. 2016. *Segment Routing Architecture*. Internet-Draft draft-ietf-spring-segment-routing-08. Internet Engineering Task Force. Work in Progress.

[21] Clarence Filsfils et al. 2017. *SRv6 Network Programming*. Internet-Draft draft-filsfils-spring-srv6-network-programming-00. Internet Engineering Task Force. Work in Progress.

[22] Clarence Filsfils, Francois Clad, Pablo Camarillo, Jose Liste, Prem Jonnalagadda, Milad Sharif, Stefano Salsano, and Ahmed AbdelSalam. 2017. IPv6 Segment Routing. In *SIGCOMM'17, Industrial demos*.

[23] Pierre Francois, Clarence Filsfils, John Evans, and Olivier Bonaventure. 2005. Achieving Sub-second IGP Convergence in Large IP Networks. *SIGCOMM Comput. Commun. Rev.* 35, 3 (July 2005), 35–44. https://doi.org/10.1145/1070873.1070877

[24] H. Gredler, J. Medved, S. Previdi, A. Farrel, and S. Ray. 2016. North-Bound Distribution of Link-State and Traffic Engineering (TE) Information Using BGP. RFC 7752. (March 2016). https://rfc-editor.org/rfc/rfc7752.txt

[25] Adiseshu Hari, TV Lakshman, and Gordon Wilfong. 2015. Path switching: reduced-state flow handling in SDN using path information. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 36.

[26] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. 2015. A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks. In *SIGCOMM '15*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/2785956.2787495

[27] T. Hollmann. 2016. A history of IPv6 challenges in Facebook data centers. (2016). Network @Scale conference.

[28] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman. 2016. Specification for DNS over Transport Layer Security (TLS). RFC 7858. (May 2016). https://rfc-editor.org/rfc/rfc7858.txt

[29] Internet2. [n. d.]. Historical Abilene Data. http://noc.net.internet2.edu/i2network/live-network-status/historical-abilene-data.html. ([n. d.]).

[30] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. 2009. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 1–12.

[31] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2015. Millions of little minions: Using packets for low latency network programming and visibility. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 3–14.

[32] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. 2002. DNS Performance and the Effectiveness of Caching. *IEEE/ACM Transactions on networking* 10, 5 (2002), 589–603.

[33] Marcus Keane. 2017. IPv6-only at Microsoft. https://blog.apnic.net/2017/01/19/ipv6-only-at-microsoft/. (Jan. 2017).

[34] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. 2007. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM Computer Communication Review*, Vol. 37. ACM, 181–192.

[35] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (Jan 2015), 14–76. https:

//doi.org/10.1109/JPROC.2014.2371999

[36] P. Lapukhov and A. Premji. 2016. Use of BGP for Routing in Large-Scale Data Centers. RFC 7938. (Aug. 2016). https://rfc-editor.org/rfc/rfc7938.txt

[37] David Lebrun. 2017. *Reaping the Benefits of IPv6 Segment Routing*. Ph.D. Dissertation. UCLouvain / ICTEAM / EPL.

[38] David Lebrun and Olivier Bonaventure. 2017. Implementing IPv6 Segment Routing in the Linux Kernel. In *Proceedings of the 2017 Applied Networking Research Workshop*. ACM.

[39] Syam Madanapalli, Jaehoon Jeong, Soohong Daniel Park, and Luc Beloeil. 2010. IPv6 Router Advertisement Options for DNS Configuration. RFC 6106. (Nov. 2010). https://doi.org/10.17487/rfc6106

[40] David A Maltz, Geoffrey Xie, Jibin Zhan, Hui Zhang, Gísli Hjálmtỳsson, and Albert Greenberg. 2004. Routing design in operational networks: A look from the inside. In *SIGCOMM'04*. ACM, 27–40.

[41] F. Martin. 2016. IPv6 Inside LinkedIn Part II. (2016). https://engineering.linkedin.com/blog/2016/08/ipv6-inside-linkedin-part-ii–back-to-the-future.

[42] Nick McKeown et al. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[43] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.

[44] Stephen Nadas. 2010. Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6. RFC 5798. (March 2010). https://doi.org/10.17487/rfc5798

[45] Mehdi Nikkhah and Roch Guérin. 2016. Migrating the Internet to IPv6: an exploration of the when and why. *IEEE/ACM Transactions on Networking* 24, 4 (2016), 2291–2304.

[46] Ben Pfaff and Bruce Davie. 2013. The Open vSwitch Database Management Protocol. RFC 7047. (Dec. 2013). https://doi.org/10.17487/rfc7047

[47] Stefano Previdi et al. 2016. *IPv6 Segment Routing Header (SRH)*. Internet-Draft draft-ietf-6man-segment-routing-header-01. Internet Engineering Task Force. Work in Progress.

[48] S. Previdi, P. Psenak, C. Filsfils, H. Gredler, and M. Chen. 2017. BGP Link-State extensions for Segment Routing. (July 2017). Internet draft, draft-ietf-idr-bgp-ls-segment-routing-ext-03, work in progress.

[49] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying middlebox policy enforcement using SDN. *ACM SIGCOMM computer communication review* 43, 4 (2013), 27–38.

[50] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. 2013. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 109–114.

[51] Yakov Rekhter and Jim Bound. 1997. Dynamic Updates in the Domain Name System (DNS UPDATE). RFC 2136. (1 April 1997). https://doi.org/10.17487/rfc2136

[52] P. Saint-Andre. 2011. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120. (March 2011). https://rfc-editor.org/rfc/rfc6120.txt

[53] Stefano Salsano, Alessio Botta, Paola Iovanna, Marco Intermite, and Andrea Polidoro. 2006. Traffic engineering with OSPF-TE and RSVP-TE: Flooding reduction techniques and evaluation of processing cost. *Computer Communications* 29, 11 (2006), 2034 – 2045. https://doi.org/10.1016/j.comcom.2005.12.007

[54] Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. 2013. Advanced study of SDN/OpenFlow controllers. In *Proceedings of the 9th central & eastern european software engineering conference in russia*. ACM, 1.

[55] Justine Sherry et al. 2012. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 13–24.

[56] William A. Simpson, Dr. Thomas Narten, Erik Nordmark, and Hesham Soliman. 2007. Neighbor Discovery for IP version 6 (IPv6). RFC 4861. (Sept. 2007). https://doi.org/10.17487/rfc4861

[57] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. 2014. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 213–226.

[58] Yu-Wei Eric Sung, Xin Sun, Sanjay G Rao, Geoffrey G Xie, and David A Maltz. 2011. Towards systematic design of enterprise networks. *IEEE/ACM Transactions on Networking (TON)* 19, 3 (2011), 695–708.

[59] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. 2012. On Controller Performance in Software-Defined Networks. *Hot-ICE* 12 (2012), 1–6.

[60] Frederic Trate. 2016. Segment Routing is crossing the chasm with real live deployments! (June 2016). https://blogs.cisco.com/sp/segment-routing-is-crossing-the-chasm-with-real-live-deployments

[61] Paul A. Vixie et al. 2015. Extension Mechanisms for DNS (EDNS(0)). RFC 6891. (14 Oct. 2015). https://doi.org/10.17487/rfc6891

[62] Daniel Voyer et al. 2017. *Insertion of IPv6 Segment Routing Headers in a Controlled Domain*. Internet-Draft draft-voyer-6man-extension-header-insertion-00. Internet Engineering Task Force. Work in Progress.

[63] Brian Wellington. 2000. Secure Domain Name System (DNS) Dynamic Update. RFC 3007. (Nov. 2000). https://doi.org/10.17487/rfc3007

[64] Peng Wu, Yong Cui, Jianping Wu, Jiangchuan Liu, and Coert Metz. 2013. Transition from IPv4 to IPv6: A state-of-the-art survey. *Communications Surveys & Tutorials, IEEE* 15, 3 (2013), 1407–1424.

[65] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. 2015. A survey on software-defined networking. *IEEE Communications Surveys & Tutorials* 17, 1 (2015), 27–51.

[66] Dr. Lixia Zhang et al. 2013. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205. (2 March 2013). https://doi.org/10.17487/rfc2205