Network Entitlement: Contract-based Network Sharing with Agility and SLO Guarantees

Satyajeet Singh Ahuja, Vinayak Dangui, Kirtesh Patil, Manikandan Somasundaram, Varun Gupta, Mario Sanchez, Guanqing Yan, Max Noormohammadpour, Alaleh Razmjoo, Grace Smith, Hao Zhong, Abhinav Triguna, Soshant Bali, Yuxiang Xiang, Yilun Chen, Prabhakaran Ganesan, Mikel Jimenez Fernandez, Petr Lapukhov, Guyue Liu*, Ying Zhang
Meta Platforms, Inc. *New York University Shanghai

ABSTRACT

This paper presents Meta's Production Wide Area Network (WAN) Entitlement solution used by thousands of Meta's services to share the network safely and efficiently. We first introduce the Network Entitlement problem, i.e., how to share WAN bandwidth across services with flexibility and SLO guarantees. We present a new abstraction *entitlement contract*, which is *stable, simple, and operationally friendly*. The contract defines services' network quota and is set up between the network team and services teams to govern their obligations. Our framework includes two key parts: (1) an *entitlement granting system* that establishes an agile contract while achieving network efficiency and meeting long-term SLO guarantees, and (2) a large-scale distributed *run-time enforcement system* that enforces the contract on the production traffic. We demonstrate its effectiveness through extensive simulations and real-world end-to-end tests. The system has been deployed and operated for over two years in production. We hope that our years of experience provide a new angle to viewing WAN network sharing in production and will inspire follow-up research.

CCS CONCEPTS

• **Networks** → **Network management**; **Wide area networks**; *Network reliability*; *Network monitoring*;

KEYWORDS

Wide-area networks, Bandwidth sharing, Network isolation

ACM Reference Format:
Satyajeet Singh Ahuja, Vinayak Dangui, Kirtesh Patil, Manikandan Somasundaram, Varun Gupta, Mario Sanchez, Guanqing Yan, Max Noormohammadpour, Alaleh Razmjoo, Grace Smith, Hao Zhong, Abhinav Triguna, Soshant Bali, Yuxiang Xiang, Yilun Chen, Prabhakaran Ganesan, Mikel Jimenez Fernandez, Petr Lapukhov, Guyue Liu*, Ying Zhang. 2022. Network Entitlement: Contract-based Network Sharing with Agility and SLO Guarantees. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3544216.3544245

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544245

1 INTRODUCTION

Meta's backbone network interconnect Point-of-Presence (PoP) sites and Data Centers (DCs), and are shared by all of Meta's services (e.g., storage, logging, AI, Ads, news feed, market place). In the past five years, the number of services supported by WAN infrastructure has increased by multiple orders of magnitude and the traffic volume has grown from a O(10)Tbps to O(100)Tbps. During this explosive growth period, one fundamental problem we have faced is *how to enable multiple services to share the backbone networks efficiently and safely?*

It is practically and economically infeasible to build capacity at the rate of increasing demand. We have to find ways to efficiently use the finite network capacity. Since network is a finite shared resource, we also had to face the reality of disruptions caused by misbehaving services (intentionally or unintentionally) e.g., a new service feature or a software bug. Even worse, service disruptions also raised accountability issues between the network team and service teams. It was hard to attribute the disruption to misuse of the network, or poor network management.

Insufficient capacity, increasing misbehaving services, and unclear operational accountability *called for* a new solution to effectively plan, manage, and operate the shared network resource. In this paper we present *Network Entitlement*, our solution to the above challenges.

Network Entitlement is a network resource reservation framework provided by the network team for all service teams. It aims to provide a *simple, stable, and operations friendly* abstraction for sharing our backbone networks. That is, *a service is guaranteed certain amount X of bandwidth with Y SLO guarantee for a given period of time*. Note that it has a significant departure from previous solutions, e.g., Google's BwE [12], which models bandwidth sharing as a real-time traffic engineering problem. Our work operates at a different time scale and provides a complimentary angle to improve WAN efficiency.

Our work also has noticeable differences from prior work on data center-based network sharing [2, 5, 7, 11, 18, 21, 25], given the three unique challenges rising from both WAN characteristics and business model. First, because our services are *internal*, the business model is fundamentally different from a cloud provider environment. The framework needs to optimize for both *network efficiency* and *service agility*. These two goals often conflict with each other (more details in §4.2). Second, most of our services are long-running network customers instead of short-term tenants, which requires us to provide *Service Level Objective (SLO) guarantees* for long time periods. We define different availability SLOs for each

class of service to standardize the network performance expectation from services[1, 24]. The availability SLO measures the uptime percentage per class of service, where uptime requires all traffic in that class of service to be admitted in the network. Such SLO guarantees are difficult to provide for WAN, sometimes they are even *infeasible* to achieve.

To address these challenges, we examine the WAN network sharing from a contract-based angle, and our framework has made the following contributions:

(1) An agile and efficient entitlement contract abstraction: We propose an *entitlement contract abstraction* which standardizes the service's network demand and SLO guarantees for a certain period. The contract is used throughout the entitlement process and provides clear accountability between the network team and service owners. The representation of the contract is based on a Hose model [6], which is used for long-term network planning [1]. A general hose model can achieve agility but is not capacity efficient. Our insight is that entitlement is a much shorter period (e.g., 3 months), thus has more predicable traffic patterns. Based on this observation, we propose a new *segmented-Hose* algorithm which incorporates service deployments to reduce uncertainty by 60%.

(2) A dynamic SLO-based entitlement granting system: To provide long-term SLO guarantees for a service, our granting system analyzes possible network failures (e.g., fiber cuts) and changes (e.g., new links) in advance. By synthesizing the service demand, potential network failures, and available capacity, the system dynamically sets reachable SLO targets for the service. While this process achieves the desired SLO guarantees, the process is computationally intensive and thus not feasible to be applied to every service. To address this issue, we identify a relatively small number (˜10) of consumers of the network that account for the majority of network usage. We call them *high-touch services* and the rest of the services are grouped into one *low-touch service*. The granting system sets an entitlement for each high-touch service and for the low-touch services on aggregate, which significantly reduces operation and computation overheads.

(3) A large-scale distributed run-time enforcement system: To enforce the entitlement contract on production traffic, we build a large-scale distributed run-time enforcement system on our end hosts (servers). Different from prior work [12, 21], our operational experiences provide two key insights: a) Traditional pure endhost-based approaches (e.g., through rate-limiting) often make immature decisions and affect network utilization; instead, our endhosts only *classify and mark* packets into different classes, and leave *final decisions* (e.g., drop or transmit) to hardware switches. b) Given two common flow-based and host-based marking approaches, our experiences show that the *host-based* approach can leverage applications' builtin resiliency mechanisms, thus achieving better performance. Moreover, it provides easier troubleshooting and better interpretability to service teams.

(4) Extensive evaluations including real-world end-to-end test drill: We demonstrate the effectiveness of our framework not only through extensive simulations but also using a real-world end-to-end test drill. With careful control on services, we introduce real congestion and observe the success of service isolation and bandwidth guarantees.

Our system has been deployed in Meta for over two years on O(100Tbps) traffic from O(100k) endhosts. To the best of our knowledge, we are the first to introduce the entire WAN entitlement process, related practical concerns, and our production solution to academia. It provides a new angle to WAN sharing at a longer time scale with a simpler and more stable interface. We hope our years of operational experience can provide a fresh contract-based angle to viewing WAN network sharing problem and inspire a new line of research. This work does not raise any ethical issues.

2 BACKGROUND AND MOTIVATION

In this section, we first provide the background on Meta's services which share the network infrastructure, and then use example incidents to show why launching the network entitlement program was urgently needed.

2.1 Meta's Service Ontology

Our network supports *thousands of* diverse applications, including both consumer-facing business apps (e.g., Ads) and internal support services (e.g., Storage). According to the business priorities and traffic types, we broadly classify traffic into a few QoS classes. Figure 1 and Figure 2 show the traffic distribution from two QoS classes.

Diverse service types: By comparing these two figures, we can see 1) the distribution of traffic *varies* significantly across QoS classes. Each QoS has a few dominating services (<10) that account for the majority of network usage, and thousands of other services that use a small fraction of capacity. 2) Traffic from one service can belong to *more than* one traffic class. For example, the majority of Warmstorage's data traffic is in Class B, but a small amount of its control traffic is in Class A. 3) Most dominating services are related to storage, e.g., Logging [10], Warmstorage [17], Coldstorage [3], Datawarehouse [22], MultiFeed [27], and Everstore (a distributed key-value store) [16]. Although storage being a top network consumer is known, we further show the variety of storage services exist in a typical Internet company. It implies a need for finer-grained network support even for storage.

Distinct traffic patterns: Besides differences in volume and traffic classes, at the micro level, services exhibit distinct traffic patterns even for a similar type of service. For example, Figure 3 shows the traffic patterns of Coldstorage service (top) and Warmstorage service (bottom) over a typical time series. Clearly, Coldstorage has regular spikes while Warmstorage has a smoother pattern. This difference is due to Coldstorage *periodically* turning on a rack of storage servers to perform data operations and rotating across all racks, to achieve more efficient power usage. In contrast, the fluctuation of Warmstorage is a consequence of the time-of-day effect. The large number of diverse services, with different priorities and distinct traffic patterns, present great challenges to sharing the network efficiently and safely.

2.2 Misbehaving Services Cause Disruptions

The services' network demand is growing at a much faster pace, roughly 30% faster than our capacity to build the network, which is fundamentally limited by the increasingly scarce fiber resources. As a consequence, there is an increase in service disruptions caused by misbehaving services. Here we show two example incidents.

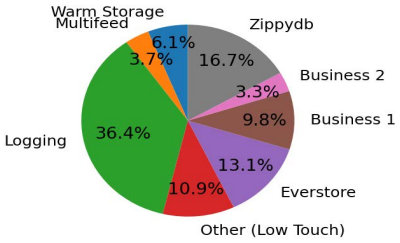

Figure 1: Service distribution of a high QoS Class

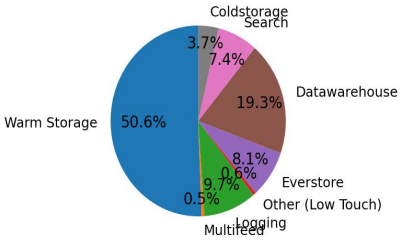

Figure 2: Service distribution of a low QoS Class

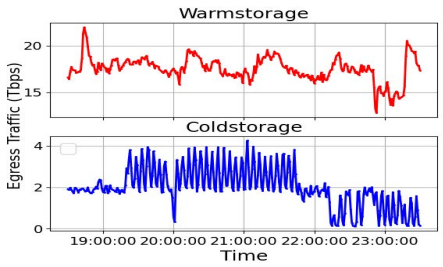

Figure 3: Two storage services with distinct patterns.

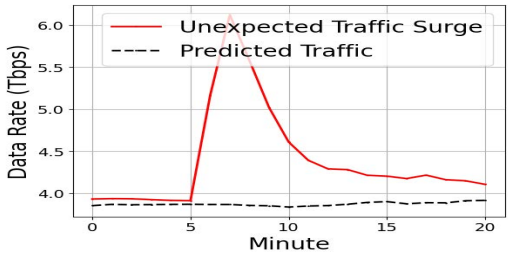

Figure 4: Misbehaving services: a service bug

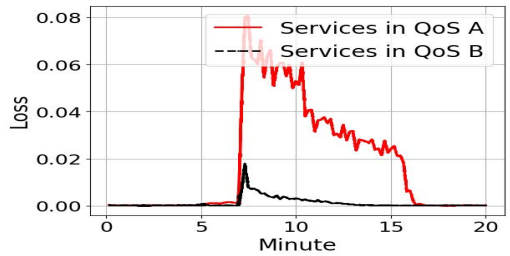

Figure 5: Loss induced to services in two QoS classes

Incident 1: Service Bug. The first incident was caused by a bug in a latest version of a video client code. The bug mistakenly downloads multiple duplicate videos in parallel. When the new version was released, a traffic spike was created as the code was deployed on a large enough number of client devices. As shown in Figure 4, this spike was formed within three minutes, and the peak volume was 50% more than predicated volume. This misbehaving service quickly caused WAN resource contention, generating noticeable loss for other services. As shown in Figure 5, all traffic of two QoS classes' that the service belong to were impacted, up to 8% loss for traffic in Class A and 2% loss for Class B was observed. Note that the figure shows a network-wide total loss, instead of just on the bottleneck links. Previously, we had deployed QoS isolation mechanisms to protect traffic across different classes, but this alone cannot safeguard well-behaved services from misbehaving ones within the same class or lower classes. This creates challenges in accountability in determining who is responsible for the incident on well-behaving services.

Incident 2: New Feature. Another incident happened when one service changed its caching strategy to add a new feature. Instead of fetching content from the cache servers at the edge, the new feature fetches content from backend servers in data centers. As soon as the feature was deployed, we observed an *unexpected surge* of backbone traffic from one region. This surge was 10% larger than the estimated *peak* volume, causing loss spikes to other services. To prevent this types of incidents caused by planned changes, the service teams and the network team need to plan the network resources *jointly*, otherwise it is hard to attribute responsibility.

These are only two representative incidents of many service disruptions caused by misbehaving services, intentionally or unintentionally. A new solution to effectively plan, manage, and operate the shared network resource was needed.

3 WAN NETWORK ENTITLEMENT

Network Entitlement aims to provide a solution to the above issues by building a *network resource reservation* framework. The framework manages a complex process involving both the network and service teams, starting from a) *service demand forecast* that estimates each service's bandwidth usage to b) *bandwidth granting* that grants maximum bandwidth for the service based on network capacity; and finally c) *runtime enforcement* that enforces granted bandwidth on the production traffic at run-time. Designing such an entitlement framework presents three outstanding challenges we discuss next.

3.1 Key Challenges

Challenge 1: Achieve Network Efficiency and Service Agility. As an *internal* framework, it needs to optimize for both *network efficiency* and *service agility*. On one side, the network team needs to adjust bandwidth allocation and make sure the network is efficiently used, e.g., high bandwidth utilization. On the other side, services need to make their business-specific decisions flexibly and independent of the network team, e.g., moving traffic across regions, changing communication patterns, or reallocating compute resources. Unfortunately, these two objectives often conflict with each other. High network utilization requires accurate traffic patterns, whereas agile traffic movement requires over-provisioned network resources to accommodate the uncertainty. Having a single team make all related decisions (e.g., server deployment, traffic allocation) is operationally infeasible.

Challenge 2: Meet long-term SLO guarantees. Most Meta services are long-running services, thus the framework needs to provide expected SLO guarantees for a long period (e.g., 3 months). Providing such guarantees cannot just rely on the current bandwidth usage, but needs to consider possible network changes and failures in advance. Prior work on data center network sharing [2, 5, 7, 11, 18, 21] cannot achieve this goal for two reasons: a) their consumers (e.g.,

cloud tenants) are short-term, so they only need to consider short-term behavior based on the current network snapshot; and (b) data centers have pre-deployed redundancy and *homogeneous* hardware (i.e., similar racks, each holding similar servers). WANs, on the other hand, have much less built-in redundancy and *heterogeneous* region capacities (i.e., each data center is built differently).

Challenge 3: Large number of services and high volume of traffic. To serve all of Meta's services, scalability is one of the main challenges, rooted on two different aspects: (a) the entitlement process involves thousands of Meta's services, each with specific requirements and distinct traffic patterns (as shown in §2.1); (b) the final granted bandwidth needs to be enforced on a large volume of production traffic (O(100T) capacity). A non-scalable solution not only fails to achieve the goal of entitlement but also may disrupt day-to-day business.

3.2 Entitlement Process Overview

In this section, we present the *entitlement contract* used through the entitlement process, and the high-level workflow we use to address the above challenges.

Entitlement Contract: An *entitlement contract* is an agreement established between the network team and each Network Product Group (short for NPG) of the service team. We use NPG and *service* interchangeably in the rest of the paper. The contract specifies a) *network SLO target*, represented by network availability, e.g., 0.9998; and b) *a list of bandwidth entitlements*. Each entitlement has five fields: <NPG, QoS class, region, entitled rate (bits/s), enforcement period>. The first three fields together delineate a set of flows; the last two fields set the *maximum* supported rate (bits/s) for specified flows during a certain period. For example, <Ads, A, M, 1Tbps, $T_1 - T_2$> means Ads service' ClassA traffic of region M has been entitled 1Tbps bandwidth during T_1 to T_2.

The contract is used through the entitlement process and provides clear *accountability* between the network team and the NPGs. If an NPG generates traffic within the entitled rate and the network cannot support it, accountability lies with the network team. If an NPG generates traffic above the entitled rate, then the responsibility lies with the NPG. Such demarcation helps troubleshoot outages and post-mortem analysis. The main goal of entitlement process is to *establish and enforce* the contract which include four key steps:

1. *Service Demand Forecast* (§4.1): the network team works closely with NPGs to estimate their demand periodically (e.g., every three months). To improve forecast accuracy, many factors need to be considered such as application characteristics, user growth, and architecture changes etc.
2. *Contract Representation* (§4.2): The service demand forecast uses a Pipe-based model, as it captures service specific requirements accurately. However, this model restricts service traffic patterns and is not flexible. Thus, we convert it into a more agile Hose-based representation to specify service demand in the draft contract.
3. *Contract Approval* (§4.3): Before generating the final contract, the demand requests from NPGs need to be assessed by the network team. Specifically, the network team a) analyzes possible network risks in advance; and b) sets up reasonable SLO targets with service teams based on the network capacity and potential risks.
4. *Runtime Enforcement* (§5): All contracts are stored in a database and the approved contracts of the current period need to be enforced on the production traffic. An enforcement agent that runs on each host monitors network usage and enforces the entitled rates on the corresponding flows.

4 ESTABLISH A CONTRACT

4.1 Service Demand Forecast

Service demand forecast affects both the network efficiency and service performance. Demand over-estimation will lead to inefficient use of the network bandwidth; under-estimation will put service's traffic needs at risk. An effective forecast framework needs to cover three aspects: (a) A *well-defined metric* agreed upon between the service and network teams to represent the demand; (b) An *accurate model* that computes the metric for different services, capturing both *organic changes* (e.g., seasonal growth) and *inorganic changes* (e.g., architectural changes); and (c) The model needs to be *flexible* to address service specific characteristics.

SLI Metric: We use a Service Level Indicator (SLI) metric to represent the forecast demand, facilitating the communication between different teams. The SLI metric is defined as *the bandwidth usage of three consecutive months*. The choice of three months (a quarter) is very important. On one hand, if the time window is too small (e.g., every day), it may not reflect the service's common behavior and also result in unnecessary entitlement overheads; on the other hand, a large time window (e.g., a year) may fail to address service demands in a timely manner. In Meta, different teams often plan and propose significant changes in a quarterly basis, thus making 3-month a good time frame to communicate across teams. The SLI metric can be represented as $(NPG, QoS, src\_region, dst\_region, bandwidth)$, for a given NPG, its QoS, the source, and destination regions. We next discuss how to compute the SLI for different services.

Organic Changes: Time is a crucial factor affecting the service demand, e.g., traffic bursts during holidays. We refer to time-related changes as *organic changes*, and they often show periodic and systematic patterns over time, thus can be captured by a time-series model. We use Prophet [13], Meta's open sourced time-series forecasting algorithm. It takes historical data as the input and decomposes the time series into 3 components: trend, seasonality, and holidays, e.g., $y(t) = trend(t) + seasonality(t) + holidays(t) + \epsilon_t$. The error term ϵ_t represents any idiosyncratic changes.

While the Prophet Algorithm serves as a basis to capture timing factors, we need to consider additional service characteristics. For example, different services need different types of daily data to feed into the model, e.g., daily max average of 6 hours for storage services, and daily p99 for ads service. For big services, we consult service owners to make more fine-grained adjustments. An example is Scribe [10], the log management service. Its traffic consists of either writes (log creation) which is almost always within region, or reads (consumption) which is mostly cross region. Thus, we refine the prediction for reads by incorporating log category, read datasets, and minimum and maximum growth expectations provided by the services to adjust the model.

Inorganic Changes: Beyond organic changes, there are other factors that can affect the accuracy of demand forecasting, we refer to these as *inorganic changes*. Some examples of these include: (1) Traffic QoS class changes; (2) Region moves, e.g., moving a service from the existing region to a new one; and (3) Architecture changes, e.g., adding a caching tier.

Unlike organic changes, inorganic changes are usually impossible to predict by patterns, thus cannot be modelled by a pure time-series model. One such case is the addition of new regions or decommissioning of existing servers that serve a particular service in some region. For example, if storage is going to start in a new region, we know of these "planned" changes in advance and can predict the demand based on the allocated power and servers in the new regions. There is a relationship between these regressors and expected traffic demand which will be captured by the machine learning model. Thus, each change is modeled using the following format and is predicted using a *tree-based model*: $(request\_type, NPG, QoS, src\_region, dst\_region, bandwidth)$

There are two types of regressors: a) the adjusted *monthly* traffic volume computed by the time-series model capturing organic factors, and b) inorganic factors such as power and regional fluidity usages, e.g., flash, disk, RCU, and sever count of different server types. These regressors are fit into a tree-based model with quantile loss (e.g., alpha = 0.5) as : $f(X_{region,NPG,t}) = f(X_{region,NPG,t-1}, X_{region,NPG,t-2}, X_{region,NPG,t-3}, Y_{region,NPG,t-1}, Y_{region,NPG,t-2}, Y_{region,NPG,t-3})$, where $X_{region,NPG,t-h}$ represents the NPG traffic of month $t-h$ in the specified region, and $Y_{region,NPG,t-h}$ represents the related inorganic changes of that month. Running this model for the *next three months* (e.g., t, $t+1$, and $t+2$) generates the final forecast demand for the next quarter, i.e., the SLI.

The computed SLI, capturing both organic and inorganic changes, is used as the *initial* request to the next module.

4.2 Hose-based Contract Representation

The estimated demand cannot be directly used for the entitlement contract across teams for two reasons: (1) it is based on the current traffic patterns and cannot accommodate possible traffic movement flexibly; (2) it doesn't consider possible network risks to provide long-term SLO guarantees. We discuss how to address the first issue in this subsection, and the second in §4.3.

Strawman1: Pipe-based model. Service forecast demands are pipe-based requests, which are defined by a pair of source-destination regions. To understand why this fails to accommodate flexible communication patterns, let's consider the example service shown in Figure 6(a). Ads service has servers in five regions A to E, and assume each server sends or receives cross-region traffic. The demand forecast is 300G from A to B, 100G from A to C, and 250G from A to D and A to E.

Without considering any risks (relaxing this assumption in §4.3 later), the network team needs to reserve 900G of capacity for Ads (Figure 6(b)). Although this reserved capacity meets the forecast traffic demand, Ads does not have the *no option* to move its traffic in the future. If Ads wants to move 200G traffic of A->B to A->C, it cannot do this movement independently of the network team. Thus, the pipe-based model is not ideal to achieve flexible communication patterns.

Strawman2: Hose-based model. Another strawman solution is the Hose-based request model which has been used for long-term network planning [1, 6]. Different from the Pipe model, it aggregates ingress and egress traffic per region. For simplicity, we only consider egress traffic here. As shown in Figure 6(c), the pipe requests can be aggregated into a Hose request, which is 900G egress for A. To satisfy this Hose request and accommodate *every* possible communication patterns, the network team needs to reserve 900G for all possible destinations. This allows the Ads team to do much more flexible traffic movement. However, the hose-based entitlement requires 3600G capacity in the worst case, four times more than the pipe-based model, thus is not capacity efficient.

Our Solution: Segmented Hose. To achieve both flexibility and capacity-efficiency, we propose an enhanced Hose model–*segmented Hose*. Our observation is to leverage compute and storage allocation, which is often stable in the short term, generating predictable traffic patterns. For example, Figure 7 shows the traffic distribution for one storage service across all source regions to a given destination. 67% of traffic is sent from 3 regions, indeed, two of them are other storage regions and one is the region hosting compute resources. This observation makes it possible to incorporate service deployments into the Hose requests and reduce the demand uncertainty.

Based on this observation, we propose a segmented Hose algorithm. The key idea is to split a given Hose into two or more segmented Hoses, with each segment covering a subset of target regions. For example, as shown in Figure 6(d), segment1 covers B and C regions while segment2 covers D and E regions. This segmentation reduces the reserved capacity needed for the same forecasted demand, e.g., 400G for B/C, and 500G for D/E, which add to 1800G in total, only half of the general Hose model. It also maintains flexibility, i.e., traffic can move between B and C without requiring changes to the entitlement. The segmented hose offers a reasonable middle ground between the pipe-based model and the agnostic hose-based model, while increasing the degrees of freedom of the model.

We formulate the segmented Hose based on the the general Hose model. The general Hose can be expressed in the form of ingress and egress constraints as follows:

$$\begin{aligned} \sum_{src \in Nodes} f(src, dst) \le constraint \text{ [ingress constraint]} \\ \sum_{dst \in Nodes} f(src, dst) \le constraint \text{ [egress constraint]} \end{aligned} \tag{1}$$

The N-segmented Hose decomposes a Hose's ingress and egress constraints into N constraints (we take N=2 and the egress constraint as an example):

$$\begin{aligned} \sum_{dst \in S_1} f(src, dst) \le \alpha * constraint \\ \sum_{dst \in S_2} f(src, dst) \le (1-\alpha) * constraint \\ where\ S_1 \cap S_2 = \emptyset\ and\ S_1 \cup S_2 = Nodes;\ 0 < \alpha < 1 \end{aligned} \tag{2}$$

As a result of a Hose's segmentation, we would reduce the volume of the convex polytope delimited by the Hose, which means we can use less capacity to build the network (or conversely, admit more bandwidth at any given availability). The larger the reduction in the polytope volume, the larger the gain to be achieved.

To find the best segmentation, a general aggregation-based approach has three main steps: (1) Aggregate all traffic for a given QoS

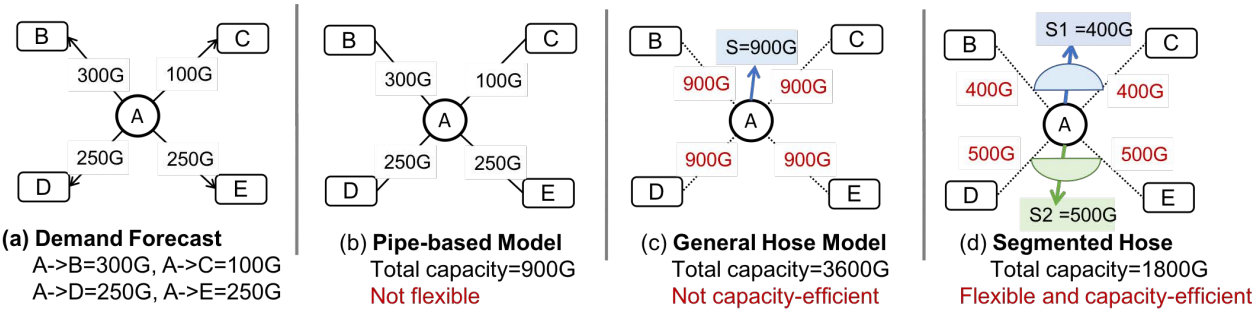

Figure 6: (a) Demand forecast of an example service. (b) Pipe-based model. (c) General Hose Model. (d)Segmented Hose.

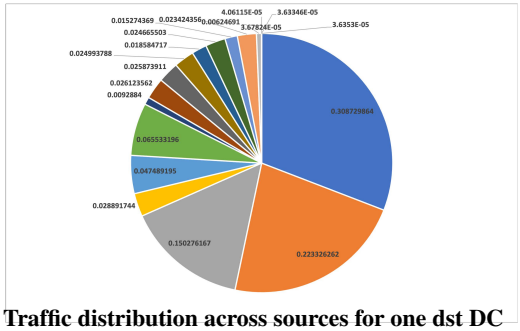

Figure 7: Traffic distribution across sources for one dst DC

class; (2) For each src region, plot the time series of flow per dst region, denoted as $F(dst, t)$; and (3) For every set of nodes S (with the complementary set being S'), we compute its ratio $R(S, t)$ and related α variables as follows:

$$R(S,t) = \frac{\sum_{dst \in S} F(dst,t)}{\sum_{dst \in N} F(dst,t)}$$
$$\alpha^+(S) = max\{R(S,t)\}; \alpha^-(S) = min\{R(S,t)\}; \quad (3)$$
$$\alpha^+(S') = max\{R(S',t)\}; \alpha^-(S') = min\{R(S',t)\};$$
$$\alpha^+(S) + \alpha^-(S') = 1; \alpha^-(S) + \alpha^+(S') = 1;$$

In terms of Hose polytope volume reduction, the highest reduction in volume can be achieved for any segmentation that includes the maximum number of dimensions in one of the new constraints (due to the curse of dimensionality). When partitioning the hose, the ratios are α and $(1-\alpha)$, which offers an optimum decomposition (i.e., the fractions sum up to 1), and avoids over-provisioning (if the hose segmentation coefficients sum up to more than 1, then the hose volume reduction would be sub-optimal). For the two segments case, splitting ratio of 50% yields the largest reduction in volume, as it scales as $\alpha * (1-\alpha)$. Thus, the best segmentation can be found by getting the largest set S (in terms of number of nodes) such that $\alpha^+(S) < 0.5$; or conversely, the smallest set S such that $\alpha^-(S) > 0.5$.

Two-Segments Algorithm: We use a greedy algorithm to split one Hose into two segments, based on the $\alpha^-(S) > 0.5$ condition. As shown in Algorithm 1, we first compute the α^- for each destination node in N using equation 3 (*line 2-3*). Then, we rank nodes in order of its α^- decreasingly (*line 4*). Finally, we add the nodes in the ranked order into the first segment set SEG (*line 5-9*). For each node added, we recompute $\alpha^-(SEG)$, until its value is larger than 0.5, meeting the condition to get the best segment. The second segment set SEG' is the rest of nodes in N (*line 10*). While our system currently uses two segments for simplicity and its good performance (details in §7),

Algorithm 1 Segmented Hose Algorithm

Input: N: set of destination nodes.
Output: SEG: set of nodes in segment1, SEG': set of nodes in segment2.
1: $SEG = SEG' = \emptyset$
2: **for** each node $n \in N$ **do**
3: $n.r = \alpha^-(\{n\})$ ▷ Compute the α^- using equation 3
4: sort the nodes of N into non-increasing order by the value of $n.r$
5: **for** each node $n \in N$, taken in non-increasing order by $n.r$ **do**
6: **if** $\alpha^-(SEG) \leqslant 0.5$ **then**
7: $SEG = SEG \cup \{n\}$
8: **else**
9: break
10: $SEG' = N \setminus SEG$
11: **return** SEG, SEG'

our algorithm can be generalized into more segments. Evaluating the effectiveness of more segments in real deployments is our future work.

4.3 Entitlement Contract Approval

Given the Hose-based draft contract, containing a list of entitlement requests <NPG, QoS class, region, entitled rate, enforcement period>, the next step is to approve them based on the available network capacity. There are two key requirements for the approval process:

- **Achieving long-term SLO guarantees.** Most Meta services are long-standing, so the approval decision should not be made only based on the current service demand, but needs to consider long-term behavior.
- **Enforcing priority between different service types.** Meta classifies backbone traffic into four classes which we refer to c1, c2, c3, and c4, with a decreasing priority. With limited capacity, a high priority class's demand should be approved before lower priority classes.

To achieve both requirements, the key challenge is the large number of services. Achieving long-term SLO guarantees requires assessing network risks, a process that *enumerates* all possible failure scenarios for each request. This process is computation heavy and *infeasible* to complete for all requests for every service individually. Luckily, we are able to identify a relatively small number of consumers of the network that account for the majority of network usage (Figure 1 & 2). We call them *high-touch services* and the rest of the services are grouped into one *low-touch service*. We try to satisfy low-touch service first given their large number of distinct services. We currently have less than 10 high-touch services, and we haven't had large changes to these definitions in recent years. A new large demand would normally fall under one of these services,

for example, network demand caused by newly developed AI clusters falls under one or more of the previously defined high-touch services.

To process different classes of requests, our approval algorithm has two main routines: *Hose_Approval* and *Pipe_Approval*. The *Hose_Approval* first converts Hose requests into a list of representative Pipe requests, which are then processed by the *Pipe_Approval* to do risk analysis [24] and enforce service priority. We specify this algorithm (Algorithm 2) in Appendix and highlight the key steps here.

Approval Process: *Hose_Approval* takes in backbone topology, entitlement contract, and Hose requests (generated by §4.2) as input, and returns a list of approved Hose requests as output. It first converts Hose requests into representative Pipe requests using an algorithm introduced by Meta's long-term network planning work[1]. Its key idea is to narrow down *infinite* possible Pipe realizations into a small set of *representative* ones, which still covers a significant portion of the Hose polytope. Given a list of Pipe requests, *Pipe_Approval* is called as a sub-routine to analyze each of them.

The *Pipe_Approval* assesses risks of the Pipe requests and enforces the service priority. It starts from Pipe requests of the *most* premium class (c1_low) and works on one class at a time until reaching the *least* premium one (c4_high). To provide long-term SLO guarantees, *Pipe_Approval* assesses risks of Pipe requests using Meta's Risk Simulation System (RSS)[24]. The RSS generates the bandwidth availability curves based on the network capacity and reliability. With the availability curves, the Pipe approval is calculated by finding the flow volume associated with the desired *SLO target*. Only when 100% of the flow meets SLO, the batch of flows is approved. If any flow fails, the batch is rejected.

Finally, *Hose_Approval* aggregates approved Pipe requests to get the Hose approval. Different aggregation approaches could affect the network usage and service guarantees, we currently sum up Hose approvals and use the *minimum* of each as the final Hose approvals.

In the approval algorithm, we approve as much demand as possible as the capacity allows. However, it is common for us to not be able to approve everything our users are asking for. It does not mean that the services cannot send traffic more than the approved entitlements, just that we don't provide guarantees. The goal of network entitlement is to provide guarantees and isolations for our users, and not to always provide 100% approvals. In those situations, there are two possible actions - (1) we work with services to explore alternative demand patterns (e.g. using different regions) or (2) we clearly communicate the implications and move forward with the under-approval. In many cases, service owners accept the risk of going over their approvals.

5 RUNTIME ENFORCEMENT

With the established contract, we next discuss how to enforce it in the production network. This is achieved by our enforcement system that runs on every host.

At a high level, our run-time enforcement system consists of three key components: (1) **Querying contract** which queries the centralized contract database to match the list of policies applicable to each host. For example, given a source host H with QoS class A, its *EntitledRate* should be X Gbps. (2) **Metering** which measures and aggregates the actual traffic rate of all hosts for each service and

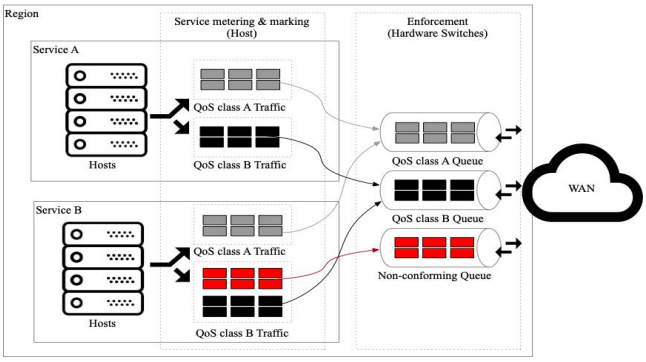

Figure 8: Current Solution Architecture

checks whether the *EntitledRate* is violated; and (3) **Enforcement** which enforces the *EntitledRate* on the specified flows in the network, e.g., dropping a portion of non-conforming traffic that is over *EntitledRate* during congestion. There are three key challenges of building a practical run-time enforcement system:

- **Scalability**: The enforcement system needs to run on every host. How can we scale the system to handle Tbps of traffic across tens of thousands of hosts?
- **Reliability**: A failure of the enforcement system can result in the enforcement contract not being honored, which could end up affecting well-behaved conforming services. How can we minimize complexity so the system can enforce the contract reliably?
- **Efficiency**: Metering and enforcement consume CPU and memory resources. How can we efficiently implement them so they have minimal impact on normal services?

In the rest of this section, we first present our early generation of the architecture and discuss how it evolved into the current architecture to address the above challenges more effectively (§5.1). We then present our experienced-based metering algorithm (§5.2) and enforcement approach (§5.3) to address practical deployment concerns.

5.1 Architecture Evolution

First Iteration: The first iteration of Meta's bandwidth manager was a *centralized* system consisting of: (a) A *Controller* that connected to a centralized contract database and all agents; and (b) *Endhost agents* installed on every host. The controller made enforcement decisions by querying the contract database and collecting traffic stats from each agent. The agents received the decision from the controller and applied rate limits to the corresponding egress traffic. For instance, if flow A's traffic rate exceeded its *EntitledRate*, the agent would rate limit the respective flow before it left the host. This implementation leveraged the iptables [23] and qdisc [14] mechanisms provided by the Linux kernel.

This architecture performed well when the system served a limited number of services (e.g., $O(10k)$ hosts) but presented two key challenges. First, computing per-host rates proved challenging to scale as the number of services (hosts) increased. Second, because the agents rate-limited traffic at the source, on occasion services ran into co-flow completion issues even when the network was not congested.

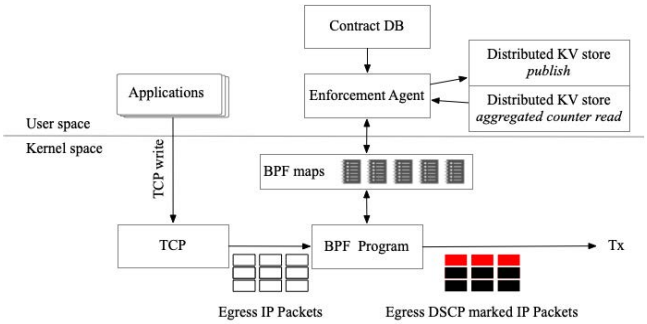

Figure 9: The architecture of enforcement agent

Current architecture: To address the issues in the first iteration, we evolved to second generation architecture (shown in Figure 8) with two major changes:

- *Centralized architecture → Distributed architecture*: Instead of relying on a centralized controller to make every decision, we evolved the system into a fully-distributed architecture in which each agent makes decisions independently. By removing the controller layer, we limit the impact of potential controller failures or the interactions between the controller and the agents. This simplified architecture improves both scalability and reliability.
- *Endhost-based enforcement → In-Network enforcement*: Instead of rate-limiting traffic at the endhost, we decided to *classify* packets at the endhost instead, and leave enforcement decisions to the hardware switches. This is based on our observation that it is difficult to know the instantaneous network capacity. Compared to the endhosts, switches have firsthand information about available capacity, thus should make the final drop decisions. This change simplifies the endhost agent since it only *marks* traffic rather than shape it, requiring much less resources.

Enforcement Agent: The agent consists a *user-space component* and a *kernel component* as shown in Figure 9. The *user-space component* is responsible for querying the centralized contract database and monitoring the traffic rates. Each agent publishes flow rate information (bits/sec) periodically using Meta's internal distributed key-value store. These rates are aggregated remotely across the entire service and read by the agent periodically. Based on the queried contract rate (*EntitledRate*) and aggregate service rate (*TotalRate*), the *user-space component* computes desired actions independently, e.g., which flows should be marked as non-conforming traffic (details in §5.3).

The *kernel component* applies the computed actions to the outgoing traffic. We use a Berkeley Packet Filter (BPF) program. Specifically, the actions are programmed in BPF maps, which are consulted by the BPF program to match packets during egress and apply the corresponding action. Figure 9 shows an example of the BPF program remarking non-conforming traffic (red) to a special DSCP value.

We choose to re-mark non-conforming traffic using agents on the endhosts instead of switches for two reasons. First, it is easier to determine the traffic attributes such as service or product-group name/IDs etc.These attributes tend to be difficult and complex to decipher on switches. Second, it is not always feasible to dynamically retrieve policy and compute traffic rates on different switches.

Network enforcement: While we don't re-mark traffic on the switch, we let the switch make the final drop decision. This is based on the marked DSCP value carried by each packet. The DSCP value of non-conforming traffic is mapped to a network queue with lowest priority in switches across both DC and Backbone[1]. When there is enough capacity, the switches transmit all packets irrespective of allocated entitlements. When there is congestion, the non-conforming traffic will be impacted before the conforming traffic.

Given the distributed enforcing architecture, we present how to compute the non-conforming traffic in §5.2 and the non-conforming traffic for remarking in §5.3.

5.2 How Much to Remark

In this section we explore how each agent can independently compute the amount of service traffic that should be remarked as non-conforming given the observed service rate (*TotalRate*) and the queried contract rate (*EntitledRate*).

Stateless metering: Naively we could assume that the amount of remarked traffic corresponds to the ratio of excess traffic above the *EntitledRate*. Then, periodically compute *NonConformRatio* based on the difference between *TotalRate* and *EntitledRate* as follows:

$$NonConformRatio = \frac{TotalRate - EntitledRate}{TotalRate} \quad (4)$$

$$ConformRatio = 1 - NonConformRatio \quad (5)$$

NonConformRatio could be used by the agent to determine the fraction of traffic to remark. As an example, assuming the Ads service has a 5 Tbps *EntitledRate* for QoS class B. If the observed *TotalRate* is 6 Tbps, then the *NonConformRatio* would be $\frac{1}{6}$ and *ConformRatio* would be $\frac{5}{6}$. Then, $\frac{1}{6}$ of the traffic should be remarked to ensure *EntitledRate* is honored.

This approach would work well during steady state when the network characteristics of both conforming and non-conforming traffic are similar. However, during *congestion scenarios* the non-conforming traffic could observe much higher loss thus any action taken on the non-conforming component of *TotalRate* would have a dampened effect.

Stateful metering: The key insight behind this approach is that conforming and non-conforming traffic can be subjected to *different congestion states* (i.e. non-conforming traffic could observe much higher loss or delay than the conforming traffic). This suggests the combined *TotalRate* should not be used to compute actions for both types of traffic. Instead the aggregate conforming rate for the service *ConformRate* should be used while at the same time keeping track of the previously computed ratio *PrevConformRatio* in each cycle to compute *ConformRatio* and *NonConformRatio* as follows:

$$ConformRatio = \frac{EntitledRate}{ConformRate} * PrevConformRatio \quad (6)$$

$$NonConformRatio = 1 - ConformRatio \quad (7)$$

Intuitively, when the *EntitledRate* is larger than the *ConformRate* (ratio > 1) for the current cycle, it means the service is remarking more traffic than necessary, thus the *ConformRatio* should be increased for the next cycle. Conversely, when the *EntitledRate*

[1]Non-conforming traffic is always mapped to the lowest priority queue regardless of the original QoS class it belongs to.

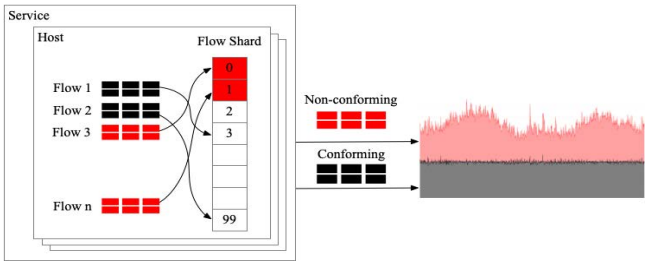

Figure 10: Flow-based remarking

is smaller than the *ConformRate* (ratio < 1), it means the service is not remarking enough traffic, thus the *ConformRatio* should be decreased for the next cycle.

In the case all the traffic for the service goes back into conformance (*TotalRate* ≤ *EntitledRate*), the metering algorithm exponentially increases *ConformRatio*, i.e. *ConformRatio* = 2∗*PrevConformRatio*. This allows rapid un-throttling but not immediate so as to avoid fluctuations if *TotalRate* rapidly exceeds or falls short of *EntitledRate*. We show this stateful algorithm can achieve the desired behavior in the next section.

5.3 What to Remark

In the previous section we computed how much traffic should be remarked as non-conforming traffic. We next need to decide *what traffic* should be re-marked as non-conforming[2]. A poorly designed marking approach may severely affect application performance.

Flow vs. Host-based re-marking: Remarking needs to done on per-flow basis to avoid packet reordering. To control how flows are re-marked, we consider two approaches: (1) a *flow-based approach* that re-marks *a fraction of flows* on each host, and (2) a *host-based approach* that re-marks all the matching service traffic from *a fraction of hosts*.

Our system implements both approaches. As shown in Figure 10, for the *flow-based approach*, we aggregate flows into groups and assign each group an identifier (e.g., from 0 to 99). If the host agent computes the *NonConformRatio* as 0.02 (like in the example figure), then flows from groups 1 and 2 will be remarked as non-conforming traffic. In the case of the *host-based approach*, all hosts are split into groups identified by a unique group number. If the host belongs to non-conforming traffic group (e.g., host 1 & 2), then *all flows* on these hosts will be remarked as non-conforming traffic.

Our operational experiences show that the *host-based approach* achieves better performance, facilities troubleshooting, and provides better visibility. While the *flow-based approach* provides fine-grained control by picking a set of flows on different hosts rather than re-marking all the traffic from the host, the result may manifest as random individual flow failures. In practice, many applications have builtin mechanisms to react to host failures, but not individual flow failures. Thus, with host-based remarking, when one host is down, the application can automatically re-balance the load to another host to mitigate failures (results in §7).

[2]Note that services signal which traffic to treat with a specific priority by marking the corresponding flows with the appropriate QoS class. Our entitlement system enforces remarking for each QoS class independently.

Besides application resilience, the *host-based approach* also helps service teams easily identify affected hosts and adjust their deployment. Given these benefits, we use the *host-based approach* as our default marking method.

6 REAL-WORLD ENFORCEMENT TESTS

We perform drill tests with the *production live traffic* every few months to ensure the correctness of the runtime enforcement system. In this section, we report the results of one of these tests from September, 2021, and we attempt to answer two key questions:

- How effective is the runtime enforcement system on enforcing the entitlement policy in production?
- What is the impact of the runtime enforcement system on service performance?

Setup: We pick one of our biggest services Coldstorage [3] as our test service. Coldstorage stores billions of photos shared daily on Facebook. This test was run on $O(10k)$ hosts, and millions of flows. All other services are run on the same backbone as normal during the test. We use the stateful host based remarking algorithm.

To evaluate our runtime enforcement system during different levels of congestion, we first decreased Coldstorage's egress entitled rate for a selected region to increase non-conforming traffic in the network. Then, we installed access control (ACL) rules in the network switches to drop an increasing percentage of Coldstorage's non-conforming egress traffic in order to mimic congestion. The drop percentage was progressively increased from 0%, 12.5%, 50%, to 100% at ˜35 mins intervals until finally all ACLs were removed after 105 minutes of test duration.

Metrics: We collected the following network-level and application-level stats during the test:

- *Network-level Stats* include packet loss, traffic rate, round trip time (RTT), and TCP stats (e.g., number of SYN/FIN/RST packets), which can be collectively used to measure the effectiveness of our enforcement system.
- *Application-level Stats* include read latency, write latency, and block errors. These stats are monitored by the Coldstorage service team and reflect the service performance. These can be correlated with the network-level stats to understand the impact on the service.

6.1 Network Stats

Loss Ratio: Figure 11 shows the loss ratio of non-conforming traffic and conforming traffic over time as we installed ACL rules following the methodology described above. We can see the loss ratio of conforming traffic (black dashed line) remains close to 0% throughout the test. For the non-conforming traffic (red solid line), there are four distinct stages with an increasing loss ratio, from 0%, 12.5%, 50%, to 100%. Finally, at close to 135 minutes, we rolled back the changes and the loss ratio returned to a small value after a short spike. This test confirms that the enforcement framework ensures guaranteed performance to the conforming traffic regardless of the severe loss of non-conforming traffic.

Traffic rate: Figure 12 shows the aggregate service total rate, conforming rate, and entitled rate as reported by the endhosts. The difference between the total rate and the conforming rate represents

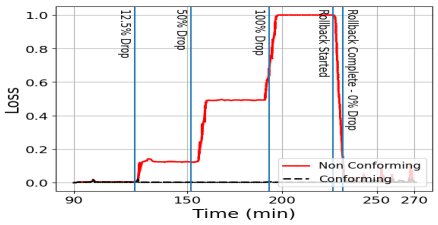

Figure 11: Packet Loss

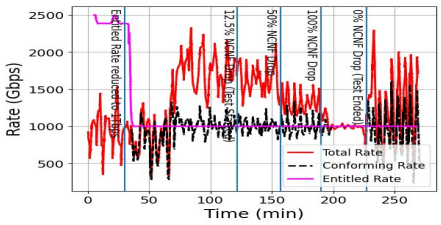

Figure 12: Traffic Rate

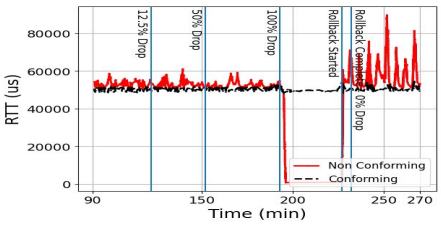

Figure 13: RTT

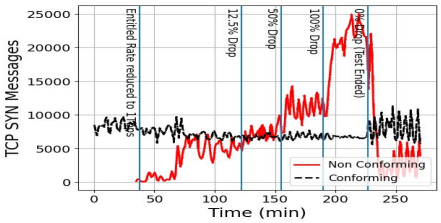

Figure 14: TCP stats: SYN Transmission

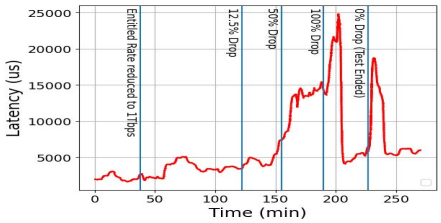

Figure 15: Storage Read latency

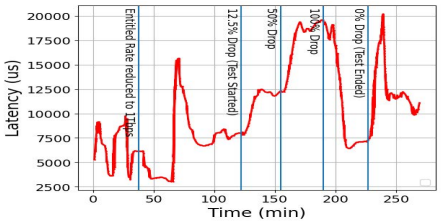

Figure 16: Storage Write latency

the non-conforming rate. At x=30 min, the entitled rate of Coldstorage is *reduced* to 1Tbps. Before x=65 min, the total rate closely matches the conforming rate as the service is not busy, but as service traffic increases, more traffic is marked as non-conforming traffic. Between x=70 min and x=195 min, as the drop percentage of non-conforming traffic increases, the total rate continues to decrease until it matches the entitled rate. After x=225 min (when all ACLs are removed), the overall rate increases to pre-test levels. As a result all traffic becomes conforming traffic and flows to the same queue. The sudden competition of queuing resources creates fluctuations, which stabilizes after a few minutes. The results from this test show that our Runtime Enforcement system is able to enforce conforming traffic to remain within the entitled rate, even during different congestion scenarios.

Round Trip Time: Figure 13 shows the average RTT for conforming and non-conforming traffic. The conforming traffic remained unaffected during the test while non-conforming traffic shows a slight increase except during 100% loss where all non-conforming traffic was dropped. These results show that our enforcement system can make decisions early on so the overall network queuing delay is not affected. It also demonstrates that network resources are used efficiently: if a packet is delivered, it is delivered without additional delays.

TCP Stats: Besides basic network stats, we also collect TCP stats including number of SYN, SYN/ACK, FIN/RST, FIN, RST, and Retransmit packets. Due to space limitations, we only show SYN stats in Figure 14. The figure shows an increase of SYN packets for non-conforming traffic as the percentage of dropped non-conforming traffic increases, with a corresponding drop once the test is completed. The metrics for conforming traffic remained unaffected during the test.

6.2 Application Stats

Application metrics such as Coldstorage restore and upload were impacted as expected. Coldstorage is the service for long-term retention of data such as image and tabular data. Coldstorage's ingress is the uploads (writes) to the storage servers and its egress is the restores (reads). Coldstorage has agreements with its users that reads will be completed less than 24 hours after a restore request is submitted. Therefore, if the network caused latency stays lower than this threshold the service itself (Coldstorage and the reader) will not see an impact on their performance metrics. The same goes for other services. Read and write latency grew proportional to the non-conforming traffic drops. The impact is similar across all clients in different regions.

Read Latency: Figure 15 shows the read latency from one remote region where clients are located. Generally, read latency grew proportional to non-conforming traffic drops. As loss created more unfinished TCP connections, there are more failed read requests which increases the average latency. This growth trend continues until the drop rate hits 100%. At this point, read latency decreases drastically, as the subset of hosts remarked as non-conforming do not establish TCP connections while the rest of hosts serve traffic without issues. The spike in latency after rollback can be attributed to a spike in TCP FIN/RST packets (omitted due to space constraints).

Something to note is that when the drop percentage is less than 50% (x<150 min), there is *little impact* on the application read latency. This can be attributed to our remarking algorithm (§5.3) that works at host-level (instead of flow-level) granularity. This makes it possible to leverage application built-in failover mechanisms to rebalance load from failed hosts to healthy hosts.

Write Latency and Error: Figure 16 shows the write latency from a remote region to the region under test. Similar to read latency, there is a gradual increase in latency matching the gradual increase in network drops. The impact on write latency is severe even when loss rate is small. This because writes are a stateful operation and sessions take some time to move away from affected hosts. Figure 17 further shows the more severe impact on write, i.e., the block error. The failure peak correlates well with TCP SYN error where the connection is difficult to establish.

7 EVALUATION

This section evaluates each component separately and quantifies the benefits in production.

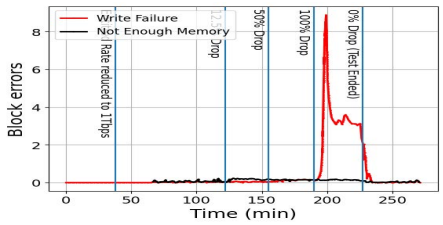

Figure 17: Block Write Error

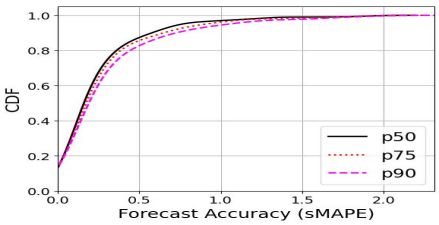

Figure 18: Forecast accuracy for QoS A

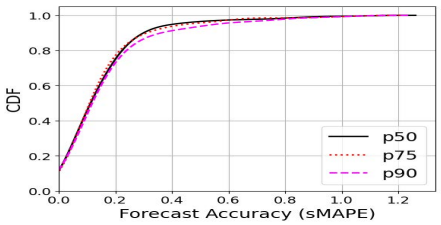

Figure 19: Forecast accuracy for QoS B

7.1 Demand Forecast Accuracy

The accuracy of demand forecast is computed by comparing actual usage (A_t) against the forecast demand (F_t). Specifically, we use symmetric Mean Absolute Percentage Error (sMAPE) computed as $sMAPE = \frac{1}{n}\sum_{t=1}^{n}\frac{|A_t - F_t|}{(A_t+F_t)/2}$. Figure 18 shows the cumulative distribution of *sMAPE* across all services in one QoS class. Note that by definition, the range of *sMAPE* is [0,2]. We evaluate the forecast result for the 50^{th}, 75^{th}, and 90^{th} percentile for each service. Figure 19 shows the same set of metrics for another QoS class. Majority of *sMAPE* is lower than 0.4. The difference of different traffic percentile is slim: p90 shows a slightly higher sMAPE. There are some anomalies where *sMAPE* values are greater than 1. They are caused by new region development, service rollout plan change, and old region decommissions.

7.2 Benefit of Segmented Hose

To quantify the benefit of Segmented Hose model, we use the metric "Hose coverage" [24] which evaluates the degree to which the generated traffic matrices (TMs) cover the entire Hose space. Ideally, we want to use a small subset of representative TMs to cover a large Hose space. To compare the Segmented Hose versus the general Hose, we compute the number of TMs needed to achieve the same coverage 75% with each approach. Figure 20 shows that in 90% of the cases, Segmented Hose needs 60% fewer TMs. The reduction of TMs needed to achieve a high Hose coverage means a drastic decrease of computation overhead.

7.3 Bandwidth Approval Tradeoffs

The set of TMs are the input to bandwidth approval engine, affecting the speed of approval computation. Another factor that controls the TM set is the Hose coverage. Figure 21 shows that the coverage increases with more TMs but with a smaller benefit when the TMs reach a certain point, e.g., 2000 TMs. It illustrates a trade off between the coverage and the approval speed, as approximated by the number of TMs. The trends are consistent across QoS classes.

The second tradeoff in bandwidth approval engine is the approval rate and network SLO. Figure 22 shows that as availability requirement increases, we have to reserve more bandwidth for each service to meet the high availability under failures. As a consequence, the total number of service requests approved is reduced. Egress and ingress approvals exhibit similar trends.

7.4 Effectiveness of Enforcement

Finally, we use simulation to evaluate the convergence of host marking algorithms. Assuming a total traffic rate of 10Tbps and an entitled rate of 5Tbps, we gradually simulate network congestion with a loss rate of 0%, 12.5%, 25%, 50% and 100% of the non-conforming traffic. We test both stateless and stateful marking algorithms described in §5. For each experiment, we observe the conforming traffic rate and compare it to the entitled rate. We then plot both the instantaneous rate observed in each iteration and the average rate computed across multiple iterations.

Figures 23 and 24 show the stateless marking algorithm results. As congestion increases, the instantaneous curve fluctuates, e.g., between 5Tbps and 10Tbps for 100% loss. The oscillation is due to the fraction of traffic the algorithm decides to remark to non-conforming gets dropped in the network and in the next cycle we see only see conforming traffic which should be within the entitled rate, the algorithm evaluates there is no need to remark any more, being stateless, and decides NOT to remark any traffic as non-conforming. This results in total rate (assuming steady demand) jumps back to the original total rate of 10Tbps. Figure 24 clearly shows the average of conforming traffic stays above the entitlement rate (5Tbps). This means the marking algorithm fails to enforce the entitled rate.

In Figure 25, we can see the stateful algorithm is able to address the issue. Both instantaneous and average curves are the same. The results for 0% to 100% are the same, which converge to 5Tbps quickly after the 10^{th} iteration. The instantaneous and average rates look similar, because the stateful algorithm already smooths out the difference across iterations.

8 FUTURE DIRECTIONS

In this section, we share a few new areas that stem from our experiences of implementing and operating network entitlement within Meta over two years.

Bandwidth Negotiation: When the contract approval engine rejects a service's request, it is currently handled manually between network and services as it can have many choices. One straightforward way is to return back to service and reduce the requested demand to try again. Alternatively, the approval engine could come up with a counter-proposal of admittable traffic to service. However, it is simply reducing the traffic directly to admittable volume, because services may have to make server allocation changes to accommodate the lack of network bandwidth, resulting a new demand. More complicated, due to dependency across services, the decision might be populated to more than one service team. Services can even perform bandwidth trading if their dependencies require so. As a part of our ongoing work, we are developing an automated negation platform to facilitate those decisions.

Unbalanced ingress and egress Hoses: Surprising challenges arise due to data quality and operational artifacts. One such example is the inequality of total ingress and egress traffic demand when aggregating all the ingress Hoses and egress Hoses together. That

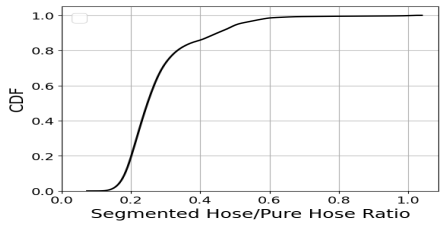

Figure 20: Efficiency of Segmented Hose

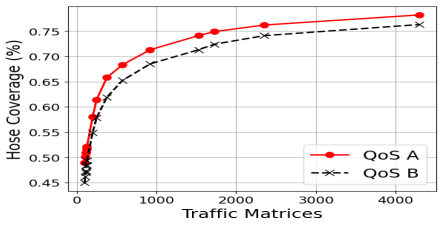

Figure 21: Hose coverage and computation overhead

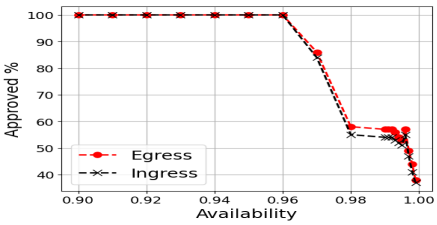

Figure 22: Approval percentage under different availability

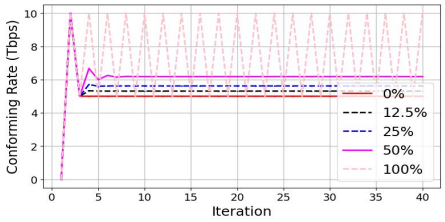

Figure 23: Stateless marking algorithm with instantaneous rate

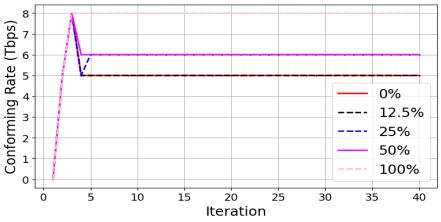

Figure 24: Stateless marking algorithm with average rate

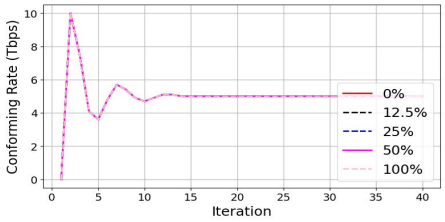

Figure 25: Stateful marking algorithm with instantaneous rate

is, the total ingress and the total egress for the global WAN must be equal. It is an artifact that the forecast of demand is performed for each hose independently. To maintain the correctness of the algorithm, we add a preprocessing to balance the ingress and egress by inflating the shortage direction. For example, if total egress is less than total ingress, we will inflate the egress so that they matches. This delta of the demand is modeled as a dummy service and is evenly attributed to all regions.

Ingress metering: In the current implementation, our runtime enforcement system takes the egress entitled rate as the input to control the amount of outgoing traffic from each host. However, in practice we observe the need to also perform metering to conform with the ingress entitled rate. Since metering can only be performed at the source, we need to translate the ingress entitlement Hose for a destination to a distributed set of meters at the sources. This requires both new algorithm design and more sophisticated centralized control.

9 RELATED WORK

There is a large body of work on enabling bandwidth sharing for cloud providers [2, 5, 7, 11, 18, 19, 21, 25]. Our work shares the similarity from two aspects. First, from the demand modeling perspective, these works take the per-VM hose based traffic demand and allocate VMs to minimize the total bandwidth consumption. These workloads are usually short-lived so that there is no need to consider failure probability and protections. For instance, Oktopus [2] proposes a VM placement algorithm based on the Hose constraints of any two sets of VMs. This model essentially adds up all the worst-case TMs and results in significant over-provisioning. Another example is TIVC [25] which extends the per-VM hose model to capture the time-varying nature of the networking requirement of cloud applications. We extend our opportunistic hose planning algorithm [1] to incorporate the short-term traffic pattern. Second, these existing work take the approach of rate-limiting at the host. We found centrally determining the throttling rate in real-time does not work well but increases management complexity greatly.

DRL [20] is a solution for distributed rate control for TCP, however, it does not account for network capacity or SLO guarantees. BwE [12] focuses on WAN bandwidth sharing but operates on a different time scale and is not SLO aware. Solutions such as SWAN [8] and B4 [9] focus on Software Defined Networking to effectively manage WAN network. Recent work has also tackled SLO and failure aware traffic engineering such as TeaVaR [4], FFC [15], and BATE [26]. These efforts are complementary to the approach in this paper.

Lastly, this system is built on top of previously published systems in Meta's infrastructure [1, 24] but is solving a completely independent problem. While hose model [1] and network risk [24] is used for long-term network planning, they have been used differently in network entitlement.

10 CONCLUSION

Network efficiency and safety has been an increasingly critical topic for the giant online service providers to keep up with the service growth in a sustainable manner. This paper tackles this problem by introducing Network Entitlement, a new contract-based approach. We describe an end-to-end system to provide agile and efficient network bandwidth sharing with SLO guarantees. We hope to bring a new angle to research on intelligent network multiplexing for further innovations.

Acknowledgments. Many people in the Network Planning and Host networking team at Meta have contributed to the Entitlement program over the years. We would like to acknowledge Steve Politis, Rajiv Krishnamurthy, Omar Baldonado, and Gaya Nagarajan for their support of the program. We thank our shepherd Sanjay Rao and anonymous reviewers for their comments. Guyue Liu and Ying Zhang are the corresponding authors.

REFERENCES

[1] Satyajeet Singh Ahuja, Varun Gupta, Vinayak Dangui, Soshant Bali, Abishek Gopalan, Hao Zhong, Petr Lapukhov, Yiting Xia, and Ying Zhang. 2021. Capacity-Efficient and Uncertainty-Resilient Backbone Network Planning with Hose. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 547–559.

[2] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards Predictable Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*. Association for Computing Machinery, New York, NY, USA, 12.

[3] Krish Bandaru and Kestutis Patiejunas. 2017. Under the hood: Facebook's cold storage system. https://engineering.fb.com/2015/05/04/core-data/under-the-hood-facebook-s-cold-storage-system/. (2017).

[4] 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 *Proc. ACM SIGCOMM'19*.

[5] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient Coflow Scheduling with Varys. 44, 4 (2014).

[6] Nick G. Duffield, Pawan Goyal, Albert G. Greenberg, Partho Pratim Mishra, K. K. Ramakrishnan, and Jacobus E. van der Merwe. 1999. A Flexible Model for Resource Management in Virtual Private Networks. In *Proceedings of the ACM SIGCOMM 1999 Conference*. ACM, 95–108.

[7] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*. Association for Computing Machinery, New York, NY, USA, 12.

[8] 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 *Proc. ACM SIGCOMM'13*. 15–26.

[9] 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. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 3–14.

[10] Manolis Karpathiotakis, Dino Wernli, and Milos Stojanovics. 2017. Scribe: Transporting petabytes per hour via a distributed, buffered queueing system. https://engineering.fb.com/2019/10/07/data-infrastructure/scribe/. (2017).

[11] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. 2017. Beyond Fat-Trees without Antennae, Mirrors, and Disco-Balls. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. Association for Computing Machinery, New York, NY, USA.

[12] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 1–14.

[13] Ben Letham and Sean J. Taylor. 2021. Prophet: forecasting at scale. https://research.facebook.com/blog/2017/02/prophet-forecasting-at-scale/. (2021).

[14] Linux. [n. d.]. Classless Queuing Disciplines. https://tldp.org/HOWTO/Traffic-Control-HOWTO/classless-qdiscs.html. ([n. d.]).

[15] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. 2014. Traffic engineering with forward fault correction. In *Proc. ACM SIGCOMM'14*.

[16] Sarang Masti. 2021. How we built a general purpose key value store for Facebook with ZippyDB. https://engineering.fb.com/2021/08/06/core-data/zippydb/. (2021).

[17] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. 2014. F4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, USA, 383–398.

[18] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. 2012. FairCloud: Sharing the Network in Cloud Computing. *SIGCOMM Comput. Commun. Rev.* 42, 4 (Aug. 2012).

[19] 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. In *Proc. ACM SIGCOMM'13*. 351–362.

[20] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C Snoeren. 2007. Cloud control with distributed rate limiting. *ACM SIGCOMM Computer Communication Review* 37, 4 (2007), 337–348.

[21] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. 2011. Gatekeeper: Supporting Bandwidth Guarantees for Multi-Tenant Datacenter Networks. In *Proceedings of the 3rd Conference on I/O Virtualization*. USENIX Association, USA.

[22] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. 2010. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 1013–1020.

[23] Wikipedia. [n. d.]. iptables. https://en.wikipedia.org/wiki/Iptables. ([n. d.]).

[24] Yiting Xia, Ying Zhang, Zhizhen Zhong, Guanqing Yan, Chiun Lin Lim, Satyajeet Singh Ahuja, Soshant Bali, Alexander Nikolaidis, Kimia Ghobadi, and Manya Ghobadi. 2021. A Social Network Under Social Distancing: Risk-Driven Backbone Management During COVID-19 and Beyond. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 217–231.

[25] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Kompella. 2012. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In *Proceedings of the ACM SIGCOMM 2012 Conference (SIGCOMM '12)*. Association for Computing Machinery, New York, NY, USA, 199–210.

[26] Han Zhang, Xingang Shi, Xia Yin, Jilong Wang, Zhiliang Wang, Yingya Guo, and Tian Lan. 2021. Boosting bandwidth availability over inter-DC WAN. In *Proc. ACM CoNEXT'21*.

[27] Yufei Zhu. 2015. Serving Facebook Multifeed: Efficiency, performance gains through redesign. https://engineering.fb.com/2015/03/10/production-engineering/serving-facebook-multifeed-efficiency-performance-gains-through-redesign/. (2015).

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A APPROVAL ALGORITHM

Algorithm 2 presents our contract approval algorithm that includes two main routines: *Hose_Approval* and *Pipe_Approval*.

Algorithm 2 Contract Approval Algorithm

Input: $topo$: The backbone network topology with network capacity and fiber reliability
Input: $contract$: The SLO target and a list of entitlement requests
Input: $hose\_requests$: The product demand forecasts in the hose format.
Output: $final\_hose\_approvals$: A list of approved hose requests.

function HOSE_APPROVAL($topo$, $contract$, $hose\_requests$)
  $init\_hose\_approvals = \emptyset$
  ▷ call Demand Generation Service to get representative pipe requests
  $pipe\_requests$ = GEN_DEMAND($topo$, $hose\_requests$)
  for each request $r \in pipe\_requests$ **do**
    $pipe\_approval$ = PIPE_APPROVAL(r, $topo$, $contract$)
  ▷ aggregate pipe approvals into the final hose approvals
  $init\_hose\_approvals$ = sum up ingress/egress $pipe\_approval$ for each hose
  $final\_hose\_approvals$ = min($init\_hose\_approvals$)
  return $final\_hose\_approvals$

function PIPE_APPROVAL($pipe\_request$, $topo$, $contract$)
  $pipe\_approvals = \emptyset$
  $tmp\_requests = \emptyset$
  ▷ enforce the priority between different QoS classes
  for each QoS class $cos \in contract$ sorted by the priority from high to low **do**
    cos_pipes = COS_PIPES(cos, $pipe\_request$, $tmp\_requests$)
    ▷ call Risk Simulation System to assess risks
    $availability\_curves$ = ASSESS_RISK(cos_pipes, $topo$)
    $tmp\_approvals$ = get approval for each cos_pipe in $availability\_curves$ based on the SLO target
    ▷ aggregate pipe approvals and requests
    MERGE_APPS (cos, $tmp\_approvals$, $pipe\_approvals$)
    MERGE_REQS(cos, $tmp\_requests$, $pipe\_request$)
  return $pipe\_approvals$