Supporting Diverse Dynamic Intent-based Policies using Janus

Anubhavnidhi Abhashkumar*
University of Wisconsin Madison

Joon-Myung Kang
Hewlett Packard Labs

Sujata Banerjee*
VMware Research

Aditya Akella
University of Wisconsin Madison

Ying Zhang*
Facebook

Wenfei Wu*
Tsinghua University

ABSTRACT

Existing network policy abstractions handle basic group based reachability and access control list based security policies. However, QoS policies as well as dynamic policies are also important and not representing them in the high level policy abstraction poses serious limitations. At the same time, efficiently configuring and composing group based QoS and dynamic policies present significant technical challenges, such as (a) maintaining group granularity during configuration, (b) dealing with network-bandwidth contention among policies from distinct writers and (c) dealing with multiple path changes corresponding to dynamically changing policies, group membership and end-point mobility. In this paper we propose JANUS, a system which makes two major contributions. First, we extend the prior policy graph abstraction model to represent complex QoS and dynamic stateful/temporal policies. Second, we convert the policy configuration problem into an optimization problem with the goal of *maximizing the number of satisfied and configured policies*, and *minimizing the number of path changes under dynamic environments.* To solve this, JANUS presents several novel heuristic algorithms. We evaluate our system using a diverse set of bandwidth policies and network topologies. Our experiments demonstrate that JANUS can achieve near-optimal solutions in a reasonable amount of time.

CCS CONCEPTS

• **Networks → Network management**;

KEYWORDS

Network Control and Management, SDN

1 INTRODUCTION

Today network policy specification is at a low infrastructure-specific level and conflict detection and resolution across multiple policy writers is typically done manually. This may have been acceptable when policy changes were infrequent and applications running on typical networks were not as diverse or as large in number. Networks are highly dynamic today with applications requiring security, performance and support for mobility in addition to reachability. In addition, increasing adoption of Software Defined Networking (SDN) and Network Functions Virtualization (NFV) technologies will further accelerate the dynamic nature of policy definition and deployment as these processes will move from humans to application programs running on network controllers.

Considerable amount of work has been done by academia and industry in the past few years to create new intent-based policy frameworks and implementations [2, 6, 9, 14, 30, 41, 44, 50, 51, 60]. New policy intent languages as well as policy compilers have been designed that efficiently convert the infrastructure-agnostic policy input to low level policy configurations on network devices and middleboxes. However, most of these early efforts have focused on static security Access Control List (ACL) policies and network function traversal [6, 9, 44]. Merlin [47] considered bandwidth polices and Kinetic [34] considered dynamic network events that could trigger different policies to be applied. To the best of our knowledge, none of them consider both *dynamic* and *performance/Quality of Service (QoS)* policies. Dynamic policies can be of two main types - (1) temporal policies that deploy specific policies depending on time of day or elapsed time since a particular event, and (2) stateful policies that take into account current network and application state - for instance, diverting traffic to a fine grained intrusion detection system if a suspicious traffic signature has been detected. QoS policies set performance requirements (like bandwidth) for specific types of traffic and sets of users. For instance, a QoS policy may dictate that all executives in an organization using a video conferencing application receive a higher priority and a minimum end-to-end bandwidth of 20 Mbps.

A natural question is how to express such policies alongside ACL policies, and compose them in the presence of multiple writers. Realizing dynamic and QoS policies presents some new technical challenges compared to ACL policies: (1) *Group-centric policies*: A central aspect of intent-based frameworks is the notion of end point groups (EPGs); all policies are specified at the group-level. In our context, we must ensure that any policy we enforce applies to a group in an all-or-nothing manner; in the above example, the policy that executives receive 20 Mbps must be configured for *all*

*Work done while at Hewlett Packard Labs

CoNEXT '17, Incheon, Republic of Korea
© 2017 ACM. 978-1-4503-5422-6/17/12...$15.00
DOI: https://doi.org/10.1145/3143361.3143380

or none of the executives. This significantly constrains our setting: if care is not taken about how policies are realized, it is easy to cause fragmentation of network resources, leaving many groups' policies unsatisfied yet causing significant network resources to remain unused. There is no existing system that ensures such group atomicity. (2) *Mobility and temporal dynamics*: An additional complication is that end-points in a logical group may not be at fixed locations - consider the executive who at the start of a video conference decides to use her laptop at a customer site instead of her office. Additionally, policies may be added, removed or modified over time. Pre-computing path configurations for all possible policy changes, all possible end-point locations and user devices to satisfy the group QoS policy would be wasteful; for new policies, pre-computation is not possible. At the same time, recomputing and provisioning a group policy in real time may cause significant disruption – currently configured policies may no longer be satisfied, and even those that continue to be satisfied may experience path changes. This is highly disruptive to users and can interact poorly with policies that rely on stateful network functions (§2).

Although works like Kinetic [34] also handle network dynamics, they do not support configuring dynamic group-based policies in an efficient and low-disruption manner. Mobility and temporal dynamics further complicate the configuration process. We propose Janus, a unified system that addresses the above issues and presents a tractable solution for realizing a diverse set of dynamic group-based policies near-optimally.

Our approach relies on three key ideas. First, we create a new Integer Linear Program (ILP) representation of the policy configuration problem that enforces group-based atomicity. Second, we accommodate mobility and temporal dynamics via a novel greedy heuristic that uses policy information to tightly control the number of path changes due to dynamic events. Third, to improve the number of groups whose bandwidth policies are satisfied, we develop a bandwidth (bw) negotiation approach that systematically weakens certain policies' requested bw guarantees in high-contention time intervals (where there is significant resource crunch) and compensates such policies by increasing bw allocations at less contended time instances, and presents these altered policies for policy writers to approve.

In this paper, we present the following contributions:

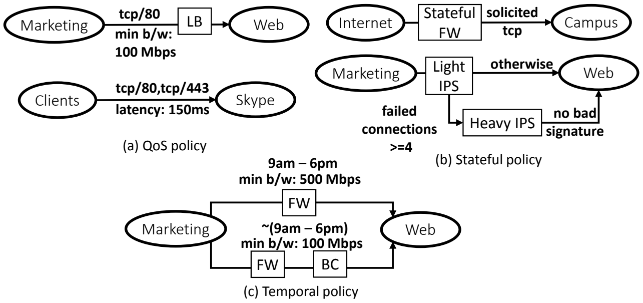
- We extend an existing intent specification called the Policy Graph Abstraction (PGA) [44] to represent QoS and dynamic policies. Our choice is motivated by the intuitive graph representation of network policies, support of network middleboxes, and the fact that PGA ideas have been included in the OpenDaylight (ODL) Network Intent Composition (NIC) project [7]. We leverage a PGA concept to express and compose intents, but our system Janus further provides how to **optimally deploy the policies based on the target network's resources.**
- We develop an optimization formulation and a heuristic algorithm that finds a solution with the objective of **maximizing the number of QoS and dynamic policy configurations** for a set of group based policies. Our secondary objective is to **minimize the number of path changes** that occur because of environment dynamics such as end-point mobility, group membership changes, network state changes, etc. This heuristic is aimed at bandwidth policies, with extensions to support latency and jitter.
- We present **bandwidth negotiation** options that achieve low disruption and low bandwidth overhead.
- We provide a comprehensive evaluation of the quality of our system for **bandwidth QoS** policies in several network topologies and scenarios. In many topologies, Janus was able to find the satisfying policy configuration for 20,000 endpoints in under 2 minutes. It was also able to avoid 90% path changes when configuring 800 policies across 5 time periods. The bandwidth negotiation protocol also assisted in satisfying additional 5.5% more policies under very congested conditions. In our experiments, each group policy typically impacts 50 end-points. Here the benefits of negotiation are amplified.

In the next section, we articulate the specific challenges in detail and further motivate this work using simple policy examples. In §3, we present a brief background of PGA and Janus system overview. Then, we provide our policy model in §4, our algorithm in §5 and a prototype implementation in §6, followed by a detailed evaluation in §7.

2 MOTIVATION AND CHALLENGES

Our research is motivated by the fact that emerging SDN and NFV applications require QoS and dynamic policies to be set in addition to basic access control policies, and the rate of policy changes is on the rise. In addition, with increasing use of bandwidth hungry and real-time enterprise applications, as well as mobile users with multiple devices, the need for efficient usage of existing capacity is also growing. Finally, the pace of business is increasing leading to rapid and often unpredictable changes/surges in workloads, and the slow paced model of over-provisioning capacity is no longer an option. All of the above facts produce some hard technical challenges that we describe in the rest of this section.

First, we briefly provide examples of QoS and dynamic stateful and temporal policies. We use an intuitive graphical representation of communication policies, in which nodes represent end-point groups (EPGs) and the directed edge between any two nodes indicates that the groups are allowed to communicate with the flow attributes annotated on the edge. Consider the examples of policies depicted in Figure 1. The two QoS policies allow a group of *Marketing* users to access a group of *Web servers* via the tcp protocol on port 80; further the web traffic is required to be load balanced by a load-balancer middlebox. Finally, this policy states that any communication between an end-point in the Marketing and a web server should be at least 100 Mbps. There could be other QoS metrics specified such as latency, hop-count, jitter, etc. The second QoS policy between end clients and the Skype server specifies that the latency should not exceed 150 msec. These high level QoS policies will need to get translated into QoS configurations such as rate-limits, priority, etc, when actually deployed on the network. Figure 1(b) and (c) depict a stateful and temporal policy respectively. Two examples of stateful policies are shown: the first is a stateful firewall that uses connection state to allow/disallow traffic between the Internet and campus users. The second stateful policy maintains state of the number of failed connections and then decides to invoke the heavy intrusion protection system. The temporal policy changes depending on the time of day to enforce the middleboxes in the path as well as the bandwidth.



(a) QoS policy

(b) Stateful policy

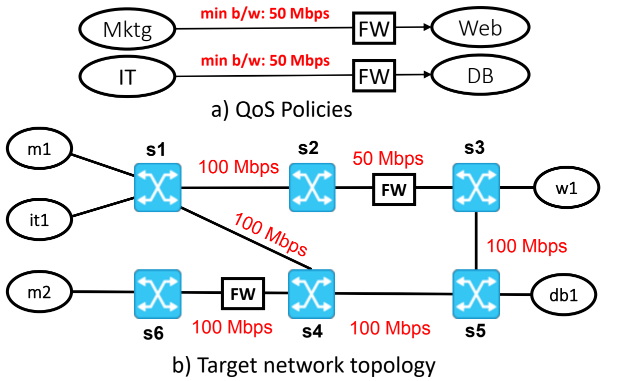
(c) Temporal policy

Figure 1: ***Examples of diverse policies***

Note that any intent-based policy framework will need the capability to express a diverse set of policies like the ones above, in addition to ACL policies. While this is an important task, extending existing systems to express QoS and dynamic policies is mostly straightforward though each system will come with its own issues. We decided to extend the PGA approach to express QoS and dynamic policies and these extensions to the policy specification are described in §4.

Our goal is to support dynamic group-based policies in an efficient manner. Implementing the composed policies is challenging because of the following requirements:

- Maintain group-based policy atomicity: Although policies are specified at group granularity, configuring them into switch rules happen at endpoint or flow granularity. We consider an EPG as an equivalent class. This allows policy writers to treat all endpoints of an EPG equally. We want to avoid partially configuring policies, i.e. it does not satisfy policies for a subset of group.
- Efficient resource provisioning: Resource requirements should be minimized to satisfy group policies. If sufficient resources are not available to satisfy all group policies, a choice needs to be made as to which policies can be rejected, which can be negotiated down (unlike ACL policies, QoS policies may have some room for negotiation) and what the protocols are to engage with the policy writers to inform and/or negotiate policy changes.
- Handling system dynamics: There are a number of system dynamics that need to be accommodated, such as state and time-dependent policy changes, end-point group membership changes, and end-point mobility. Reactively computing an optimal configuration triggered by frequent changes can incur high system overheads. At the same time, planning ahead and provisioning resources proactively may overbook resources unnecessarily.
- Low disruption incremental configuration: When policy configuration has to be changed incrementally, the goal should be to disrupt as few users as possible by minimizing path changes and re-negotiations.



a) QoS Policies

b) Target network topology

Figure 2: ***Example of policies contending for network resources. Endpoints m1 & m2 belong to Marketing group, it1 to IT, db1 to DB and w1 to Web***

We provide three examples below to demonstrate the above challenges.

2.1 Example 1: QoS policy configuration

QoS policies with bandwidth requirements depend on the network topology and link resources. Different policy writers can create policies that either conflict with each other or contend for more resources (e.g. network bandwidth) than available ones in the network.

For example, one policy requires the *minimum bandwidth guarantee* to be 100 Mbps and the other requires *maximum allowed bandwidth* to be 50 Mbps. Clearly one of these policies needs to be rewritten and the system needs to have conflict checking rules implemented.

Figure 2 shows an example of different policies contending for network bandwidth. Both policies require the traffic to go through a FW with a minimum bandwidth guarantee of 50 Mbps. There is only 1 valid path to satisfy traffic policies for "$m1$ to $w1$" ($s1 \rightarrow s2 \rightarrow s3$) and "$it1$ to $db1$" ($s1 \rightarrow s2 \rightarrow s3 \rightarrow s5$). The link $s2 \rightarrow s3$ has a bandwidth capacity of only 50 Mbps, hence it can be used to configure only 1 policy. Existing systems like Merlin [47] convert policy configuration into a flow constraint problem and inform the policy writers whether the constraint problem has a feasible solution or not, i.e. whether the current set of policies can be enforced in the current network or not. We on the other hand want to satisfy as many group policies as possible, and inform those policy writers whose policies have been violated to make small changes to their policies.

In Figure 2, the first policy can be satisfied only if we can configure flow rules for both "$m1$ to $w1$" and "$m2$ to $w1$". Maintaining group granularity during configuration (flow rule installation) is missing in the existing solution and presents additional constraints as mentioned earlier.

2.2 Example 2: System Dynamics

There are two types of changes that can happen at runtime after configuration of network policies.

Endpoint changes. These include changing the location of endpoints in the network (VM migration, endpoint mobility), adding new endpoints to the group, and changing the group and hence the policy associated with an endpoint.

Graph changes. These include modifying, adding and removing input policy graphs. All of these actions will change the composed policy graph. This will modify the policy edges between many endpoint groups, and hence the configuration for many endpoints.

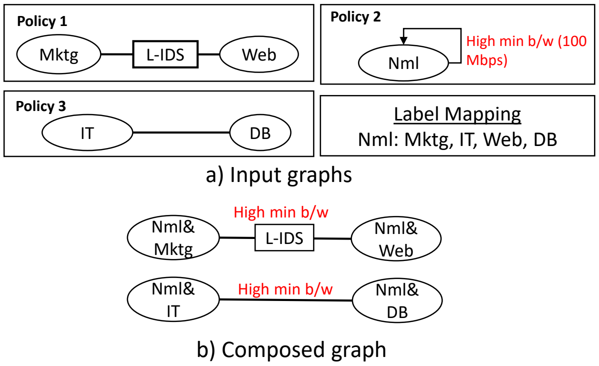


Figure 3: ***QoS composed policy graph***

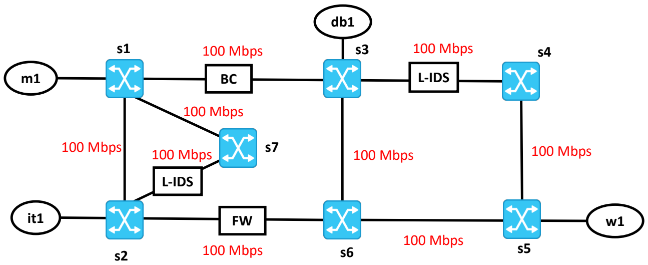


Figure 4: ***Example Topology. m1 belongs to Nml&Mktg, it1 to Nml&IT, w1 to Nml&Web, db1 to Nml&DB***

For example consider configuration of the policies mentioned in Figure 3 in the topology given in Figure 4. Traffic from $m1$ to $w1$ should go through an $L-IDS$ and it can be satisfied by 2 paths: *path1* ($s1 \rightarrow s3 \rightarrow s4 \rightarrow s5$), *path2* ($s1 \rightarrow s7 \rightarrow s2 \rightarrow s6 \rightarrow s5$), and traffic from $it1$ to $db1$ can be satisfied by the following 2 paths: *path3* ($s2 \rightarrow s7 \rightarrow s1 \rightarrow s3$), *path4* ($s2 \rightarrow s6 \rightarrow s3$). Assume the paths chosen for $m1$ to $w1$ is *path1* and for $it1$ to $db1$ is *path4*.

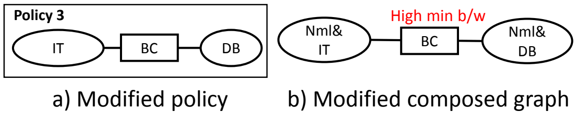


Figure 5: ***Policy 3 gets modified which changes the composed policy graph***

Assume policy graph 3 changes as shown in Figure 5. The new path which can satisfy traffic from $it1$ to $db1$ must go through a Byte Counter(BC), which is *path5* ($s2 \rightarrow s1 \rightarrow s3$). But path assigned for $m1$ to $w1$, *path1*, is already using $s1 \rightarrow s3$. So now we'll have to modify the path assigned for $m1$ to $w1$ to *path2*. This will also require us to transfer the state of L-IDS corresponding to $m1$-$w1$ traffic seen thus far from the L-IDS between $s3 \rightarrow s4$ to L-IDS between $s7 \rightarrow s2$, which is non-trivial.

As seen above, modifying a policy can also change the paths configured for other unchanged policies. Changing paths requires updating rules in multiple switches. The latency involved in updating switch rules is high [24] in all OpenFlow switches. There has been a lot of work focused on doing fast, consistent rule updates [28, 39] in a set of switches but they still incur a significant overhead. Changing paths may also require transferring states of multiple Network Function boxes (NFs) which will also result in significant downtime [22]. We want to mitigate these issues by minimizing the number of path changes.

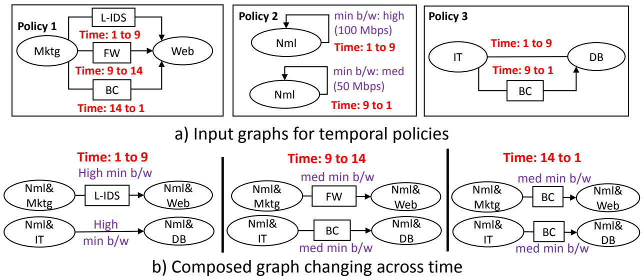


Figure 6: ***Time based composed policy graph***

2.3 Example 3: Temporal Policy

By associating time with policies, the composed policy graph will change periodically across time. Similarly the paths, NFs and network resources used by policies will also vary across time. Consider the modified input policies in Figure 6. This creates a composed policy graph that periodically changes 3 times every day. Similar to the argument in §2.2, modifying a single policy graph may change paths for other unmodified policy graphs and the frequency of these changes increases with temporal policies.

3 JANUS SYSTEM OVERVIEW

We build our Janus system on top of the PGA [44] policy management architecture, which currently only supports static ACL policies. Thus it is expected that there will be architectural similarities between the two systems. PGA is a graph-based approach to represent high-level and application-level policies, and proactively compose multiple policies by resolving possible conflicts [44]. Users (or SDN applications) naturally express their network policies using a graph through a PGA API or PGA GUI like drawing diagrams on a whiteboard [29, 37] as the examples depicted in the prior section.

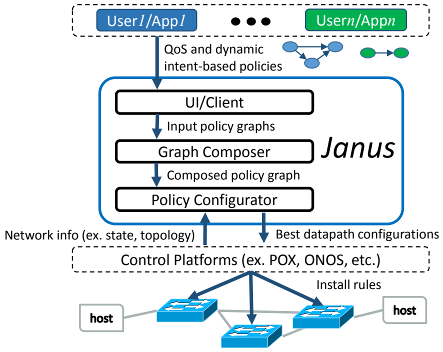

User1/App1
Usern/Appn
QoS and dynamic intent-based policies
UI/Client
Input policy graphs
Graph Composer
Janus
Composed policy graph
Policy Configurator
Network info (ex. state, topology)
Best datapath configurations
Control Platforms (ex. POX, ONOS, etc.)
Install rules
host
host

Figure 7: Janus ***architecture overview***

Figure 7 illustrates an overview of Janus architecture with key components. Like other intent-based management frameworks such as PGA, Janus conceptually resides between users/SDN applications who specify their intents via the UI and the underlying control platforms which implement the intents in the target network. In addition to security ACL policies handled by PGA, Janus allows the users/SDN applications to *express QoS and dynamic intent-based policies* based on the extended policy graph model (see §4) and constructs a composed graph from multiple input graphs based on the composition method presented in §4.

PGA [44] does not consider physical resource constraints while deploying static security policies. The implicit assumption made is that there are always enough resources to install the rules for policies. QoS policies are restricted by the target network topology and its link capacities. We introduce a new component, Policy Configurator, which is responsible for finding the dataplane configuration for all endpoints belonging to all policies. The policy configurator uses both the composed policy graph and the network topology information to generate the best dataplane configuration. This should maximize the number of policies configured in the network and minimize the number of path changes that occur when a new policy is added or modified. It uses novel heuristic algorithms to create this configuration. Janus then uses the underlying controller platform to install rules into switches based on this configuration. We will present more details about policy configurator in §5.

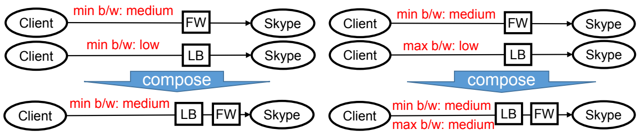
4 POLICY GRAPH MODEL

In this section, we present the policy graph model extensions, followed by the policy composition mechanism for QoS and dynamic policies respectively. Janus follows a similar graph composition model as PGA [44] with the additional support of dealing with QoS and dynamic policies.

4.1 QoS Policies

QoS policy definition. We represent and compose policies independent of the network-specific QoS values (e.g., specify bandwidth as 50 Mbps). We use *logical labels* to represent QoS levels in the QoS policies. For example, three logical labels to represent bandwidth QoS metrics could be *low* (< 100 Mbps), *medium* (> 100 Mbps and < 500 Mbps) & *high* (> 500 Mbps). The labeling system could be specific to each deployment and based on application requirements. Further, the mapping from the network independent label to the network specific value is done separately at run-time, depending on the target network environment and application requirements. Using abstract logical labels to specify policy intents makes our proposed system more flexible and extensible. Figure 1(a) provided examples of QoS policies, where the edge attributes can now have QoS metrics specified, either using logical labels or the actual desired value of the metric.

QoS policy composition. When combining QoS policies, there are two types of compositions that we need to handle: the edges being composed either have (a) the same QoS metric or (b) different QoS metrics that may or may not be related. Unlike security ACL policies, where it is somewhat easier to define policy conflicts ("deny" and "allow" are obvious conflicts), for QoS policies, the definition of conflict is more subtle and may need administrator input. For example, if there are two policies specifying the same metric - "min b/w", as in Figure 8(a), the labels *medium* and *low* are not necessarily conflicting. The administrator could institute the rule that for such policies, the composed edge should pick the label that provides better performance - *medium* for this example. In the rest of the paper, we assume this principle. To compose edges that have different QoS metrics, the same principle of picking the label representing a higher QoS level could be used. The composed edge will need to specify labels for the set of metrics from individual policies being composed. For this, we leverage prior work [5, 58] on QoS to understand the dependency relationship between different metrics (e.g., higher bandwidth also provides lower latency) and pick the labels for the individual metrics such that the composed policy does not cause the overall performance to degrade. One such simplified example with "min b/w" and "max b/w" is shown in Figure 8(b). If the metrics cannot co-exist with each other, then the conflict resolution could be to either reject one of the policies, or negotiate an acceptable QoS level .

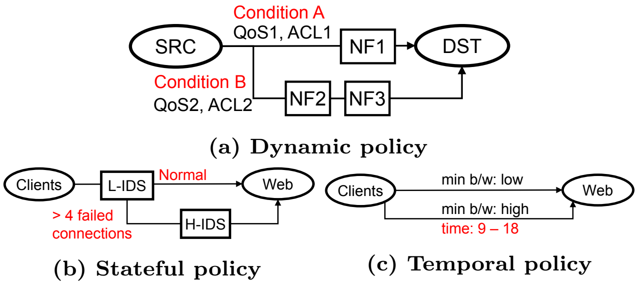


(a) Same QoS metrics **(b) Different QoS metrics**

Figure 8: ***Composition of QoS policy graph***

4.2 Dynamic Policies

Some network policies depend on the state of network flows [10, 16] and may only be valid for a specific period of time [1, 3, 33]. Janus handles dynamic policies by adding conditions in the policy graph model as shown in Figure 9a. In this example, if condition A is satisfied, traffic from the EPG SRC is allowed to the EPG DST by matching ACL1 and guaranteeing QoS1 via NF1. Whereas, if condition B is satisfied, traffic from the EPG SRC is allowed to the EPG DST by matching ACL2 and guaranteeing QoS2 via NF2 and NF3. Janus handles two types of dynamic policies: stateful and temporal.



(a) Dynamic policy

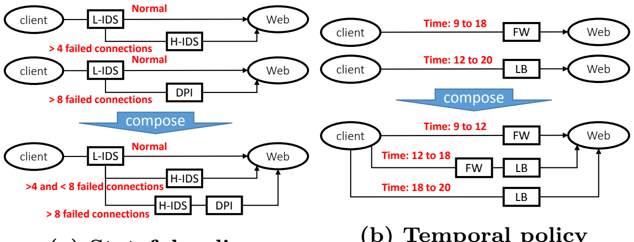
(b) Stateful policy

(c) Temporal policy

Figure 9: *Dynamic policy graph examples*

Stateful policies. A stateful policy has conditions/events associated with NF boxes (i.e. middleboxes) and the network traffic as shown in Figure 9b. The conditions on each edge are events that can be detected in the function box based on the traffic that passes through it, or these could be external events as well. In this example, if the Light-Intrusion Detection System (L-IDS) detects 4 failed connections from the EPG Clients to the EPG Web, it reroutes the network flow to the Heavy-IDS (H-IDS). The policy writer can also represent policies that require a sequence of events to occur.

Temporal policies. Temporal policies are only valid for a certain period of time. The time-period will be represented in the edge of policy graphs shown in Figure 9c. This example shows that a minimum bandwidth guarantee from the Clients to the Web is "high" from 9 to 18, otherwise, "low".



(a) Stateful policy

(b) Temporal policy

Figure 10: *Dynamic policy graph composition*

Composing Dynamic Policies. When no input graph has a dynamic policy, this reduces to the problem of policy composition for QoS and ACL policies that has been discussed earlier. Given two input policies A and B, if only one of them is a dynamic policy, (say A), then traffic goes through composed policy only when dynamic policy A is satisfied. Traffic that does not satisfy policy A goes through policy B. When both the input policies A and B have dynamic policies associated with them, traffic goes through the composed policy only when both policies are satisfied. Traffic that satisfies only policy A goes through policy A, and vice versa.

After composition, some policies may always be unsatisfiable and hence cannot exist together. In Figure 10a, > 8 and < 4 failed connections cannot be satisfied simultaneously. Such policies should be removed from the graph. Figure 10b shows an example of composing temporal policies. Network traffic from the Clients to the Web is only allowed via FW and LB during the overlapped time period (12-18).

5 POLICY CONFIGURATOR

The policy configurator is responsible for synthesizing the policy configuration specified by the composed policy graph in the dataplane of the network. The primary goal of the policy configurator is to *maximize the number of satisfied policy configurations* and its secondary goal is to *minimize the number of path changes.* In this section, we explain the optimization problem and the heuristic algorithm to achieve the primary goal, and how it can be modified to achieve the secondary goal. We mainly focus on bandwidth QoS and then propose extensions for jitter and latency metrics §5.7. We first satisfy the bandwidth requirements and then express jitter and latency as additional constraints.

5.1 Preliminaries

Input data. The main inputs required to create the optimization problem are the network topology, the composed policy graph along with the endpoint to EPG mapping, and the set of <src,dst> endpoint pairs whose policies Janus needs to configure. The network topology is a graph of nodes and links. The nodes can be a "switch" or an "NF". The topology should also have information about the link bandwidth. Paths are a natural abstraction to express these policies [25] as certain properties like the service chain ordering of NFs (waypoint constraint) are path based properties. We use similar methods as [25, 47] to generate all valid paths between endpoints. The valid path must satisfy the waypoint constraint of the policy. These paths can be pre-computed offline. The composed policy graph gives us information about the QoS requirement and the waypoint constraint of each policy, and the endpoint to EPG mapping can be used to infer the policy associated with each <src,dst> endpoint pair.

Symbols and functions. Table 1 and 2 lists the symbols and functions used in our optimization problem, along with their meaning. All the variables used in our optimization problem are indicator variables. Indicator variables, also known as binary variables, are variables which can take the value of '0' or '1'. We associate policies and paths used to configure those policies with indicator variables. For example, I_i is an indicator variable for policy i. It will take the value 1 if i gets configured in the network, and 0 if violated.

5.2 Maximize Policy Configurations

QoS policies depend on both the network traffic flow and the link bandwidth/capacity. Different policies could contend for more resources than available in the network and it may not be possible to satisfy all policies. The primary goal of Janus is to *maximize the number of policy configurations*. It uses weights to represent priority of policies.

$$\textbf{Objective:} \quad \texttt{maximize} \sum_{i \in pols} W_i \times I_i \qquad (1)$$

There are two basic constraints that needs to be considered in formulating the optimization problem.

Policy constraints. To satisfy policy i between two endpoints (x, y), Janus needs to reserve exactly 1 valid path in the network. Since Janus is satisfying policies at group

Type	Symbol	Meaning
Variable	I_i	indicator variable set as 1 if policy i is satisfied
	$P_{i,p}$	indicator variable set as 1 if path p is chosen to satisfy policy i
	ξ_i	slack variable for policy i
	$\alpha_{i,p}$	penalty to represent change of path assignment for policy i
Constant	W_i	weight assigned to policy i based on importance
	BW_i	bandwidth required to satisfy policy i
	CAP_l	capacity of link l
	ϕ	weight of penalty for violating soft constraints
	ρ	weight assigned to the secondary objective of minimizing path changes
	J_i	jitter level required to satisfy policy i
	$PR_{s,t}$	number of policies allowed at priority queue level t for switch s

Table 1: *Symbols and notions. All variables used here are indicator variables*

Function	Meaning
$pols$	returns the set of all policies
$dpols(i)$	returns the default edge policy of policy i
$ndpols(i)$	returns the set of non-default edge policies of policy i
$vp(i,x,y)$	returns the set of all valid paths between endpoints x and y to satisfy policy i
$paths(l)$	returns the set of all paths that go through link l
$spaths(s)$	returns the set of all paths that go through switch s
$links$	returns the set of all links in the network
$switches$	returns the set of all switches in the network
$eps(i)$	returns the set of all endpoint pair in policy i
$satp$	returns the set of all policies satisfied in the current network
$selp(i)$	returns the set of all path selected to satisfy policy i in the current network

Table 2: *Functions and notions*

granularity, the policy can be satisfied, i.e. $I_i = 1$, only if Janus can reserve 1 path for all the endpoint pairs $eps(i)$ associated with that policy.

$\forall i \in pols, \forall (x,y) \in eps(i)$:

$$\sum_{p \in vp(i,x,y)} P_{i,p} = I_i \qquad (2)$$

Resource constraints. The paths selected to configure all policies should not oversaturate any network link. The total bandwidth of paths traversing the link should be less than the link bandwidth.

$\forall l \in links$:

$$\sum_{i \in pols} \sum_{p \in paths(l)} (P_{i,p} \times BW_i) \leq CAP_l \qquad (3)$$

One obvious problem with using indicator variables for paths is that there could be a variable explosion. An ILP that considers all valid paths $vp(i,x,y)$ will be inefficient since the number of paths grows exponentially with the size of the network, and the ILP will become too large to solve in reasonable time. We use a heuristic algorithm which uses a random subset [25] of valid paths as the candidate paths. Similar to SOL [25], we believe choosing paths randomly can provide a high degree of edge-disjointedness among selected paths.

In §7, we show that our heuristic algorithm can give near-optimal results in a reasonable amount of time.

5.3 Configuring Stateful Policy

By not representing stateful policies, policy graphs will change with change in traffic/NF state. This may require changing the paths assigned to configure these policies. As presented in §2, this can change paths assigned for other policies too. By representing these policies, Janus knows exactly how policies can change and hence also how paths can change. It could reserve paths for changed policy (say i) beforehand, so that no other policy is using these paths and hence no other policy will have to change its path when policy i changes. The problem is that this could reduce the total number of policies configured in the system, as we have increased the number of paths to be reserved for each policy. We also know that only one of the edge conditions of policy i will be satisfied for a particular <src,dst> endpoint pair at a particular instance of time, and hence only one path would actually be used. Most of our reserved paths may be unused, which could have configured other policies.

Each stateful policy has a default edge that represents policy for normal traffic, and a set of non-default edges that represent different policies for different conditions. We aim to reserve paths for as many non-default policy edges as possible (secondary goal) while still maximizing the number of configured default edge policies (primary goal).

Soft constraints. Typically in constrained optimization problems, there are 2 types of constraints: a) hard constraints that must be satisfied, and b) soft constraints that we would like to satisfy but not at the expense of other constraints. Such optimization problems penalize the objective function for violating soft constraints. To achieve our objective, we require that to satisfy a policy i, configuring the default edge policy is a hard constraint and configuring the non-default policy edge is a soft constraint.

5.3.1 Modifying optimization problem. We introduce soft constraints using slack variables [32] in both the objective and the constraint equations. We divide policy i into $dpols(i)$ and $ndpols(i)$. The first change is that Eqn 2 will become a soft-constraint for the non-default edges and stay as hard-constraint for the default edge of a policy. This is done using slack variable ξ_i (Eqn 4). If ξ_i is 1, policy i can be satisfied even if no path is assigned for non-default edge of policy i.

$\forall i \in pols, \forall ndp \in ndpols(i), \forall (x,y) \in eps(ndp)$:

$$\sum_{p \in vp(ndp,x,y)} P_{ndp,p} = I_i - \xi_i \qquad (4)$$

$\forall i \in pols, \forall (x,y) \in eps(dpols(i))$:

$$\sum_{p \in vp(dpols(i),x,y)} P_{dpols(i),p} = I_i \qquad (5)$$

The second change is incorporating slack variables as penalty in the objective function (Eqn 6). We use variable ϕ

to control the weight of the penalty associated with violating a soft constraint. By assigning ϕ a low value, we can ensure maximizing policy configurations is the main objective.

$$\textbf{Objective:} \quad \texttt{maximize} \sum_{i \in pols} W_i \times (I_i - \phi \times \xi_i) \qquad (6)$$

5.4 Minimizing Path Changes at Runtime

As mentioned in §2.2, path changes can be tedious, as it requires changing rules in multiple switches and also transferring states of many NFs. Our secondary goal is to *minimize the number of path changes.* There are two types of runtime events that lead to path changes: endpoint and graph-based.

Endpoint related changes. These include endpoints being added/moved and changing EPG of endpoints. All these events will change the constraints in Eqn 2, 3 and we will have to re-run our heuristic algorithm. LP solvers like Gurobi uses "warm start", which allows them to start from the existing solution. For minor changes in constraints or objective functions, "warm start" can be significantly faster [59], and can produce a solution closer to the already existing solution. We show this empirically in §7.2. Hence for small endpoint related changes, we can achieve our secondary goal without modifying or re-running the heuristic algorithm from start.

Graph related changes. Graph related changes will modify the policies for all the endpoints associated with that graph. This will significantly change the constraints (Eqn 3) of the optimization problem. Here warm start can incur many path changes. In this scenario we will have to modify our heuristic algorithm to incorporate our secondary goal.

Modifying optimization problem. Indicator variables associated with paths can be used as a signal to represent path changes. The value of $P_{i,p}$ for path p and policy i can change from 1 in the initial solution to 0 in the new solution in two scenarios: a) policy i is satisfied by some other paths, and, b) no path is configured for policy i and it is violated. Both involve path changes and require modifying switch rules. We minimize such changes using variables $\alpha_{i,p}$ in Eqn 7 and 8. Eqn 7 considers only those paths ($selp(i)$) which were configured in the initial solution. Changing the value of $P_{i,p}$ from 1 to 0 in the new solution will set $\alpha_{i,p}$ to 1. These variables represent path changes and are used to create the penalty function in Eqn 8.

$$\forall i \in satp, \forall p \in selp(i): \qquad P_{i,p} = 1 - \alpha_{i,p} \qquad (7)$$

The indicator variable I_i represents the primary goal of policy i and $\alpha_{i,p}$ represents its secondary goal. Since the number of paths ($selp(i)$) required to satisfy policy i can be more than 1, we have to normalize the summation of both these indicator variables in our objective function.

Objective: `maximize`

$$\frac{\sum_{i \in ps} W_i \times I_i}{\sum_{i \in ps} W_i} - \rho \times \frac{\sum_{i \in satp} \sum_{p \in selp(i)} \alpha_{i,p}}{\sum_{i \in satp} \sum_{p \in selp(i)} 1} \qquad (8)$$

Similar to Eqn 6, Janus uses another variable ρ to control the penalty associated with path changes. The main objective is still maximizing policy configurations. This can be achieved by assigning ρ a low value.

5.5 Configuring Temporal Policy

Similar to stateful policies, temporal policies can also specify how policies can change and hence how paths can change. But for temporal policies we cannot categorize any edge as default/normal traffic edge.

We propose to convert our optimization problem into a *time-based optimization problem.* Janus already knows the various time periods, TP, at which the composed policy graph will change. Each time-period t has a separate LP/heuristic algorithm, say LP_t. And each LP_t's main goal is to configure all non-temporal policies and temporal policies which are valid for time period t. Each LP_t would also want to reduce the number of path changes that will need to happen across all time periods.

Joint optimization. One way to solve this problem is using joint optimization, where all the LP_ts will be combined into a single LP as shown in Eqn 9.

Objective: `maximize`

$$\sum_{t \in TP} \left(\frac{\sum_{i \in ps(t)} W_i \times I_i}{\sum_{i \in ps(t)} W_i} - \rho \times \frac{\sum_{i \in satp(t)} \sum_{p \in selp(i,t)} \alpha_{i,p}}{\sum_{i \in satp(t)} \sum_{p \in selp(i,t)} 1} \right) \qquad (9)$$

All the functions specified in Table 2 will become time-based functions, where the values/sets returned depends on time period t. This does not scale well with increase in number of time periods, so we propose a greedy approach which can give us near-optimal results in reasonable time.

Greedy approach. For the initial LP, LP_{t0}, Janus will try to configure all the non-temporal policies and temporal policies that are valid during $t0$. During time-period $t0$, the temporal policies for remaining time period ($TP - \{t0\}$) are analogous to the non-default policies in §5.3 as they can tell us exactly how paths can change. For time period $t0$, Janus will try to avoid using paths which may be used to configure *temporal policies of other time periods* as much as possible while still maximizing the number of configured policies for time period $t0$. The remaining LPs (LP_{t1}, LP_{t2}, LP_{t3}, ...) will also do the same with the additional goal of "minimizing the path changes from the previous LP solution".

Dealing with runtime changes. Earlier, when a new graph was added/changed, Janus tried to reduce the number of path changes in a single LP. With temporal policies, we will have to reduce the number of paths changes across *all* time periods. To incorporate this, we have to re-solve the greedy approach with the additional constraint of minimizing the number of path changes between all LP_{t_old} and LP_{t_new}.

5.6 Negotiation strategy

The greedy algorithm that we propose in §5.5 may leave out some policies (i.e., will not enforce them) in each time period.

This happens because the network does not have sufficient bandwidth to accommodate the bandwidth requirement of all policies. Recall that our formulation makes a binary decision with regard to each policy; i.e., a policy either gets its full bandwidth requirement or is not configured at all. As a result, not all links in the network will be fully utilized. Further, for any given time period, t, if we reduce the bandwidth requirements of policies, we can accommodate more policies in each time period. To compensate for the reduction in bandwidth, we can also allocate more bandwidth in future time periods whenever possible. We think our negotiation method is useful for long-lived applications which are not time-sensitivity such as backup.

Let $bpols(t)$ be the set of policies configured at time t. The policies in $bpols(t)$ are ranked (in descending order) based on the number of *bottleneck links* they utilize. For each time period t starting from $t0$, we select the top $K\%$ of policies in $bpols(t)$. For each policy p, we find if there exists a time period in future, say ft, where we can increase the bandwidth requirement of p by $N\%$ without violating the bandwidth resource constraint of any link at time ft. If such time period exists, we decrease the bandwidth requirement of p at time t by $N\%$ and increase it's bandwidth requirement at time ft by $N\%$. This is repeated for top K% policies for all time periods. The bottleneck links represent the constrained links in the network. We employ sensitivity analysis [56] on the ILP formulation(§5.2) to identify bottleneck links. By selecting policies that occupy more bottleneck links, we increase the effective available bandwidth. In §7.6, we discuss the trade-offs involved with different values of K and N. JANUS notifies the policy writers about the proposed changes.

5.7 Extension to other QoS Metrics

We cannot guarantee jitter and latency to specific values. Hence we configure them at the label abstraction.

Jitter. Using multi-level priority queues and restricting the number of policies that can be assigned to each level, we can control jitter. Eqn 10 constraints the number of policies assigned to each level of priority queue per switch. Jitter will be the lowest if traffic is assigned to the highest priority queue.

$\forall s \in switches:$

$$\sum_{i \in pols} \sum_{p \in spaths(s)} (P_{i,p} \times J_i) \leq Pr_{s,J_i} \quad (10)$$

Latency. We use number of hops as a proxy for latency. The candidate paths selected by JANUS for these policies is based on its hop-count.

A more detailed treatment of these and other QoS metrics is left to future work.

6 PROTOTYPE

Our prototype is implemented in Python and Pyretic [42], and it uses the POX Openflow controller to install rules in the network. Users create EPGs and specify the set of endpoints that belong to an EPG. The function boxes are created using the extended Pyretic language that was proposed in PGA. It supports both static and dynamic function boxes. It allows input policies to be expressed in form of EPG. Openflow switches use queues for QoS implementation by rate-limiting packets destined for different queues [21]. Although Pyretic supports dynamic policies, it cannot handle QoS policies. We extended both Pyretic and POX to install queue based rules, which enforces QoS policies.

We use Gurobi [4] to solve ILPs used in our heuristic algorithm. The input to the heuristic algorithm is a network topology and the composed policy graph created by graph composer §4. The composed policy graph [44] is stored as a hash table with source EPG, destination EPG and state as keys. Our prototype is reactive. The first packet of the new flow is matched against the composed policy graph, and the pyretic-compiled version of the rule is then installed into the underlying network topology. The configuration specified by the Policy Configurator §5 is used to select the switches in the underlying topology to install these rules in.

7 EVALUATION

In this section, we evaluate the quality and runtime of JANUS's heuristic algorithm using a diverse set of bandwidth policies and network topologies. We use topologies of various sizes from the Topology Zoo dataset [35][1]. We synthetically create our policy dataset, where each policy can be randomly assigned 0 to 2 NFs and a QoS bandwidth requirement between 10 to 30 Mbps. In all our experiments, we randomly attach different endpoints and NFs to different nodes in the network. We also randomly assign different NFs to 10-30% of nodes in the network. Optimality gap is the percentage difference between the number of policies satisfied by the original ILP (which considers all paths as candidates) and our heuristic algorithm (which considers a random subset of paths). We used this to evaluate our heuristic algorithm. Unless specified otherwise, the number of candidate paths considered by JANUS is 5, the topology is Internode, the number of endpoints belonging to each policy is 50 and the number of policies to be configured in the network is 1000. All our experiments were done on machines with 32 cores, 2.4 GHz Intel Xeon Processor and 128 GB RAM. The values shown in all our experiments are the averages taken over 10 runs. None of the existing systems tackle group based QoS and dynamic policies, hence we did not compare it with other systems.

7.1 Optimality and Scalability

We first show that our heuristic algorithm can provide a near-optimal solution in reasonable time across different network topologies. In our first experiment, we keep the number of endpoints belonging to each policy fixed as 20 and increase the number of policies in 4 different networks. We were able to achieve an optimality gap of 0% throughout this experiment.

[1] The number in parenthesis represent the number of nodes in the topology, e.g. *Internode(66)* is a 66-node topology. The topology also has bandwidth capacity of its links

As shown in Figure 11, JANUS is significantly faster across *all topologies* and the difference increases with increase in number of policies. In some cases, the difference is $2x$ in magnitude. This trend continues in Figure 12, where we keep the number of policies fixed to 1000 and vary the number of endpoints. Here we perform worse than ILP (Figure 13) but the optimality gap is still under 20%. Finally we evaluated the optimality gap and time gap (% reduction in runtime compared to ILP) for various topologies as a function of number of candidate paths. We fixed the number of policies as 1000 with 40 endpoints assigned for each policy. As shown in Table 3 and 4, JANUS with *5 paths* achieves the right balance of (a) generating *near-optimal* solution and (b) being *significantly faster* than the ILP, even when dealing with 40,000 endpoints.

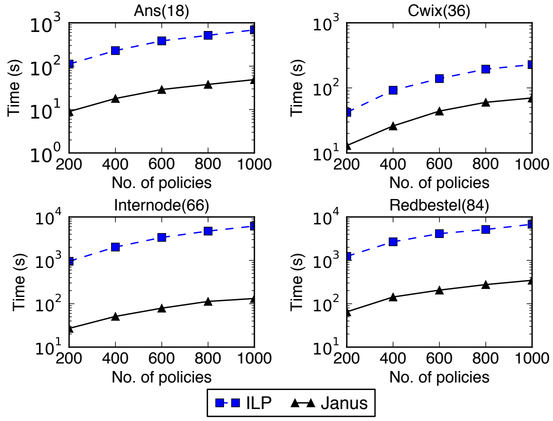


Figure 11: JANUS *VS ILP with varying the number of policies in the network. Each policy has 20 endpoints*

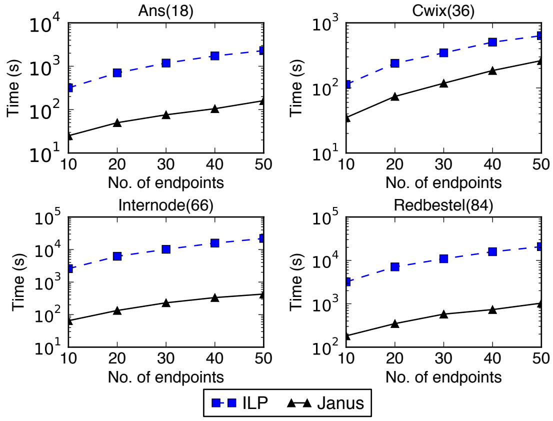


Figure 12: JANUS *VS ILP with varying the number of endpoints belonging to each policy. The number of policies to be configured is 1000.*

7.2 Warm Start for Small Changes

We claimed in §5.4 that for small endpoint related changes, "warm start" will be significantly faster with minimal number of path changes. In this experiment, we configure 600 policies and calculate the number of path changes that happen with warm-start, and the % decrease in time compared to solving from start. We randomly change the location of endpoints (endpoint changes) after deploying the initial configuration. As shown in Figure 14, when the number of endpoint related changes are less than 200, warm start will have 0 path change. For small endpoint related changes (<10), warm start is faster by over 40%. But when the number of changes are greater than 350, warm start is actually slower than solving from start. When the number of endpoint changes are large, the number of constraint changes become significant. Here warm start may require lot of backtracking to find a new solution. This increases the runtime of the solver.

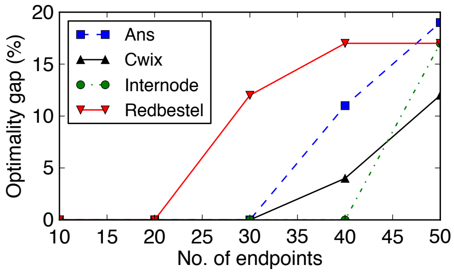


Figure 13: *Number of endpoints VS Optimality gap(%)*

Topology	Optimality Gap(%)				
	20 paths	10 paths	5 paths	2 paths	1 path
Ans(18)	0	0.6	*10.3*	23.2	33.5
Agis(25)	0	0	*0*	14.6	28.9
CrlNetServ(33)	0	0.9	*10.7*	25.8	36.9
Cwix(36)	0	0	*4*	19.8	32.8
Garr201008(55)	0	0	*3.3*	12.4	24.8

Table 3: *No. of paths considered VS Optimality gap(%)*

Topology	Percentage reduction in time (%)				
	20 paths	10 paths	5 paths	2 paths	1 path
Ans(18)	74	77.4	*93.8*	97.3	98.9
Agis(25)	11.6	49	*61*	88.9	95.3
CrlNetServ(33)	6.9	37.8	*66.8*	87.9	94.9
Cwix(36)	16.4	42	*58.5*	87.4	94.3
Garr201008(55)	95	97	*99*	99	99

Table 4: *No. of paths considered VS % reduction in time*

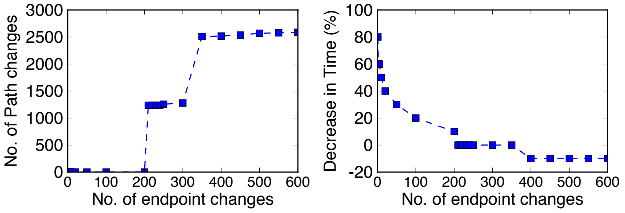


Figure 14: *Performance of Warm start*

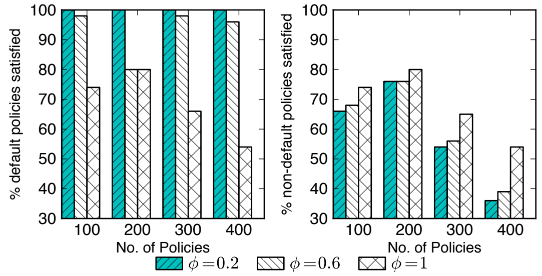


Figure 15: ***The percentage of default/non-default policies configured with different penalty weight***

7.3 Configuring Stateful Policies

Here we show that having a low penalty weight (ϕ) associated with violating soft constraints allows us to reserve paths for many non-default policy edges while still maximizing the number of configured default edge policies. In this experiment, each policy has 1 default and 2 non-default policy edges. We vary the number of policies to be configured in the topology. As shown in Figure 15, the number of configured default policies is inversely proportional to ϕ and number of configured non-default policies is directly proportional to ϕ. Setting ϕ to 0.2, allows Janus to configure all default policies while still enabling it to reserve paths for 35% to 75% of non-default policies.

7.4 Configuring Temporal Policies

Here we evaluate the performance of our greedy heuristic algorithm §5.5 for configuring temporal policies. We increase the number of policies from 500 to 800 and spread them across 5 time periods. We measure the runtime and the percentage decrease in total number of path changes compared to rerunning our original heuristic algorithm §5.2 for each time period. We set ρ *as 0.2* in all our experiments. As shown in Table 5, our greedy approach reduces the number of path changes by over 90%. The joint optimization algorithm did not complete even after running for over 20 hours.

No. of Policies	No. of Configured Policies	Reduction in path changes(%)	Time(s)
500	500	98.2	492
600	600	94.7	675
700	691	92.6	1438
800	741	91.3	4157

Table 5: ***Performance of greedy heuristic algorithm***

7.5 Weights represent priority

In this experiment, we show that weights can be translated directly into priorities. We increase the number of policies from 600 to 1300 and split them evenly across 3 priority classes *high*, *med* and *low*. The weights assigned to *high*, *med* and *low* priority class are 8, 4 and 2. As shown in Figure 16, most of the unconfigured policies belong to *low* priority class. We did not see any unconfigured *high* priority policy until the number of unconfigured *med* and *low* priority policies became greater than 100 and 200 respectively.

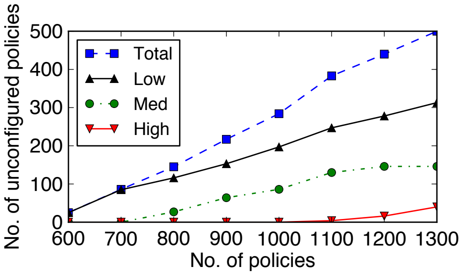


Figure 16: ***Distribution of number of unconfigured policies based on priority class/weights***

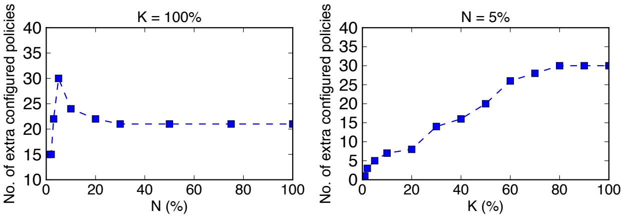


Figure 17: ***Negotiation strategy over increasing N, K***

7.6 Negotiation strategies

In §5.6, we proposed a negotiation strategy for temporal policies. In this section, we evaluate it by varying the value of N (% of bandwidth to be shifted) and K (% of policies considered). In this experiment we tried to configure 600 policies across 4 time periods. Without our negotiation strategy we were able to configure 536 policies. We make two interesting observation from Figure 17: (a) with $K = 100\%$, we can configure a maximum of 30 extra policies when $N = 5\%$. When $N > 5\%$, the number of policies that can be shifted to other time periods decreases due to lack of network bandwidth. (b) After $K = 60\%$, the increase in the number of extra policies configured is not significant.

8 DISCUSSION

Dealing with topology changes. In this paper, we do not focus on handling reconfiguration of policies due to topology changes/failures. We could extend our system to handle this in a manner similar to §5.4 (using similar constraints and objective function), where we aim to minimize reconfiguration due to endpoint/policy graph related changes. Also there are works [49] that either pre-install backup paths to handle a certain number of link failures or do minimal repair where they find new paths but with the aim of minimizing the number of switches whose rule have to be modified.

Work Conservation. Janus is a reservation system that maximally allocates bandwidth, i.e. after allocation there is no policy that can be atomically allocated. Our bandwidth allocations are rigid. We do not focus on ensuring work conservation.

Fast/consistent bulk rule update. Installing large number of rules in a network topology is not trivial. Some of the main challenges include (a) maintaining consistency and correctness by avoiding ACL violations, blackholes and transient forwarding loops, and (b) doing fast rule update to avoid low network utilization (He et al [24] have shown that path changes caused by policy reconfiguration can result in significant downtime). This is an important research problem and outside the scope of this paper. Hence we did not evaluate the runtime impact of reconfiguration. Both Dionysus [28] and McClurg et al [39] ensure consistency during the transition between old and new configuration by carefully selecting the order of rule updates in each individual switch. Dionysus also makes reconfiguration faster by considering runtime differences in update speeds of different switches.

Practicality. Our main idea of Janus has been started by adding real values from existing network management solutions to existing intent-based approaches. As a survey stage, we have reviewed major commercial network management systems [1, 3] and have obtained many feedback from real network administrators. As a result, QoS, stateful network functions, and temporal policies handled by Janus in this paper are the major and important factors. Using Janus, administrators can express their performance and dynamic policies easily, deploy them to the target network with optimization, and receive a feedback for negotiation. We believe Janus will be easily integrated with any intent-based system.

9 RELATED WORK

Intent-based management. Policy-based management helps network administrators to simplify management tasks by separating the rules governing the behavior of a system from its functionality, and it has been well-adapted in the field [48, 53]. However, it has some limitations of scalability and flexibility for managing complex heterogeneous network systems. In order to overcome the limitations, many latest work have proposed to create new intent-based policy frameworks by designing new intent (or high-level) languages and compilers/interpreters [8, 20, 26, 42, 47, 51, 54], application-level abstractions [11, 14, 18, 30, 44, 50, 55, 60], group-based approaches [6, 9, 31, 44] and specifying new northbound interfaces [2, 19, 41]. However, most of those approaches have focused on security-related ACL or reachability policies without considering resource limitations.

QoS/SLA management. There is a huge body of work on network QoS. However, none of them have addressed dynamic group-based policies at an intent level. QoS frameworks such as IntServ/DiffServ have been proposed to support QoS in the Internet. Other frameworks support QoS/SLA provisioning for SDN/Cloud/NFV [12, 13, 23, 27, 36, 38, 43, 45, 46]. Our approach can leverage and be deployed through these frameworks. Openflow [21, 40] also supports QoS by mapping a flow to a pre-configured queue and we have used it for our prototype implementation.

Dynamic policies. Some works have discussed dynamic policies in terms of modeling, verification, and testing [10, 15–17, 52, 57, 61] of network function middleboxes. In order to specify, test, and verify the stateful policies, we can use the previous approaches. Procera [33] has proposed to help operators express event-driven network policies that react to various types of events such as time, data usage, and flow but it was still limited to supporting simple allow/drop actions.

Configuring policies. There are some works [25, 34, 47, 49] on configuring policies or synthesizing data planes. But none of them maintain *group-based* policy atomicity during configuration. For example, SOL [25] provides a near-optimal solutions and device configurations to implement *path-based* optimization. However, Janus is located on a higher level than them using logical label-defined groups and can use them to deploy polices to the target infrastructure.

10 CONCLUSION

There is a growing demand for networks to support a variety of rich performance/QoS and security requirements. However, existing intent based management frameworks only support security policies. In this paper we proposed Janus, a system to configure QoS (mainly bandwidth policies with an extension to support latency, jitter) and dynamic intent-based policies at group granularity. We show how to extend an existing policy graph abstraction to express these policies and to compose them when they conflict. We develop a variety of novel heuristic algorithms which maximize the number of configured policies and minimize the number of path changes caused by either intrinsic dynamics in policies or due to policy churn. Our evaluation has shown that Janus can offer near-optimal solutions in a reasonable amount of time for several network topologies and scenarios. Further, Janus's negotiation feature can increase the number of configured policies. Future directions include integrating Janus's features into existing intent-based systems such as OpenDaylight/NIC, extending it to support other QoS policies and evaluation on real diverse policy datasets from production enterprise environments.

11 ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Laurent Vanbever for their insightful comments. We also thank Raajay Vishwanathan and Kausik Subramanian for their useful feedback. This work is supported by the National Science Foundation grants CNS-1302041, CNS-1330308, and CNS-1345249. This work was initiated and performed during Anubhavnidhi Abhashkumar's internship and while Sujata Banerjee, Ying Zhang and Wenfei Wu were employed at Hewlett Packard Labs.

REFERENCES

[1] 2017. Aruba Networks: Aruba ClearPass Policy Manager. https://tinyurl.com/j98p6xk. (2017).
[2] 2017. Boulder Intent-based NBI. http://tinyurl.com/zzvolsb. (2017).
[3] 2017. Cisco: Creating Time-of-Day QoS Service Policies. https://tinyurl.com/j7l8tq9. (2017).
[4] 2017. Gurobi. http://www.gurobi.com/. (2017).
[5] 2017. NETWORK PERFORMANCE: LINKS BETWEEN LATENCY, THROUGHPUT AND PACKET LOSS. http://tinyurl.com/j5jo7zr. (2017).
[6] 2017. OpenDaylight Group Policy. https://wiki.opendaylight.org/view/Group_Policy:Main. (2017).
[7] 2017. OpenDaylight Network Intent Composition (NIC) Graph Implementation. https://tinyurl.com/gld2qzn. (2017).
[8] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014.

NetKAT: Semantic Foundations for Networks *(POPL '14)*. New York, NY, USA, 113–126. DOI:https://doi.org/10.1145/2535838.2535862

[9] Anu Mercian, Felipe Yrineu, Joon-Myung Kang, Raphael Amorim, Saket M Mahajani, Mario Sanchez and Sujata Banerjee. 2016. Network Intent Composition (NIC) Be Feature Update and Demo: Intent Compilation, Lifecycle Management and Automated Mapping. http://sched.co/7RBY. Presented in OpenDaylight Summit 2016.

[10] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2015. SNAP: Stateful Network-Wide Abstractions for Packet Processing. *CoRR* abs/1512.00822 (2015). http://arxiv.org/abs/1512.00822

[11] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don'T Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations *(SIGCOMM '16)*. New York, NY, USA, 328–341. DOI:https://doi.org/10.1145/2934872.2934909

[12] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. HUG: Multi-resource fairness for correlated and elastic demands. In *USENIX NSDI*.

[13] Freddy C Chua, Julie Ward, Ying Zhang, Puneet Sharma, and Bernardo A Huberman. 2016. Stringer: Balancing latency and resource usage in service function chain provisioning. *IEEE Internet Computing* 20, 6 (2016), 22–31.

[14] R. Cohen, K. Barabash, B. Rochwerger, L. Schour, D. Crisan, R. Birke, C. Minkenberg, M. Gusat, R. Recio, and V. Jain. 2013. An intent-based approach for network virtualization. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. 42–50.

[15] Seyed K Fayaz and Vyas Sekar. 2014. Testing stateful and dynamic data planes with FlowTest. In *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 79–84.

[16] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. 2016. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 275–289. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/fayaz

[17] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. 2013. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 19–24.

[18] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. 2012. Hierarchical Policies for Software Defined Networks. In *HotSDN (HotSDN '12)*.

[19] Andrew D. Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. 2013. Participatory Networking: An API for Application Control of SDNs. In *SIGCOMM*.

[20] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 279–291.

[21] Open Networking Foundation. 2014. OpenFlow Switch Specification Version 1.5.0. (2014).

[22] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. ACM, 163–174.

[23] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. 2010. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*. ACM, 15.

[24] Keqiang He, Junaid Khalid, Sourav Das, Aaron Gember-Jacobson, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. 2015. Latency in software defined networks: Measurements and mitigation techniques. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 43. ACM, 435–436.

[25] Victor Heorhiadi, Michael K Reiter, and Vyas Sekar. 2016. Simplifying software-defined network optimization using SOL. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 223–237.

[26] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. 2009. Practical Declarative Network Management. In *WREN*.

[27] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 435–448.

[28] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic scheduling of network updates. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 539–550.

[29] Joon-Myung Kang, Sujata Banerjee, Jeongkeun Lee, and Mario Sanchez. 2017. Policy Canvas: Draw Your Policies for OpenStack Services. OpenStack Summit 2016 Austin, http://tinyurl.com/zsszpb7. (2017).

[30] Joon-Myung Kang, Jeongkeun Lee, Vasudevan Nagendra, and Sujata Banerjee. 2017. LMS: Label Management Service for Intent-driven Cloud Management. In *Proc. IM*. 177–185.

[31] Nanxi Kang, Ori Rottenstreich, Sanjay Rao, and Jennifer Rexford. 2015. Alpaca: Compact network policies with attribute-carrying addresses. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 7.

[32] Eric C Kerrigan and Jan M Maciejowski. 2000. Soft constraints and exact penalty functions in model predictive control. In *Control 2000 Conference, Cambridge*.

[33] H. Kim and N. Feamster. 2013. Improving network management with software defined networking. *IEEE Communications Magazine* 51, 2 (February 2013), 114–119. DOI:https://doi.org/10.1109/MCOM.2013.6461195

[34] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. 2015. Kinetic: Verifiable Dynamic Network Control. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 59–72. http://dl.acm.org/citation.cfm?id=2789770.2789775

[35] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The internet topology zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (2011), 1765–1775.

[36] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, and others. 2015. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 1–14.

[37] Jeongkeun Lee, Joon-Myung Kang, Chaithan Prakash, Sujata Banerjee, Yoshio Turner, Aditya Akella, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. Network Policy Whiteboarding and Composition. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 373–374. DOI:https://doi.org/10.1145/2829988.2790039

[38] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. 2014. Application-driven bandwidth guarantees in datacenters. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 467–478.

[39] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. 2015. Efficient synthesis of network updates. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 196–207.

[40] 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.

[41] Minh Pham and Doan Hoang. 2016. SDN applications - the intent-based Northbound interface realisation for extended applications. In *IEEE Workshop on SDN and IoT (SDN-IoT 2016)*.

[42] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing Software Defined Networks. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 1–13. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto

[43] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. 2013. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. *ACM SIGCOMM Computer Communication Review*

43, 4 (2013), 351–362.

[44] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 29–42. DOI: https://doi.org/10.1145/2829988.2787506

[45] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying middlebox policy enforcement using SDN. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 27–38.

[46] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival O Guedes. 2011. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks.. In *WIOV*.

[47] 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.

[48] John Strassner. 2003. *Policy-based network management: solutions for the next generation*. Morgan Kaufmann.

[49] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2017. Genesis: synthesizing forwarding tables in multi-tenant networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 572–585.

[50] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 426–439. DOI:https://doi.org/10.1145/2934872.2934874

[51] C. Trois, M. D. Del Fabro, L. C. E. de Bona, and M. Martinello. 2016. A Survey on SDN Programming Languages: Toward a Taxonomy. *IEEE Communications Surveys Tutorials* 18, 4 (Fourthquarter 2016), 2687–2712. DOI:https://doi.org/10.1109/COMST.2016.2553778

[52] Brendan Tschaen, Ying Zhang, Theo Benson, Sujata Banerjee, Jeongkeun Lee, and Joon-Myung Kang. 2016. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *IEEE SDN-NFV Conference*.

[53] D. C. Verma. 2002. Simplifying network administration using policy-based management. *IEEE Network* 16, 2 (Mar 2002), 20–26. DOI:https://doi.org/10.1109/65.993219

[54] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. 2013. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. ACM, New York, NY, USA, 87–98. DOI:https://doi.org/10.1145/2486001.2486030

[55] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. 2016. Ravel: A database-defined network. In *Symposium on Software Defined Networking (SDN) Research, SOSR 2016*. ACM. DOI:https://doi.org/10.1145/2890955.2890970

[56] James E Ward and Richard E Wendell. 1990. Approaches to sensitivity analysis in linear programming. *Annals of Operations Research* 27, 1 (1990), 3–38.

[57] Wenfei Wu, Ying Zhang, and Sujata Banerjee. 2016. Automatic Synthesis of NF Models by Program Analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*. ACM, New York, NY, USA, 29–35. DOI:https://doi.org/10.1145/3005745.3005754

[58] Xipeng Xiao and L. M. Ni. 1999. Internet QoS: a big picture. *IEEE Network* 13, 2 (Mar 1999), 8–18. DOI:https://doi.org/10.1109/65.768484

[59] E Alper Yildirim and Stephen J Wright. 2002. Warm-start strategies in interior-point methods for linear programming. *SIAM Journal on Optimization* 12, 3 (2002), 782–810.

[60] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. 2015. Scenario-based programming for sdn policies. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 34.

[61] Ying Zhang, Wenfei Wu, Sujata Banerjee, Joon-Myung Kang, and Mario A Sanchez. 2017. SLA-Verifier: Stateful and Quantitative Verification for Service Chaining. In *Proc. INFOCOM*.