MIT Libraries | DSpace@MIT

MIT Open Access Articles

Tuneman: Customizing Networks to Guarantee Application Bandwidth and Latency

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Sharma, Sidharth, Kushwaha, Aniruddha, Alizadeh, Mohammad, Varghese, George and Gumaste, Ashwin. 2022. "Tuneman: Customizing Networks to Guarantee Application Bandwidth and Latency." ACM Transactions on Internet Technology.

As Published: http://dx.doi.org/10.1145/3575657

Publisher: ACM

Persistent URL: https://hdl.handle.net/1721.1/147674

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of Use: Article is made available in accordance with the publisher's policy and may be subject to US copyright law. Please refer to the publisher's site for terms of use.

Massachusetts Institute of Technology

Tuneman: Customizing Networks to Guarantee Application Bandwidth and Latency

SIDHARTH SHARMA*, IIT Indore, India
ANIRUDDHA KUSHWAHA*, IIT Indore, India
MOHAMMAD ALIZADEH, MIT, USA
GEORGE VARGHESE, UCLA, USA
ASHWIN GUMASTE, IIT Bombay, India

We examine how to provide applications with dedicated bandwidth and guaranteed latency in a programmable mission-critical network. Unlike other SDN approaches such as B4 or SWAN, our system Tuneman optimizes *both* routes and packet schedules at each node to provide flows with sub-second bandwidth changes. Tuneman uses node-level optimization to compute node schedules in a slotted switch, and does dynamic routing using a search procedure with QoS-based weights. This allows Tuneman to provide an efficient solution for mission-critical networks that have stringent QoS requirements. We evaluate Tuneman on a telesurgery network using a switch prototype built using FPGAs, and also via simulations on India's Tata Network. For mission-critical networks with multiple QoS levels, Tuneman has comparable or better utilization than SWAN while providing delay bounds guarantees.

CCS Concepts: • **Networks → Programmable networks**; **Packet scheduling**; **Network experimentation**.

Additional Key Words and Phrases: QoS, scheduling, network programmability, FPGAs, mission-critical networks, segment routing

1 INTRODUCTION

The Internet has been an amazing success for nearly 50 years, partly because of a key design decision to make the network "application-agnostic." In other words, IP treats all applications – whether remote telesurgery or Facebook – as a group of addresses that sends a bag of bits to each other. This allowed the Internet to support applications like email and later accommodate unforeseen applications like video. Today we take for granted applications like Uber for hailing a cab but these run on mobile devices that are connected by an IP-centric network. There are many more exciting networked applications around the corner, whether networks of self-driving cars, remote telesurgery, precision agriculture, and more. Many of these new applications have critical requirements from the network. They are so essential for society that they demand unprecedented levels of performance and reliability from the network. Consider a surgeon doing remote telesurgery in AIIMS, a hospital in Delhi to a village 100km away in Haryana, India. Such a network has stringent requirements on latency (< 10 msec) and fixed bandwidth (≈ 10 Mbps) [14]. Consider another example of a trading network with stringent real time constraints connecting

*The majority of the work was done when the authors were with IIT Bombay.

Authors' addresses: Sidharth Sharma, IIT Indore, Indore, India, sidharth@iiti.ac.in; Aniruddha Kushwaha, IIT Indore, Indore, India, aniruddha@iiti.ac.in; Mohammad Alizadeh, alizadeh@csail.mit.edu, MIT, Cambridge, USA; George Varghese, varghese@cs.ucla.edu, UCLA, Los Angeles, USA; Ashwin Gumaste, ashwing@ieee.org, IIT Bombay, Mumbai, India.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
1533-5399/2022/12-ART $15.00
https://doi.org/10.1145/3575657

Just Accepted

trading centers in the New York area. Providing guaranteed Quality of Service (QoS) is critical for such networks as delay variation can have severe impacts [33, 50]. We call these types of networks "mission-critical" and focus on the problem of automatically configuring them to provide applications with both dedicated bandwidth and small guaranteed latencies. Note that the primary objective of designing such a network is to provide guaranteed delay and bandwidth for a specific class of service (such as telesurgery). However, for economic viability, other types of services are also added to saturate the network.

Existing solutions only solve part of the problem. For example, centralized SDN implementations like B4 [27] and SWAN [26] maximize traffic utilization while guaranteeing application flow bandwidths and but do not guarantee flow delay bounds. As in SWAN [26] our definition of a flow is a higher level "application flow" (called a service in SWAN) consisting of a source-destination pair and an application. It is not a TCP flow; an application flow will typically comprise many TCP flows that come and go.

Our solution starts by replacing classical router output scheduling (using algorithms such as DRR [46] and iSLIP [35]) with a slotted scheduler at each output link that can be programmed by a centralized SDN controller. Slotted schedulers are cheap to implement and allow fine tuning of node delay bound without the expense of a classic Weighted Fair Queuing (WFQ) scheduler [19] needed for Parekh-Gallagher style delay bounds [40]. The SDN controller also controls routes. When an application flow or service makes a request for bandwidth B and delay bound D, the controller searches for a route, starting with the shortest path and then moving on to the next shortest paths as in SWAN [26].

However, unlike B4 or SWAN, at each switch in the prospective route, the controller does a node level optimization to see whether the current output link schedule can be adjusted to accommodate a node delay bound for the new flow. If the sum of the node delay bounds meets the requested application delay bound D, the controller allocates the flow on the route. Otherwise, it moves on to consider other routes. If all routes fail, the controller uses more Draconian measures and considers shifting an existing flow (with a more relaxed delay bound) to an alternate route to accommodate the new flow. The algorithm avoids thrashing using a measure called *mobility factor* that estimates the likelihood of a flow repeatedly evicting other flows.

In some sense, our approach *functionally emulates SWAN [26] (traffic engineering, bandwidth bounds) but augments it with delay bounds while doing so in an incremental, distributed manner*. In our approach, as in the classical SDN vision [45], distributed protocols such as routing are eliminated. Our approach goes further, however, and also eliminates distributed scheduling mechanisms such as DRR [46] and iSLIP [35]. Instead a single controller orchestrates the routes *and* schedules of all application flows at every router. The controller is responsible for jointly optimizing routes and schedules. Since the general problem is computationally hard [30], we decompose it into a node optimization and a route search procedure as described above. We call the resulting framework Tuneman because it customizes the network to meet each application's QoS needs. We demonstrate Tuneman with application mixes such as remote telesurgery, video traffic, and Web-surfing on a testbed consisting of Terabit routers built from FPGAs, and also using simulations on a much larger country-wide network.

The Tuneman framework is completely automated to minimize the amount of operational expertise needed to run a mission-critical network. This is because operational expertise is hard to obtain, especially for applications like telesurgery, where the network spans villages. Since the Internet protocols by design provide no application/service level support, router vendors over the years have coped by adding mechanisms like MPLS and tuning knobs like QoS and DiffServ [7] to guarantee performance. However, these knobs must be set manually on each device by experts in order to make the network application-aware, which we seek to avoid. We also focus on minimizing the cost of a switch. This is partly because of our design choices: segment routing [24] and label switching to avoid prefix and ACL lookups, and slotted schedulers to avoid complex QoS mechanisms. We employ segment routing in Tuneman because it provides flow-level QoS guarantees and has a significantly less control overhead than the TE protocols used on the internet today [17, 55].

Just Accepted

Figure 1 positions Tuneman with respect to other approaches from the literature. The first choice is whether to use distributed routing and scheduling as in MPLS-TE. Comparisons with SWAN (see Figure 14 in [26]) show that such distributed traffic engineering solutions are far from optimal in utilization compared to centralized traffic engineering. When we turn to centralized solutions, the controller can either control routes only (as in B4 and SWAN) or can control both routes and schedules. As we will see in our evaluation, controlling only routes as in SWAN [26] or B4 [27] cannot guarantee latency, as they allocate flows to routes based on bandwidth needs and not latency. Silo [29] also only controls routes but uses rate limits and delay calculus [15] to provide bandwidth guarantees and delay bounds. However, Silo uses VM placement to maximize utilization, which is possible in a data center but not in a mission-critical network.

Next, if we choose to have centralized control of both routes and schedules, we can choose to control routes and schedules at hosts or at routers. Fastpass [41] is the main prior work we know of that uses an SDN approach to schedule flows, but at end nodes not at routers as we do. However, Fastpass's edge coloring algorithm only works in topologies such as Fat Trees that are reconfigurably non-blocking. Fastpass also does not provide application bandwidth guarantees. Fastpass does, however, work with existing switches. By contrast, Tuneman requires a new switch design (but which allows us to optimize for switch cost because of our design choices) with a slotted link scheduler that can be programmed by a centralized SDN controller. Similarly, [30] does not have the hardware switch capabilities that Tuneman leverages for node-level scheduling in order to provide bounds on both delay and bandwidth for each application. A broader comparison for related work is in Table 3, and detailed experimental comparisons with SWAN can be found in § 7.

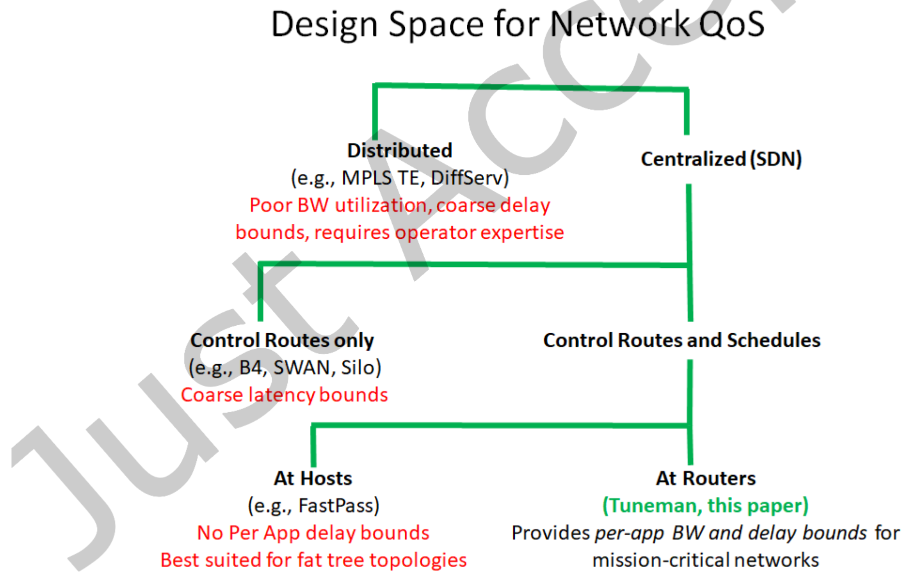


Fig. 1. Positioning of Tuneman.

Just Accepted

To use Tuneman, besides the physical topology, applications need only specify their traffic demands (e.g., bandwidth) and performance objectives (e.g., latency) to the controller.

The contributions of this paper are:

1. Architecture: We show that a fully programmable, application-aware network design – where both routes and packet schedules are under the control of a central SDN scheduler – is both computationally scalable (Figure 12a) and effective (Figure 12b, § 7).

2. Algorithms: We introduce a new optimization framework for node scheduling (§ 3) and a search algorithm for rerouting flows that leverages a new metric: mobility factor (§ 4).

3. Evaluation: We describe a terabit router testbed that emulates an actual Telesurgery setup (§ 6) and simulations on India's Tata network to show that Tuneman has better utilization and latency for mesh networks with several QoS levels than MPLS-TE and SWAN (§ 7).

2 TUNEMAN ARCHITECTURE

Interface: Tuneman provides an API to applications for specifying their needs, and has interfaces to configure routers and end-nodes. Tuneman converts application bandwidth and delay needs to a set of schedules that are sent to each router, and routes that are sent to end-nodes. Note that applications are service requests (flows), of granularity at least a few Mbps, and not TCP sessions. Though nodes and routers are connected to Tuneman, they are not required to be synchronized with each other. Applications use a 5-tuple to specify their needs to the controller consisting of source(s), destination(s), bandwidth, latency guarantees and QoS values, which constitutes a flow. Packets with lower QoS values are dropped before packets with higher QoS values in times of overload or when a route is not found.

Design Choices: First, consider the schedulers at individual routers. Observe that simple solutions like strict priority can only guarantee latency bounds for the highest priority flow even at one router. Simple weighted round robin schemes like Deficit Round Robin [46] have poor delay bounds [30], while Generalized Processor Sharing (GPS) approximations like W2FQ [5] require expensive sorting. Instead, our FPGA-based programmable dataplane routers simply cycle through a slotted allocation pattern.

Creating a large number of source-destination classifiers at each router is expensive. MPLS-like tunnels suffice but require lookup tables at each switch. Segment routing makes router forwarding trivial with no forwarding state at routers; the state mapping each flow to paths is stored at end-nodes (which need to store such state information anyways for transport protocols).

In summary, to keep routers simple and low-cost for mission-critical networks, the "backend" of the Tuneman compiler is a segment-routed network with data-plane programmable routers (implemented as FPGAs) that use slotted schedules.

Design Overview: At the highest level, the Tuneman controller solves a bin-packing problem where the ”bins” are time slots on each output link in the network and the ”items” are flows (Figure 3). In classical bin packing [6], the only constraint is that the sum of the item sizes allocated to each bin must not exceed the bin size. In our problem, not only must the sum of allocated flow bandwidths not exceed the bandwidth of each link, but the slot allocations should guarantee flow latency bounds. Further, our problem is a multi-resource bin packing problem, where the resources (network links) are coupled by the network graph structure.

Since bin-packing by itself is $\mathcal{NP}$-hard [6] and the additional latency constraint along with the graph structure only makes the problem harder [10], exact solutions (even using solvers) are infeasible [30]. Instead, Tuneman hierarchically decomposes the problem into path selection and node scheduling: it uses search heuristics to find a feasible path, and an exact local optimization to compute node schedules. Since the optimization requires milliseconds, it is infeasible to redo the optimization when packets enter and leave; instead the optimization is only done when the delay or bandwidth requirements of a service changes or a new service is added.

To evaluate a prospective route, Tuneman selects a plausible path (§ 4), and then computes a schedule for every node along the path by including the new flow's 5-tuple. The new flow's parameters are checked against existing provisioned flows. Schedules are computed on a per-node basis. If nodes along a selected path can accept their new schedules (inclusive of the new request), then that path is chosen, and the request is provisioned. If not,

Just Accepted

Tuneman attempts to find another path by invoking routing algorithms presented in (§ 4). Tuneman achieves sub-1-second provisioning.

To compute a schedule for a port at a node, Tuneman uses an optimization mechanism that takes as a constraint the permissible latency through each node. The latency of an end-to-end path is bounded by a threshold set by the application, and this end-to-end bound is divided equally among all nodes in the path to determine the latency for the flow in each node. Tuneman guarantees bandwidth (through schedules) and latency (through node-constraints). Tuneman assumes that flows are shaped at the ingress nodes to avoid bursts at switches that would invalidate flow delay guarantees.

For reasons explained earlier, the current Tuneman data-plane uses segment routing. Ports are node-wise uniquely numbered, and a path is identified by the conjoining of ingress and egress port numbers of nodes along the path. Each end-node (edge devices, not routers, routers do not need such tables) has a table that maps an incoming packet to a segment-routed label. (called megalabel, §6). For a flow entering the Tuneman-controlled network, the ingress node has a table entry of the form <protocol identifier: segment-routed label>. The segment-routed label is pushed on to incoming packets (like MPLS). From then on, packet forwarding at each router happens just by examining the node-relevant portion of the label and avoiding time costly match table look-ups.

We now describe the detailed design as to how node schedules are computed, followed by the details of routing in §4.

3 NODE LEVEL SCHEDULING

Before delving into the details of node level scheduling, let us consider the global picture of how Tuneman handles flow requests. Figure 2 shows that flow requests are handled by one of the following scenarios.

1. Start up: In this scenario, there is no traffic in the network and a routing algorithm is invoked that calculates the shortest path for a new traffic flow and provisions the flow. Upon provisioning a flow, a schedule is obtained by computing epoch time (ϵ_j) for each output port j for all the nodes along a desired path.

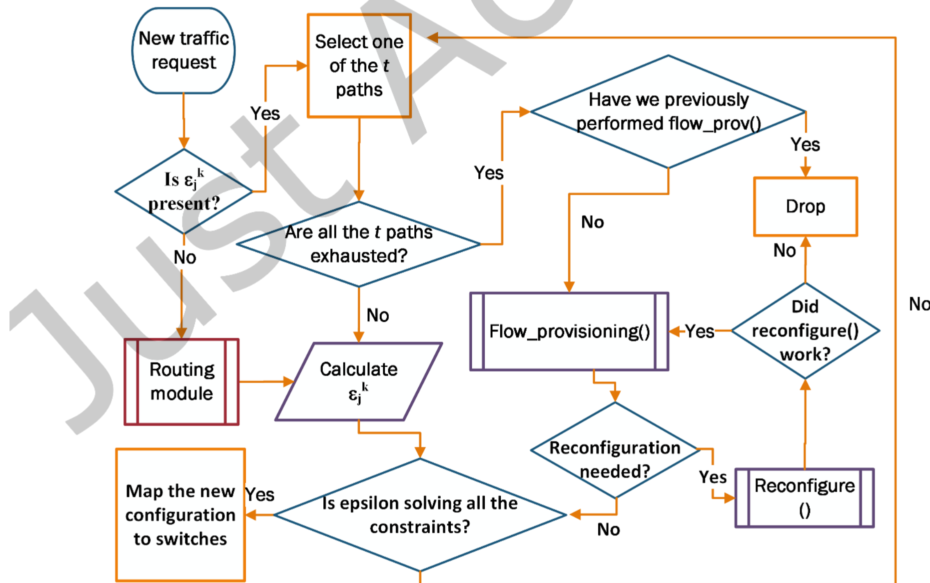


Fig. 2. A flow chart for the Tuneman provisioning system.

Just Accepted

2. Feasible provisioning: In this case there is traffic in the network, and a new flow can be provisioned readily by using the routing algorithm. Specifically, for a new traffic request, one of t possible paths is chosen, such that the path satisfies provisioning constraints by computing ϵ_j at every node along the path.

3. Feasibility subject to reconfiguration: In this case, a network is heavily loaded and a new request arrives. However, this new request cannot be provisioned only by the routing algorithm and choosing one of the t paths. This is because the bandwidth/delay for the new request is not met. In such a case, we make use of a Dynamic Flow Provisioning (DFP) algorithm (described in §4) that consists of two modules: reconfiguration and flow provisioning, which together attempt to re-allocate existing traffic in order to find a suitable path. Finally, if despite attempts to reconfigure existing flows, the new request cannot be provisioned, then it is dropped.

For all three scenarios, computing ϵ_j is required to find a schedule at every node (along the provisioning path) to meet the bandwidth and delay requirements for all the provisioned flows. This step is referred to as epsilon solving in Figure 2. The details of epsilon solving are as follows.

Each output port j at a node supports a cyclic schedule of duration ϵ_j. This cycle consists of slots A_{ij} (time allocated to send data from an input port i to an output port j) for all input ports i that intend to send data to output port j. The epoch repeats as long as there is no change in the flow bandwidth (BW)/delay at port j. However, if a flow using port j requests the controller for a change in bandwidth or delay guarantee, then the epoch duration needs to change. Assume the end-to-end delay for a particular flow T^m_{ab} is Δ^m_{ab}, where V_a and V_b are the source and destination nodes for the flow, and m is a flow instance between (V_a, V_b). If the path to provision the flow is of h hops, then we constrain the maximum delay at each node on that path to be $\delta^m_{ab} \leq (\overline{\Delta^m_{ab}})/(h-1)$, where $\overline{\Delta^m_{ab}}$ is obtained after subtracting propagation delay (for the chosen path) from Δ^m_{ab}.

Figure 3 is a toy example that shows a 3x3 node, with Virtual Output Queues (VoQs) supporting a non-blocking switch. With the VoQs, the switch is essentially a $3(3-1)$x3 or 6x3 switch. There are 6 flows, 2 arriving at each input port and some combination of 2 flows exiting at an output port. Flows arrive as multiple packets and are buffered. To transfer packets from an input port i to the output port j, a fix slot time is provided. We store these slot times in a matrix A. An element of the matrix A_{ij} represents the slot time available for input port i to transfer

Just Accepted

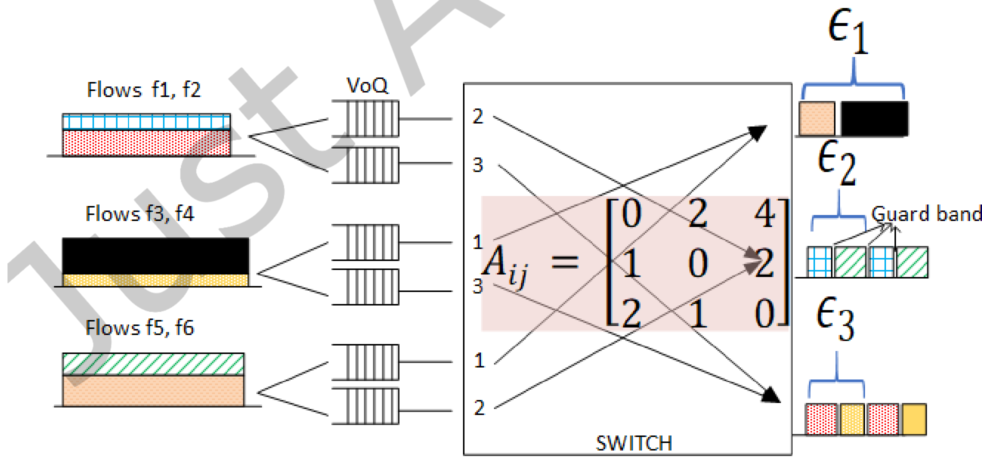


Fig. 3. Scheduling at a 3-port switch. At the input side, each port has 2 VoQs. Schedules and time slots for data destined to output port from different input ports are shown. The matrix A_{ij} provides slot time information from an input port i to an output port j. Since the switch does not forward a packet on the same port where it has been received. Therefore, $A_{ij} = 0$, if $i = j$.

Table 1. Optimization model parameters

Parameter	Description
T_{ab}^m	The mth traffic request (flow) from node V_a to node V_b.
B_{max}	Maximum size of a VoQ buffer.
P	Number of ports.
P_{c_j}	Output rate at port j.
A_{ij}	Time-slot duration for input port i at output port j .
ϵ_j^k	Epoch time for output port j at node k (sometimes also referred as ϵ_j) .
Δ_{ab}^m	End-to-end delay requirement for the flow T_{ab}^m.
δ_{ab}^m	Maximum allowable delay at a node for the flow T_{ab}^m.
δ_{ij}^k	Maximum allowable delay at node k between port i to port j.

packets to output port j. In Figure 3 matrix A is shown, where A_{21} and A_{31} represent the time slots available for transferring packets from input ports 2 and 3, respectively, to output port 1. Also, the summation of the elements of a column constitutes the epsilon value for the respective output port. For instance, summation of column-1 of matrix A in Figure 3 is $A_{11} + A_{21} + A_{31} = 3$, where 3 is ϵ_1 value for output port 1. The figure also shows a penalty for switching from one input port to another. We refer to this as a *guard-band* interval G; no transfer of data takes place during this interval. These guard-band intervals are also accounted for in the epsilon computation. Therefore, the final epsilon value for output port 1 becomes $\epsilon_1 = 3 + G$. Thus the optimization seeks to maximize ϵ_j, the cycle time for port j independently for each port, in order to minimize wasted throughput (due to guard bands). Note that in our architecture, the ϵ_j for each port need not be equal. In Figure 3, the cycle ϵ_1 is bigger than ϵ_2 .

We now describe how to compute the epoch time ϵ_j. We have listed the optimizations parameters and their description in Table 1.

Objective: Intuitively, we wish to maximize throughput (i.e. reduce the number of interchanges between the ports–guard-band) subject to delay and buffer constraints. Hence, the objective function we use is to maximize the cyclic schedule for each port j is: $\mathbf{max}[\epsilon^j]$

Constraints:

Delay constraint: $\epsilon_j/2 \leq \delta_{ij}^k$. This constraint implies that the average waiting time within a switch should be less than the maximum permissible delay for that switch. The maximum permissible delay δ_{ij}^k is the tightest delay bound among all the flows for which packets enter switch k from port i and exit from port j, i.e., $\delta_{ij}^k = min[\delta_{ab}^m]$, $\forall T_{ab}^m$ passes from port i to port j at node k.

Rate constraint: $\forall i, \;\; T_{ab}^m.\epsilon_j \leq P_{C_j}.\left[A_{ij}\right]$. The rate constraint implies that the total arrival in time ϵ_j for every traffic request at each of the ingress ports i is less than or equal to the departure at egress port j. Note that A_{ij} is computed by the optimization (see below).

Buffer constraint: $T_{ab}^m.\epsilon_j \leq B_{max}$. This constraint ensures that the rate of packets entering the buffer and leaving through the VoQ switch is within the buffer threshold.

Slot constraint: $\forall i, j, \quad A_{ij} \geq (\epsilon_j - G \times (P-1)) \times (\sum_{a,b,m} T_{ab}^m / \sum_{a,b,m,i} T_{ab}^m)$.

This constraint ensures that a slot time is allocated in proportional fraction of traffic bandwidth. In the first term of the above equation, the guard-band G is of 5-20 *ns* and P is the port count of a switch. For a P port switch, at every output port, we need to assign a slot for each of the P input ports. Therefore, for a given output port j, once a slot for input port i gets over, the next input port in the sequence (i.e., $(i+1)^{th}$ port) gets the chance

Just Accepted

to transmit packets to j. In this process, there exists a small interval (gap) while switching from port i to port $i+1$. This gap corresponds to guard band interval G. For a P port switch, there will be $(P-1)$ such gaps during which no packets can be forwarded and the switch remains idle. Subsequently, the second term in the constraint provides a fraction of total traffic destined for port j from port i. The slot time proportional to the computed fraction is assigned to port i from the epoch time ϵ_j after removing the time wasted due to guard-band intervals.

Node optimization ensures that the switch by itself is work conserving as long as there is conformance to the committed information rate promised by the controller to the application. Appendix A enhances this simple optimization module above to incorporate traffic that varies from its average value to a peak value using robust optimization [4]. The idea is that any x of the y requests at a node are at their peak value, while the other (y-x) are allowed to be at a known average value, though we do not know which of the x requests are at their peak values.

The node optimization model was implemented in Matlab + Gurobi and used as an executable in a Python simulator. This LP assumed the following major parameters: switch port count 4-128; line-rate 1-100Gbps; max buffer size of 1 ms per port. Other parameters include: ϵ_j is bounded to 250μs at a node; $\delta_{ij}^k \leq 150\mu$s; bandwidth of flows from 1 to 100Gbps, in increments of 100Mbps; number of flows at a node for the non-robust case 400 × port count $\sim (1600 - 51200)$. For a sweep of the bandwidth and flow parameters, Matlab finds a feasible result in under 2.5 ms for the worst case.

4 DYNAMIC FLOW PROVISIONING

Our routing algorithm (see flow chart in Figure 2) accommodates a new request either directly by considering the ϵ_j values of all the ports of nodes along the shortest path and then provisions the request, or by reconfiguration. Reconfiguration involves re-routing already provisioned flow(s) on an alternative path(s), thus incorporating the new request. For re-routing, only those flows are considered that have lower priority (QoS) than the new request. In more detail, following the flow chart of Figure 2, when a request arrives, Tuneman first attempts to provision the request over one of the t shortest paths if there is node-level compliance: in other words, the node optimization results in a valid schedule at all the nodes in the path.

We note that as the product of the flows and bandwidth assigned to a path increases, A_{ij} decreases for all nodes on the path. But to avoid fragmentation, A_{ij} cannot fall below the time to forward 1 MTU sized packet. Thus even if the node optimization procedure in the last section works and computes a A_{ij}, a node schedule is considered *invalid* if the time-slot corresponding to A_{ij} falls below 1 MTU. If for all t shortest paths, even one node on each path is invalid, we say all paths are exhausted (see Figure 2) and we invoke the DFP algorithm for reconfiguration.

The DFP algorithm makes use of ϵ_j value that is available from the last successful run of the optimization model. This value is useful in determining if latency bounds and bandwidth needs are met for the new and affected flow requests. We compute latency over a path in a graph $G(V,E)$ as:

$$l_y = \sum_{k:k \in P^{ab}(y)(V)} \epsilon_j^k/2 + \sum_{e:e \in P^{ab}(y)(E)} \phi_e, \tag{1}$$

where $\epsilon_j^k/2$ is the average latency at an output port j at node k, given node k is on path y. Further, $P^{ab}(y) : y \in \{1, 2, ..., t\}$ is the yth path between node V_a and V_b. The second term in the above equation is the sum of the propagation delay over all the links in path $P^{ab}(y)$, denoted by ϕ_e. The available bandwidth over a h-hop path is calculated as:

$$min\{V_1^{bw}, V_2^{bw}, ..., V_h^{bw}\}, \text{where } V_k^{bw} = P_{C_j} - \Sigma T_{ab}^m \tag{2}$$

The DFP algorithm consists of two functions that work in tandem called *flow_provision()* and *reconfigure()*, shown in Algorithms 1 and 2. We now step through the DFP algorithm in detail which takes as input the network state

Just Accepted

(i.e., already provisioned traffic, denoted by T, and the corresponding set of paths P). For a new request T_{ab}^m, between nodes V_a and V_b the algorithm starts with *flow_provision().*

flow_provision() function: In step 1, t paths for a new traffic request T_{ab}^m are computed. Thereafter, P^{ab} is computed by eliminating a path p from the computed t paths. For a new request, p is null; for re-provisioning an existing flow to an alternate path, p is set to the existing path of the flow. Removing path p from P^{ab} ensures that the flow provision function does not provision an existing flow on the same path again. In step 2, the algorithm computes whether sufficient bandwidth is available (using equation 2) on the yth available path $P^{ab}(y) : P^{ab}(y) \in P^{ab}$ to accommodate the new request. If sufficient bandwidth is available and delay bound is met (using equation 1) on path $P^{ab}(y)$, then the algorithm provisions the new request on this path. In case sufficient bandwidth is not available or the delay bound is not met on any of the available paths in P^{ab}, then the algorithm calls *reconfigure()* with a path $P^{ab}(y)$ as an input (step 3). The *reconfigure()* function may return the updated network state (T and P), such that sufficient bandwidth is now available on path $P^{ab}(y)$ to provision the request. This provisioning of a request may require additional re-routing for some of the already provisioned requests from path $P^{ab}(y)$ to some other path(s) because of insufficient bandwidth on $P^{ab}(y)$. In step 4, the state of the network (T and P) is updated by adding the new request and its path.

reconfigure() function: A *reconfigure()* function is used for selecting a provisioned traffic request in the network for re-routing over an alternate path. Re-routing enables sufficient bandwidth to be available on a selected path $P^{ab}(y)$. The selected path is provided as an input from *flow_provision().* In step 1 of the *reconfigure()* function, a list (S_{ab}) of all the nodes over path $P^{ab}(y)$ is generated. In step 2, the bottleneck node(s) that cannot accommodate the new request on the path $P^{ab}(y)$ are identified. The requests provisioned through a bottleneck node (V_i) are then sorted in descending order (set T_i), based on their value of a measure called mobility factor M_f (described later in this section). For re-routing, a request corresponding to the highest mobility factor (i.e. $T_i(1)$, which was provisioned over a path $P_i(1)$) is selected from the sorted list of flows and the *flow_provision()* function is invoked. The new network state is denoted by $T - T_i(1), P - P_i(1)$. This state implies that neither the traffic $T_i(1)$ nor the path $P_i(1)$ exist in the new network state and thus we avoid any possibility of provisioning same service

Algorithm 1: Function *flow_provision().*

```
   Input  : T, P, T^m_ab, Δ^m_ab, p  Output: T, P
1  P^ab = find_t_shortest_paths(a, b) − p
2  provision = 0
3  Step 2: // Check for bandwidth and delay on paths
4  for y = 1 to |P^ab| do
5  |   if avail_bw(P^ab(y)) ≥ T^m_ab and delay(P^ab(y)) ≤ Δ^m_ab then
6  |   |   provision = 1
7  |   |   avail_bw(P^ab(y)) = avail_bw(P^ab(y)) − T^m_ab
8  |   |   Break
9  Step 3: // In case sufficient bandwidth is not available or delay bound is not met, choose a path in P^ab for
     reconfiguration.
10 if provision = 0 then
11 |   (P, T) = reconfigure(T, P, T^m_ab, P^ab(y))
12 |   avail_bw(P^ab(y)) = avail_bw(P^ab(y)) − T^m_ab
13 Step 4: //Update network state (T,P).
14 P = P ∪ P^ab(y), T = T ∪ T^m_ab
15 Step 5:Return P, T
```

Just Accepted

Algorithm 2: Function *reconfigure()*

Input : $T, P, T_{ab}^{m}, P^{ab}(y)$, **Output :** T, P

1 **Step 1:** // Find all the nodes over the path P_s^{ab}
2 $S_{ab} = get_nodes_on_path(P^{ab}(y)), Z = 0$
3 **Step 2:** list all bottleneck nodes
4 **for** i *in* S_{ab} **do**
5 | **if** $avail_bw(i) < T_{ab}^{m}$ **then**
6 | | $BS_{ab} = BS_{ab} \cup V_i$
7 **for** i *in* BS_{ab} **do**
8 | $(T_i, P_i) \leftarrow$ provisioned requests at node i having bandwidth $\geq T_{ab}^{m}$ and $QoS \leq QoS(T_{ab}^{m})$
9 | **if** $(T_i, P_i) == \emptyset$ **then**
10 | | Terminate Algorithm //reconf. not possible
11 | $(T_i, P_i) \leftarrow$ requests sorted in descending order of M_f at node i.
12 | // provision first request on another path.
13 | $(P, T) = Flow_provision(T - T_i(1),$
14 | $P - P_i(1), T_i(1), \Delta_i(1), P_i(1))$ //passing values
15 | **if** $T_i(1)$ *provisioned on path other than* $P_i(1)$ **then**
16 | | De-provision $T_i(1)$ across nodes along $P_i(1)$
17 | | $T \leftarrow T - T_i(1), P \leftarrow P - P_i(1), Z \leftarrow Z + 1$
18 | **else**
19 | | call $Flow_provision$ with the next request from the sorted set (T_i, P_i) until exhausted
20 **if** $Z \neq |BS_{ab}|$ **then** Terminate Algorithm
21 **Step 3** Return P, T

over the same path again. The current path of the traffic $P_i(1)$ is also used as an input to the *flow_provision()* function. This ensures that traffic to be re-routed does not use its current path for reconfiguration. This procedure is repeated for each bottleneck node along the path $P^{ab}(y)$.

Time complexity of DFP algorithm: To find t-shortest paths in a network graph $G(V, E)$ the complexity is in the order of $O(tV(E + V \log V))$ using Yen's algorithm [56]. Tuneman calculates t-shortest paths for each source-destination pair in the network and stores it. For every new request to be provisioned, Tuneman checks whether sufficient bandwidth is available on the t-shortest paths for the request. To check for sufficient bandwidth on t paths, we need to traverse every node along those paths. A path can have up to V nodes. Hence the time complexity for this step is $O(t.V)$. In case, none of these t-shortest paths have sufficient bandwidth, Tuneman starts the reconfiguration process. To this end, the first shortest path is chosen, and an attempt to find bottleneck nodes is made. There could be V bottleneck nodes and each node may have, on average S services. Tuneman identifies candidate services for reconfiguration in O(S) and calculates the mobility factor for each candidate service. To select a service for reconfiguration, Tuneman sorts candidate services based on the mobility factor in $O(S \log S)$. Finally, there is recursion for re-routing; as a result, there can be maximum $V.S$ iterations. Hence, the total time complexity of our algorithm is computed as $O\big((tV(E + VlogV) + VS.(t.V + V.S \log S)\big)$.

Performance micro-benchmarks in §5 show that DFP is able to keep most flows of higher priority provisioned on their shortest path, penalizing lower QoS flows by sending them over longer paths if required. By contrast, a naive algorithm drops flow requests beyond a network load of 60%; however using DFP and a mobility factor, flow requests are dropped only after 80% load. These results are sensitive to the topology used; this dependency is examined in §5.

Just Accepted

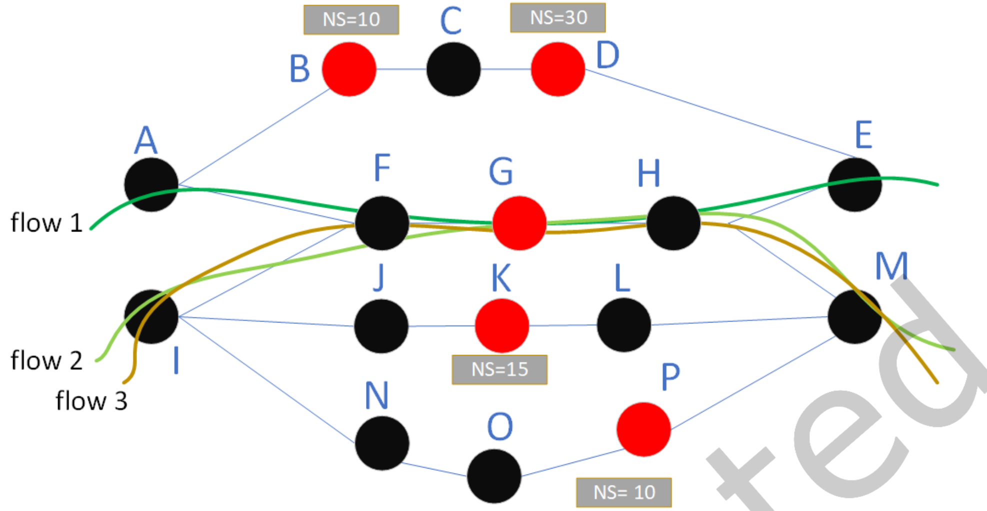


Fig. 4. Motivating mobility factor: choosing the best flow to reconfigure.

Just Accepted

Recursive deadlock: The process of provisioning a candidate request may require re-routing of other provisioned request(s). This could lead to recursive calls of the two functions and an infinite loop. To avoid this, we allow our algorithm to terminate when either a flow is successfully configured or there are no other flows available for re-routing with a lower QoS value than the new flow request. For the algorithm to select a candidate flow for rerouting, we associate a numerical value with each routed request called M_f or *mobility factor*, which when high, implies that the request is a suitable candidate for rerouting (based on QoS, path availability and latency). Selecting a lower QoS flow for re-routing limits the pool of choices and ensures that after several iterations our algorithm will select the lowest QoS flow for re-routing. By continuously choosing lower QoS flows, our algorithm exhausts candidates that can be selected, leading to termination.

To motivate the mobility factor measure, consider a network with 16 nodes as shown in Figure 4. Suppose a new flow request needs to be configured over the path F-G-H. However, there is no bandwidth available at (red) node G; this is when Tuneman invokes the DFP algorithm. At node G, three different flows (flow 1, 2 and 3) are configured with lower priority but with higher bandwidth requirement than the new request. Hence the DFP algorithm can select a flow for shifting out from among these three flows and move it to an alternate path. Assume the next shortest path for flows 1, 2 and 3 are A-B-C-D-E, I-J-K-L-M and I-N-O-P-M, respectively. Given the three choices, Tuneman attempts to choose a flow whose shifting does not lead to a large chain of reconfigurations. While reconfiguring a flow on its alternate path, Tuneman still needs to guarantee the latency requirement of the flow. Therefore, only those flows are considered for reconfiguration, for which latency criteria are fulfilled at one of the alternate paths. For instance, if the path I-J-K-L-M does not meet the flow 2 latency requirement, then flow 1 and flow 3 are the only candidates for reconfiguration.

Next, assume nodes B and D on flow 1's alternate path do not have enough capacity to accommodate flow 1. Also, nodes K and P do not have enough bandwidth for flows 2 and 3 respectively on their next shortest paths. We refer to these nodes as bottleneck nodes. Hence if we choose to move flow 1, we need to free-up bandwidth at both nodes B and D. By contrast, choosing flow 2 or 3 requires freeing up bandwidth only at a single node (node K or P). Intuitively, the larger the number of bottleneck nodes on a path, the larger is the chain of reconfiguration calls and vice-versa. Therefore, flows 2 and 3 with a single bottleneck node are better candidates for shifting.

Further, to select among flows 2 or 3, Tuneman checks the number of existing flows at bottleneck nodes K and P which can be shifted to their next shortest paths if required. A higher count implies a larger pool to select a flow to shift. Therefore, Tuneman selects flow 2 to reconfigure on path I-J-K-L-M because it has a pool of 15 flows (NS = 15) on bottleneck node K in contrast to 10 flows on bottleneck node P for flow 3.

Motivated by the above example, we propose a mobility factor, which is instrumental in DFP choosing the "best" flow for reconfiguration among a pool of candidate flows.

Computing Mobility Factor M_f: Based on the example, the mobility factor of a flow is computed using three factors.

(1) *Latency slack* is defined as the difference in latency between the worst-case path and an existing path from among t paths. We calculate latency slack as:

$$l = (l_w - l_y)/l_w \tag{3}$$

Where, l_w represents the worst latency across the t paths. Further, l_y is the latency over a path on which a flow is currently provisioned i.e. $y \in \{1, 2, ..., t\}$.

(2) The *bottleneck node* over a path (on which we are attempting to provision this flow) is defined as a node that does not have sufficient bandwidth available to provision this request. The total number of bottleneck nodes over a path is B.

(3) M_s is the *minimum number of flows* provisioned across all the bottleneck nodes that have lesser QoS and higher bandwidth than the requested flow.

To summarize, the larger the latency slack of a flow, the more likely it is to find an alternative path such that the latency requirement of the flow is satisfied. On the other hand, a higher value of M_s, implies more options (larger pool of lower QoS flows are available) from which a flow can be selected for re-routing over an alternate path. Therefore, M_f should be directly proportional to both latency slack and M_s. By contrast, a large number of bottleneck nodes (B) implies a correspondingly higher requirement to reconfigure a larger number of flows on an alternate path. This suggests that we make M_f inversely proportional to the number of bottleneck nodes. Putting the pieces together, we define the mobility factor of a flow as:

$$M_f = \frac{l * (M_s + 1)}{B + 1} \tag{4}$$

5 PATH COMPUTATION BENCHMARKS

A Python based simulation model was developed for *reconfigure()* and *flow_provision()* functions and applied to a 200-node topology modelled as a power-law graph with average degree of connectivity across nodes as 4. Nodes have 2-16 ports at 10Gbps rate. Flows were generated between random source-destination pairs and have one of the 8 QoS values. The probability of selecting any QoS value for a flow is 0.125 (equal distribution among all 8 QoS classes). The arrival of the flows follow exponential distribution (with $\lambda = 6.53$ ms) and the Cumulative distribution function (CDF) of the inter-arrival times is shown in Figure 5. The bandwidth requirements for the flows are normally distributed [18] with a mean of 300 Mbps and variance of 200 Mbps. Between each source destination pair, we calculate and store sixteen shortest paths ($t = 16$) before provisioning any service. Load is calculated as the ratio of actual occupancy across all the links to the sum of all link capacities divided by the average path-length; hence it lies in [0, 1]. We refer to this definition of load throughout the paper. The average hop-count is 4.25 (representing a typical tier-2 network).

Effective traffic shifting: In Figure 6a we show the capability of our algorithm to keep the flows of higher priority provisioned on their shortest path, penalizing lower QoS flows by sending them over longer paths (if required). Figure 6a shows that 93% of the flows having the highest QoS value are provisioned on the shortest path. This validates the *reconfigure()* function and the intuition behind the computation of M_f. In the absence of our algorithm, flows are evenly distributed over the available paths. The result shows that our algorithm always attempts to keep the traffic of the highest QoS value on the shortest path. Due to reconfiguration, flows having

Just Accepted

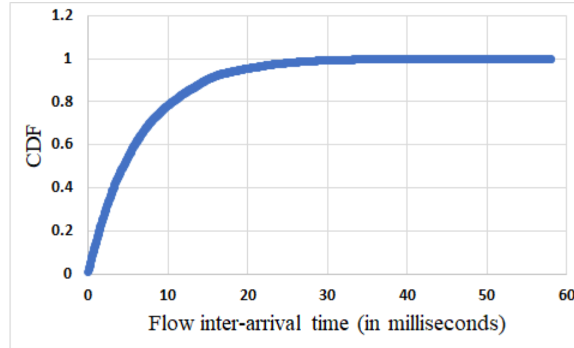
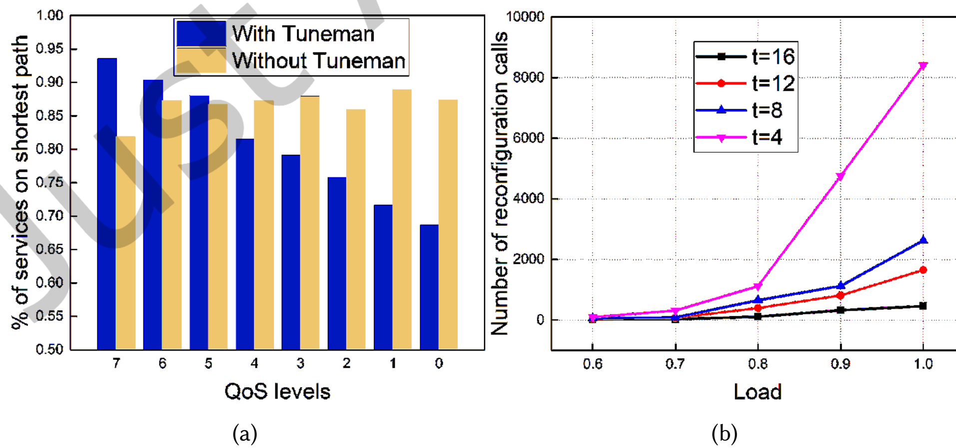


Fig. 5. CDF of the inter-arrival times of the flows.

lower QoS values may not be provisioned on their shortest path as compared to an approach where there is no reconfiguration. The lower the QoS value of a flow, the lower the likelihood of its being provisioned on the shortest path.

Next, in Figure 6b, we observe the relationship between the frequency of reconfiguration as a function of number of paths t and load. The value of t impacts the number of reconfiguration calls with increasing load. We experimented with four different values of t: 4, 8, 12 and 16. Below a load of 80%, all flows are successfully provisioned without requiring any reconfiguration. Note that the higher the value of t, lower the number of reconfigurations. For lower values of t (such as $t = 4$), the number of reconfiguration calls increases drastically (at higher loads). This is because more flows will result in higher chances of reconfiguration.

Next in Figure 7a, we show how the use of mobility factor improves traffic provisioning time by reducing the number of recursive calls in the algorithm. To obtain this result, we computed the number of flows re-routed by the algorithm with and without using mobility factor at different loads. We then calculate the percentage improvement seen in the number of recursive calls by the use of mobility factor for different values of t. For



(a) (b)

Fig. 6. a) Fraction of flows on their shortest path for different QoS values. b) Number of reconfiguration function calls for different values of t (number of shortest-paths considered) with load.

Just Accepted

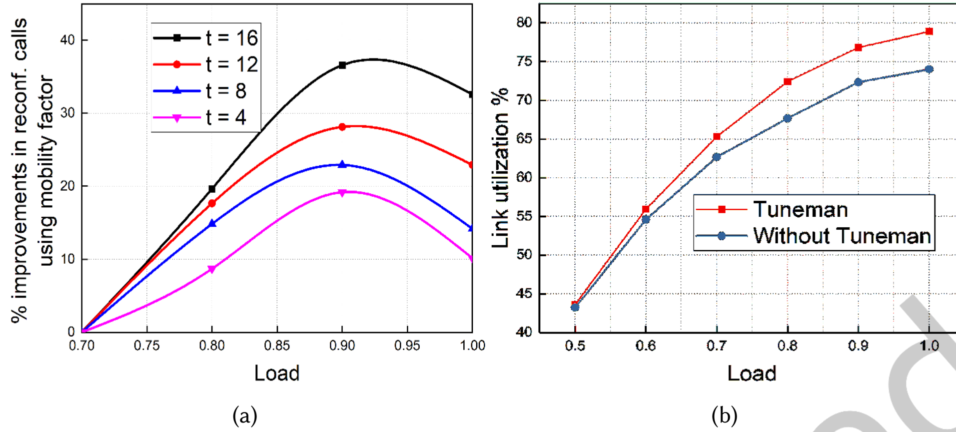


Fig. 7. a) Percentage improvement in number of reconfiguration function using mobility factor with load. b) Average link utilization (with and without Tuneman).

Just Accepted

$t = 16$, an improvement of 36.5% is observed at a load of 90%. Even for smaller values of t, such as $t = 4$ the improvement is around 20% at this load. However, for all the values of t, improvement due to mobility factor falls after a load of 90%. This is because all the candidate flows lead to similar reconfigurations at heavy loads and less available paths.

Next, in Figure 7b, we show the effect of the reconfiguration algorithm on average link utilization in the network. Tuneman achieves better link utilization as compared to the traditional approach. The improvement is much more noticeable at the high loads. This is because Tuneman drops fewer services and pushes low-priority services on the longer paths to increase path diversity.

Next, in Figure 8, we show how Tuneman improves the acceptance ratio of the service requests. For this experiment, we show the percentage of requests dropped for every considered QoS class, when services are provisioned with and without Tuneman (baseline). We say a service request is dropped if it cannot be provisioned in the network because of insufficient available bandwidth at all of the t-shortest paths. The bar chart in Figure 8 has 4 groups of bars corresponding to the network load of 0.4, 0.6, 0.8, and 1.0. For each group, we have 16 bars – eight each for Tuneman and baseline case. Within a group, for each case, the eight bars represent the percentage of services dropped for all eight considered priority classes. The first bar in each group denotes the percentage of requests dropped belonging to QoS level 7 (highest QoS) for the baseline case. The next adjacent bar corresponds to the Tuneman case for the same QoS level. Similarly, the last two bars in each group represent dropped requests belonging to the lowest priority class (QoS level 0) for the baseline and Tuneman case, respectively. At a low load of 0.4, even without Tuneman, the acceptance percentage is quite good and only $\sim 2\%$ requests are dropped across all priority classes. However, with Tuneman this rate even improves, and hardly any high-priority service is dropped. However, for the load of 1, the drop percentage is around $\sim 15\%$ for the baseline case across all priority classes. Tuneman significantly improves this percentage for the high-priority classes as only $\sim 5\%$ services are dropped. However, the improvement is not much for low-priority classes such as QoS level 0. This is because Tuneman's reconfiguration algorithm does not shift higher priority flows (on a longer path) to adjust a low priority flow. The minute improvement for the lowest priority services (QoS 0) comes from shifting other lowest priority services on a longer path.

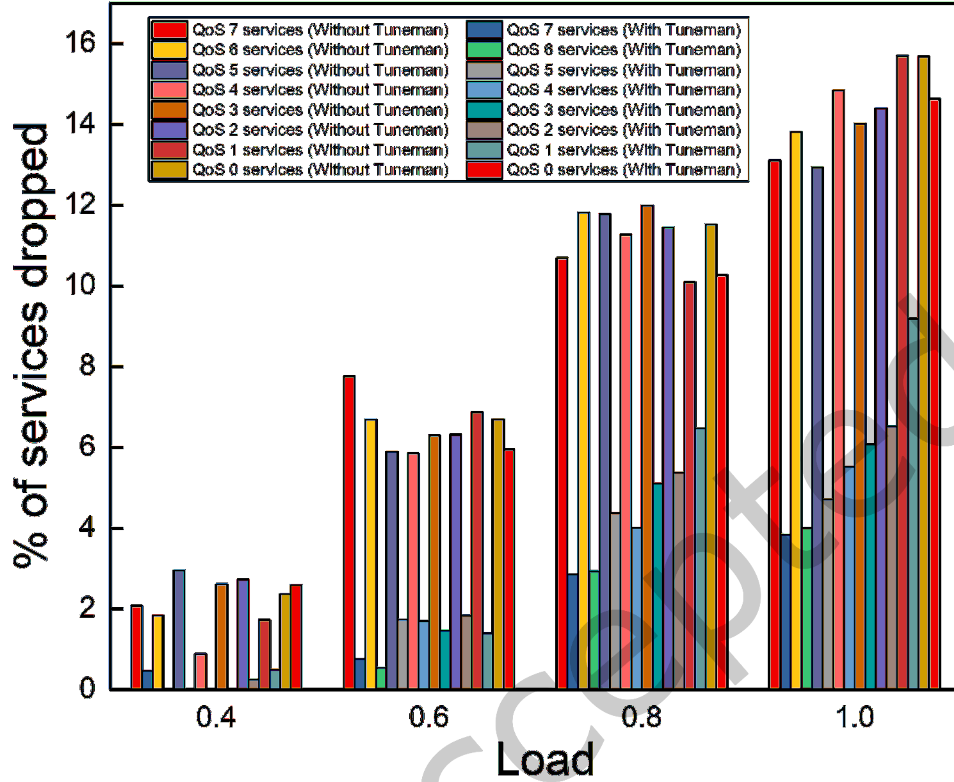


Fig. 8. Percentage of services dropped for each QoS level with and without Tuneman (at different network loads).

Just Accepted

6 SYSTEM EVALUATION

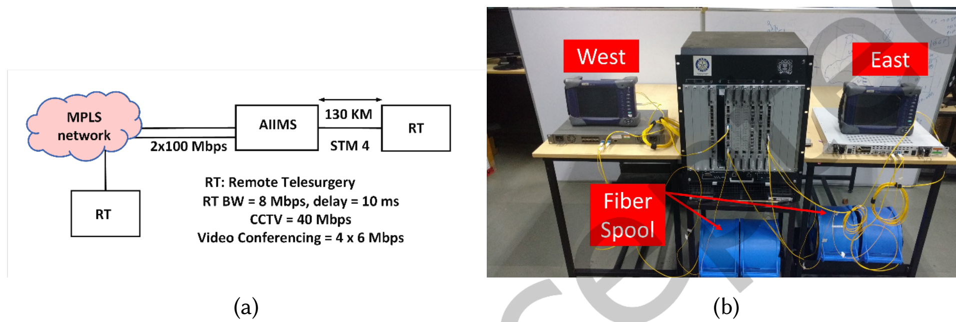
Initially, Tuneman discovers all nodes using layer 2 discovery protocol (similar to Cisco Discovery Protocol) via its immediately connected nodes (and then its neighbors and so on) and constructs an adjacency graph of the entire network. Thereafter each port at a node is given a unique number. Tuneman periodically pings all nodes to verify the adjacency matrix. We use a re-arrangeable and non-blocking VoQ crossbar. When a new request arrives, a schedule is computed based on §3. A path is then found that can accommodate a new request after which tables are populated.

Tuneman computes a *megalabel* by concatenating ingress and egress port numbers for all the routers along the request's path. The megalabel is a segment-routed information appended to every packet belongs to the request. An entry in the megalabel is 13-bits long: the first 6-bits correspond to an ingress port, while the last 6-bits correspond to an egress port. As a packet traverses a node, the node eradicates its active portion of the megalabel by converting the middle bit from a 0 to a 1 allowing downstream nodes to know which portion of the megalabel to work on next. As a flow enters a Tuneman-supported network, a table at the ingress node maps an application flow identifier to a megalabel, which is present in the table if the flow was provisioned.

The scheduler takes into account the small buffer sizes. In the FPGA we use, buffers are implemented using block RAMs that are not more than 300 microseconds or about 250 MTUs at 10 Gbps.

(a) (b)

Fig. 9. a) The edge and regional SDN routers. b) The core SDN router. Chassis (above left), PCB (above right), the switching card (bottom left), the IO card (bottom right).



(a) (b)

Fig. 10. a) AIIMS existing telesurgery set-up. b) Emulation of AIIMS telesurgery set-up.

Tuneman hardware: Three kinds of SDN whiteboxes were used in the testbed (shown in Figure 9a and 9b): an edge box that has 10×1 Gbps and 2×10 Gbps IOs (Input/Output), a regional platform, whose individual stackable element has 4×10 Gbps and 8×1 Gbps IOs and a core whitebox that supports 144×10 Gbps IOs. The edge box is built using a Virtex 6 FPGA (240T-1) and an AMCC PHY that multiplexes 3.125 Gbps IO lines from the FPGA into 10 Gbps IOs. The regional SDN platform uses a Virtex 6 365T FPGA along with 4 Pemaquid OTN capable ASICs. The larger core platform has 2 switching cards, each supporting multiple Virtex 7 690T FPGAs, and up to 12 IO cards, each supporting 12×10Gbps IOs with each IO card built around two parallel units of Virtex 7 FPGAs. The FPGA-based hardware interacts with a Java-based network management system (NMS) (i.e., Tuneman) through a control state machine. The Tuneman controller consists of a provisioning module, a telemetry module and a network discovery module.

6.1 Telesurgery Emulation Experiment

We emulated the topology and telesurgery setup of a leading hospital (AIIMS) in India on our testbed. Figure 10a shows that the hospital has 2 telesurgery (TS) locations, East and West of itself at a distance of 96 and 130 km respectively. The West location is currently connected through an MPLS cloud using 2x100 Mbps best-effort traffic, while the East location is connected using a SDH (Synchronous Digital Hierarchy) network of a STM-4 (622 Mbps) link. We emulated this set up in our lab with spools of fibers corresponding to the distances as shown in Figure 10b. We used all three routers we had designed – a terabit router at the hospital location, a mid-size router at the Eastern location and the smallest router at the Western location. The Tuneman controller was connected to the terabit router, through which it was also connected to the other two edge routers. All traffic

Just Accepted

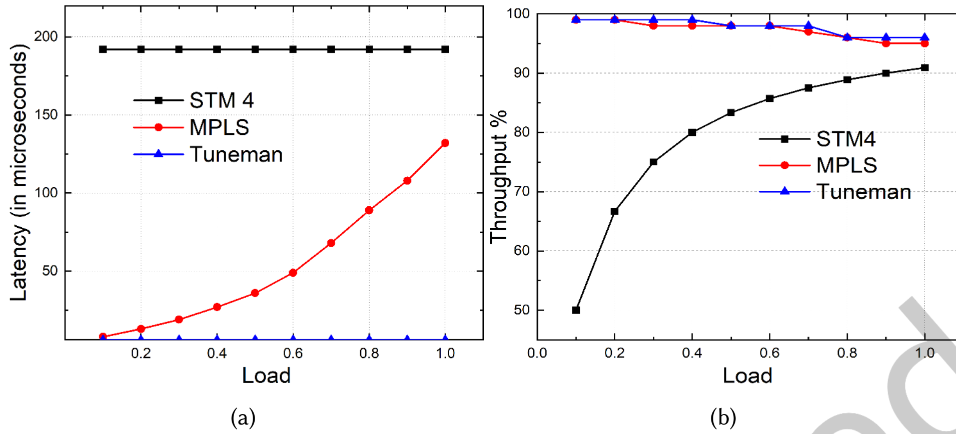


Fig. 11. a) Latency comparison of Tuneman vs. existing solutions. b) Throughput comparison of Tuneman vs. existing solutions.

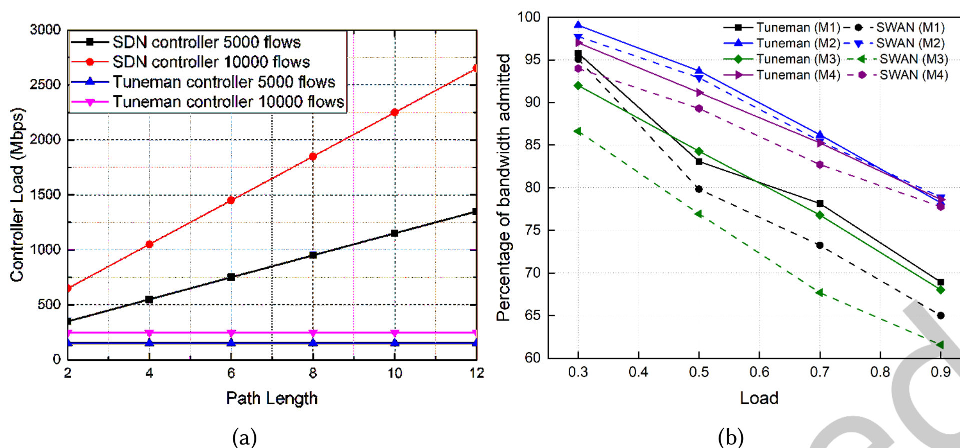
Just Accepted

was in integer multiples of 10 Mbps and over 10 Gbps links. The TS traffic was multiplexed with other traffic. Two traffic generators (JDSU8000) were used at the far ends to generate multiple traffic streams such that the network was loaded with not just TS, but other background traffic as well. Shown in Figures 11a, 11b are latency and throughput respectively as load varies.

We observe that the circuit switched STM-4 has the worst latency (6-hops with 32 microseconds at each hop), though it remains constant with load. MPLS latency grows with load, and hence will adversely impact Telesurgery (TS). Though MPLS latency is less than the strict cut-off for TS, it is variable, implying that there is perceivable change in user experience for TS. Deterministic latency is hence much more important for the application of TS than just meeting a bound as the data rate is intense (HD video) and precision in the form of two-way feedback is critical. Tuneman exhibits very low and deterministic latency, ideal for TS. Note that we have neglected the constant propagation delay for each case which would increase the user perceived delays by a few msec. Note that Tuneman is almost 1/4-th the price of STM-4 and yet gives much more BW (622Mbps vs 10Gbps on a link). The last graph shows the throughput comparison between STM-4, MPLS and Tuneman. MPLS and Tuneman have similar throughput In STM-4, the throughput is a direct function of network load. When viewed together, the *deterministic* latency and high throughput of Tuneman makes it a promising solution for TS (and other applications).

6.2 Controller Scalability

We compare the performance of the Tuneman controller with another SDN controller, OpenDaylight (ODL). To do so we created a linear topology of 14 nodes in Mininet. Mininet interacts with ODL, which writes the network configurations. We captured different sized packets exchanged between ODL with Mininet using Wireshark. We also capture the packets exchanged between Tuneman and the node in the network. Different sized packets are exchanged between controller and the nodes; therefore for a fair comparison, we compute the total number of bytes exchanged by varying the number of flows in the network as shown in Figure 12a. Note, to exhibit similar functionality as Tuneman, there is significant increase in load on the ODL controller because Tuneman uses segment routing which results in updates of match tables *only at the ingress node* but ODL has to update all the routers. Observe that as the number of flows increases, there is an increase in controller load for both Tuneman



(a) (b)

Fig. 12. a) Controller load comparison between Tuneman and a standard SDN controller. b) Percentage of requested bandwidth admitted by SWAN and Tuneman.

and ODL, but this increase is not as significant for Tuneman as in ODL; therefore, Tuneman controller scales better.

7 COMPARISON WITH OTHER SCHEMES

In this section, we compare the end-to-end latency and network utilization of Tuneman to SWAN (that only attempts to maximize utilization). For this reason, one might expect Tuneman to at most match the utilization of SWAN. This is a key comparison because as we said in the introduction, Tuneman functionally emulates SWAN but adds delay bound guarantees.

To examine scalability in the following experiments, we used simulations on a larger countrywide network, India's Tata Network rather than the smaller testbed that we used to emulate the AIIMS telesurgery setup. The Tata network is a WAN (TATA network [61]) comprising 145 nodes and 186 bidirectional fiber links. Links have 10Gbps capacity.

SWAN vs. Tuneman: Figure 12b, compares the percentage of requested bandwidth allocated by SWAN and Tuneman. We generated flows between random source-destination pairs. We assume four priority classes: (1) telesurgery and similar applications (latency bound of 5 ms [14, 58]), (2) video traffic (latency bound of 35 ms [52]), (3) voice traffic (latency bound of 100 ms [39]) and, (4) web-surfing traffic (latency bound of 200ms). Note that the latency bounds are in msec as opposed to μs earlier because we also consider realistic end-to-end propagation delays for mission-critical networks. The individual bandwidth requirement of the flows for these priority classes are as follows: 100 Mbps for telesurgery (multiple sessions), 5-10 Mbps for video flows, 1-5Mbps for voice flows and 1-20 Mbps for web-surfing. We have experimented with different traffic mixes of these four classes. The traffic mixes are as follows: M1 = [0.1, 0.1, 0.4, 0.4], M2 = [0.7, 0.1, 0.1, 0.1], M3 = [0.1, 0.2, 0.2, 0.5] and M4 = [0.3, 0.3, 0.2, 0.2]. An entry M_{ij} in the traffic mix M_i determines the fraction of traffic for priority class j to the total generated traffic. For SWAN's traffic allocation algorithm, we set U = 10 and alpha = 1.5 (see Figure 6 in [26]). SWAN focuses on inter-DC WANs, whereby there are multiple paths between a chosen source-destination pair. Hence to be fair to SWAN, we have only included flows between those source-destination pairs that have at least 10 paths between them.

Just Accepted

Table 2. Product Features

Feature	Cisco ASR	Juniper MX	Tuneman 1-Tbps
IPv4/IPv6 forwarding	Yes	Yes	Yes
MPLS TP/Carrier Ethernet	Yes	Yes	Yes
OpenFlow1.5	No	No	Yes
L2 switching	Yes	Yes	Yes
BGP + OSPF	Yes	Yes	Yes
802.1ag OAMP (50ms protection)	No	No	Yes
OTN	No	No	Yes
Bandwidth on demand	Yes	No	Yes
Deterministic latency	No	No	Yes

Just Accepted

Figure 12b shows that for traffic mixes M3 and M1, where the proportion of low priority flows are higher than the high priority flows, Tuneman provisions roughly 5% more bandwidth than SWAN. This is because SWAN always provisions high-priority flows on their shortest paths, irrespective of whether the latency guarantee is barely met or met by a large margin. By contrast, Tuneman provisions flows based on their latency tolerance: as long as the latency guarantee is met, the application can be provisioned on a longer path, thus creating room for application flows on otherwise congested links. Tuneman can provision high-priority flows on slightly longer paths, allowing Tuneman to allocate comparatively shorter paths to low priority flows, hence allowing more low priority flows while meeting delay bounds for every provisioned flow.

Even for traffic mixes where high-priority flows dominate (M2, M4), Tuneman performs slightly better in terms of bandwidth admission. More fundamentally, SWAN cannot guarantee delay bounds for applications like telesurgery. SWAN's priority mechanism is simply best effort, and can fail without the application being aware of the failure. *Thus, Tuneman's real advantage is better or comparable traffic utilization while guaranteeing delay bounds.* Without latency bounds, SWAN's traffic engineering algorithm always outperforms Tuneman for all traffic mixes as SWAN uses a network-level optimization model for bandwidth utilization while Tuneman relies on the (heuristic) DFP algorithm.

Our evaluation only compares Tuneman with SWAN. Fastpass is host based and uses edge coloring (which in turn requires reconfigurably non-blocking networks) and so cannot directly be compared to Tuneman which works in mesh networks. We also do not compare with MPLS TE because it is well known that centralized schemes like SWAN are better than distributed schemes [26]. Shown in Table 2 is a comparison of our developed Tuneman hardware to commercial grade SDN/IP routers.

8 RELATED WORK

Table 3 summarizes Tuneman versus related work. Most earlier work attempts *either* to provide delay bounds or do traffic engineering to improve utilization. Notable exceptions are Silo [29] (which improves utilization by VM placement and hence is not applicable to mission-critical networks) and [30] (which uses SDN switches and

Table 3. Related works

Scheme	Delay Bounds	BW Utilization	Throughput Bounds	Low cost switches	Applicable to Mesh Networks?
TE solutions [26, 27, 44]	No	Good	Yes	No	Yes
SDN based Host schedule [41])	Yes	No	No	No	No[1]
Delay Solutions [2, 22, 25]	Yes	No	No	No	Yes
Delay Calculus Based + Pacing [29]	Yes	Yes[2]	Yes	No	No
Optimization + WFQ switches [30]	Yes	Yes	Yes	No[3]	Yes
Tuneman: SDN control of routes and schedules	Yes	Yes	Yes	Yes	Yes

[1]reconfigurably non-blocking topologies only, [2]by VM Placement, [3]SDN switches

WFQ approximations to provide coarse delay bounds). Also, there are a few earlier works based on QoS routing [9, 53] considered both bandwidth and delay criteria to provision services in ad-hoc and multimedia networks. However, the latency requirements of these services are often in the range of 100s of milliseconds as opposed to the strict requirements in mission-critical networks. These works compute a set of paths that satisfies both delay and bandwidth criteria and thereafter provisions a request on a link with either maximum bandwidth or minimum delay. However, such a strategy leads to congestion on the shortest path, eventually forcing a drop of service requests with stringent bandwidth and delay requirements.

Traffic Engineering: Multiple traffic engineering proposals exist for maximizing bandwidth utilization in WANs and DCs. The goal of B4 [27] and SWAN [26] is to utilize links effectively by changing data-plane configurations based on changing flow bandwidth requirements. The optimization model does not consider latency but does consider bandwidth changes and flow priority classes. SWAN focuses on avoiding congestion while updating the rules in the switches and enforces max-min fairness within a class of traffic. SWAN does not consider switch-level scheduling of flows and hence does not guarantee delay bounds. A model that maximizes bandwidth allocation while respecting availability is presented in [20]. Edge Fabric [44] avoids link congestion at the edge in Facebook's network.

Delay Bounds: HULL [2] uses phantom queues to leave bandwidth headroom to support latency-sensitive flows. QJUMP [22] uses priority queuing along with rate-limiting to handle throughput-latency trade-off. pFabric [3] uses a remaining flow size metric for scheduling packets in DC switches to optimize flow completion time. PDQ [25] uses preemption to quickly finish data-center flows. These approaches are fundamentally different from Tuneman as they are primarily focused on low-latency communication within a DC and do not perform traffic engineering.

SDN Control of Host Packet Sending: FastPass [41] uses an SDN controller to allocate packet sending time-slots at hosts to minimize packet queues and latency at switches. FastPass does not simultaneously optimize flow bandwidth and latency as Tuneman does. Fastpass also does not do traffic engineering to maximize utilization, and requires reconfigurably non-blocking topologies used in data-centers [41] for its edge coloring algorithm.

Delay and Throughput Bounds with High Utilization: Kumar [30] propose a delay and bandwidth guarantee model for SDNs. However, our work differs significantly because [30] provides delay guarantees in commodity SDN switches using WFQ approximations and one queue per flow which is expensive and infeasible (as they themselves note). They also provide only coarse node delay guarantees. By contrast, we build our own slotted SDN switches, which allow us to scalably and cheaply control node delays at fine granularities, which in turn allows us to perform node-level bandwidth and delay optimizations. Thus the algorithm proposed by [30] is best-effort, while we guarantee delay bounds. Silo [29] leverages VM placements and pacing to guarantee latency bounds, bandwidth guarantees, and bandwidth utilization in a multi-tenant DC. By contrast, Tuneman provides bandwidth and latency for each application in a WAN environment.

Just Accepted

Router Scheduling: Tuneman's node scheduling optimization module is comparable to a fast scheduler like iSLIP [35] but works at a coarser time scale. While iSLIP is not cast as an optimization, it does provide fast work-conserving scheduling. Native iSLIP, however, does not consider latency bounds, which Tuneman does. Tuneman is computationally slower than iSLIP but provides latency guarantees for Tuneman flows. Similarly, variants of DRR [46] provide poor delay bounds. The celebrated Parekh-Gallager [40] delay bound and Cruz's network calculus [15] are techniques to ensure and calculate fairly coarse delay bounds in a multi-hop network. By contrast, our focus is on designing and controlling the nodes of a network so that flow delay are explicitly met. Our analysis does not require the sophistication of the delay calculus because of the simple slotted schedulers we employ at nodes. Recent Switch schedulers include hardware schedulers [34, 37, 47, 49, 57] and software schedulers [42, 43, 49]. Tuneman's slotted scheduler is different and more closely related to techniques such as [1, 11, 28]. DRILL [21] does scheduling and load balancing.

Other approaches: Cameo [54] is an approach to meet deadlines while ensuring performance isolation in streaming systems. Homa [38] is a transport protocol for small messages in low-latency DCs, that assigns priority dynamically at a receiver and over-commits receiver's links to increase BW utilization. Low-latency routing is considered by [23] by dimensioning the physical topology. [32] is a measurement module for software switches. We have a similar module that provides latency and throughput. Tuneman can be implemented using other data-plane programmability techniques [8, 12, 48, 60].

9 DISCUSSION

Before we conclude, we discuss the limitations, the impact of rescheduling flows on TCP and the scheduling complexity of Tuneman.

Limitations: We identified some limitations in our work. The first limitation of our work is that we assume the applications specify their bandwidth and latency requirements at the time of provisioning. Applications can spoof their QoS and bandwidth requirements. In the present version of Tuneman, we expect applications to convey their QoS and bandwidth requirements correctly. We assume to have a mechanism that can convey QoS and bandwidth requirements (based on the application) to Tuneman. Application Aware Routing is one such alternative as presented in [16, 31, 36, 51, 59], which aims to identify the applications and perform routing based on application's requirement. This may require a standardization process for applications such that they can be categorized into classes and a specific field is allocated in a packet for the class IDs. Another limitation is that Tuneman equally distributes the target delay for a service among all switches along the chosen path. This assumption simplifies the problem as it does not consider the existing load at a switch while distributing the delay budget. In the future, we aim to rectify these limitations by introducing mechanisms that consider application awareness and individual switch's load.

Impact of flow rescheduling on TCP: Flow scheduling and reconfiguration impacts the TCP flows and their windowing mechanism. This impact is prominent when reconfiguration of flows is done following the break before make philosophy, where new configurations for flows are populated in the match table after tearing down the existing flow configurations. In contrast, Tuneman follows make before break philosophy, where new configurations are populated first before tearing down the existing configurations. Therefore impact on TCP flows and throughput is negligible. However, this reconfiguration impacts the TCP window sizing. As with reconfiguration, paths of the flows become longer, TCP attempts to increase its window size to sustain the throughput.

Scheduling complexity: As per our analysis, packet scheduling in Tuneman can be directly correlated to the scheduling presented in [13, 46]. For a round robin scheduling work required to process a packet has complexity of O(1) as presented in [13, 46]. Therefore, the work required to process a packet in Tuneman is also O(1), as Tuneman also uses round robin to schedule the packets from input ports to output ports. In Tuneman, an epoch

Just Accepted

is divided into smaller time slots for scheduling the packets from all incoming ports; this slot time corresponds to a "quanta" in [46] or "weight" in [13].

10 CONCLUSION

This paper explores the consequences of *programmability* of a network, where routes, switch schedules, QoS priorities are all at the command of a centralized controller. Earlier work (e.g., SWAN, B4, FastPass) assumed the network was partially programmable. B4/SWAN assumes routes can be dynamically selected but not switch schedules, while FastPass assumes routes and packet schedules at end-nodes can be programmed but not switches. In existing routers, both routes and QoS parameters can only be coarsely tuned using link-weights and queue parameters.

We present a path towards full programmability using packet schedules and dynamic routing. Our results show that SWAN's coarse latency knobs (priority levels) can actually decrease utilization. Full programmability also paradoxically simplifies matters compared to having a controller restrained with existing switches and limited QoS knobs. Tuneman thus can be used in mission-critical networks with increasing demands for stringent latency bounds (e.g., telesurgery).

Just Accepted

REFERENCES

[1] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. 2018. Sincronia: near-optimal network design for coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 16–29.

[2] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 253–266.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 435–446.

[4] Aharon Ben-Tal, Laurent El Ghaoui, and Arkadi Nemirovski. 2009. *Robust optimization*. Vol. 28. Princeton University Press.

[5] Jon CR Bennett and Hui Zhang. 1996. WF/sup 2/Q: worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM'96. Conference on Computer Communications*, Vol. 1. IEEE, 120–128.

[6] Judith O Berkey and Pearl Y Wang. 1987. Two-dimensional finite bin-packing algorithms. *Journal of the operational research society* 38, 5 (1987), 423–429.

[7] Yoram Bernet, Peter Ford, Raj Yavatkar, Fred Baker, Lixia Zhang, Michael Speer, Robert Braden, Bruce Davie, John Wroclawski, and Eyal Felstaine. 2000. A framework for integrated services operation over diffserv networks. *RFC2998, November* (2000).

[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[9] Jing Cao and Wei Wu. 2008. A Multi-metric QoS Routing Method for Ad Hoc Network. In *2008 The 4th International Conference on Mobile Ad-hoc and Sensor Networks*. 99–102. https://doi.org/10.1109/MSN.2008.25

[10] Shigang Chen and Klara Nahrstedt. 1998. On finding multi-constrained paths. In *ICC'98. 1998 IEEE International Conference on Communications. Conference Record. Affiliated with SUPERCOMM'98 (Cat. No. 98CH36220)*, Vol. 2. IEEE, 874–879.

[11] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 239–252.

[12] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. 2017. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 1–14.

[13] Guo Chuanxiong. 2001. SRR: An O (1) time complexity packet scheduler for flows in multi-service packet networks. *ACM SIGCOMM Computer Communication Review* 31, 4 (2001), 211–222.

[14] Burak Cizmeci, Xiao Xu, Rahul Chaudhari, Christoph Bachhuber, Nicolas Alt, and Eckehard Steinbach. 2017. A multiplexing scheme for multimodal teleoperation. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 13, 2 (2017), 1–28.

[15] Rene L Cruz. 1991. A calculus for network delay. II. Network analysis. *IEEE Transactions on information theory* 37, 1 (1991), 132–141.

[16] Guo-Cin Deng and Kuochen Wang. 2018. An application-aware QoS routing algorithm for SDN-based IoT networking. In *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 00186–00191.

[17] Yoann Desmouceaux, Pierre Pfister, Jérôme Tollet, Mark Townsley, and Thomas Clausen. 2018. 6lb: Scalable and application-aware load balancing with segment routing. *IEEE/ACM Transactions on Networking* 26, 2 (2018), 819–834.

[18] Homa Eghbali and Vincent WS Wong. 2015. Bandwidth allocation and pricing for SDN-enabled home networks. In *2015 IEEE international conference on communications (ICC)*. IEEE, 5342–5347.

[19] Peixuan Gao, Anthony Dalleggio, Yang Xu, and H Jonathan Chao. 2022. Gearbox: A Hierarchical Packet Scheduler for Approximate Weighted Fair Queuing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 551–565.

[20] Jeremy Bogle Nikhil Bhatia Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. 2019. TeaVaR: Striking the Right Utilization-Availability Balance in WAN Traffic Engineering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM.

[21] Soudeh Ghorbani, Zibin Yang, P Godfrey, Yashar Ganjali, and Amin Firoozshahian. 2017. DRILL: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 225–238.

[22] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. 2015. Queues Don't Matter When You Can {JUMP} Them!. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 1–14.

[23] Nikola Gvozdiev, Stefano Vissicchio, Brad Karp, and Mark Handley. 2018. On low-latency-capable topologies, and their impact on the design of intra-domain routing. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 88–102.

[24] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. 2015. A declarative and expressive approach to control forwarding paths in carrier-grade networks. *ACM SIGCOMM computer communication review* 45, 4 (2015), 15–28.

[25] Chi-Yao Hong, Matthew Caesar, and P Brighten Godfrey. 2012. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 127–138.

[26] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 15–26.

[27] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 3–14.

[28] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. 2015. Network-aware scheduling for data-parallel jobs: Plan when you can. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 407–420.

[29] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. 2015. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 435–448.

[30] Rakesh Kumar, Monowar Hasan, Smruti Padhy, Konstantin Evchenko, Lavanya Piramanayagam, Sibin Mohan, and Rakesh B Bobba. 2017. End-to-end network delay guarantees for real-time systems using sdn. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 231–242.

[31] Aniruddha Kushwaha, Naveen Bazard, and Ashwin Gumaste. 2021. IPv6 Flow-Label based Application Aware Routing in SDNs. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 1–6.

[32] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*. ACM, 334–350.

[33] Phumzile Malindi. 2011. QoS in telemedicine. *Telemedicine Techniques and Applications* (2011), 119–138.

[34] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2018. Learning scheduling algorithms for data processing clusters. *arXiv preprint arXiv:1810.01963* (2018).

[35] Nick McKeown. 1999. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM transactions on networking* 2 (1999), 188–201.

[36] Hesham Mekky, Fang Hao, Sarit Mukherjee, Zhi-Li Zhang, and TV Lakshman. 2014. Application-aware data plane processing in SDN. In *Proceedings of the third workshop on Hot topics in software defined networking*. 13–18.

[37] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Universal packet scheduling. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 501–521.

[38] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 221–235.

[39] Songun Na and Seungwha Yoo. 2002. Allowable propagation delay for VoIP calls of acceptable quality. In *International Workshop on Advanced Internet Services and Applications*. Springer, 47–55.

[40] Abhay K Parekh and Robert G Gallager. 1994. A generalized processor sharing approach to flow control in integrated services networks: The multiple node case. *IEEE/ACM transactions on networking* 2, 2 (1994), 137–150.

Just Accepted

[41] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2015. Fastpass: A centralized zero-queue datacenter network. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 307–318.
[42] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Carlo Contavalli, Amin Vahdat, et al. 2017. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 404–417.
[43] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen W Zegura, Mostafa H Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and Flexible Software Packet Scheduling.. In *NSDI*. 17–32.
[44] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. 2017. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 418–431.
[45] Scott Shenker, Martin Casado, Teemu Koponen, Nick McKeown, et al. 2011. The future of networking, and the past of protocols. *Open Networking Summit* 20 (2011), 1–30.
[46] Madhavapeddi Shreedhar and George Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking* 3 (1996), 375–385.
[47] Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*. ACM, 367–379.
[48] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 15–28.
[49] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 44–57.
[50] Ted H Szymanski and Dave Gilbert. 2010. Provisioning mission-critical telerobotic control systems over internet backbone networks with essentially-perfect QoS. *IEEE Journal on selected areas in communications* 28, 5 (2010), 630–643.
[51] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. 2019. Is advance knowledge of flow sizes a plausible assumption?. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 565–580.
[52] IT UNION. 2001. ITU-T G. 1010 Quality of service and performance. *Switzerland: Geneva* (2001).
[53] Zheng Wang and Jon Crowcroft. 1996. Quality-of-service routing for supporting multimedia applications. *IEEE Journal on selected areas in communications* 14, 7 (1996), 1228–1234.
[54] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. 2021. Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 389–405.
[55] Shujie Yang, Changqiao Xu, Lujie Zhong, Jiahao Shen, and Gabriel-Miro Muntean. 2019. A QoE-driven multicast strategy with segment routing—A novel multimedia traffic engineering paradigm. *IEEE Transactions on Broadcasting* 66, 1 (2019), 34–46.
[56] Jin Y Yen. 1970. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quart. Appl. Math.* 27, 4 (1970), 526–530.
[57] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. 2016. CODA: Toward automatically identifying and scheduling coflows in the dark. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 160–173.
[58] Qi Zhang, Jianhui Liu, and Guodong Zhao. 2018. Towards 5G enabled tactile robotic telesurgery. *arXiv preprint arXiv:1803.03586* (2018).
[59] Shuai Zhao, Ali Sydney, and Deep Medhi. 2016. Building application-aware network environments using SDN for optimizing Hadoop applications. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 583–584.
[60] Noa Zilberman, Gabi Bracha, and Golan Schzukin. 2019. Stardust: Divide and conquer in the data center network. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 141–160.
[61] Topology Zoo. 2011. TATA. *Available http://www.topology-zoo.org/maps/TataNld.jpg. Last accessed 28 Jan 2020* (2011).

Just Accepted

APPENDIX

A ROBUST OPTIMIZATION

Robustness in optimization is incorporated when we consider some connections to be at a peak value, while others are at a fixed ("average") value. However, we do not know which of the connections are going to be at their peak values. Incorporating robustness allows a statically designed system to support some degree of traffic dynamism.

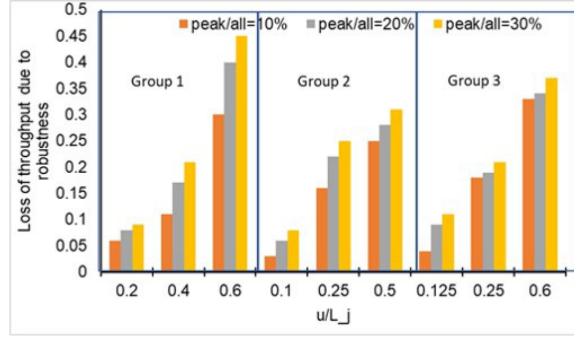


Fig. 13. Loss of throughput due to robustness

We say that of all the flows at node k for each input port i.e. $\forall i, \Sigma_{\forall j} A_{ij}$, we allow any u of the connections to be at peak value, which is T^m_{ab} (peak). The other connections are at a value T^m_{ab}, which is their static or average value. The corresponding robust optimization formulation requires the following additional constraints:

We define, $\forall \gamma^{mkij}_{ab} = 1$, L^j_k as the number of connections assigned to node k, which will exit port j. Then the capacity constraint under robustness changes to:

$$\gamma^{mkij}_{ab}.T^m_{ab}(peak).\epsilon^k_j \leq P_{C_j}.[A_{ij}]$$

And for other $L^j_k - u$ connections:

$$\gamma^{mkij}_{ab}.T^m_{ab}.\epsilon^k_j \leq P_{C_j}.[A_{ij}]$$

The buffer occupancy and rate constraint change for $\underbrace{max}_{u} T^m_{ab}$ to:

$$\gamma^{mkij}_{ab}.T^m_{ab}(peak).\epsilon^k_j \leq A_{ij}.P_{r_c} \quad (Rate\ constraint)$$

$$\gamma^{mkij}_{ab}.T^m_{ab}(peak).\epsilon^k_j \leq B_{max} \quad (Buffer\ constraint)$$

Similarly, the buffer occupancy and rate constraints for the other $L^j_k - u$ connections (denoted by $\underbrace{min}_{L^j_k - u} T^m_{ab}$) at a port are given by:

$$\gamma^{mkij}_{ab}.T^m_{ab}.\epsilon^k_j \leq A_{ij}.P_{r_c} \quad (Rate\ constraint)$$

$$\gamma^{mkij}_{ab}.T^m_{ab}.\epsilon^k_j \leq B_{max} \quad (Buffer\ constraint)$$

Figure 13 measures the impact of robustness. For this experiment, we use $50 \leq L^k_j \leq 200$; $10 \leq u \leq 120$; On the x-axis, we have the ratio of the number of flows at peak-value to the total number of flows at the output port. We begin from 0.1 and go up to 0.6. We create 3 groups with 3 bars in each group: the first 3 bars are for small number of flows at the output port, the next three are for medium number of flows at an output port, while the last three are for a heavy loaded output port. In each group, we further have three bar-graphs for different ratios of peak-to-average bandwidth values.

The result can be interpreted as follows: Each bar has two indicators – the peak-to-average bandwidth value and the number of flows at peak to total number of flows at the port. We measure for these two indicators the loss of throughput (defined as how much extra padding in terms of A_{ij} we need to reserve to take into consideration

Just Accepted

robustness). The 3rd, 6th, 9th bars i.e. at $u/L_i > 0.5$ are difficult to realize as the loss in throughput is high since we incorporate almost 50% churn.

The remaining results show the merit of the robustness model – the loss of throughput is less than 20% for even relatively high churn of 40% of the connections varying their bandwidth by 10-20% from their steady-state value (we assume both increase and decrease from the steady-state value). A plausible reason for this seemingly lower loss of throughput is because of better fitting of schedules in the ϵ^k. What is happening is that with churn, there is some interchange between slots (and hence a new schedule), though the epoch time is hardly impacted.

Just Accepted