A Scalable Monitoring Approach for Service Level Agreements Validation

Mun Choon Chan Yow-Jian Lin
Networking Research Laboratory
Bell Laboratories, Lucent Technologies
{munchoon,yjlin}@research.bell-labs.com

Xin Wang
Electrical Engineering Department
Columbia University
xwang@ctr.columbia.edu

Abstract

In order to detect violations of end-to-end service level agreements (SLA) and to isolate trouble links and nodes based on a unified framework, managers of a service provider network need to gather Quality of Service (QoS) measurements from multiple nodes in the network. For a network carrying over thousands of flows with end-to-end SLAs, the information exchanged between network nodes and a central network management system (NMS) could be substantial. Moreover, in situations where only a small number of flows violate their respective SLAs, simple polling mechanisms can lead to huge unnecessary overhead in identifying these ill-behaved flows. In this work, we propose an algorithm called **ARM** *(Aggregation and Refinement based Monitoring) to reduce the amount of information exchange.* **ARM** *uses a histogram-based dynamic QoS data aggregation/refinement technique at each network node and a reasoning engine at the NMS to minimize the amount of data exchange between network nodes and NMS.* **ARM** *not only reduces unnecessary reporting through selective refinement, it also performs well across a wide range of traffic loads. Our simulation results show that* **ARM** *is at least an order of magnitude more efficient than a simple polling scheme. It also outperforms two centralized, highly optimized schemes that cannot be implemented in practice.*

1 Introduction

This paper describes a scalable framework for monitoring end-to-end Quality of Service (QoS) with an emphasis on detecting flows that have violated their respective Service Level Agreement (SLA). QoS guarantee has become a highly desirable feature in Internet service offering. To meet an SLA offered to a customer, an Internet Service Provider (ISP) must provision and monitor the usage of its network resources. Traditionally ISPs have been over-provisioning resources to meet their SLAs, an approach that is not cost effective. Recent works on resource allocation [9] and [5] that build on both deterministic and statistical models have yielded interesting results. Nevertheless, the provision based on these results is still conservative.

The measurement-based approach for managing resources is becoming an attractive alternative in ensuring QoS offering [6]. It is based on a basic monitoring-control loop. With a roughly-estimated initial provisioning, the approach relies on constant interactions between measurement and provisioning adjustment. The main advantage of measurement-based approach is its dynamic adaptation to changes in resource needs. On the other hand, the approach poses some challenges. One of the challenges is the efficient collection of measurement data, in particular, when managing a large network.

The mechanisms for collecting measurement data vary, depending on the amount and the type of data transmitted. We assume that in an ISP's network, measurements are collected at routers and forwarded to a network management system (NMS). The amount of measurement data forwarded from a router to the NMS could be *exhaustive*: a router forwards all measurement data collected from all flows passing through it to the NMS, or it could be *selective*: a router forwards only a subset of the measurement data as needed. The exhaustive data collection easily yields a complete picture but at the cost of excessive overhead, whereas the selective one enables scalability with added complexity in selecting proper subsets of information.

The type of measurement data conveyed to the NMS could be *end-to-end*: QoS data are accumulated along the route from the source to the destination before being forwarded to the NMS. Alternatively, the data conveyed could be *hop-by-hop*: QoS data are collected on a per-hop basis and the NMS must assemble the received data to determine the end-to-end QoS. End-to-end data tend to provide more accurate measurement on an end-to-end basis, but reveal less information in helping NMS locate performance degradation in intermediate nodes. On the other hand, hop-by-

hop data can be accurate in some QoS measurement such as loss rate, but can introduce inaccuracies in other measurement such as delay. Nevertheless, since the NMS receives per-hop data, it can easily identify the problematic links in ill-behaved flows.

In accordance with network management terminologies, we refer to the object that collects and sends measurement data at each router as an *agent* . We also use the terms NMS and *manager* interchangeably.

2 Problem Statement and Key Innovations

The main focus of this paper is the design of an efficient and scalable monitoring algorithm that is capable of detecting QoS violation in a large network. The monitoring approach proposed is based on *selective, hop-by-hop* measurement data. The main technique used is *data aggregation*.

We assume that the network of an ISP consists of a large number (> 100) of network devices. Each of these devices supports a large number of flows ($> 1,000$) and is capable of collecting detail information concerning all flows of interest. The kind of flows concerned here, called SLA flows, is between any two end points in an ISP network, and is an aggregated traffic governed by an SLA. We classify each SLA flow by its source, its destination, and its SLA. SLA flows are long-lasting; once an SLA flow is provisioned, it usually stays up for an extended period of time.

A naive approach to monitoring the performance of SLA flows is to collect performance measurements of each flow from every network device. While this approach may be reasonable for a small network, it is inefficient, not scalable, and can cause severe overload as well as congestion at the network manager during a monitoring cycle. Data aggregation is one approach to achieve scalable monitoring.

The main objective of data aggregation is to use a controlled amount of information to convey a close approximation of a set of data. By aggregation we mean to use a value range (*minimum* and *maximum* values) to represent the many QoS measurements associated with a set of flows. In order for the manager to properly extract information from aggregated data, it needs to know how each agent aggregates QoS measurements. In particular, the manager must figure out the mapping between each aggregated data point and its corresponding set of flows.

Given the measurement data of a set of flows, we refer to the problem of partitioning the set for aggregation as a *flow grouping* issue. Alternatives exist to address the flow grouping issue, each with different trade-offs. One way is to statically assign flows to groups. However, without the proper means to predict performance similarity among flows, the static group assignment tends to yield poor approximation. Another way is to let both the manager and agents use the same random group assignment function (with same random seed each time) to add dynamics. Nevertheless, without taking into account the real measurement value distribution, this approach too could fail badly. A third approach is to let each agent groups flow measurements dynamically based on their values and notify the manger each group's membership along with the aggregated data. One major problem with this approach is that the overhead of conveying such membership is now in the same order as that of conveying individual flow data, which defuncts the purpose of data aggregation.

The key idea of the proposed monitoring approach, called **ARM** (Aggregation and Refinement based Monitoring), is based on a dynamic hierarchical aggregation mechanism which allows selective incremental refinement on reporting details. The approach assumes that NMS is aware of the route for each individual SLA flow, a commonly available information through VPN or MPLS provisioning. The proposed approach also assumes that NMS and agents maintain the same ordering view of SLA flow identifications. After an agent has collected QoS measures of all its flows, the agent forwards an approximation of these measures to the NMS as a histogram with a small number of bars. The NMS then constructs a view of the QoS that each SLA flow is experiencing based on the relevant per-hop values represented in these histograms. The NMS will ask agents to refine portions of their histograms only if it needs more precise values to determine if some flows in those portions have violated their respective SLAs.

ARM exhibits several key advantages. With its dynamic histogram-based data aggregation it requires minimum data exchange between agents and NMS in creating a coarse but informative picture of the network status. Its selective refinement procedure reduces unnecessary data reporting. More importantly, **ARM** performs well across a wide range of traffic loads.

We conducted extensive simulations to study the performance of **ARM** in terms of monitoring overhead reduction. In particular, we studied its performance under various network load, aggregation granularity, and aggregation selection functions. Note that the reduction of data exchange overhead between network manager and agents comes at the cost of additional aggregation computation at network devices. Given that modern routers are beginning to provide hardware-assisted packet accounting and have large processing capabilities, this appears to be a reasonable trade-off.

The monitoring algorithm proposed here is independent of other SLA management mechanisms, such as admission control and bandwidth/buffer allocation schemes. The aggregation and refinement are also independent of the QoS parameter being monitored; the NMS maintains its responsibility for interpreting the data end-to-end.

This document is organized as follow. Section 3 presents our monitoring framework, **ARM**, followed by the simulation results in Section 4. Section 5 discusses related work, with concluding remarks in Section 6.

3 The Algorithm

This section describes the proposed monitoring algorithm - **ARM** . We first outline the QoS measures of interests for each SLA flow and the violation conditions associated with them. After that, we present the algorithm, discussing how **ARM** incorporates a novel dynamic histogram-based aggregation technique for exchanging measurement data, how the NMS interprets the aggregated data, how the refinement takes place, and when the algorithm terminates.

3.1 QoS Measures for SLA Flows

Typical parameters of a Service Level Agreement (SLA) for a flow i include: average throughput (SLA^i_{thr}); end-to-end packet loss ratio (SLA^i_{loss}); and average end-to-end packet delay (SLA^i_{delay}). Out of these three parameters, our work has centered around the loss ratio SLA^i_{loss} and the delay SLA^i_{delay}. We assume that routers at the edge of a network can perform policing function to ensure that each flow will not exceed a certain peak rate and burst size while allowing it to enter the network at no more than the average throughput SLA^i_{thr}.

The routers in an ISP network collect measurements of SLA flows passing through them. As mentioned, **ARM** is based on hop-by-hop monitoring. The router at hop j of an SLA flow i collects its local measurements:

- Loss Ratio ($Local^{ij}_{loss}$) = packet drop count of flow i at hop j / packet arrival count of flow i at hop j;
- Average Delay ($Local^{ij}_{delay}$) = total packet delay sum of flow i at hop j / packet departure count of flow i at hop j

Packet delay at a router is defined as the time difference between a packet entering and leaving the router. Similarly, the end-to-end packet delay SLA^i_{delay} is the time difference between a packet entering an ingress router and leaving an egress router. Here we assume zero transmission delay. With the current technology, a router can compute this time difference by tagging all incoming packets with a 16-bit timestamp with 1ms resolution. Such a timestamp allows packet delay for up to 65 seconds, which should be sufficient for most, if not all, reasonable router performance. Note that this also assumes that the clocks on the interface cards are synchronized to within 1ms. If a 16-bit timestamp is too expensive, a 8-bit timestamp with 2ms resolution is another option, which supports up to 512ms packet delay. This alternative may be sufficient for some routers depending on their queuing discipline and buffer size.

Given the local loss ratio and average delay measurements, $Local^{ij}_{loss}$ and $Local^{ij}_{delay}$, of an SLA flow i at each hop j, assuming that $Local^{ij}_{loss}$ and $Local^{ij}_{delay}$ are small for all i and j, we define flow i's end-to-end measurements as follows:

$$EtoE^i_{loss} = \sum_j Local^{ij}_{loss} \quad (1)$$

$$EtoE^i_{delay} = \sum_j Local^{ij}_{delay} \quad (2)$$

It follows that a flow i meets its SLA requirements if $EtoE^i_{loss} \leq SLA^i_{loss}$ and $EtoE^i_{delay} \leq SLA^i_{delay}$.

3.2 ARM, Aggregation and Refinement-based Monitoring

ARM addresses the scalability and overhead issues in forwarding local measurements to the NMS for SLA violation detection. Monitoring sessions are performed periodically (or on demand). In order to detect and correct violations in time, the interval between periodically performed monitoring sessions should be smaller than the SLA measuring period. Each **ARM** session operates based on the following steps:

1 Each agent computes and forwards an aggregation of local measurements to the manager;
2 The manager processes the aggregated data to detect flows violating their SLAs;
3 **While** the violation status of some SLA flows is still in doubt **do**
4 The manager requests, and the agents respond with refined aggregated data;
5 The manager recheck the violation status based on the refined data;

ARM consists of three major components: histogram-based aggregation, violation detection, and selective refinement. It uses the histogram-based aggregation in Steps 1 and 4, the violation detection procedure in Steps 2 and 5, and the selective refinement in Step 4. Each session ends when the manager is clear about the violation status of every SLA flow. The following subsections discuss these components in detail.

3.2.1 Histogram-based Aggregation

We propose a histogram-based aggregation technique to convey an approximation that has reasonable overhead and

yet captures the dynamics in QoS measurements. Moreover, the technique provides adjustable parameters to adapt to changes in network load (and thus the number of flows violating their SLAs). We assume that the manager and each agent share information about the SLA flows running through the agent and their identifiers. Changes on such shared information occur at a much slower time scale than the monitoring session. For each QoS parameter such as loss ratio or average delay, this information serves as the basis of creating an ordered list of values according to the ascending order of flow identifiers.

Histogram is well suited to approximate a curve, which in our case is a series of QoS values sorted according to the ascending order of flow identifiers. In **ARM** each segment of a histogram has both an upper bound and a lower bound that represent the maximum and the minimum QoS value of the flows included in the segment. We use three values to encode each segment in a histogram, the upper bound U, the lower bound L, and the length S (i.e., the number of flows in the segment). When we merge two consecutive segments i and $i+1$, the resulting segment i^{new} carries the encoding $U_{i^{new}} = max(U_i, U_{i+1})$, $L_{i^{new}} = min(L_i, L_{i+1})$, and $S_{i^{new}} = (S_i + S_{i+1})$. We also define

$$\begin{aligned} &\textit{difference}(i, i+1) = \\ &\quad [(max(U_i, U_{i+1}) - min(L_i, L_{i+1})) * (S_i + S_{i+1})] \\ &\quad -[(U_i - L_i) * S_i] - [(U_{i+1} - L_{i+1}) * S_{i+1}] \end{aligned} \quad (3)$$

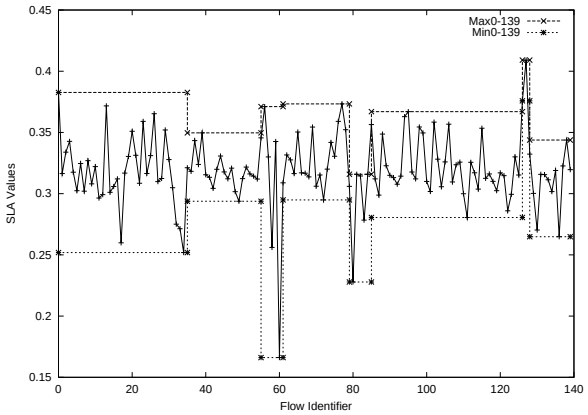
That is, *difference(i, i+1)* represents the increase in the area of uncertainly after merging segments i and $i+1$. In general, the more segments there are in a histogram, the better the approximation is, though at the cost of additional overhead.

Histogram Construction Algorithm

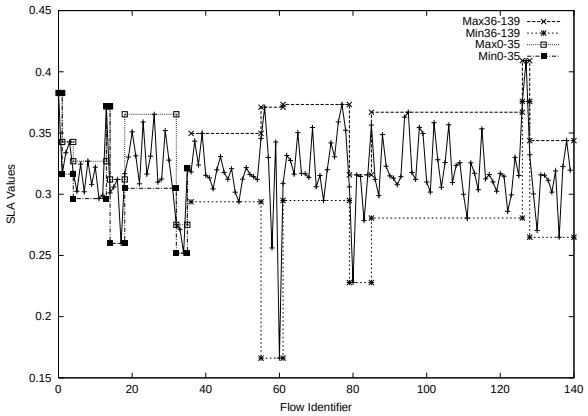
Input Parameters: a list of M data points, an initial aggregation threshold T, and the maximum number of segments N

1. Initialize a histogram of M segments, where each segment corresponds to 1 data point. The upper and lower bound of each segment is the data point itself, and the segment length is 1.
2. Merge neighboring segments i and $i+1$ if $(U_{i^{new}} - L_{i^{new}}) \leq T$. Let the number of segments remained after this step be C.
3. **while** $C > N$ **do**
4. Select a segment k such that *difference(k, k+1)* is the smallest;
5. Merge segment k and $k+1$ and subtract C by one;

Figure 1(a) shows a graphical representation of applying our histogram construction algorithm to a set of 140 values. The N in this case is 8.



(a) Example of Histogram Approximation - First Iteration



(b) Example of Histogram Approximation - Second Iteration

Figure 1. Example of Histogram Approximation and Refinement in ARM

In general, when the differences among data points are *small enough*, it is more efficient to merge the histograms in the initial phase. This is particularly true, for example, when flows are all having minimum loss ratio or similar average delay. This merging process is performed in Step 2, where the threshold T define the *small enough* difference.

Note that this algorithm limits the number of output segments to be at most N. The choice of N impacts both the data exchange overhead as well as the number of iterations needed to complete a session.

3.2.2 Violation Detection

Once the manager receives histograms from agents (one histogram per QoS parameter from each agent), it must interpret the aggregated data to detect flows that have violated their respective SLAs. The following describes how the manager makes such decisions.

To extract the upper and lower bound QoS value, say the loss ratio, of an SLA flow at an agent is fairly straightforward. The histogram sent by each agent to the manager approximates a curve, of which the flow identifier is the x-axis and the QoS value is the y-axis. Note that the manager knows exactly the set of flows passing through each agent and their identifiers. Hence, the manager knows the index of the x-axis. With the triplet (U, L, S) for each segment, the manager can compute from left to right along the x-axis the upper and lower bound QoS values of each flow reported in the histogram.

We distinguish an *exact* value from a *bound* for each flow; A flow has an exact value if its upper bound and lower bound are the same. By applying equations 1 and 2 each twice, first using the per-hop upper bound and then the per-hop lower bound, the manager derives the upper and lower bound of the estimated end-to-end loss and delay for each SLA flow.

Let $EtoE_x^i(upper)$ and $EtoE_x^i(lower)$ be the upper and lower bound respectively of the QoS parameter x (loss or delay) for SLA flow i. $EtoE_x^i(exact)$ exists if $EtoE_x^i(upper) = EtoE_x^i(lower)$. The manager acts in one of the following five cases listed under two broad categories:

- $EtoE_x^i(exact)$ exists, then
  - **Case I:** $EtoE_x^i(exact) > SLA_x^i$.
    Flow i has violated its x QoS. The manager must take immediate actions to ease the problem.
  - **Case II:** $EtoE_x^i(exact) \leq SLA_x^i$.
    Flow i is fine for now.
- $EtoE_x^i(exact)$ does not exist, that is, some of the reported values for flow i are SLA bounds, then
  - **Case III:** $EtoE_x^i(upper) > SLA_x^i \geq EtoE_x^i(lower)$.
    The manager cannot infer anything definitely in this case.
  - **Case VI:** $EtoE_x^i(lower) > SLA_x^i$.
    Flow i is definitely in violation of its SLA.
  - **Case V:** $SLA_x^i \geq EtoE_x^i(upper)$.
    Flow i is fine for now.

In cases II and V, depending on how close it is to a violation, the manager may choose to take some actions such as re-routing the flow. In general, since the manager has per-hop information, it can spot problems at some hops even when the end-to-end measure is fine.

As the objective is to detect possible SLA violations, the only set of flows that require further investigation are those in Case III. The following subsection describes our refinement algorithm.

3.2.3 Selective Refinement

The purpose of our selective refinement approach is to refine the *coarse* network status pictures that the manager constructed based on reported histograms. As long as some flows are in Case III, the manager must selectively ask agents to refine segments of their reported histograms.

Manager Selective Request Algorithm

Input Parameters: a list FS ($|FS| = k$) of case III flows in the current round, the maximum number N_{poll} of flow/segment identifiers to be sent back to an agent, and the current histogram H reported by the agent

1 Let $CS = \{s_i | (FS \cap s_i) \neq \emptyset, s_i \in H\}$ be the set of segments in H each of which contains at least one flow in FS, $|CS| = m$;
2 **if** $k \leq N_{poll}$ **then**
3 Poll the agent for the exact QoS values of flows in FS;
4 **else**
5 Ask the agent to refine the segments in some set $RS \subseteq CS$, where $RS = CS$ if $m \leq N_{poll}$.

The strategies for choosing RS from CS in Step 5 vary. We currently pick the first m segments of CS when $m > N_{poll}$. The parameter N_{poll} allows the manager to limit the communication overhead between the manager and the agents. The manager can also use it as a threshold to decide whether it is time to poll an agent for exact flow data instead, as we have shown in Steps 2–3. Note that if $CS = \emptyset$, then the manager does not poll the corresponding agent.

When the manager does polling in Step 3, the agent simply replies with the exact values for the flows listed in FS. Otherwise, the agent performs the following algorithm to refine the histogram.

Agent Selective Refinement Algorithm

Input Parameters: a list of segments RS in the current histogram to be refined, and the maximum number N_i of new segments to be reported at round i

1 Use heuristics to select a number b_k for refining segment $s_k \in RS$ into b_k new segments, where $1 \leq k \leq |RS|$, and $\sum_{k=1}^{|RS|} b_k \leq N_i$;
2 Apply Histogram Construction Algorithm to each segment s_k and create a set of new segments $\{s_{ki} | 1 \leq i \leq b_k\}$;
3 Add a triplet (U_{ki}, L_{ki}, f_{ki}) to a reply message for each newly created segment s_{ki};
4 Send the reply message to the manager;

Note that in our implementation scheme the rightmost

flow identifier in a segment serves as the segment identifier. We also use it to signal segment boundary, instead of using the length of each segment. Hence, in Step 3 above f_{k_i} is the rightmost flow id of s_{ki}.

The choices of b_k in Step 1 are not crucial. It is more important to strike a balance between maximum increases in total number of segments (so that some flows can get a best approximation quickly) and fair distribution of refinement to all segments (so that more Case III flows can get some value refinement).

For example, assume that a histogram has N segments, all need refinement. Furthermore, assume that an agent is due to send back $2N$ triplets in reply. Should the agent choose to evenly allocate 2 to each existing segment, then the manager gets a new histogram of 2N segments, with a moderate refinement on each flow value. However, should the agent choose to refine only one existing segment, assume it is possible, then after receiving $2N$ triplets from the agent the manager now has an updated histogram of $N + (2N - 1) = 3N - 1$ segments, with no refinement for other segments. One example of choosing a set of b_k is given in Section 4.

Figure 1(b) shows the result (of 15 segments total) after refining the first segment of the histogram in Figure 1(a) into 8 additional segments. Observe that in order to reconstruct the *refined* histogram, the manager only needs the 8 triplets (U_{1i}, L_{1i}, f_{1i}), $1 \leq i \leq 8$ from an agent.

3.2.4 Summary

Here is a recapitulation of each monitoring session in our proposed framework. To initiate a session, either the manager starts polling all agents or the agents periodically send their initial histogram.

1 Each agent uses the Histogram Construction Algorithm to generate N_1 segments and sends the results to the manager;
2 The manager applies the analysis outlined in Section 3.2.2 to identify a set of Case III flows;
3 **While** some Case III flows exist **do**
4 The manager decides whether to poll or to ask for refinement based on the Manager Selective Request Algorithm;
5 Each agent that receives a refinement request sends back a response based on the Agent Selective Refinement Algorithm;
6 The manager re-check the violation status of the Case III flows based on the refined data;
7 The manager **returns** all the flows being classified as Case I and Case VI, i.e., flows with SLA violations;

Since we use a triplet to encode each segment, there is intrinsic 50% overhead if we would have to report each data point as a segment versus as a (flow_id, value) pair.

Corollary 1 *Let the number of flows to be reported by an agent be M and the maximum number of segments reported at each round be N. In the worst case,* **ARM** *needs $(2M/N) - 1$ rounds to complete a session, and the total overhead is $3(2M - N)$.*

Here the worst case assumes that we evenly divide N to all segments that need refinement. When $M = N$, **ARM** finishes in one round, and the overhead is $3M$ compared to $2M$ of a naive method that reports (flow_id, value) pairs directly. As N becomes smaller, not only **ARM** needs more rounds to complete a session, the worst case overhead approaches $6M$. However, as we will illustrate in the next section, our experimental results show that **ARM** preforms much better on average. Due to information aggregation **ARM** even outperformed some ideal but unrealistic intelligent schemes that only report flows with SLA violation.

4 Experimental results

In order to evaluate the effectiveness of the proposed algorithm in monitoring the service performance of a network with QoS guarantees, we conducted extensive experiments in a simulated network domain. The result reported in this section addresses the following issues:

- the advantage of using the proposed monitoring algorithm in terms of overhead reduction;
- the effect of changing N_i, where N_i is the maximum number of new segments each agent reports at round i (cf. the Agent Selective Refinement Algorithm).

4.1 Testbed Setup

As a first step test, the experiments were carried out over a randomly generated 30 nodes topology shown in Fig. 2.

The topology is organized as a single three level hierarchy. The highest level is the core routers consisting of nodes 0, 1 and 2. The next level routers are nodes 6, 7, 12, 15, 16, 20, 23, 24 and 28. The rest are edge routers.

All links are duplex. The one-way bandwidth of each link depends on the type of routers it connects at both ends. It is 20 Mb/s for links connecting two core routers, 15 Mb/s for ones between a core and a next level router, and 10 Mb/s for the rest.

An on-off model is used to generate traffic with different average rate and burst size. Leaky bucket is used for policing at the edge routers. Input traffic is selected from the four classes listed in Table 1, which shows the leaky

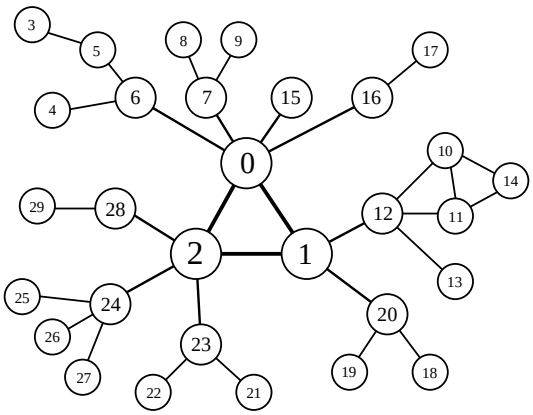


Figure 2. Simulation network topology

bucket parameters associated with each traffic class used in the simulations. All flows in the simulation have the same SLA, which allows average end-to-end delay of 150ms and loss ratio of 0.02. Within the network, packets are scheduled using the FIFO discipline.

	Class 1	Class 2	Class 3	Class 4
r (kb/s)	128	128	64	64
busy time (s)	0.3	0.5	0.3	0.5
idle time (s)	0.7	0.5	0.7	0.5

Table 1. Traffic parameters for the four classes used in the simulations

There is a local network management agent on each router and a centralized network manager. For simplicity, in our simulation we placed a management link between each node and the central network manager. Loss ratio and average delay were collected at each network node and samples of the statistics were reported to the manager periodically based on the algorithm presented in Section 3.

All experiments are performed using ns-2.

4.2 Random Load Generator

We developed a random load generator to generate network traffic with various loading conditions. The load generator mimics admission control procedures in practice. It runs a flow generation loop. For each iteration in the loop, it randomly selects two edge routers as the source-destination pair for a flow, and selects traffic class for the flow. The generator then attempts to "admit" the flow by securing its resources (in this case, bandwidth) along its route.

To create overload situations on some number of links, flows are admitted even when there is insufficient link bandwidth along portion of the path. Nevertheless, a list of links that have been "over-subscribed" is maintained. The flow generation loop terminates when the number of admitted flows is at least X and the number of over-subscribed links is more than Y.

The above steps result in reasonable traffic pattern variations, but only within a range of overload conditions. In order to generate a wide range of network load, where the number of flows violating their SLAs varies from none to almost entire set of flows, admission control is performed using the *virtual* bandwidth of each link. By sizing the virtual bandwidth of all links up and down by multiplying the actual link bandwidth by a constant factor, the generator now terminates when it has used up the virtual bandwidth from more than Y links (and admitted more than X flows, too).

The constant sizing factor reflects how willingly an ISP wants to risk SLA violations. The smaller the factor, the more conservative the admission control is, and the lesser SLA violations the network may observe.

We set $X = 1000$ and $Y = 8$ for all traffic loads generated in our experiments. The virtual bandwidth sizing factor is from 0.5 to 1.5.

4.3 Comparison of Monitoring Performance

The performance of our monitoring scheme is compared to 2 centralized off-line schemes which are expected to perform well. In both schemes, it is assumed that all flow status are known by a single *virtual* management agent and this virtual agent only sends to the network manager data relating to flows with SLA violations. In *scheme-1*, for each flow with SLA violation the virtual agent sends to the manager QoS data of the flow collected at all hops. In *scheme-2*, instead of sending data collected at all hops for those flows with SLA violations, only sufficient information is sent such that the manager can confirm their violation status. That is, if there are 3 hops and a significant loss is occurring only on a single congested link, then only the loss ratio on that link is sent. This is the minimum information required to identify a SLA violation without resorting to some form of aggregation.

Comparison is based on the total count of all data items sent from the agents to the manager. Each data item, regardless of its type, has a count of 1. For the idealized schemes, each update consists of 2 data items, one for the flow identifier and the other for the measured value. For **ARM**, the overhead for one update is 3 (maximum value, minimum value and flow identifier). In addition, a minimum overhead of 2 data items is incurred in all **ARM** message exchanges to indicate the number of delay and loss updates.

Let I be the set of flows with loss violation in a session, and J be the set of flows with delay violation in the same

session.

- For scheme-1, the count per session is $2 \times (\sum_{i \in I} Hop_i + \sum_{j \in J} Hop_j)$, where Hop_i is the hop count of flow i.
- For scheme-2, the count per session is $2 \times (\sum_{i \in I} Min_i + \sum_{j \in J} Min_j)$ where Min_i is the minimum number of hops to decide if violation occurs in flow i.
- For **ARM**, the count per session is $\sum_{r=1}^{R}(2 + \sum_{k=1}^{K}(3 \times (H_{rk}^{loss} + H_{rk}^{delay}) + 2 \times (P_{rk}^{loss} + P_{rk}^{delay})))$ where R = number of rounds, K = number of links, and, for each link k in round r, H_{rk}^{loss} = number of new loss histogram segments, H_{rk}^{delay} = number of new delay histogram segments, P_{rk}^{loss} = number of loss polling updates, and P_{rk}^{delay} = number of delay polling updates.

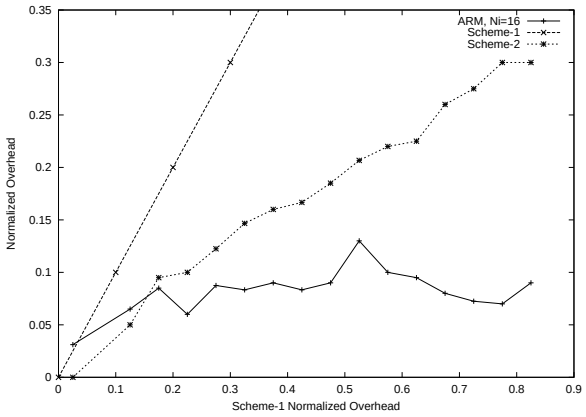
For comparison, a simple polling approach requires a total count of $4 \times (\sum_{i \in F} Hop_i)$, where F is the entire set of flows, and Hop_i is the hop count of flow i. Each entry consists of the two pairs (flow_id, loss value) and (flow_id, delay value). Note that if the manager and agents share the sorted list of flows, as we have assumed for **ARM**, a better polling approach will be to send only the sorted QoS data without flow identifiers. Nevertheless, it will only change the normalized values reported in our experimental results, but not the relative measures between **ARM** and the other two idealized schemes.

In all the experiments, b_k in Step 1 of the **Agent Selective Refinement Algorithm** described in Section 3.2.3 is chosen as follows.

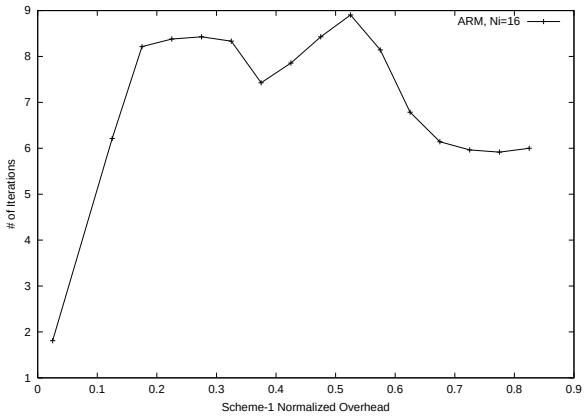
We first assign b_k to each segment s_k, where $b_k = \max(\lfloor |s_k|/4 \rfloor, 2)$. If $\sum_{k=1}^{|RS|} b_k > N_i$, we randomly decrease some b_k so as to comply with the restriction that $\sum_{k=1}^{|RS|} b_k \leq N_i$. In any case, b_k could be reduced to 0 (i.e., no refinement on segment s_k), but must not be set to 1. On the other hand, if $\sum_{k=1}^{|RS|} b_k < N_i$, we randomly increase some b_k till the sum equals N_i.

All experiments ran for 100 seconds simulation time excluding a 5 seconds warmup time. The performance of various algorithms are evaluated by running each algorithm 50 times using different traffic loads generated by the random load generator. The minimum number of flows in an experiment is 1000, the maximum is 1864 and the average is 1306. The minimum total data item count using simple polling in an experiment is 12944, the maximum is 25868 and the average is 18344.

In the first experiment, the parameter N_i is fixed at 16 for the entire simulation run. N_{poll} is fixed at 32 for all experiments.



(a) Monitoring overhead incurred



(b) Number of Iterations Required for **ARM** for N_i = 16

Figure 3. Performance Comparison between Scheme-1, Scheme-2 and ARM

The measurement overhead of all three schemes are normalized by dividing the monitoring data item count by the total count required in a simple polling approach. That is, if the count for simple polling is 1000 and the count for **ARM** is 100, the normalized overhead for **ARM** is $100/1000 = 0.1$.

Figure 3 shows the performance of Scheme-1, Scheme-2 and **ARM** relative to that of Scheme-1 for the same traffic load. The x-axis is the normalized Scheme-1 overhead, and the y-axis is the respective normalized overhead of Scheme-1, Scheme-2, and **ARM**. The choice of Scheme-1 normalized overhead for the x-axis serves as an indication of the number of SLA violations in the network, though the relationship is not exact because the monitoring overhead also depends on the number of hops the flows go through. To make the data easier to read, for each scheme we display only the mean value of the data within each 0.05 segment along the x-axis. In other words, the value depicted at

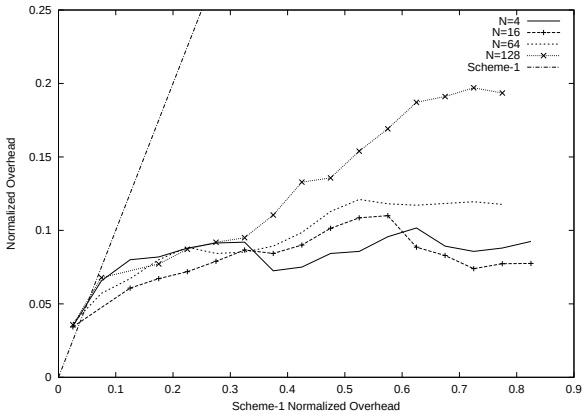
$x + .025$ corresponds to the mean of all values collected within the segment x to $x + .05$. As an example, in Figure 3(a) the set of traffic loads that generates average normalized overhead between 0.45 to 0.5 using Scheme-1 generates average normalized overhead of 0.09 using **ARM** with N_i = 16.

When there is no SLA violation, **ARM** incurred a minimum normalized overhead of 0.02 whereas Scheme-1 and Scheme-2 have no overhead. However, as the number of SLA violations increases, normalized overhead for **ARM** increases slowly and performs better than Scheme-1 for normalized overhead larger than 0.06. Beyond normalized overhead of 0.15, **ARM** performs even better than Scheme-2. This may come as a surprise since Scheme-1 and Scheme-2 are highly optimized schemes with very low redundant information exchanged. The difference is that in these two cases, exact values are exchanged. On the other hand, **ARM** provides only bounds on these values and can thus aggregate many values into a single segment. Another advantage of **ARM** is that as the number of violations increases, the normalized overhead does not increase linearly with the number of violations. It is due to the fact that once the lower bound of the QoS values violates the SLA, the computation can terminate and there is no need to obtain the actual values.

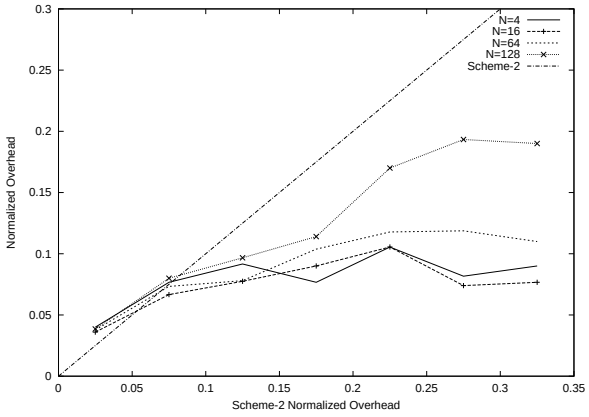
Figure 3(b) shows the average number of iterations it takes before **ARM** terminates using the same x-axis segments and y-axis averages. When the number of violations is small, it takes much longer to detect all violations because it is harder to aggregate values and a much finer picture of the network is needed before SLA validation can be completed. However, as the number of violations increases, it becomes easier to detect violations as aggregation of *similar* values becomes more common.

Figure 4(a) shows the improvement of **ARM** over Scheme-1 and Figure 4(b) shows the the improvement of **ARM** over Scheme-2 for N_i set to 4, 16, 64 and 128. Figure 5(a) shows the average number of rounds for N_i set to 4, 16, 64 and 128. In these experiments, N_i is fixed during a single simulation run. The same segment size of 0.05 is used on the x-axis, so is the segment mean value on the y-axis.

Figure 4(a) and 4(b) show that the overhead incurred by **ARM** increases with N_i. This is because when N_i is large, the number of measurements collected in each round may be much more than what is needed. The extent of such excessive measurement increases as N_i gets larger. The average overhead incurred when N_i = 128 is about twice the amount of overhead incurred when N_i = 16. A smaller N_i is more efficient. On the other hand, when N_i is too small, the amount of new information collected in each round may be too little and many rounds are needed before the algorithm can terminate. In Figure 5(a), the average number of



(a) Overhead incurred by Scheme-1 and **ARM**



(b) Overhead incurred by Scheme-2 and **ARM**

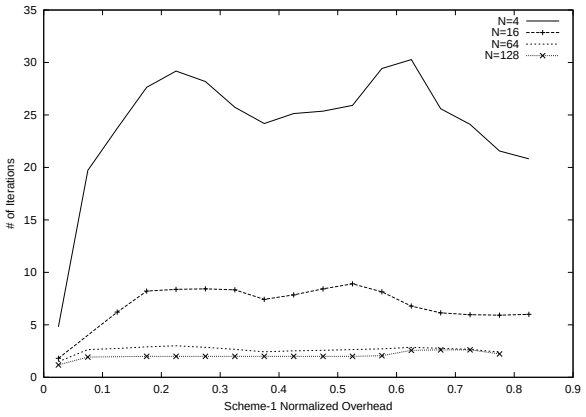
Figure 4. Comparison of Scheme-1, Scheme-2 and ARM for N_i = 4,16,64 and 128

rounds required when N_i = 4 is about 10 times that required when N_i = 128. A larger N_i can thus lead to much shorter termination time.

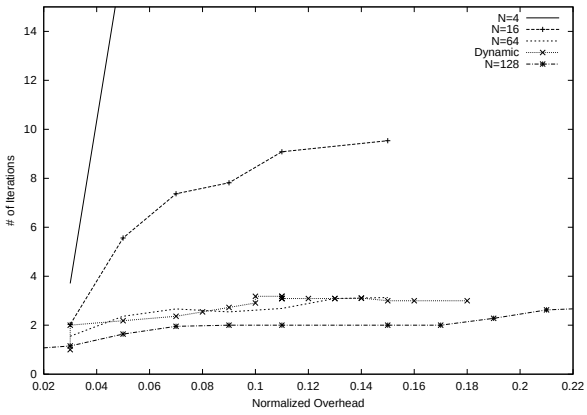
The results can be summarized as follow:

- **ARM** performs better than Scheme-1 and Scheme-2 in most cases except where the number of violations is very low. This is true for all N_i shown.
- A small N_i reduces the normalized overhead but increases the number of round required. The reverse is also true.

The trade-off in our scheme is between data collection overhead and termination time. Figure 5(b) shows a plot of the normalized overhead vs. number of round for various values of N_i that clearly illustrates this trade-off. The figure indicates that decreasing N_i decreases the basic overhead of the algorithm but at the same time increases the number of

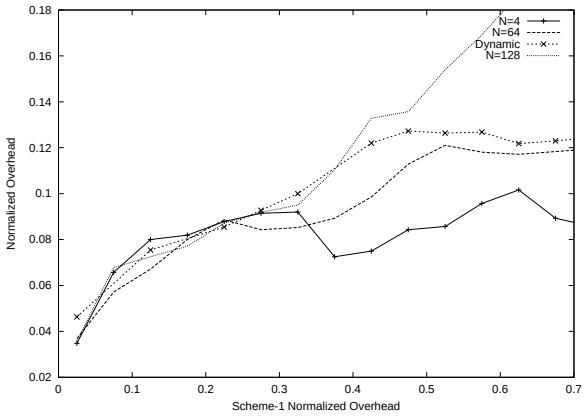


(a) Number of Rounds Required for **ARM**

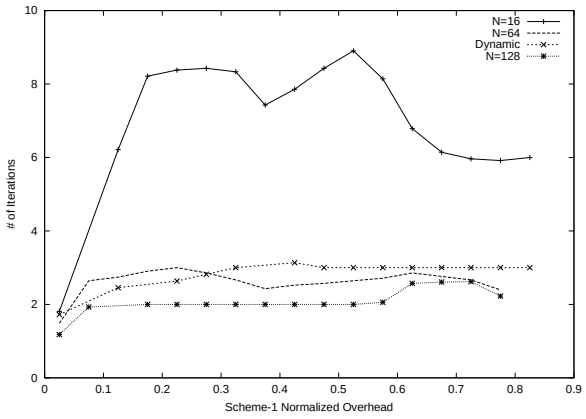


(b) Trade-Off of Overhead vs Convergence Time

Figure 5. Performance of ARM and Dynamic ARM for N_i = 4,16,64 and 128



(a) Overhead Comparison



(b) Number of Rounds Comparison

Figure 6. Performance Comparison of Dynamic ARM and ARM for N_i = 4,16,64 and 128

rounds it takes for the algorithm to converge. On the other hand, increasing N_i to 128 keeps the number of rounds to a very small value but increases the normalized overhead. In addition, since more data are sent in a single cycle, a larger N_i increases the load at the network manager. Thus, N_i should not be set beyond some threshold in order to avoid degenerating **ARM** into a simple polling scheme.

Given the trade-off characteristics of **ARM**, a *dynamic* version of **ARM** is evaluated. In dynamic **ARM**, a small N_i of 16 is used in the first round in order to reduce the risk of *over-sampling*. If a second round is required, N_i is increased to 32. If a third or subsequent round is required, N_i is set to 128 so that the algorithm will terminate quickly. The performance of this dynamic **ARM** is shown in Figure 5(b) and Figure 6.

The results show that dynamic **ARM** performs much better than **ARM** with N_i = 16 and N_i = 128 in terms of the combine performance of iteration and overhead. Dynamic **ARM** is similar in performance to N_i = 64. Compare to **ARM** with N_i = 64, dynamic **ARM** terminates faster when the overload is low to moderate due to the use of a large N_i in later rounds. However, when the overload is high, it tends to over-sample and incurs a higher overhead.

Before concluding this section, it is important to point out that while the performance of **ARM** is fairly robust over a wide range of traffic load, the quantitative result of **ARM** may change if the total number of flows in the network increases by more than an order of magnitude. Hence, if the number of flows is much larger, larger N_{poll} and N_i may be more appropriate than the values given in this section.

5 Discussion and Related Work

The term, service level agreement, is still subject to definition. Not only the choice of QoS parameters is a fre-

quent debate, the ways that these parameters are measured or interpreted can vary significantly as well. For example, different lengths of measurement intervals could influence even a simple term such as an average delay or a connection availability. While a precise description of each QoS parameter is important to an SLA, the issue is orthogonal to our work. In our simulation experiments, we collect QoS values over a range of 100-200 seconds. For networks in operation the interval could be in the order of minutes. To get an accurate end-to-end measurement the agent would have to synchronize their clocks to within seconds, especially if data collection is triggered periodically.

The collection of performance data serves many purposes. This work focused on the detection of SLA violation, which does not need to know the exact performance of every single flow. Hence, we can apply the aggregation technique to achieve tremendous savings in information exchange. The same aggregation technique will not apply to usage accounting, for example, where the QoS of every flow must be evaluated individually. However, monitoring for detection and control occurs at a shorter time scale and requires quicker reactions. For accounting purpose service providers can transfer measurement data to log servers for off-line processing.

Network monitoring is an essential management requirement and much effort has been devoted to providing a unified monitoring framework including common protocols for fetching information, syntax for defining monitoring information and management information.

The most popular protocols for network monitoring are the IETF Simple Network Management Protocol (SNMP) ([1], [2]) and the ISO Common Management Information Protocol (CMIP). Many management Information Bases (MIBs) have been defined, including the Remote Network Monitoring Management Information Base (RMON MIB) ([12], [13]). RMON provides significant expansion in SNMP functionality, including support for off-line operations, more sophisticated data processing and multiple managers. A drawback with these MIBs is that they describe the information available on individual devices and also tend to be fairly low levels and focuses on counters for hardware statistics and errors. A recent development is the definition of a MIB module for performance management of Service Level Agreements [14]. It is assumed that SLA is defined via policy schema definitions and the MIB defines statistics related to a policy rule definition.

Given that the amount of management information in a large network can be very large and the inefficiency of continuous polling, a number of proposals have been made to reduce the management information overload. A common approach to reducing monitoring overhead is to vary the polling frequencies base on the state and characteristics of variable being monitored. In [4], two algorithms are proposed for changing polling frequencies, either based on past history or based on how close a measured value is to a threshold. In [15], Discrete Fourier Transform is applied to each sequence of collected data and the polling frequency is selected as 2 times the largest frequency if the management bandwidth is sufficiently large. In [7], the authors presented a model in which the behavior of the network states is captured by state transition diagrams. They showed that a greedy algorithm that delays measurement as much as possible is correct and optimal.

In [8], the amount of information to be collected is reduced by only collecting information that is required to satisfy the objective of monitoring. For example, if the end-to-end delay of a specific path is required, then only performance data of delay along the specific path will be collected. An inference engine is used to map a request to the individual measurement components.

End-to-end measurements per SLA flow is ideal for deciding if a flow meets its SLA. A large scale end-to-end measurement of packet dynamics over the Internet can be found in [10]. A discussion of using Operation and Management (OAM) cells to measure end-to-end performance over a ATM network can be found in [3]. While such measurements are appropriate for determining the end-to-end Quality of Service, there are two potential problems. First, the number of measurements taken is equal to the number of flows with SLA and may not be scalable for a large network. In addition, when problems are detected, locating the congestion links is not straightforward. Additional measurements in the core of the network are still needed. It is precisely these problems that motivated our work.

Finally, the IETF IP Performance Metrics (IPPM) Working Group has attempted to develop a set of standard metrics that can be applied to the quality, performance, and reliability of Internet delivery services. For more details, refer to [11].

6 Conclusion

We have presented a monitoring framework to address the scalability in detecting SLA violations. The monitoring is based on hop-by-hop measurement of QoS values, aiming at deriving qualitative status of flows and links, i.e., finding which flows have SLA violations and which links are having long delay or high loss rate. With a dynamic histogram-based data aggregation technique and an iterative refinement process, the proposed framework, **ARM** achieved substantial reduction in overhead and scaled well over a wide range of traffic loads. **ARM** constantly remains at 10% overhead compared to a simple polling monitoring scheme, and often outperforms two other schemes that represent the most optimized but un-implementable approaches without data aggregation.

Two future directions are of immediate interests to us. We plan to look at the monitoring issues with routers that employ more sophisticated queuing mechanisms such as WFQ or support differentiated services. We also plan to look into ways of using the monitoring results to trigger management actions. In particular, we can easily extend **ARM** to identify not only flows that violate their SLAs, but also those that receive significantly better services than what their SLA stated. Based on such a monitoring tool we plan to develop an SLA management application to adjust provisioning among these flows. After all, it is a provider's best interest to utilize available resources to satisfy as many SLA flows as possible.

References

[1] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (SNMP). *IETF RFC 1157*, May 1990.

[2] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Protocol operations for version 2 of the simple network management protocol (SNMPv2). *IETF RFC 1905*, Jan 1996.

[3] T. Chen, S. Liu, M. J. Procanik, D. Wang, and D. Casey. INQUIRE: A software approach to monitoring QoS in ATM networks. *IEEE Network*, pages 32–37, March/April 1998.

[4] P. Dini, G. V. Bochmann, T. Koch, and B. Kramer. Agent based management of distributed systems with variable polling frequency policies. In *Proceedings of Integrated Network Management V*, pages 553–564, San Diego, California, May 1997. IFIP.

[5] A. Elwalid and D. Mitra. Design of generalized processor sharing schedulers which statistically muiliplex heterogeneous QoS classes. In *Proceedings of the IEEE INFOCOM*, pages 1220–1230, New York, NY, March 1999.

[6] M. Grossglauser and D. N. C. Tse. A time-scale decomposition approach to measurement-based admission control. In *Proceedings of the IEEE INFOCOM*, pages 1539–1547, New York, NY, March 1999.

[7] J. Jiao, S. Naqvi, D. Raz, and B. Sugla. Minimizing the monitoring cost in network managemment. In *Proceedings of Integrated Network Management VI*, pages 155–170, San Diego, California, May 1999. IFIP.

[8] S. Mazumdar and A. Lazar. Objective-driven monitoring for broadband networks. *IEEE Transaction on Knowledge and Data Engineering*, 8(3), Jun 1996.

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

[10] V. Paxson. End-to-end internet packet dynamics. In *Proceedings of ACM SIGCOMM'97*, pages 139–152, Cannes, France, Sep 1997. ACM.

[11] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. *IETF RFC 2330*, May 1998.

[12] S. Waldbusser. Remote network monitoring management information base. *IETF RFC 1757*, Feb 1995.

[13] S. Waldbusser. Remoate network monitoring management information base version 2 using smiv2. *IETF RFC 2021*, Jan 1997.

[14] K. White. Definitions of managed objects for service level agreements performance monitoring. *IETF RFC 2758*, Feb 2000.

[15] K. Yoshihara, K. Sugiyama, H. Horiuchi, and S. Obana. Dynamic polling scheme based on time variation of network management infomation values. In *Proceedings of Integrated Network Management VI*, pages 141–154, San Diego, California, May 1999. IFIP.