2018-10-01

Multi-layer virtual transport network management

Multi-Layer Virtual Transport Network Management

Yuefeng Wang Ibrahim Matta

***Abstract*—Nowadays there is an increasing need for a general paradigm which can simplify network management and further enable network innovations. Software Defined Networking (SDN) is an efficient way to make the network programmable and reduce management complexity, however existing SDN solutions do not realize the full potential of SDN because of their lack of scoping in network management. In response to limitations of current Software Defined Networking (SDN) management solutions, we propose a recursive approach to enterprise network management, where network management is done through managing various Virtual Transport Networks (VTNs) over different scopes ($i.e.$, regions of operation).**

Our approach offers a generalized and structured framework to realize the full potential of SDN using recursive scoped management, and our VTN-based approach provides communication service with explicit Quality-of-Service (QoS) support for applications via transport flows. Each VTN involves all mechanisms ($e.g.$, addressing, routing, error and flow control, resource allocation) needed to support such transport flows. Based on this approach, we design and implement a management architecture, which recurses the same VTN-based management mechanism for enterprise network management. Our experimental results show that our management architecture achieves better performance demonstrated through unicast and multicast video applications.

***Index Terms*—Enterprise Network Management, Multi-layer Network Design, Virtual Transport Network, Recursive Network Management**

I. Introduction

Traditionally network management is a complicated and error-prone process that involves low-level and vendor-specific configurations of physical devices. Nowadays computer networks have become increasingly complex and difficult to manage, and new cloud-based service models [1] have become the norm in networking economics, $e.g.$, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). These trends increase the need for a general management paradigm to simplify network management and further enable network innovations.

Extensive research has been done to address the complexity of network management by providing high-level network abstractions and hiding low-level details of physical devices. Among those efforts, making the network programmable is an efficient way to that end. Active Networking [2] pioneered the research in programmable networking, but it failed to gain popularity due to lack of an immediately compelling problem (or a killer application) [3]. Recently Software Defined Networking (SDN) [4] has drawn considerable attention due to the popularity of OpenFlow [5], a protocol that allows the configuration of switches without exposing their internal details. SDN focuses on programming the control plane through a network management layer and has been widely deployed in enterprise and data center networks [5], [6]. An OpenFlow-based SDN management layer provides a high-level interface, and network managers can easily manage the network without dealing with the complexity of low-level network devices.

However, existing SDN solutions do not realize the full potential of SDN due to their lack of recursive scoped management. In response to this limitation, we propose an application-driven recursive approach to enterprise network management. In our approach, network management is done through managing various Virtual Transport Networks (VTNs), inspired by and built atop a new network architecture, RINA [7], [8], [9], [10], which aims to solve current TCP/IP limitations.

In our approach, we consider Network Function Virtualization (NFV) [11] as a distributed application whose function is to provide a certain service (such as load balancing, firewalling, $etc.$), and SDN is just a special case of NFV whose function is to manage the network. Our approach inherently supports NFV and SDN, and generalizes NFV to include both application-specific functions (which are often referred to as "NFV" in the literature) and network-level functions (which are often referred to as "SDN"). Besides, our approach can be overlaid on top of TCP/IP to impose a recursively managed network structure and create new layers/VTNs that can run non-TCP/IP protocols.

Different from the traditional virtual network model which mainly focuses on routing/tunneling, a VTN provides communication service with explicit QoS support[1] for applications via transport flows, and it includes all mechanisms ($e.g.$, addressing, routing, resource allocation) needed to support such transport flows. Furthermore, a VTN is application-driven, *i.e.*, a VTN can be dynamically formed with different policies to meet different application-specific requirements. One of the biggest advantages of VTN is that it enables scoping at the transport level, thus we can either aggregate multiple transport flows into a single transport flow or split a transport flow into multiple transport flows, which enables better resource allocation and utilization in support of various requirements.

In this paper, we focus on how to program a single VTN ($i.e.$, the basic building block for network management) and how to program the whole network by composing the services provided by different VTNs to support communication flow requests. The contributions of this paper are: (1) we propose an application-driven recursive approach to enterprise network management, where network management is done through managing various Virtual Transport Networks (VTNs); (2) we present the design and implementation of a management

Yuefeng Wang (yuefeng.wang@akamai.com) is with Akamai Technologies, Inc, Cambridge, MA, and this work was done while he was at Boston University. **Ibrahim Matta** (matta@bu.edu) is with the Computer Science Department at Boston University, Boston, MA.

[1]Details of how QoS is supported in our approach is out of the scope of this paper.

architecture based on our approach, which allows management and dynamic formation of such VTNs to support flow requests from regular users and to meet application-specific requirements; and (3) we show the advantages of our VTN-based management architecture through experimental results.

The rest of the paper is organized as follows. We review existing SDN management solutions in Section II. Details of our VTN-based management approach are explained in Section III. The implementation and evaluation of our VTN-based management architecture are presented in Section V. In the end, we conclude this paper with future work in Section VI.

II. Related Work

The core of an SDN-based management solution is the management layer (such as [12], [13]). This layer itself does not manage the network but provides a global network view and general programming interface (the so-called "Northbound" API [14]) to management applications (designed or programmed by network managers), which actually manage the network. Access control, virtual machine (VM) migration, traffic engineering, and routing are examples of management applications. The network management layer translates high-level network policies specified by the management applications into low-level and vendor-specific configurations of network devices (switches or routers) through the OpenFlow protocol [5] (or any of so-called "Southbound" API [14]).

SDN reduces management complexity by providing high-level network abstractions. However most SDN management layers (such as [12], [15]) only provide the management-level interface, used by network managers to write management applications to monitor and control the network. They lack a user-level interface, used by regular users to write general applications (such as video application) and achieve better performance for their applications.

Network virtualization allows multiple isolated virtual networks to be built on top of the same physical infrastructure. It can improve resource utilization through network consolidation while providing isolation for security purposes or for developing and testing new network features. Some SDN management layers (such as [16], [17], [18], [19]) support network virtualization, but they mainly focus on routing and access control, and do not consider other mechanisms (such as error and flow control and resource allocation) for transport purpose, which is important for network resource utilization.

The most important contribution of this paper is defining the concept of layered scopes in network management. A network management layer manages a network over a certain *scope*, where *scope* is a collection of processes running on a subset of nodes ($e.g.$, switches, routers, and hosts) in the network. Members in the same scope follow the same network policies, and collectively provide a particular communication service. New management scopes can be dynamically defined for different purposes ($e.g.$, application performance or network performance).

Layered scoping is important in network management as it enables fine-grained control over the network and better support for policy-based management. However, with existing SDN solutions, scope is flat and only one-level, and cannot be dynamically defined, thus they do not realize the full potential of SDN because of their lack of scoping through recursive management. In other words, they lack levels of management scope, and every component is part of the same and only management scope. This is due to SDN's reliance on the TCP/IP architecture, which notably lacks resource allocation and flow/error control over limited scopes [7]. There is existing SDN work that provides recursive control ($e.g.$, [20], [21]), but it lacks transport-level flow/QoS control over limited scopes and does not support dynamic management of scopes.

III. VTN-based Network Management

In this section, we explain the details of our VTN-based approach to enterprise network management. Our approach is inspired by and built on top of a new network architecture, the Recursive InterNetwork Architecture (RINA) [7], [8], [9], [10].

RINA is based on the principle that *networking is Inter-Process Communication (IPC) and only IPC*. RINA solves shortcomings of the TCP/IP architecture by addressing the communication problem in a more fundamental and structured way. It provides communication services with explicit QoS support via transport flows by using a recursive building block (the IPC layer, which we call VTN). This building block is repeated over different scopes to provide different communication services. The building block involves all kinds of mechanisms ($e.g.$, enrollment, authentication, addressing, routing, error and flow control, resource allocation) to support transport flows over a certain scope. RINA separates mechanisms and policies, and each building block can have its own policies ($e.g.$, routing, naming, access control, $etc.$) while using the same mechanisms. Our contribution over existing work on RINA is that we exploit the usage and benefits of such building block ($i.e.$, VTN) for the purpose of network management. We also extend the RINA specification [9] to include the allocation of multi-layered VTNs.

The concept of *transport network* is not new, and a lot of work has been done on how to build such a transport network, $e.g.$, Optical Transport Network (OTN) [22] and Multiprotocol Label Switching-Transport Profile (MPLS-TP) [23]. The IPC layer in RINA has the properties of both virtual network and transport network, thus we use the term *Virtual Transport Network (VTN)* in this paper to denote this IPC layer.[2]

A. Virtual Transport Network (VTN)

A Virtual Transport Network (VTN) is the basic building block in our network management. The job of a VTN is to provide communication service with QoS support via transport flows for user applications. Unlike a regular virtual network which mainly focuses on routing/tunneling, a VTN involves all kinds of mechanisms ($e.g.$, enrollment, authentication, routing, addressing, error and flow control, resource allocation) needed to support transport flows over a certain management scope. A transport flow provides end-to-end communication service

[2]A VTN is termed a DIF in [7], [8], [9], [10], to mean a Distributed Inter-Process Communication (IPC) Facility.

with QoS parameters, which differs from a tunnel that is usually hard-coded, and just provides best-effort service over an overlayed routing path (tunnel) without resource allocation, flow and error control.

A transport process is a process that is capable of establishing transport flows across the VTN at requested QoS levels. Each VTN consists of a set of transport processes which run on different nodes (or hosts), and the operations of its member processes are contained in the VTN itself. VTN is a secure container, where every process has to be explicitly enrolled through an authentication procedure [8], [10]. Each transport process contains a data transfer component supporting transport flows between different applications. The VTN provides communication service to application processes by exposing a flow allocation interface.

Each VTN has its management scope, $i.e.$, each VTN includes a limited number of transport processes running on a limited number of physical nodes. Each VTN maintains the mapping between applications and transport processes, $i.e.$, application name resolution within its scope. Note that in our management approach, there is no global address space for application processes and an application process is only reachable over certain scopes (instead of the global scope). Thus we need the VTN structure to support communication between application processes, $i.e.$, two application processes are able to communicate if only if they have a common underlying VTN. The same VTN mechanism can be repeated to provide a larger-scope transport service for applications by recursively using the smaller-scope transport services provided by existing VTNs. Namely, we can build VTNs of different levels, $i.e.$, multi-layered VTNs, to provide transport services over different scopes. Different VTNs use the same mechanisms but may use different network policies ($e.g.$, policies for routing and error and flow control), and the transport processes inside the same VTN follow the same policies specific to the particular VTN.

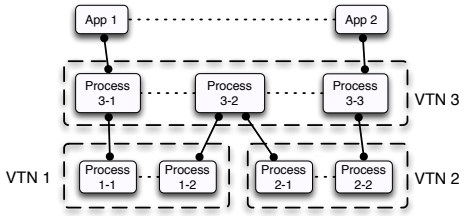

Fig. 1: Two levels of VTNs, and *VTN 3* spans a larger scope. Each process inside a VTN is a transport process.

Figure 1 shows a simple example of VTNs providing transport service over different scopes. *VTN 1* and *VTN 2* each spans a smaller scope, and can provide transport services to applications inside its scope. If an application *App 1* in one scope wishes to communicate with another application *App 2* in another scope, and since *VTN 1* and *VTN 2* cannot satisfy such request, we need a higher level *VTN 3* which spans both scopes and provides a transport service across the larger scope. Recursively, we can repeat VTN to provide an even larger-scope transport service, $i.e.$, any two application processes can communicate as long as a common underlying VTN can be found or built.

In our VTN-based network management, VTN is the basic building block, which modularizes network management. VTN encapsulates a range of operation (scope) by exposing a transport service specification that can be composed to form a larger-scope (high-level) VTN that ultimately meets user/application requirements. Our VTN approach provides the opportunity to control management complexity. Each VTN has its own scope, and can be managed without interfering with the operations of other VTNs. By allowing the dynamic formation of VTNs, we can either break a larger (lower-level) management scope into smaller ones or aggregate smaller scopes into a larger one. This gives us more flexibility for network programmability compared to existing SDN solutions.

Most importantly, our VTN-based management approach provides a generalized and recursive structure to impose scoping of management functions ($i.e.$, integrated routing and transport within each scope defined by the VTN). Our work in [24] shows that our VTN-based management approach, through layered scoping, can improve network performance by reducing routing overhead and transport overhead. Specifically, our approach is able to limit the scope in which routing messages are propagated, and so avoid unnecessary communication with remote nodes, thus reducing routing overhead. Also, our approach is able to break a large transport scope into small scopes, so retransmission of lost/corrupted packets is done over each smaller scope (rather than end-to-end over the whole large scope), thus reducing transport overhead. *In this paper, we present the architecture, and associated interfaces, that support the VTN design algorithms in [24]. We also demonstrate, experimentally on the GENI testbed, NFV/SDN capabilities inherently supported by our VTN-based management architecture, through (unicast and multicast) video case studies.*

B. Naming and Addressing

With our naming and addressing mechanisms, our VTN-based approach allows transport flows to start and end anywhere compared to only end-to-end as in the legacy Internet architecture.

A process within a VTN has an address that is (1) unique within the VTN/scope, and (2) independent of the interface or path that is used to reach that process (unlike TCP/IP). In addition, a transport flow is assigned an identifier that is (1) unique in each endpoint/process, and (2) decoupled from the "port" used by higher level VTN (or application) processes. So multiple higher level flows can multiplexed by the lower level (underlying) VTN by mapping their ports onto a single (aggregated) lower level flow.

We show an example in Figure 2 to explain how naming and addressing is done in our VTN-based approach. There are 2 levels of VTNs, and 4 application processes on top of that. *VTN 1* belongs to Level N, and *VTN 2* and *VTN 3* belong to Level N-1.

There are 3 transport processes within *VTN 1*: *TP A, TP B and TP C*. Each of them is assigned a unique address within

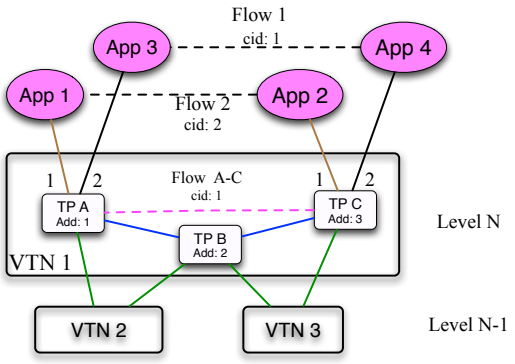

Fig. 2: Two levels of VTNs and four applications on top.

VTN 1, $i.e.$, 1 for *TP A*, 2 for *TP B*, and 3 for *TP C*. Routing within *VTN 1* is done using these assigned addresses, and we can use any routing protocol such as Link State or Distance Vector. There are two (virtual) links inside *VTN 1*, one link between *TP A* and *TP B*, and another one between *TP B* and *TP C*.

On top of the two levels of VTNs, there are 4 application processes. *App 1* and *App 3* use *TP A*, and *App 2* and *App 4* use *TP C*. *TP A* assigns *App 1* "port" number 1, and "port" number 2 to *App 3*. On the other side, *TP C* assigns *App 2* "port" number 1, and "port" number 2 to *App 4*.

There is a flow between *App 1* and *App 2*, $i.e.$, $Flow$ 2, and another flow between *App 3 and App 4*, $i.e.$, $Flow$ 1. $Flow$ 1 and $Flow$ 2 are mapped onto a single flow ($i.e.$, $Flow\ A-C$) between *TP A* and *TP C* within *VTN 1*, and the multiplexing/demultiplexing is done using the previously assigned "port" numbers.

Note that $Flow\ A-C$ is identified with a connection ID (cid: 1) between *TP A* and *TP C*. There may be multiple flows between them, and each of them has a unique connection ID. Furthermore, $Flow$ 1 and $Flow$ 2 can be mapped onto different flows (if they existed) between *TP A* and *TP C*.

The flow between *TP A* and *TP C* within *VTN 1* is routed over the path consisting of two (virtual) links $A-B$ and $B-C$, where each of these links is supported by a underlying VTN, $i.e.$, *VTN 2* and *VTN 3*, respectively. Note that the flow may be routed over a different path (if existed) within *VTN 1* to choose the path with best performance.

C. Multi-Layer VTN Design Problem

Our VTN-based management approach is able to support application flow requests and improve network performance for different aspects ($e.g.$, routing, transport) through multi-layered VTNs. How to design such multi-layered VTNs is out of the scope of this paper. In [24], the *Multi-Layer VTN Design Problem* is defined, and by solving this problem we can determine the VTN structure needed to support application flow requests. Also, a unified design framework is provided to allow different instantiations to meet specific application and network goals. More details can be found in [24].

IV. Design of VTN-based Management Architecture

In this section, we explain the design of our VTN-based management architecture which enables the VTN-based network management. First, we present the management components, and external API provided by our management architecture, as well as the details of the VTN formation protocol which enables dynamic VTN formation. Then, we explain the design of other components and internal API of our management architecture. This paper extends the RINA specification [9] to include the allocation of multi-layered VTNs.

A. Management Components

We describe the management components of our VTN-based management architecture. To this end, we realize the management function through distributed management applications. Next we explain the components ($i.e.$, distributed applications) for managing (1) a single VTN; and (2) all VTNs.

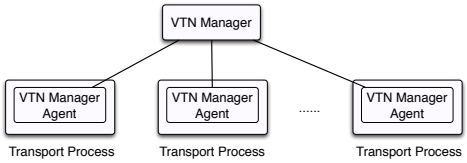

Fig. 3: VTN Manager and its agents for a single VTN.

1) VTN Manager and VTN Manager Agent:

As shown in Figure 3, the distributed application for managing a single VTN includes a *VTN Manager* and its *VTN Manager Agents*. Every VTN has a VTN manager, which is a process that can be implemented in a centralized or distributed fashion, and it manages the whole VTN by specifying different network policies inside the VTN such as routing, access control, and transport policies. A VTN manager agent is part of each transport process (see Figure 8) inside the VTN, and it exposes a programmable interface for the VTN manager to translate high-level network policies to transport process's configurations.

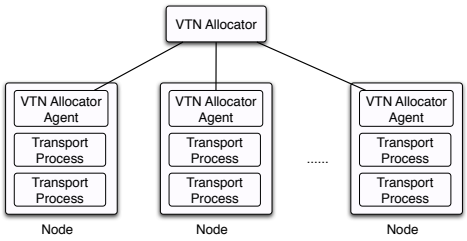

Fig. 4: VTN Allocator and its agents for an enterprise network.

2) VTN Allocator and VTN Allocator Agent:

As shown in Figure 4, the distributed application for managing all VTNs inside the enterprise network includes a *VTN Allocator (VA)* and its *VTN Allocator Agents (VAA)*. The

network has one VA, which is a management process that can be implemented in a centralized or distributed fashion, and it manages all VTNs as a whole. Each node inside the network has a VAA, which exposes a programming interface allowing the VA to create new transport processes on the node and thus build new VTNs across multiple nodes within the network. VA manages the network by managing existing VTNs and building new VTNs dynamically to support different application flow requests.

Fault tolerance of the VTN Manager and VTN Allocator is beyond the scope of this paper and left for future work.

3) **VTN Resource Manager (VRM)***:*

Every application process has a component called *VTN Resource Manager (VRM)*, which manages the use of all VTNs available to this particular process. It is the job of VAA of the node to decide which VTNs are accessible to particular processes. An application process uses its VRM to allocate transport flows with QoS requirements to other processes, and the VRM in turn passes the flow allocation requests to VTNs via the flow allocation interface exposed by VTN.

4) **Walk-through of Transport Flow Allocation***:*

When an application wants a transport flow with a certain QoS requirement to another application process, it uses the flow allocation interface exposed by its VRM. When the VRM gets the request, it first checks whether any of its available VTNs can reach that application. If the VRM finds such a VTN, it uses the VTN interface to allocate the flow through the transport process belonging to that VTN on the same node. If no VTN is found, the VRM sends the flow request to the VAA of the node, which then forwards the flow request to the VA of the network, and eventually the VA determines how to build a new VTN which consists of new transport processes running on the source node, destination node and some intermediate nodes.

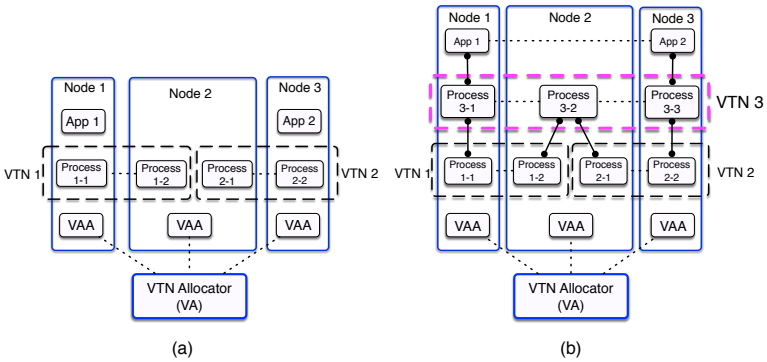

Fig. 5: (a) An enterprise network consists of three nodes (*Node 1*, *Node 2* and *Node 3*), and a centralized VTN Allocator. (b) A new VTN (*VTN 3*) is formed to support the flow between *App 1* and *App 2*.

As shown in Figure 5(a), when *App 1* on *Node 1* asks its VRM (not shown) for a flow to *App 2*, and its VRM cannot find an existing VTN to reach *App 2*, *App 1*'s VRM then sends the request to *Node 1*'s VAA, which forwards the request to the VA. The VA figures out that a new VTN is needed to support the flow, then it builds a new VTN (*VTN 3*) spanning all three nodes (Figure 5(b)). After the new VTN is ready, the VAA of *Node 1* notifies the VRM of *App 1*, which eventually uses this new VTN (*VTN 3*) to create a flow to *App 2*. Note than links between processes of *VTN 3* constitute *virtual transport* links and not simply routing tunnels.

B. Network API

Our management architecture provides two sets of APIs: (1) the management-level API, used by network managers to manage the network by programming the VTN Allocator and VTN Manager; and (2) the user-level API, used by regular users to program their own applications – it helps users affect their application traffic to improve user application performance.

1) **Management-level API***:*

Network managers manage networks through management applications, and our management-level API includes three sets of APIs as shown in Table I.

(1) VTN Formation API
public boolean createVTN (VTNRequest vtnRequest); public boolean deleteVTN (VTNRequest vtnRequest);
(2) Flow Allocation API

public int allocateFlow(Flow flow);
public boolean deallocateFlow(int handleID);
public void send(int handleID, byte[] msg) throws Exception;
public byte[] receive(int handleID);

(3) Information API

public int createEvent(SubscriptionEvent subscriptionEvent);
public boolean deleteEvent(int subscriptionID);
public Object readSub(int subID);
public void writePub(int pubID, byte[] obj);

TABLE I: Three sets of management-level APIs in Java.

VTN Allocator Agents (VAAs) expose the *VTN Formation API*, which is used by the VTN Allocator (VA) to create new VTNs or delete existing VTNs. A VTN is instantiated with different policies such as routing policies, addressing policies and flow and error control policies.

The VRM of each application process exposes the *Flow Allocation API*, which is used to create/delete transport flows with QoS requirements as well as to send/receive data messages over existing flows between applications.

Based on a publish/subscribe model (a pulling mechanism to retrieve information is also supported), the *Information API* allows management applications to retrieve or publish network information from/to other management applications. This API (similar to NIB API in Onix [15]) allows management applications to access network information.

2) **User-level API***:*

The user-level API includes two sets of APIs: (1) the Flow Allocation API; and (2) the Information API. They are the same as the ones in the management-level API.

Users use these two APIs to write their own applications, where they use the Flow Allocation API to create/delete transport flows with QoS requirements as well as to send/receive data messages over existing flows between user-defined applications; and they use the Information API to retrieve or publish user-specific information between user-defined applications.

C. VTN Formation Protocol

New VTNs may need to be formed in support of transport flows (Section IV-A4). In this section, we explain how the VTN Allocator (VA) and VTN Allocator Agent (VAA) interact with each other via a VTN formation protocol to create new VTNs.

1) Objects Exchanged in the Protocol:

The key aspect of the VTN formation protocol is two objects exchanged between the VA and VAA.

Flow Request Object: the VAA on a node sends a flow request object to the VA when a certain transport flow cannot be supported using existing VTNs on the node. The flow request object specifies the source and destination application information as well as QoS requirements including throughput, delay, and loss rate. The flow request object also supports advanced flow requirements (policies) such as which nodes to bypass or go through, or whether encryption is needed or not. The flow policies inside the request object are specified when the application uses the Flow Allocation API (shown in Table I (2)) exposed by its VRM to allocate the transport flow.

VTN Request Object: the VTN request object supports two operations: VTN creation and deletion. A VA sends a VTN request object to multiple VAAs on different nodes once it determines how the new VTN should be formed, $i.e.$, the new VTN should have new transport processes running on which nodes. The VTN request object specifies policies for the new VTN, including policies for routing, addressing, error and flow control, $etc.$ Also, it supports other policies such as which application can use this VTN, the lifetime of this VTN as well as resource allocation policies. Furthermore, the VTN request object can specify the connectivity among transport processes, and the enrollment and authentication policies for new transport processes to join the VTN.

2) Protocol Details:

A new VTN needs to be formed when existing VTNs cannot satisfy a new transport flow request. Next we explain how the VTN formation protocol works step by step. When the VA receives a flow request object from a VAA, it first inspects the object to see if it is a valid request. If valid, it checks the network state and determines whether there exists a path (a chain of nodes) from the node where the source application runs to the node where the destination application runs. Once a path is found, the VA can decide which nodes the new VTN should span, $i.e.$, the design of the VTN. We call this procedure (including finding a path and designing the new VTN) the *Multi-Layer VTN Design Problem* (Section III-C).

Once the VA determines the design of the new VTN, it sends the VTN request object specifying the policies of the new VTN to all VAAs of the nodes along the path (including source and destination nodes). When a VAA receives the VTN request, it first inspects the object to see if it is a valid request. If valid, the VAA creates a new transport process as a member of the new VTN, then the VAA sends a VTN response to the VA indicating that the new transport process on this node is ready. After the VA receives the responses from all VAAs to which it sent the VTN request, and if all responses indicate that all new transport processes are ready, the VA sends a flow response to the VAA that sent the initial flow request indicating that a new VTN is ready and the associated VRM can use it to create the transport flow. If any of the VAA's responses indicates failure of the creation of the new transport process, the VA sends a VTN delete request to all other VAAs to delete the newly created transport processes for that VTN. Then the VA sends a negative flow response to the VAA that asked for the new transport flow indicating that the flow request cannot be satisfied.

D. Other Components

Next we explain the design of other components in our VTN-based management architecture. In our management architecture, a *node* is a container where *application processes* and *transport processes* reside. There is a *node process* running on each physical node.

1) Node Process:

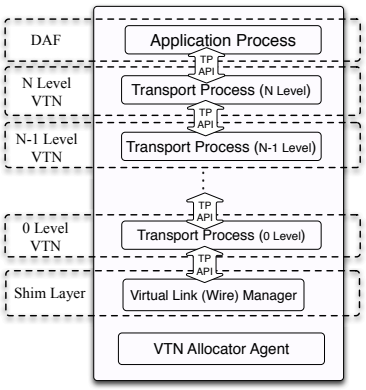

Fig. 6: Components of a node process.

As shown in Figure 6, application processes or high-level transport processes communicate with their peers using the communication service provided by underlying low-level transport processes, which expose the *Transport Process (TP) API* and serve as points of attachment. The mapping from an application process (or higher-level transport process) to the lower-level transport process is maintained and resolved by the underlying VTN. Note that in Figure 6, we only show one application process and one transport process at each level, however, practically there might be multiple application processes on the same node, and multiple transport processes of the same level.

Physical connectivities between transport processes in level-0 VTNs are emulated by a shim layer. More generally, using the shim layer enables our management architecture to run on top of Ethernet, TCP, UDP, or SDN-based networks.

As discussed in Section IV-A2, the enterprise network has a VTN Allocator (VA), and each node has a VTN Allocator Agent (VAA), which exposes a programming interface allowing the VA to create new transport processes on the node and thus build new VTNs across multiple nodes within the network.

2) **Application Process:**

A *Distributed Application Facility (DAF)* [9], [10] is a collection of distributed application processes with shared states. Each DAF performs a certain function such as video streaming, weather forecast or communication service. Different DAFs use the same mechanisms but they may use different policies for different purposes and over different scopes. For example, a *VTN*, $i.e.$, a collection of transport processes, is a special DAF whose job is only to provide communication services for application processes. Also, the management components for managing a single VTN (or an enterprise network) form a management DAF whose job is to manage a single VTN (or an enterprise network).

Actually, the DAF is the container that enables our approach to inherently support NFV/SDN. The distributed function provided by a DAF can be a Virtualized Network Function (VNF), and SDN is realized in our approach by programming the VTN, which is a special case of a DAF that implements a communication service.

The *Common Distributed Application Protocol (CDAP)* [9] is the only application protocol required, and it is used for both network management applications and regular user applications. For example, the objects exchanged for the VTN formation protocol are encapsulated in CDAP messages. CDAP defines a set of operations: create/delete, read/write, and start/stop on remote objects, and connect/release to enable authentication and coordination among management applications. Users are free to define new types of objects for their own application purposes, as long as instances of such application agree on the same data representation.

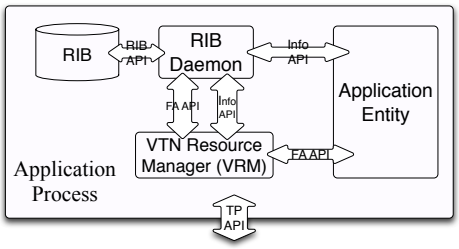

Fig. 7: Components of an application process.

Figure 7 shows the common components of an application process. The *Resource Information Base (RIB)* is the database that stores all information related to the operations of an application process, and RIB can be accessed via *the RIB API*. The *RIB Daemon* helps other components of the application process access information stored in the local RIB or in a remote application's RIB. As discussed in Section IV-A3, each application process also has an *VTN Resource Manager (VRM)*, which manages the use of underlying transport processes belonging to lower-level VTNs that provide communication services for this application process, and the VRM manages underlying VTNs via the *Transport Process (TP) API* exposed by each transport process. The *Application Entity* is the container in which users can implement different management or application-specific functionalities.

The *Flow Allocation (FA) API* and *Information API* are provided for users to write management (or regular) applications and to support new network management policies. The Information API (provided by the RIB Daemon) is based on a publish/subscribe model, which supports the creation and deletion of a subscription event (a *Pub* or *Sub* event), the retrieval of information through a *Sub* event, and the publication of information through a *Pub* event. The RIB Daemon also supports the traditional pulling mechanism to retrieve information. The Flow Allocation (FA) API (provided by the VRM) allows allocating/deallocating a connection (transport flow) to other application processes, and sending/receiving messages over existing connections.

3) **Shim Layer Process:**

In order to deploy our VTN-based management architecture on top of legacy networks (such as over Ethernet, TCP or UDP), we have a shim layer in our design. Physical connectivity between transport processes at level 0 are emulated by the shim layer, and the shim layer includes functionalities such as resolving a user-defined level-0 transport process name to an IP address and a port number. The shim layer consists of a collection of shim layer processes, $i.e.$, the virtual link (wire) manager on each node, and each virtual link (wire) manager manages the emulated physical wires for that particular node.

Because of the shim layer, our approach can be overlaid on top of TCP/IP or any other network architecture to impose a recursively managed network structure and create new layers/VTNs that can run non-TCP/IP protocols.

4) **Transport Process Components:**

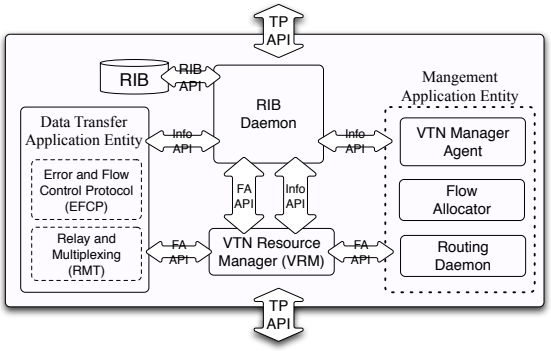

Fig. 8: Components of a Transport Process.

Figure 8 illustrates how different components of a transport process interact with other components. Note that a transport process is just a special application process whose job is only to provide communication services, so it has *RIB*, *RIB Daemon* and *VRM*. Recursively, each transport process manages the use of underlying VTNs via the *Transport Process (TP) API*. The application entity for a transport process consists of two parts: (1) data transfer application entity, and (2) management application entity.

The *Data Transfer Application Entity* is responsible for data transfer for each existing flow, and for the Relay and Multiplexing Task (RMT). The Error and Flow Control Protocol (EFCP) [9] is the data transfer protocol used within a

VTN. The functions of this protocol ensure reliability and flow control as required. The original design of EFCP is modeled after Richard Watson's Delta-t transmission protocol [25], and includes a Data Transport Protocol (DTP) and a Data Transport Control Protocol (DTCP). When a DTP packet is received, the data transfer application entity inspects its header information, and delivers it to the corresponding application process using this transport process based on the connection ID if the transport process is the destination. If the destination is not itself, it checks the forwarding table and sends the packet to the next-hop toward the destination.

The *Management Application Entity* includes 3 components: (1) *VTN Manager Agent*, (2) *Flow Allocator*, and (3) *Routing Daemon*. The *VTN Manager Agent* is a part of each transport process inside the VTN, and it exposes a programmable interface for the *VTN Manager* to translate high-level network policies to transport process's configurations. The *Flow Allocator* is the component that is responsible for the *Transport Process (TP) API* invocation from application processes (or higher-level transport processes), and maintains every transport flow allocated by this transport process. The *Routing Daemon* is responsible for the routing inside the VTN, so transport processes are able to communicate with other members in the same VTN.

The *Transport Process (TP) API* is used by an application process's VRM to access an underlying transport process: (1) via its Flow Allocator to create and delete a flow to another application process, (2) via its Data Transfer Application Entity to send and receive messages over an existing flow, (3) to register in the underlying VTN such that this application process can be reached through the transport process. Recursively, an N-level transport process is seen as an application process which uses an (N-1)-level transport process's communication services.

More details about the design of our VTN-based management architecture can be found at [26].

V. Implementation and Evaluation of VTN-Based Management Layer

We implement our VTN-based management architecture, which enables VTN-based management on real networks. Our implementation allows not only managing an enterprise network by programming management applications but also creating new user applications by programming user-defined applications. Our implementation is at the user level, and it supports dynamic formation of VTNs and multiple management policies ($e.g.$, naming and routing policies). The current implementation, called ProtoRINA (version 2.0), consists of about 70k lines of Java code excluding support libraries and configurations. Also in our implementation, all objects (CDAP messages, data transfer messages, and other messages) are serialized and deserialized using the Google Protocol Buffers [27]. More details (including the code) of our implementation can be found at [26]. Note that QoS is supported in our VTN-based management, but in our current implementation we focus on reachability only, where a flow request is specified by the source and destination application names.

Our implementation is tested both on our Boston University campus network and the GENI [28] testbed. GENI (Global Environment for Network Innovations) [28] is a nationwide suite of infrastructure that supports large-scale experiments, and it enables research and education in networking and distributed systems. Through GENI, users can obtain computing resources ($e.g.$, virtual machines (VMs) and raw PCs) from different physical locations (GENI aggregates), and connect these computing resources with layer-2 (stitched VLAN) or layer-3 (GRE Tunnel) links. GENI allows users to install customized software or even customized operating systems on these computing resources, and also provides users control over how network switches handle traffic flows. GENI also enables experiments on SDN such as providing support for OVS [29] switches and other OpenFlow support. What's more, GENI provides a variety of instrumentation and measurement tools (such as jFed, Jacks, Omni, GENI Desktop, LabWiki, Flack, $etc.$), to configure, run and measure user-specific experiments.

Next, we show the evaluations of our implementation of the VTN-based management architecture. Our evaluations are performed for two different setups: (1) unicast video application, and (2) multicast video application.

A. Video Unicast Performance

In this section, we show how our VTN-based management architecture can improve application performance through application-specific policies by using video unicast streaming as an example.

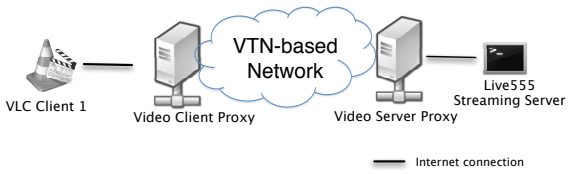

Fig. 9: Video clients (VLC players) are connected to the video streaming server (Live555 streaming server) through RTSP proxies over a VTN-based network.

In order to support Real-Time Streaming Protocol (RTSP) video streaming application running on legacy nodes, two video proxies are used as shown in Figure 9. The video client proxy is connected to the video client over the Internet, and the video server proxy is connected to the video streaming server also over the Internet. These two proxies are connected over a VTN-based network that is composed of VTNs, where routing policies (and other policies) can be easily configured for each VTN. Video proxies support RTSP and redirect all traffic between the video client and the video streaming server to the communication service provided by the VTN-based network. The details of these two proxies can be found at [30]. In this experiment, we use the VLC player (version 2.0.4) [31] as the video client, and the Live555 server (version 0.78) [32] as the video streaming server. The video file used in our experiment is encoded in the H.264/MPEG-4 AVC (Part 10) format, and can be found at [33].

As mentioned earlier, our VTN-based management approach separates mechanisms and policies, where different VTNs use the same mechanisms but can be instantiated with different policies. For the VTN that directly provides communication service between the two video proxies, we use the link-state routing protocol, and we test two link-cost policies (hop and jitter), and then observe how they affect the performance of the unicast video application.

1) **Experiment Design:**

As shown in Figure 10, we reserve GENI resources (VMs, VLANs, and GRE tunnels) from four GENI aggregates (Georgia Tech, UIUC, NYU, and Cornell University). VMs in different aggregates are connected using GRE tunnels, and VMs in the same aggregate are connected using VLANs. Each node process is running on a GENI VM, and we use 13 nodes (`Node 1` to `Node 13`). `Node 1`, `Node 2`, and `Node 3` are running on VMs from the NYU aggregate. `Node 4`, `Node 5`, `Node 6`, and `Node 7` are running on VMs from the Georgia Tech aggregate. `Node 8`, `Node 9`, and `Node 10` are running on VMs from the UIUC aggregate. `Node 11`, `Node 12`, and `Node 13` are running on VMs from the Cornell aggregate.

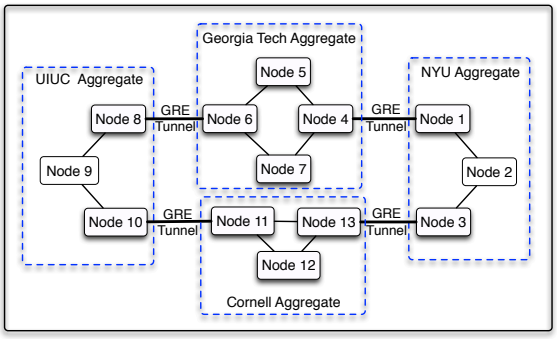

Fig. 10: Each node process is running on a GENI VM. Nodes in different aggregates are connected via GRE tunnels, and nodes in the same aggregate are connected via VLANs.

To provide communication service for the video client proxy located at `Node 9` and the video server proxy located at `Node 2`, in our experiment, we use a VTN design shown in Figure 11, and focus on different link-cost policies.

As shown in Figure 11, there are three level-1 VTNs (`VTN 1`, `VTN 2`, and `VTN 3`), and one level-2 VTN (`VTN 4`). `VTN 1` has three members: `Process 9` (on `Node 8`), `Process 10` (on `Node 9`), and `Process 11` (on `Node 10`). `VTN 2` has five members: `Process 12` (on `Node 10`), `Process 13` (on `Node 12`) , `Process 14` (on `Node 11`), `Process 15` (on `Node 13`), and `Process 16` (on `Node 3`). `VTN 3` has eight members: `Process 1` (on `Node 1`), `Process 2` (on `Node 2`) , `Process 3` (on `Node 3`), `Process 4` (on `Node 4`), `Process 5` (on `Node 5`) , `Process 6` (on `Node 6`), `Process 7` (on `Node 7`), and `Process 8` (on `Node 8`). `VTN 4` has five members: `Process 17` (on `Node 9`), `Process 18` (on `Node 10`), `Process 19` (on `Node 3`) , `Process 20` (on `Node 2`), and `Process 21` (on `Node 8`). Processes hosted on the same node are depicted in the same color in Figure 11.

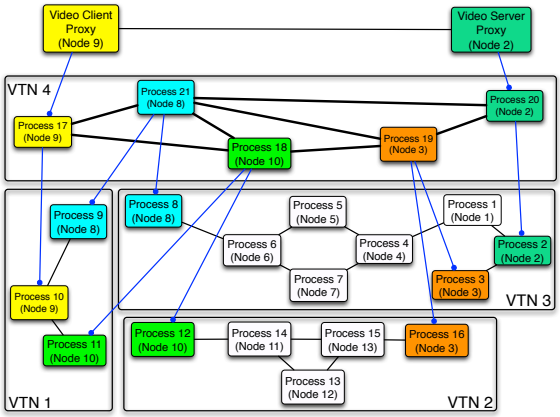

Fig. 11: Video client proxy (on `Node 9`) and video server proxy (on `Node 2`) communicate through a level-2 VTN (`VTN 4`), which is built on top of three level-1 VTNs (`VTN 1`, `VTN 2`, and `VTN 3`).

In Figure 11, the video client proxy on `Node 9` uses `Process 17`, which recursively uses `Process 10`. The video server proxy on `Node 2` uses `Process 20`, which recursively uses `Process 2`. `Process 21` on `Node 8` uses `Process 8` and `Process 9`, `Process 18` on `Node 10` uses `Process 11` and `Process 12`, and `Process 19` on `Node 3` uses `Process 3` and `Process 16`.

So the video client proxy and video server proxy communicate through a connection supported by the underlying `VTN 4`, which recursively uses the communication services provided by three level-1 VTNs. The connection between the video server proxy (using `Process 20`) and the video client proxy (using `Process 17`) is mapped to a path inside `VTN 4`, and each (virtual) link in `VTN 4` is supported by a level-1 VTN.

We use the network emulation tool, *NetEm* [34], to emulate link delay and jitter. Delay (300ms) with variation ($\pm$200ms) is emulated on two physical links between `Node 8` and `Node 9`, and between `Node 1` and `Node 2`. This emulation leads to jitter on link `Process 9-Process 10` in `VTN 1`, and on link `Process 1-Process 2` in `VTN 3`. This in turn is reflected as link jitter in the higher-level `VTN 4`, where four links (`Process 17-Process 21`, `Process 18-Process 21`, `Process 20-Process 21`, and `Process 19-Process 21`) exhibit jitter that they inherit from underlying paths.

2) **Experimental Results:**

We run our experiments on a network reserved on GENI as shown in Figure 12, which corresponds to Figure 10.

As we can see from Figure 11, the connection between the video server proxy and video client proxy can be routed via one of the seven possible loop-free paths inside `VTN 4` between `Process 20` and `Process 17`. These paths are: *path 1* (`Process 20 - Process 21 - Process 17`), *path 2* (`Process 20 - Process 19 - Process`

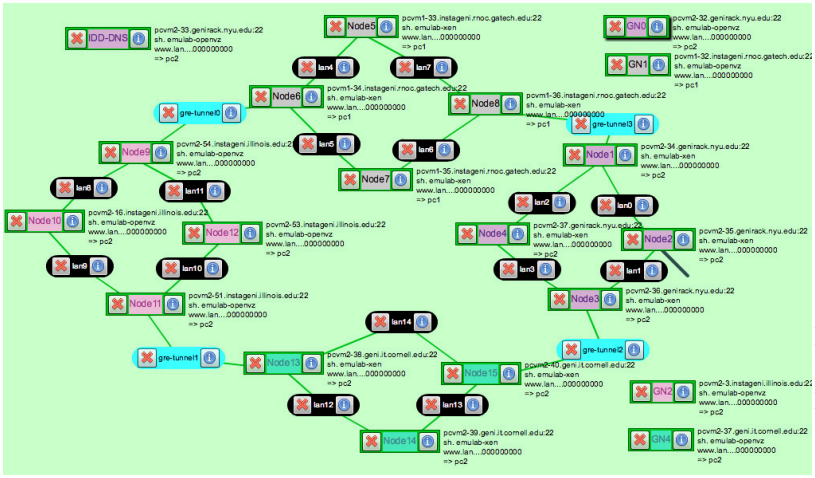

Fig. 12: GENI resources from four aggregates shown in Flack.

`18 - Process 17`), *path 3* (`Process 20 - Process 21 - Process 19 - Process 18 - Process 17`), *path 4* (`Process 20 - Process 21 - Process 18 - Process 17`), *path 5* (`Process 20 - Process 19 - Process 21 - Process 17`), *path 6* (`Process 20 - Process 19 - Process 21 - Process 18 - Process 17`), and *path 7* (`Process 20 - Process 19 - Process 18 - Process 21 - Process 17`).

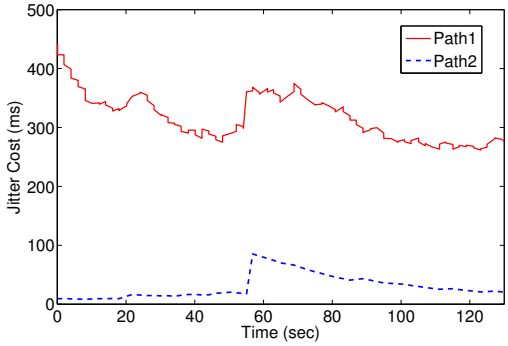

Fig. 13: Path jitter of *path 1* (the least-hop path) is larger than *path 2* (the least-jitter path), where path jitter is calculated as the sum of jitter on links along the path (collected by the routing task of `Process 20`).

Figure 13 shows the path jitter of *path 1* (least-hop path) and *path 2* (least-jitter path), where the jitter of a path is calculated as the sum of jitter on all links along that path (collected by the routing task of `Process 20`). If jitter is used as link cost, the connection between `Process 20` and `Process 17` is routed on *path 2*. On the other hand, if hop is used as link cost, the connection is routed on *path 1*.

For video applications, it is important to choose a path with least jitter, otherwise video quality degradation is observed. Figure 14 shows the measured instantaneous jitter that is experienced by video packets from the video server proxy to the client proxy, as the video server streams the same video to

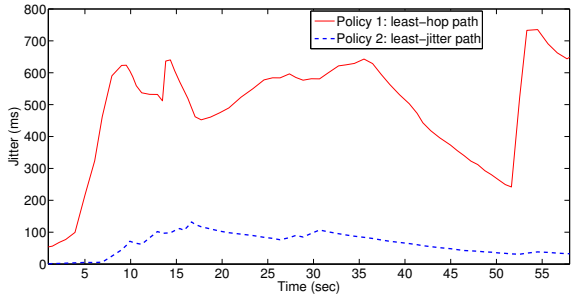

Fig. 14: Measured instantaneous jitter for video packets from the video server proxy to the client proxy when `VTN 4` uses hop or jitter as link cost.

a player client. To calculate the jitter, we sample video packets received by the client proxy at the rate of one every 60 packets. We can see that under least-jitter routing, the video packets experience much less jitter compared to least-hop routing.

As a consequence of this increased jitter, we indeed observe that the video freezes more often when using hop rather than jitter as the link-cost policy. Measuring video quality at the client side (player) using metrics such as signal-to-noise ratio (SNR) and mean opinion score (MOS), is left for future work.

B. Video Multicast Performance

In this section, we explain how video can be efficiently multicast to different clients on demand as an example of application-driven network management.

In order to support RTP (Real-time Transport Protocol) video streaming over the VTN-based network, RTP proxies (server proxy and client proxy) are used as shown in Figure 15. The RTP server proxy is connected to the video server over the Internet, and each RTP client proxy is connected to a video client also over the Internet. The RTP server proxy and RTP client proxies are connected over the VTN-based network which consists of VTNs. Namely, the RTP server proxy redirects all RTP traffic between the RTP server and RTP client to the communication channel provided by the VTN-based network. In our experiments, we use the VLC player [31] as the video client, and the Live555 server [32] as the RTP video server. The video file used in the experiments is an MPEG Transport Stream file, which can be found at [35].

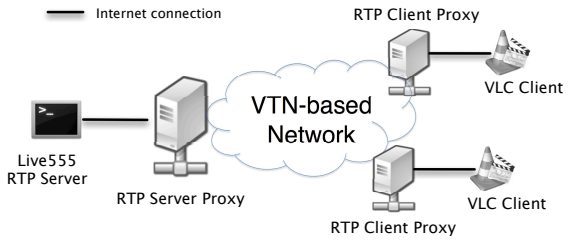

Fig. 15: Video clients (VLC players) are connected to the RTP video server through RTP proxies over a VTN-based network.

Figure 16 shows a scenario, where the enterprise network is made up of four subnetworks. The RTP server and RTP

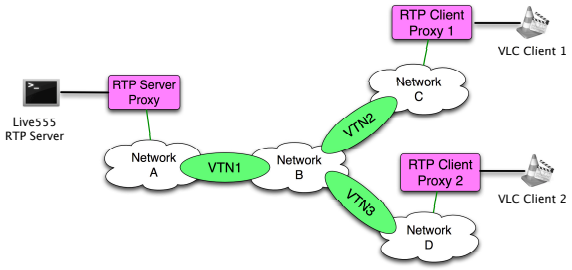

Fig. 16: Video server providing a live video streaming service is running in `Network A`. One client is in `Network C`, and one is in `Network D`.

server proxy are running in `Network A`, and they provide a live video streaming service. There are two video clients along with RTP client proxies (one in `Network C` and the other one in `Network D`) that would like to receive video provided by the RTP video server. `Network A` and `Network B` are connected through `VTN 1`, `Network B` and `Network C` are connected through `VTN 2`, and `Network B` and `Network D` are connected through `VTN 3`. `VTN 1`, `VTN 2` and `VTN 3` are three level-zero VTNs that can provide communication services for two connected networks. For simplicity, the Live555 RTP server and VLC clients are not shown in the following figures.

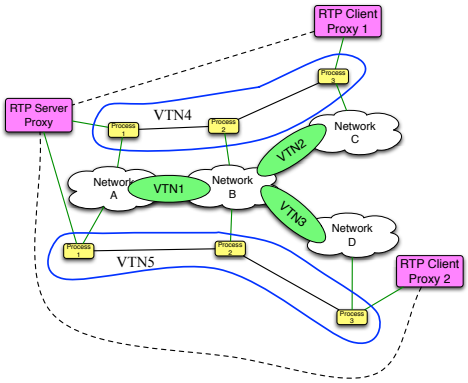

Fig. 17: Video streaming through unicast connections, where the same video traffic is delivered twice over `VTN 1` consuming unnecessary network bandwidth.

A very simple way to meet clients' requirements is as follows. Two video clients can receive live streaming service from the video server through two unicast connections supported by two separate VTNs as shown in Figure 17. The unicast connection between `RTP Client Proxy 1` and the video server proxy is supported by `VTN 4`, which is a level-one VTN formed based on `VTN 1` and `VTN 2`. The unicast connection between `RTP Client Proxy 2` and the video server proxy is supported by `VTN 5`, which is a level-one VTN formed based on `VTN 1` and `VTN 3`. However, it is easy to see that the same video traffic is delivered twice over `VTN 1`, which consumes unnecessary network bandwidth. In order to make better use of network resources, it is necessary to use multicast to stream the live video traffic.

1) **VTN-based Multicast Solutions:**

Next we show two different VTN-based solutions to support multicast, *i.e.*, two ways of application-driven network management: (1) application-level multicast, and (2) VTN-level multicast.

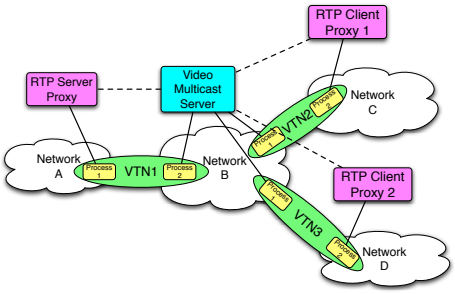

Fig. 18: Video multicast through an RTP multicast video server.

The first solution, *application-level multicast*, is enabled through a video multicast server as shown in Figure 18. The connection between the video server proxy and the video multicast server is supported by `VTN 1`. The connection between the video multicast server and `RTP Client Proxy 1` is supported by `VTN 2`, and the connection between the video multicast server and `RTP Client Proxy 2` is supported by `VTN 3`. The video server proxy streams video traffic to the video multicast server, which multicasts video traffic to each client through two unicast connections supported by `VTN 2` and `VTN 3`, respectively. We can see that the video traffic is delivered only once over `VTN 1` compared to Figure 17. In this case, we only rely on existing level-zero VTNs, and no new higher-level VTN is created.

Actually the video multicast server provides a VNF (Virtual Network Function [11]) as in NFV (Network Function Virtualization), *i.e.*, our VTN-based management approach can implicitly support NFV. In a complicated network topology with more local networks, if there are more clients from different local networks accessing the live streaming service, we can instantiate more video multicast servers, and place them at locations that are close to the clients, thus provide better video quality and network performance (such as less jitter and bandwidth consumption).

The second solution, *VTN-level multicast*, is supported using the multicast service provided by the VTN mechanism. As shown in Figure 19, we form a level-one VTN (`VTN 4`) on top of existing level-zero VTNs. The video server proxy creates a multicast channel through `VTN 4`, and streams live video traffic over this multicast channel. Each client joins the multicast channel to receive the live video traffic. Note that the allocation of a multicast connection is the same as the allocation of a unicast connection.

Here we can see that our VTN-based management approach implicitly supports SDN [4] by allowing the dynamic formation of new VTNs. Furthermore, it allows instantiating different policies for different VTNs. In a complicated network

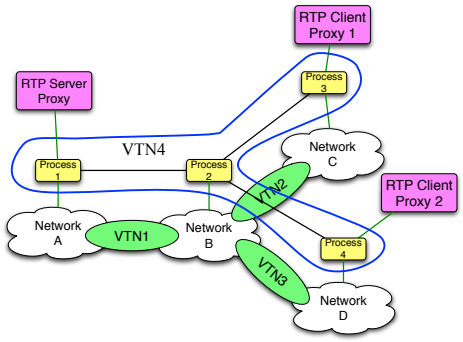

Fig. 19: Video multicast through multicast service provided by the VTN.

topology with more local networks, if there are more clients from different local networks accessing the live streaming service, we can either dynamically form new higher-level VTNs or expand the existing VTNs providing the multicast service.

2) **Experimental Results:**

In this section, we show the experimental results of our VTN-based multicast solutions over the GENI [28] testbed.

As shown in Figure 20, we reserve four VMs from four InstaGENI aggregates (Rutgers, Wisconsin, Chicago and NYSERNet), and we connect the VMs using stitched VLANs. Each aggregate corresponds to one network in Figure 16, where the RTP server and RTP server proxy are running on VM `N1` in the Rutgers aggregate, the `RTP Client Proxy 1` is running on VM `N4` in the Chicago aggregate, and the `RTP Client Proxy 2` is running on VM `N3` in the NYSERNet aggregate.

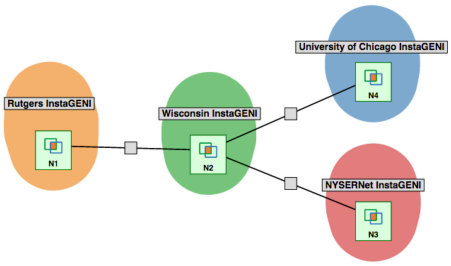

Fig. 20: GENI resources from four InstaGENI aggregates shown in Jacks.

Figure 21 shows the bandwidth usage for the unicast solution and the two multicast solutions over `VTN 1` (cf. Figure 16), *i.e.*, the link between VM `N1` in the Rutgers aggregate and VM `N2` in the Wisconsin aggregate in Figure 20. We can see that, as expected, the bandwidth usage for the two multicast solutions are close to half of that of the unicast solution.

As shown in Figure 22, we reserve five VMs from five InstaGENI aggregates (GPO, Chicago, NYSERNet, Stanford, and Wisconsin), and we connect the VMs using stitched VLANs. The RTP server and RTP server proxy are running on VM `N1` in the GPO aggregate, the `RTP Client Proxy 1`

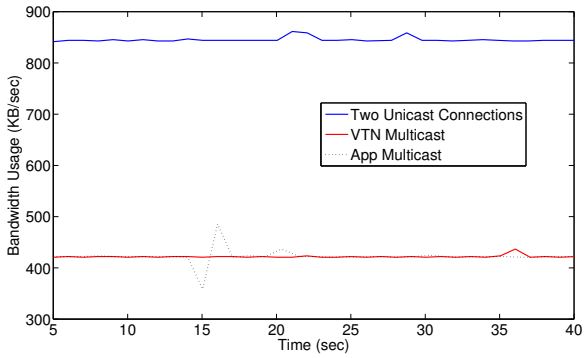

Fig. 21: Comparison of bandwidth usage over `VTN1`: unicast vs. multicast.

Fig. 22: GENI resources from five InstaGENI aggregates shown in Jacks.

is running on VM `N3` in the Stanford aggregate, and the `RTP Client Proxy 2` is running on VM `N5` in the Wisconsin aggregate. The goal is to observe the effect on the video quality at the video client side when placing the video multicast server in different locations, *i.e.*, placing the video multicast server either on VM `N2` in the Chicago aggregate or VM `N4` in the NYSERNet aggregate.

Since GENI does not yet allow specifying parameters when reserving stitched VLANs, such as capacity, packet loss and latency, we use a network emulation tool, NetEm [34] to add delay (1000ms $\pm$500ms) on the link between VM `N1` in the GPO aggregate and VM `N2` in the Chicago aggregate. In order to observe video quality, we have VLC players running locally on our BU campus network and connect them to the RTP client proxies running on GENI aggregates (*i.e.*, VM `N3` and `N5`) via Internet connections. Note that the jitter on the Internet connections is negligible, and the jitter in our experiments is mainly from jitter emulated on GENI links.

We run a VLC player locally and connect it with the `RTP Client Proxy 1` running on VM `N3` in the Stanford aggregate. Figure 23 shows the video observed when placing the multicast server on VM `N4` in the NYSERNet aggregate. Figure 24 shows the video observed when placing the multicast server on VM `N2` in the Chicago aggregate. We can see that by placing the video multicast server at a location experiencing less jitter, we can achieve better video quality.

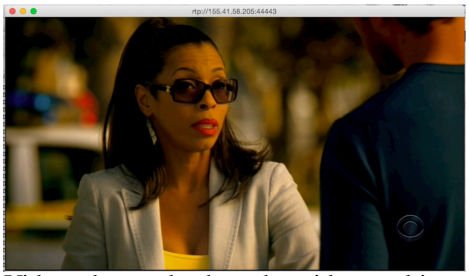

Fig. 23: Video observed when the video multicast server is placed on VM N4 in the NYSERNet aggregate resulting in a path with less jitter.

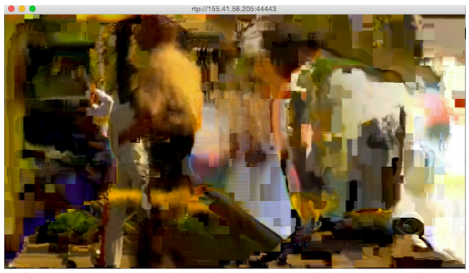

Fig. 24: Video observed when the video multicast server is placed on VM N2 in the Chicago aggregate resulting in a path with more jitter.

VI. Conclusion and Future Work

In this paper, we present a recursive application-driven approach to network management, which is able to realize the full potential of SDN. Our approach manages the network using Virtual Transport Networks (VTNs), where each VTN is a virtual network that can provide communication service via transport flows with explicit QoS support. A VTN can be dynamically formed and instantiated with different policies to satisfy different application-specific requirements. We present the design and implementation of a management architecture based on our approach, and show the performance and advantages of our management architecture through experimental results on real networks.

One important future work is to keep adding new components (such as a complete error and flow control component) to our implementation and enable QoS support for more metrics (such as throughput and delay). Also, it would be interesting to investigate the algorithms and mechanisms needed for QoS support to enable better resource allocation and utilization on real networks. Our VTN-based approach can enable better application performance, and another important future work is to investigate how to design and apply application-specific VTN policies to improve performance for more real applications. Furthermore, it is important to use formal methods to verify the correctness of our protocols and guarantee the proper usage of our API (both management-level and user-level), which contribute to the reliability and robustness of the management architecture.

Acknowledgment

This work has been partly supported by National Science Foundation awards: CNS-0963974 and CNS-1346688.

References

[1] National Institute of Standards and Technology, "The NIST Definition of Cloud Computing," 2011.

[2] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A Survey of Active Network Research," *Comm. Mag.*, vol. 35, no. 1, pp. 80–86, Jan. 1997. [Online]. Available: http://dx.doi.org/10.1109/35.568214

[3] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN," *ACM Queue*, vol. 11, no. 12, p. 20, 2013.

[4] Open Networking Foundation White Paper, "Software-Defined Networking: The New Norm for Networks," April, 2012.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[6] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-deployed Software Defined Wan," *SIGCOMM CCR*, vol. 43, no. 4, pp. 3–14, Aug. 2013.

[7] J. Day, *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall, 2008.

[8] J. Day, I. Matta, and K. Mattar, "Networking is IPC: A Guiding Principle to a Better Internet," in *Proceedings of ReArch'08 - Re-Architecting the Internet (co-located with CoNEXT)*, New York, NY, USA, 2008.

[9] Pouzin Society, "RINA Specification Handbook," 2013.

[10] Y. Wang, I. Matta, F. Esposito, and J. Day, "Introducing ProtoRINA: A Prototype for Programming Recursive-Networking Policies," *ACM SIGCOMM Computer Communication Review (CCR)*, July 2014.

[11] ETSI, "Network Functions Virtualisations (NFV) - White Paper," https://portal.etsi.org/Portals/0/TBpages/NFV/Docs/NFV_White_Paper3.pdf.

[12] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1384609.1384625

[13] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 1–12, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1282427.1282382

[14] Open Networking Foundation, "https://www.opennetworking.org/."

[15] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A Distributed Control Platform for Large-scale Production Networks," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924968

[16] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "FlowVisor: A Network Virtualization Layer," in *Technical Report, OpenFlow-TR-2009-1, OpenFlow Consortium*, 2009.

[17] D. Drutskoy, E. Keller, and J. Rexford, "Scalable Network Virtualization in Software-Defined Networks," *Internet Computing, IEEE*, vol. 17, no. 2, pp. 20–27, March 2013.

[18] E. Salvadori, R. Doriguzzi Corin, A. Broglio, and M. Gerola, "Generalizing Virtual Network Topologies in OpenFlow-Based Networks," in *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, Dec 2011, pp. 1–6.

[19] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A Compositional Hypervisor for Software-defined Networks," in *NSDI 2015*. Berkeley, CA, USA: USENIX Association.

[20] J. McCauley, A. Panda, M. Casado, T. Koponen, and S. Shenker, "Extending SDN to large-scale networks," *Open Networking Summit*, 2013.

[21] The SPARC Project: Split Architecture for Carrier Grade Networks, "http://www.fp7-sparc.eu/."

[22] ITU, "ITU-T G.709: Interfaces for the Optical Transport Network," https://www.itu.int/rec/T-REC-G.709/en, 2003.

[23] B. Niven-Jenkins, D. Brungard, M. Betts, N. Sprecher, and S. Ueno, "Requirements of an MPLS Transport Profile," RFC 5654, 2009.

[24] Y. Wang and I. Matta, "Muti-Layer Virtual Transport Network Design," in *Journal of Network and Systems Management*, vol. 26, no. 3. Springer, 2018, pp. 755–789.
[25] R. W. Watson, "Timer-based Mechanisms in Reliable Transport Protocol Connection Management," *Computer Networks (1976)*, vol. 5, no. 1, pp. 47–56, 1981.
[26] Y. Wang, "ProtoRINA 2.0," http://csr.bu.edu/rina/protorina/2.0, 2016.
[27] Protocol Buffers, https://developers.google.com/protocol-buffers/.
[28] GENI, http://www.geni.net/.
[29] Open vSwitch, http://www.openvswitch.org/.
[30] Y. Zhu, "RINA Video Streaming Application Proxies," https://github.com/yuezhu/rina_video_streaming.
[31] VLC Media Player, http://www.videolan.org/.
[32] LIVE555 Media Server, http://www.live555.com/.
[33] http://csr.bu.edu/rina/videoSample/.
[34] NetEm. Linux Foundation, "http://www.linuxfoundation.org/collaborate/workgroups/networking/netem."
[35] http://csr.bu.edu/rina/RTPVideoSample/.