# Goodput Control for Heterogeneous Data Streams

Jia-Ru Li  Xia Gao  Leiming Qian  Vaduvur Bharghavan
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{*juru, xiagao, bharghav*}*@timely.crhc.uiuc.edu, lqian@uiuc.edu*

**Abstract**

Congestion control tries to answer the question: "At what rate should a sender transmit data under current network conditions?" While answering this question is sufficient to maximize the goodput of traditional bulk data streams, emerging multimedia applications generate heterogeneous data streams where different frames have different quality of service requirements in terms of priority, deadlines, and inter-frame dependence. Consequently, the goodput of a multimedia stream depends not only on answering the above question, but also the question "Which packets should the sender transmit given its current transmission rate?" In this paper, we make the case that the "goodput control" mechanisms that answer this question should be considered as a critical component of future multimedia transport protocols.

We present an objective definition of goodput at the transport layer, and show that optimizing this goodput function has exponential complexity in the general case. We then present a set of simple online packet dropping algorithms that can be used at the sender side in order to approximate the optimum within a bounded ratio, and show that our goodput control mechanisms also help to improve application level reception quality for the flow.

## 1 Introduction

Emerging multimedia applications generate data streams that have significantly more complex characteristics than traditional bulk data applications. Consider a multimedia streaming application that requires the transmission of an MPEG video stream from a server to a client. The server and client exchange control packets that must be delivered reliably, and the MPEG video stream contains I-, P-, and B-frames that are prioritized and possibly deadline bounded. Further, there are dependencies across the frames: a P-frame depends on the preceding I-frame, while a B-frame depends on the preceding and following non B-frames. To summarize, multimedia applications generate *heterogeneous data streams* wherein frames/packets have diverse quality of service (QoS) requirements in terms of reliability, deadline, utility, and inter-frame dependency.

What are the special requirements that heterogeneous data streams impose on the underlying data transport mechanisms? In traditional bulk data transfer applications such as `ftp`, the transport must provide rate control and support reliable and sequenced delivery of packets – all packets are equally important, and the timeliness of delivery is not critical. Therefore the transport protocol must determine the sustainable sending rate for a connection (i.e., perform rate control), and then send packets in sequence while making sure that lost packets are retransmitted (i.e., enforce reliability). However, for the heterogeneous data streams under consideration, timeliness of data delivery may be critical – packets that arrive after their deadline are useless at the receiver. Therefore, the data transport mechanisms must perform rate control as before, but send only the "most useful" packets that can be delivered at the sustainable sending rate. In other words, *the transport must transmit the highest utility packets that satisfy their deadline and dependency requirements in order to maximize the "goodput" (or broadly speaking, aggregate utility) of the heterogeneous data stream at the receiver.*

Comparing the requirements of "homogeneous" and "heterogeneous" data streams, we see that the data transport mechanisms must consider two questions:

1. *At what rate* must the sender transmit packets given the current network conditions? This is the standard rate control problem, and has been extensively addressed for both reliable and unreliable data streams in related literature [3, 5, 7, 10]. In this work, we use a rate control mechanism that is TCP-friendly and stable, similar to the work in [5], and we will not address this issue further.

2. *Which packets* should the sender transmit, given the current transmission rate and packets in the send buffer? While the answer to this question is straightforward for homogeneous data streams (transmit the next packet in sequence), it is non-trivial for heterogeneous data streams because it depends on the relative values of the application sending rate and the transport sending rate, the QoS requirements of the packets in the buffer, and which packets have already been transmitted (in order to take into account the dependency requirements across frames). In short, different packets in the same stream have different utilities, and the transport needs to implement a set of mechanisms for selecting packets for transmission such that the aggregate utility of the received packets at the receiver is maximized.

   We call this set of mechanisms *goodput control* in this paper, and make the case that data transport for multimedia streams must provide goodput control as an integral part of the systems support for multimedia application designers[1].

It is now widely accepted that rate control must be a part of transport protocols designed to support both reliable and unreliable flows . The current approach for multimedia system design calls for multimedia applications to sit on top of rate-controlled unreliable transport, receive rate and delay feedback periodically, and then adapt within the application to the dynamics of the connection.

In this paper, we consider a different model. We concur with conventional wisdom that the application best knows the QoS metrics of its frames. Thus, we believe that the responsibility of assigning QoS requirements in terms of reliability, utility, deadline, and dependency constraints must remain with the application. On the other hand, once the QoS requirements of a frame are specified by the application, we argue that the "goodput control" mechanisms which maximize the perceived utility of the stream at the receiver can be best implemented in the data transport/middleware outside of the application. This approach provides for a clean separation between application-specific QoS policies (which are set and controlled by the application) and the general mechanisms that are used to implement these QoS policies (which are provided by the data transport).

*In this paper, we make the case for considering goodput control as a fundamental component of multimedia transport* as opposed to a part of the application, motivated by three reasons. First, it makes writing multimedia applications much simpler, because the application designer only needs to think about the policy-level issues and not the mechanisms for achieving effective adaptation to rate/delay fluctuations. Second, it makes the modeling of, and structured reasoning about, goodput control easier if these mechanisms constitute a separate component and are not wrapped into the application. Third, fine-grained interaction between the rate control and goodput control can improve the aggregate utility of the transmitted stream under network dynamics. Motivated by both the efficacy of the mechanisms and the convenience of writing applications, we argue for the approach wherein the application sends a heterogeneous data stream at its "desired sending rate" with per-frame QoS specifications, and the transport provides the mechanisms for rate control and goodput control to maximize the aggregate utility of the heterogeneous data stream. Our initial experimentation shows that maximizing the objective goodput control metric at the transport level also leads to improved perceptual quality of the data stream at the application level.

[1] For much of this paper, we will not distinguish between the transport layer and the middleware that sits between the transport and the application, generically using the term "data transport mechanisms" to refer to both. In Section (), we will discuss trade-offs between placing the goodput control mechanisms within the transport protocol as opposed to the middleware.

The rest of the paper is organized as follows. Section 2 presents the network model and the framework for goodput control. Section 3 describes the model for optimal goodput control and a set of simple buffer management mechanisms that approximate the optimal model. Section 4 evaluates the goodput control mechanisms for three different types of application-level coding schemes: motion JPEG, MPEG, and 3DSPIHT. Section 5 discusses some unresolved issues and concludes the paper.

## 2 Model and Framework

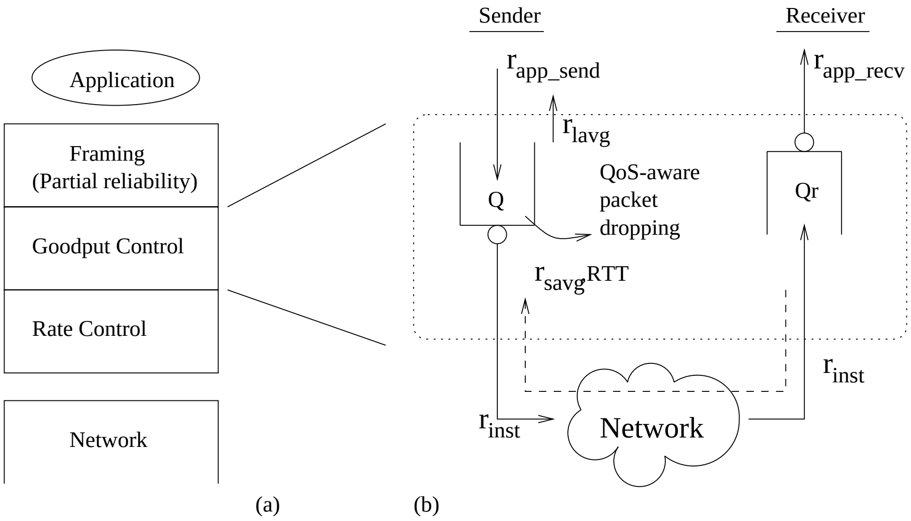


Figure 1: Framework for Goodput Control. Figure (a) positions goodput control in the protocol stack. Figure (b) shows the key components of goodput control. The goodput control mechanisms described in this paper provide QoS-aware packet dropping in the sender side buffer.

Figure 1.a positions goodput control in context. As a part of neneric data transport, we consider three functions: framing, goodput control, and rate control. The application deals with frames and frame-specific QoS policies. The framing component provides frame-to-packet segmentation/reassembly, translation of frame-to-packet QoS, and partial reliability. The rate control component provides estimates of the rate and delay for the connection. Goodput control bridges the potential rate mismatch between the application sending rate and the transport sending rate by determining which packets to send and which packets to drop at the source. The first order of business is to determine what the goodput control mechanisms are, and how they can effectively use the application-specified QoS parameters of frames/packets in order to maximized the aggregate received utility at the receiver. The second order of business is to determine how to structure the three components, i.e. which components belong to the transport layer and which belong to middleware. For simplicity of discussion, we will initially assume that goodput control is a part of a transport protocol such as HPF [6], and then revisit the structural trade-offs in Section 5.

In the rest of this work, we will assume the following: (a) a higher layer already has performed the frame-to-packet translation (so the goodput control component deals only with packets) and (b) a lower layer performs rate control and provides the goodput control component with short term and long term running averages of rate and round trip time estimates. In all our simulations and analytical evaluation, we use the standard additive increase-multiplicative decrease rate control algorithm, similar to the implementation in RAP [5]. Figure 1.b. shows the three aspects to goodput control: buffer management at the sender, rate and delay feedback, and buffer management at the receiver. Packets flow into the sender buffer at a rate of $r_{app\_send}$, and are drained at a rate of $r_{inst}$[2]. The rate control component provides the

[2] All rates are time dependent, though we drop the time dependence for convenience of notation.

sender side buffer manager with the short-term average rate $r_{savg}$, the long-term average rate $r_{lavg}$, and the smoothed round trip time $rtt$. If the application is adaptive, then it may periodically probe the lower layer for $r_{lavg}$ and adjust $r_{app\_send}$ accordingly. The buffer size at the sender is $B$. At the receiver, packets arrive into the receiver buffer and are drained from the receiver buffer at a rate of $r_{app\_recv}$. Our focus is on the set of packet dropping mechanisms at the sender buffer that bridge the potential rate mismatch between the application and the network in a way that maximizes the goodput of the flow, which we now formally describe below.

## 2.1 Conditional Utility and the Goodput Control Objective

Let us consider a multimedia application that generates a multi-resolution image/video stream.Eeach packet $i$ is associated with a "utility" $u(i)$ and length $l(i)$. For applications with real-time constraints, each packet may be associated with a "deadline" $d(i)$ by which it must be delivered. Finally, there are dependencies between data packets. In other words, a packet $i$ is useful at the receiver only if the receiver also receives the set of packets $D(i)$. We abstract these QoS characteristics of a packet by means of a *conditional utility* $v(i)$, defined as follows:

$$v(i) = \begin{cases} 0 & \exists \text{ packet j} \in D(i), \text{ j is not received at the receiver} \\ 0 & \text{packet } i \text{ is predicted to miss its deadline } d(i) \\ u(i) & \text{otherwise} \end{cases}$$

Conditional utility captures the requirements that a packet must be received before its deadline[3], and that all the packets on which it depends must also be received. Given this definition of conditional utility, we define the "goodput" $G[t1, t2]$ of a heterogeneous data stream over a time window $[t1, t2]$ as

$$G = \sum_{i \in S[t1,t2]} v(i), \tag{1}$$

where $S[t1, t2]$ is the sequence of packets that is transmitted in the time window. The goodput control problem is thus an optimization problem: *maximize* $G[t1, t2]$ *for a desired time window* $[t1, t2]$ *such that*

$$\sum_{i \in S[t1,t2]} l(i) \leq \sum_{t=t1}^{t2} r(t) * \Delta t \tag{2}$$

*where the rate* $r(t)$ *adapts in discrete time intervals of* $\Delta t$*, a packet is eligible for transmission only after it arrives at the send buffer, and the sequence* $S[t1, t2]$ *is a subsequence of the sequence of transmissions from the application layer.* In essence, the goal is to choose the sequence of transmissions $S[t1, t2]$ in a way that maximizes the aggregate conditional utility at the receiver. Note that in this problem formulation, we do not consider losses in the network in the ideal model (i.e. every packet that is selected for transmission is optimistically expected to be received at its destination)[4]. Of course, in the practical instantiations of the mechanisms and in evaluation, we do not make this assumption.

Let us now consider how the utility and dependency relations are specified by the application. $u$ is an application-specific utility function, and $D$ is an application-specific dependency relationship between frames. The precise mapping between the application-level perceptual usefulness of a frame and its utility assignment is beyond the scope of the paper. However, in the evaluation section, we use the PSNR metric for assigning per-frame utility. Dependency across frames is inherent to the structure of the coding

[3] Decision is made on the sender side based on the estimated RTT and estimated flow rate. The deviation of the actual delay from estimated RTT is ignored in the theoretical analysis but considered in the simulation

[4] While it is possible to account for network losses using a multiple description-based forward error correction below the goodput control layer, in this work we assume an idealized network model that does not drop packets so long as the sender does not violate its rate estimate.

scheme which is faithfully reflected in the inter-packet dependency. We consider three coding schemes in our evaluation to demonstrate the generality of the goodput control work: Motion JPEG, MPEG, and 3D-SPIHT. We see in our evaluation that maximizing the objective goodput function in the transport also generally achieves higher PSNR on the receiver side and the perceptual quality of the stream in the application is also improved.

# 3 Goodput Control Mechanisms

In Section 2, we presented a high-level overview of the optimization criterion, viz. *select* $S[t1, t2]$ *to maximize* $\sum_{i \in S[t1,t2]} v(i)$ *such that the transmission of the packets in S does not violate the rate bound of the connection.* In this section, we first explore the detailed ideal model of goodput control; then we show that goodput maximization with deadline and dependency constraints is exponential in nature; then we investigate a simple greedy solution that achieves the best known competitive ratio with respect to the optimum. The result of our work is that we propose a very simple set of online packet dropping mechanisms that effectively approximate goodput optimization.

## 3.1 Ideal Goodput Control

Each packet $i$ has a utility $u(i)$, length $l(i)$, deadline $d(i)$, and dependency relationship $D(i)$, where $D(i)$ is a set of preceding packets on which it depends. Let us first consider a simple offline version of the goodput control problem wherein the task is to determine at time $t1$ which packets from the queue to select for transmission in a time window $[t1, t2]$, with fixed connection rate of $r$ at time in $[t1, t2]$ and no further arrivals after $t1$. First, let us simplify the problem further, and assume that $\forall i,\ \ d(i) = \infty$ and $D(i) = \Phi$. Then this problem reduces to the *0-1 Knapsack problem*, and has a well known dynamic programming solution in $O(N_Q * r * (t2 - t1))$ time where $N_Q$ is the number of packets in the sender buffer $Q$. The key point to note is that the 0-1 Knapsack problem satisfies the *optimal substructure property* (i.e. partial solutions of the optimal solution are also optimal) – hence, dynamic programming is a valid approach for solving this problem. Let us now generalize this approach to account for deadlines and dependencies.

We define the recursive relation for the goodput control problem as follows:

$$G[Q] = max_j\{v_{S(Q-\{j^*\})}(j) + G[Q - \{j\}]\} \tag{3}$$

$$j^* = arg\ max_j\{v_{S(Q-\{j^*\})}(j) + G[Q - \{j\}]\} \tag{4}$$

$$S(Q) = \{j^*\} \cup S(Q - \{j^*\}) \tag{5}$$

$$\text{feasible } S(Q - \{j^*\}) \Rightarrow \text{feasible } S(Q) \tag{6}$$

where $G[Q]$ is the optimal goodput considering the set of packets $Q$, $S(Q)$ is the subsequence of packets selected for transmission from the set of packets $Q$, and $v_S(i)$ is the conditional utility of packet $i$ given that a sequence $S$ has already been selected for transmission.

Essentially, the above dynamic programming problem setup yields a correct solution if the recursive relationship for $G[Q]$ and the feasibility condition for $S(Q)$ hold. Under such circumstances, we can obtain a simple polynomial time optimal solution for goodput.

Unfortunately, when we introduce deadlines, then the feasibility condition may be violated because adding a packet $j^*$ to the transmission sequence in any iteration of the solution may violate the deadline of some previously chosen packet in $S(Q - \{j^*\})$ that follows $j^*$ in the sequence of transmissions.

Further, when we introduce dependencies, the optimal substructure property may be violated because $S(Q - \{j\})$ may depend on whether $j$ is selected or not in $S(Q)$. In other words, adding a packet to the transmission sequence may change the optimal transmission subsequence chosen thus far, because it can enable the transmission of other packets that were not hitherto considered.

To summarize, the optimal substructure property is violated when we introduce both deadline and dependency constraints, and consequently the solution moves from a polynomial time to an exponential time computation. We will discuss this issue more precisely in the final paper.

### 3.2 Goodput Control Mechanisms under Some Simplifications

Given the fact that the goodput maximization algorithm is exponential without restrictions on how the $u(i)$, $l(i)$, $d(i)$ and $D(i)$ are specified, we investigate what the time complexity of the goodput maximization algorithm would be if the $u(i)$, $l(i)$, $d(i)$ and $D(i)$ observe some simple relations.

We represent the packet dependency via a the digraph $Gr = (V, E)$ the nodes in set $V$ are all the packets in the queue $Q$. Ordered pair $(i, j) \in E$ if packet $j$ depends on packet $i$. Packet $i$ depends on packet $l$ explicitly or implicitly if there exists a path from $l$ to $i$ and no such $k$ such that $(k, i) \in E$ and $(j, k) \in E$. Let $P(i)$ be the set of nodes which depends on node $i$ explicitly or implicitly.

Let the "utility per bit" of packet $i$ be $w(i)$, $w(i) = u(i)/l(i)$.

**Proposition 3.1** *If packet $i$ has $w(i)$ larger than that of any packets depending on itself, then*

$$w(i) > \frac{\sum_{k \in SP(i)} u(k)}{\sum_{k \in SP(i)} l(k)} \tag{7}$$

*the average utility per bit for any subset $SP(i)$ of $P(i)$.*

If the deadline $d(i)$ is not $\infty$ but can be any arbitrary value, the feasible condition of $S(Q)$ is not guaranteed even with the assumption of Proposition 3.1. Hence the time complexity of the optimal solution is still exponential. We has considered heuristic approximation in the following section.

### 3.3 Practical Goodput Control Mechanisms - Selecting Algorithm

In this section we proposed a greedy algorithm to approximate the optimal result whose output has the lower bound of $\log(\frac{maxU}{minU} * \frac{maxl}{minl})$.[5]. The algorithm has no assumption about the utility function and deadline. Step 1: Let Set $I \leftarrow$ {all the packets which don't depend on other packets}.
Step 2: Pick the packet $P(i)$ in $I$ with largest bit utility $w(i)$ and satisfy the deadline constrict.
Step 3: Add the packets whose dependent packet are already selected to the set $I$.
Step 4: Remove the packets which miss the deadline and all their dependency from $I$.
Step 5: If $I$ is empty, the algorithm terminates. Otherwise it goes to Step 2.

The idea is always to pick the packet with largest bit utility $w(i)$ under the constraints that it is within its deadline bound and its transmission won't make the packets already picked miss the deadline. Those packets already picked have higher $w(i)$ than itself.

### 3.4 Practical Goodput Control Mechanisms - Dropping Algorithm

While we have the selecting algorithm for the goodput, this is not good for the online algorithm since it assume no further arrival. Here we propose an online dropping algorithm which is equivalent to the

[5] We will give the proof in the final paper

previous algorithm in the online case but is more amenable to online adaptation. We also prove in next section that the output sequence of the two algorithm are the same for the offline scenario. We consider a suite of 4 packet dropping algorithms here.

1. Discard packets that are not "useful" to the receiver.
   (a) *Deadline drop*: Discard a packet that is predicted to violate its deadline by the time it is received at the receiver.
   (b) *Dependency drop*: Discard a packet for which the sender has already discarded some packet on which it depends (e.g. discard a B-frame packet whose preceding P-frame has been dropped).
2. Preferentially discard lower utility packets in favor of sending higher utility packets.
   (a) *Utility drop*: When the sender buffer is full, discard queued packets with lower normalized utility in favor of incoming packets with higher normalized utility, where we define "normalized utility" of packet $i$ as $u(i)/l(i)$ (i.e. utility per-bit).
   (b) *Anticipated deadline drop*: If a packet with higher normalized utility is predicted to miss its deadline, but discarding one or more packets with lower normalized utility preceding it in the sender buffer will enable the packet to meet its deadline, then discard those lower normalized utility packets.

### 3.5 The equivalence of selecting and dropping algorithm

As we have seen above, the selecting algorithm is used to derive the lower bound of the goodput but the we propose to use the dropping algorithm in practice because of its simplicity. Here we will prove in fact, these two algorithms have the same output sequence for offline operation. Hence the dropping algorithm has the same competitive ratio.

**Proposition 3.2** *For any given queue, the dropping algorithm and the selecting algorithm have the same output sequence.*

Proof: (by induction).
*step 1*: Considering the packet $P_1$ with the highest bit utility $w(1)$ in the queue, the selecting algorithm selects it first and gurantees it can be transmitted by checking the lately selected packet not to make this packet miss the deadline. The dropping algorithm scan the queue and when it scans $P_1$ it will drop some packets before $P_1$ if necessary because $P_1$ has the highest bit utility. The only case that $P_1$ can't be included in the output sequence is that even $P_1$ being sent immediately it will still miss the deadline. In this case $P_1$ can't be included in both case. Then the packet with the second highest bit utility is considered and either all the packets in the queue miss the deadline or the same packet in the time bound with highest bit utility is included in the output sequence of both algorithms.
*step 2*: Assuming the packets of 1st to kth highest bit utility of the output sequence are the same for the two algorithm, we claim the $(k+1)$th highest packet are also the same. Considering the $(k+1)$th highest packet $P^s_{k+1}$ of the selecting algorithm, any packets in the queue with bit utility function between $P^s_{k+1}$ and $\{P_1, ..., P_k\}$ can't be selected because they will make at least one packet of $\{P_1, ..., P_k\}$ miss the deadline. The dropping algorithm also has to satisfy this constaint so it can't do better than selecting algorith. Furthermore when scanning $P^s_{k+1}$ it will drop some packets with lower bit utility before $P^s_{k+1}$ if necessary to make $P^s_{k+1}$ to be within the deadline. This will succeed because even when all the $\{P_1, ..., P_k\}$ precede $P^s_{k+1}$, $P^s_{k+1}$ is guranteed transmitable by selecting algorithm. When droppong algorithm scanning the packets following the $P^s_{k+1}$, it won't drop $P^s_{k+1}$ to make room for packets which has higher bit utility than $P^s_{k+1}$. So $P^d_{k+1}$ is the same as $P^s_{k+1}$. This conclude the proof.

### 3.6 Anticipated deadline drop

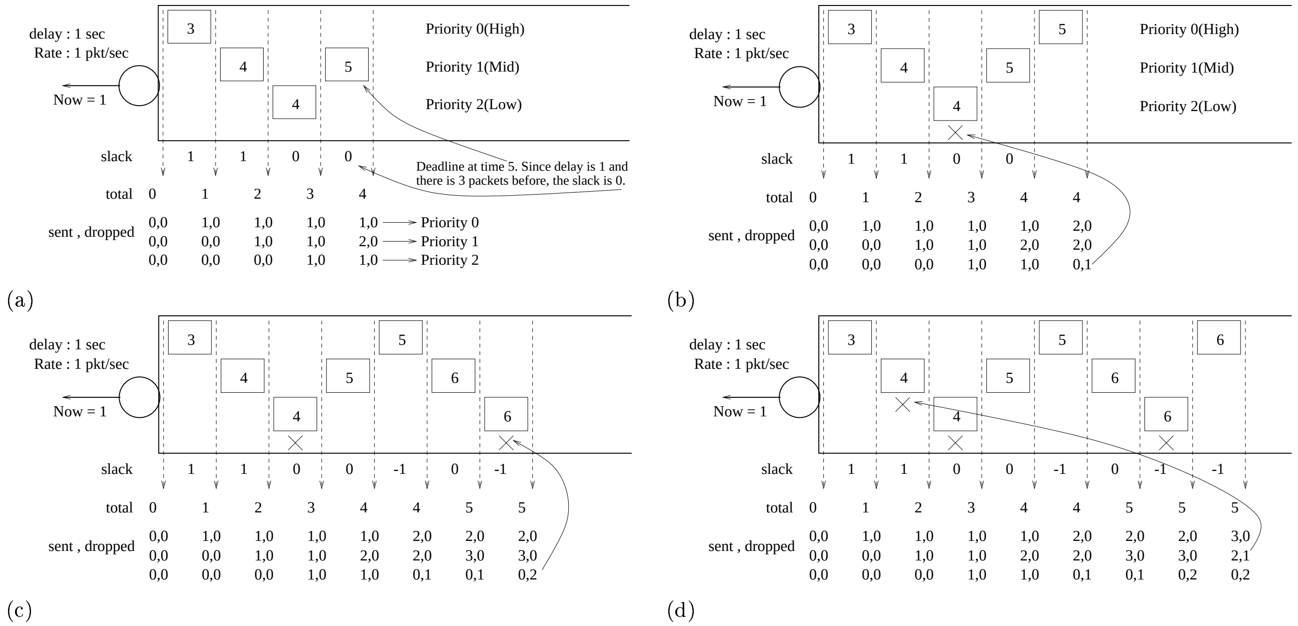


Figure 2: Anticipated Deadline Drop. Figure (a) shows the data structures. Each packet is labeled with its deadline. The slack denotes the available slack for the packet (slack of -1 causes either the packet to be dropped, or anticipated deadline drop). We also show the updated state of the send[] and dropped[] arrays as we traverse the queue. Figures (b), (c), and (d) show the evolution of the queue and the actions of the anticipated deadline drop as four packets with different priority and deadline arrive.

We now consider the case of a data stream with multiple priorities and packet deadlines. In this case, suppose a high priority packet is predicted to miss its deadline, but there are enough lower priority packets preceding it such that dropping them will cause the high priority packet to make its deadline, then the anticipated deadline drop mechanism will delete the preceding lower priority packets to make room for the following high priority packet. Figure 2 illustrates a sequence of scenarios, and Figure 3 specifies the pseudocode.

Anticipated deadline drop can be accomplished with a two-pass algorithm through the send queue. In the first pass, we scan the queue in order to determine how many packets in each layer must be dropped to make room for following higher priority packets to meet their deadlines. In the second pass, we drop the designated number of packets from the head of each priority level.

The space complexity of this algorithm is $O(p)$ for $p$ priority levels, since we need to maintain two arrays – *sent[level]* and *dropped[level]* – in the algorithm in Figure 3. Anticipated deadline drop is invoked under two circumstances: (a) when a packet is enqueued, and (b) when the connection parameters have changed beyond a threshold value since the last invocation of the anticipated deadline drop. The time complexity for case (a) is $O(p)$ (since we only need to perform the two-pass algorithm for the incoming packet), and the time complexity for case (b) is $O(np)$.

### 3.7 Dependency drop

As we have described before, frames in a heterogeneous data stream may have application-specific dependencies. At the receiver, a frame cannot be usefully received unless the frames on which it depends have also been usefully received. The motivation for dependency drop is to discard the packets of a frame at

```
FIRST PASS:

while (p)
   allow = (p.deadline - now - rtt) / rate
   before = sum(i: p.level to max_level)  sent[i] ; head drop
   if total <= allowed ; enough slack, accept
      sent[p.level] ++
      total ++
   else
      if before < total - allowed ; not enough slack, discard
         mark p
      else    ; must discard lower priority packets to accept
         to_drop = total - allowed ; number of lower priority
         j = max_level             ; packets to drop
         while (to_drop > 0)
            drop_at_level -= min{sent[j], to_drop}
            to_drop -= drop_at_level
            sent[j] -= drop_at_level
            dropped[j] += drop_at_level
            j --
         sent[p.level] ++
         total = allowed + 1         ; equal to total-to_drop+1
   next p
```

```
SECOND PASS:

total_drops = sum(i:0 to levels) dropped[i]
while (total_drops > 0)
   if dropped[p.level] > 0 and p unmarked
      mark p
      dropped[p.level] --
      total_drops --
   next p
```

| | |
|---|---|
| sent[levels] | : number of packets sent for this priority level |
| dropped[levels] | : number of packets dropped for this priority level |
| total | : total number of packets sent |
| max_level | : number of priority levels |

Figure 3: Pseudocode for the Anticipated Deadline Drop Mechanism

source if the sender knows that any of the frames on which this frame depends have not been usefully received at the receiver.

Providing dependency drop requires the transport protocol to understand the in-built application structure of the data stream, and the description of dependency drop in this section is necessarily more closely coupled to the specific transport protocol implementation as compared to previous mechanisms.

One of the more interesting uses of inter-frame dependency is to *emulate layering*. A "layer" in our model is simply a *dependency chain*. If the application does not want to adapt frequently, it can simple "chain" a sequence of frames that belong to a "layer". Once a frame in the layer is dropped, subsequent frames in the layer will also be dropped. The chain must be broken periodically to enable the "join" experiment for the layer. Using this approach, we can emulate layering and bound the frequency of adaptation for applications that do not wish to have frequent quality fluctuations. It turns out that we can emulate finer granularity adaptation than layering. For example, by chaining every $n^{th}$ frame in a layer (and having $n$ such chains), we can essentially enforce adaptation at the granularity of $1/n^{th}$ of a layer. Thus, dependency allows the application to control the frequency and granularity of adaptation.

## 4 Simulation and Analysis

We now present an evaluation of the goodput control mechanisms through simulation and analysis. First, we present two adaptation mechanisms that are commonly used in the practice. That will further movivate the necessity of goodput control. Second, we present the results for Progressive Motion JPEG and characterize the utility among packets within one frame. Third, using the MPEG trace from Bellcore, we show an example to characterize the utility across different frames in one GOP and formularize the relationship between deadline and buffer size. Finally, we evaluate in detail every component in goodput control and show that network characteristic does not affect the functionality of goodput control.

### 4.1 Goodput Control versus Application-Level Adaptation

In this experiment, we compare goodput control with application-level adaptation, and understand the trade-offs involved.

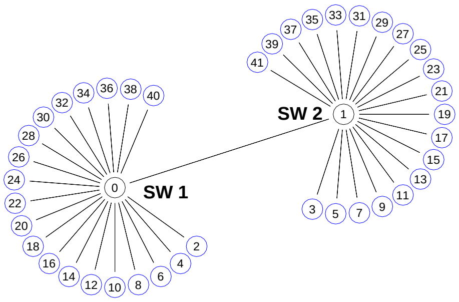


Figure 4: Simulation Topology with two switches

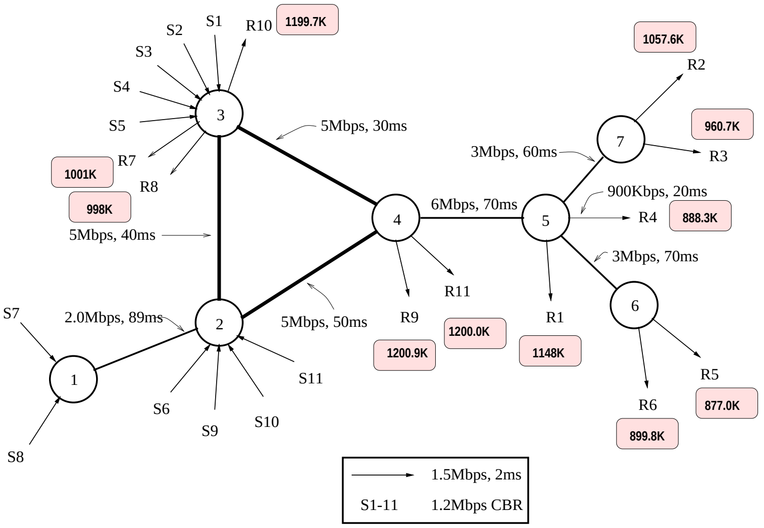


Figure 5: Complex Multi-hop Link Topology

| Sender | Expected rate | Observed rate | Frame sent | *Useful frame*$^*$ | $PSNR^*$ (db) | deadline drop | priority drop | I frame | P frame | B frame | *miss deadline*$^*$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1000 | 1147.98 | 28684 | 28675 | 32.34 | 5 | 1376 | 1879 | 5638 | 21181 | 0 |
| 2 | 1000 | 1057.63 | 26596 | 26328 | 31.61 | 452 | 3071 | 1839 | 5477 | 19064 | 36 |
| 3 | 1000 | 960.73 | 24008 | 23882 | 30.88 | 227 | 5958 | 1863 | 5540 | 16603 | 29 |
| 4 | 900 | 888.27 | 22309 | 22178 | 30.53 | 188 | 7554 | 1858 | 5522 | 14838 | 106 |
| 5 | 1000 | 899.76 | 22571 | 22412 | 30.57 | 431 | 6836 | 1865 | 5492 | 15263 | 10 |
| 6 | 1000 | 877.07 | 22050 | 21779 | 30.35 | 477 | 7403 | 1850 | 5463 | 14616 | 19 |
| 7 | 1000 | 1011.26 | 25147 | 24880 | 31.34 | 284 | 4808 | 1865 | 5584 | 17469 | 51 |
| 8 | 1000 | 988.45 | 25086 | 24730 | 31.29 | 301 | 4862 | 1862 | 5578 | 17399 | 46 |
| 9 | 1200 | 1200.03 | 29990 | 29989 | 32.72 | 0 | 0 | 1875 | 5623 | 22492 | 0 |
| 10 | 1200 | 1200.95 | 30004 | 30001 | 32.72 | 0 | 0 | 1876 | 5625 | 22503 | 0 |
| 11 | 1200 | 1199.70 | 29978 | 29977 | 32.72 | 0 | 0 | 1874 | 5621 | 22483 | 0 |

Table 1: Results of experiment on topology in Figure . (The superscript * is measured at receiver side. The rate has unit Kbps.)

we present the simulation results for goodput control in a randomly generated network configuration composed of 7 backbone links with different link bandwidths and latencies, and 11 connections as shown in Figure 5. Note that the goodput control mechanisms are not affected by the network characteristic as we will show in the Section 4.3.

All the 11 application senders send MPEG traffic at constant bit rate with 16 frames per group of picture (GOP). Each GOP has one I frame, three P frames, and 12 B frames with one packet per frame. P frames depend on preceding I frames while B frames depend on preceding and following non B frames. The aggregate rate for each sender is 1.2Mbps. 11 CBR connections share the available bandwidth as follows. The bandwidth of $S_4$ is limited by the bandwidth of the global bottleneck link from node 5 to R4. Connections $S_1$, $S_2$, $S_3$, $S_5$, and $S_6$ share the link 4-5 with constrained connection $S_4$. On the other hand, connection $S_9$ and $S_{11}$ share the link 2-4 with connection $S_6$. Since the bandwidth of $S_6$ was constrained by link 4-5, connections $S_9$ and $S_{11}$ shared the remaining bandwidth which makes the $R_9$ and $R_{11}$ receive all the CBR frames. Connections $S_7$ and $S_8$ share link 1-2, which is the bottleneck link on their paths. Since $S_{10}$ shares the link 2-3 only with connection $S_7$ and $S_8$, whose transmission rate where already constrained by link 1-2, it gets all the remaining bandwidth of link 2-3.

Table 1 shows the results for the experiment using goodput control mechanisms and the additive increase-multiplicative decrease rate control algorithm. Note that, in column 5, we introduce a notion of utility called "useful frame". Because of the inter-frame dependency in MPEG, utility can be defined to be the number of decodable frames that satisfy the dependency constrain at the receiver. As we can see in the column 5 and 6, it is closely related to PSNR value. Thus, we use the "useful frame" as a looser notion of metrics to measure the quality. From this point on, we present MPEG test in term of useful frame instead of PSNR for simplicity.

To see the impact of goodput control, we also perform the same experiment, but with two other application behaviors: (a) adaptive application with additive increase and loss proportional decrease(LIPD): the application will change its CBR rate based on monitoring the connection progress, and send at the peak estimated rate, and (b) adaptive application with additive increase and multiplicative decrease(LIMD): the application will change its CBR rate according to an additive increase multiplicative decrease algorithm based on monitoring the connection progress. Figure 6 compares goodput control versus the former scenario, while Figure 7 compares goodput control versus the latter scenario. In both cases, we test with different adaptation intervals for the application.

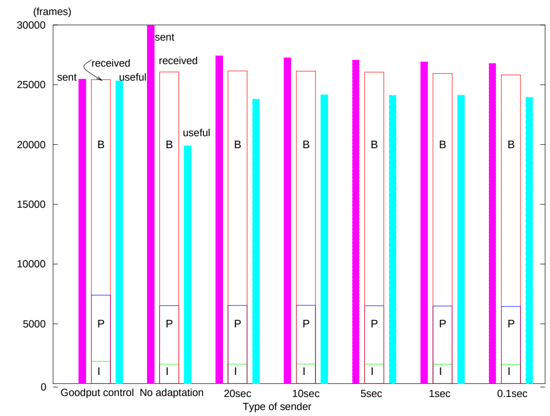


Figure 6: Goodput Control versus LIPD Adaptive Application over UDP

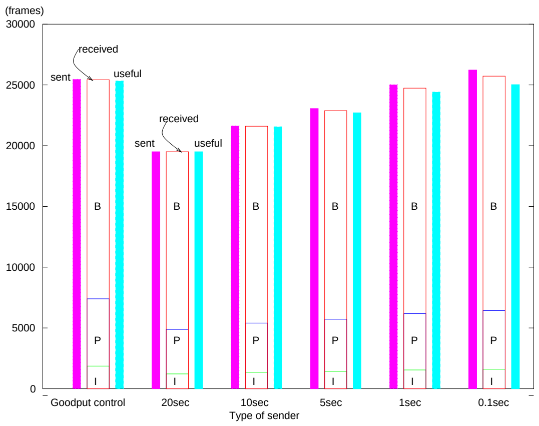


Figure 7: Goodput Control versus LIMD Adaptive Application over UDP

From Figure 6, we see two results:

1. The adaptive application is far more aggressive (in a non-TCP friendly manner) and sends 17% to 5% more packets than the goodput control case, but useful frames received is 21.5% to 4.6% less than that in goodput control case. Specifically, the goodput control application sends 11.8% more I frames, and 11.4% more P frames, even for the best case adaptation. In words, sending more does not help the application itself.
2. The total number of packets received at receiver is similar but the goodput can be significantly different depending on the presence of goodput control mechanisms at the sender. Further, for the same number of received packets, the adaptive application sends a lot more packets, which would choke other TCP flows (the rate control algorithm used in goodput control is TCP friendly [5]).

From Figure 7, we see two results:

1. The adaptive application in this case behaves in a TCP friendly manner but it is highly susceptible to the adaptive interval. The applications will underutilize the channel after the multiplicative decrease and take serval adaptation periods to recover. Even in the best case, the goodput control application sends 13.2% more I frames and 14.3% more P frames. The aggregate number of useful received frames with goodput control is 5.4% more.
2. Shorter adaptation intervals enable the application to become more responsive to the dynamics of the network. In fact, goodput control outperforms adaptive applications precisely because of the same reason – it is closely coupled with the rate adaptation component and has the same granularity of adaptation.

## 4.2 Test Results for Progressive Motion JPEG and MPEG Trace

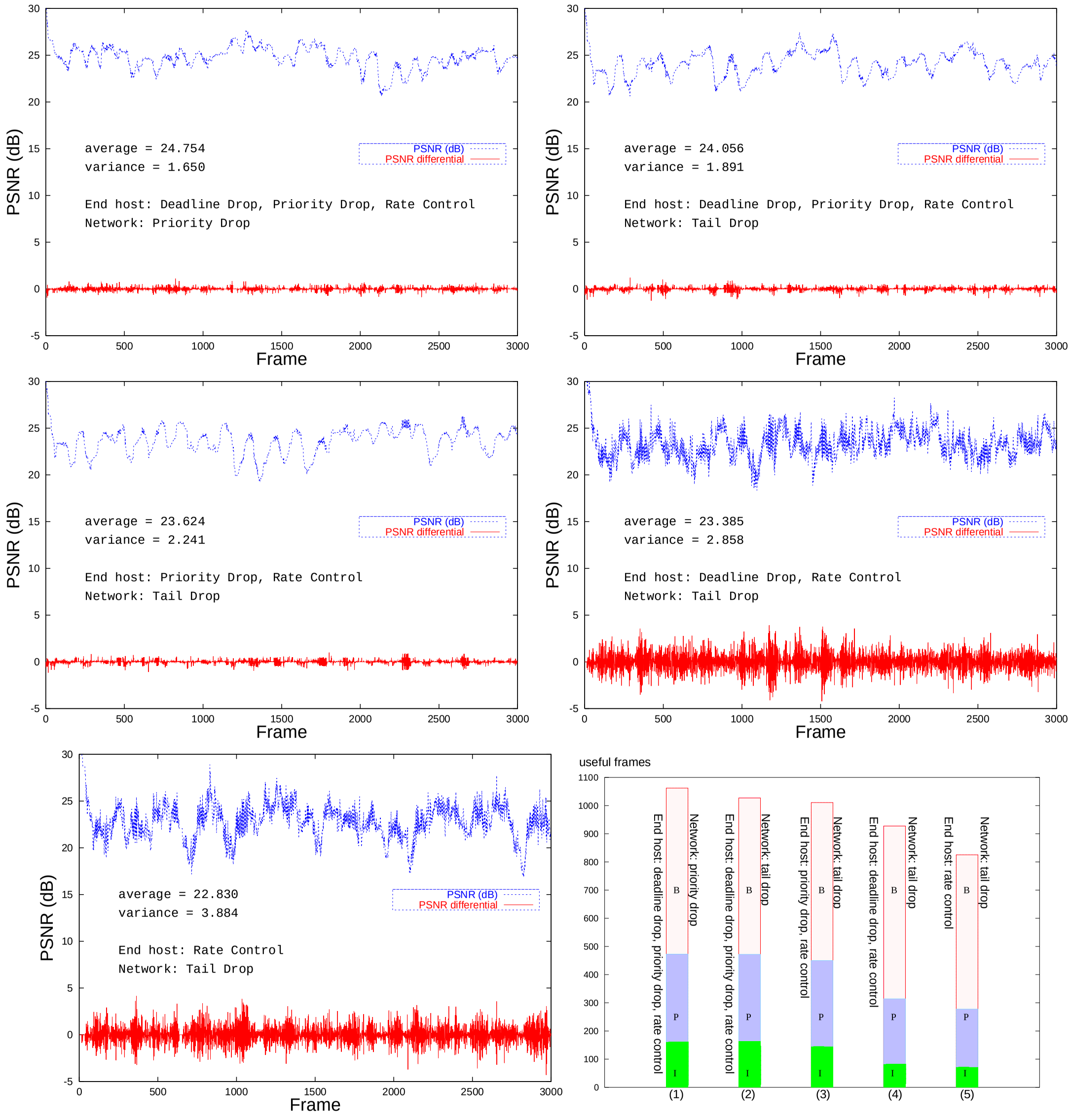


Figure 8: Plot for buf=420 deadline=2.0 and delay=30

For the simulations in this section, we encode a football sequence (grayscale, 342x240, 31 frames per second) using Progressive Motion JPEG. The resulting data is fragmented into 13 packets with payload size of 1000 bytes. In the other words, the traffic is about 3.6Mbps. We send this traffic over the topology as shown in Figures 4. The bottleneck bandwidth is 55Mbps with latency 30ms. There are one Pareto and one Exponential ON/OFF source among 20 senders. The mean for both the ON and OFF period are 10ms. During the ON period, CBR with rate 5Mbps is sent.

We measure the mean and variance of PSNR over 18 senders/receiver pairs. In Figure 8.a, it demostrates the perfect world with smart transport and priority network. In Figure 8.b, we do not assume network has priority drop and we show slightly quality degradation for 0.7db. In Figures 8.c, the average PSNR is lower for 1.7% because the deadline drop does not delete the deadline-violated frames in order to save bandwidth for other packets in the buffer, ie. the anticipated drop. This further delays the packets in the sender buffer and reduces usefulness of received frame. In Figures 8.d, except that the PSNR is lower for 2.7%, there is another important source for the quality degradation. The difference in PSNR for any two consecutive frames is much higher because there is no priority drop to remove less important data to make room for higher utility packets. Therefore, users will perceive jittery quality even receivers receive the same amount of packets. This hehavior can also be characterized quantitatively by the variance of the PSNR as well. The variance increases from 51.1% between case b and d. Finally, in Figure 8.e, without smart transport and network support, the quality is dramatically degraded. The PSNR drops from 24.056 to 22.830 while the variance incresaes from 1.891 to 3.884. Note that in all 5 examples shown here, the network rate stay approximate the same. This validate the arguments that congestion control is of the interest to the network but it does not optimize the application goodput.

In Figure 8.f, we use the MPEG trace from Bellcore. The sequence consists 12 frames in one GOP with the 240X352 (Luminance - Y) and 120X176 (Crominance - U & V). The frame rate is 24 frames per second. We again show the five scenarios as that in MPJPEG case except that we use the useful frame as a metrics to characterize the quality. In MPJPEG, there is no inter-frame dependency. The quality is purely depending on the packets received for that frame. Showing quality by PSNR can best represent the perceived quality.

In contrast, in MPEG, inter-frame dependency plays such an important role in the perceived quality. Thus, "number of useful frames received" captures perfectly the utility and dependency in MPEG streams. As shown in Figure 8.f, between bar (1) and bar (2), the useful frame decreases slightly(3.31%) from perfect case to goodput control case. However, there is literally no difference in received I and P frames. In bar (5), useful frames drop dramatically. Especially, number of I frames received drop more than 50% that lead to poor quality.

### 4.3 Simulation-based Evaluation of Goodput Control

In Section 4.3.1 we observe the impact of priority drop as a function of the sender buffer for a heterogeneous stream with no anticipated deadline drops. In Section 4.3.2 we observe the impact of anticipated deadline drop as a function of the sender buffer for a heterogeneous stream with deadlines, and understand the relationship between priority drop, deadline drop, and buffer size. In Section 4.3.4 we observe the impact of the accuracy of round trip time estimation on deadline violation. In Section 4.3.5 we observe the impact of router parameters on end-to-end goodput control.

For the simulations in this section, we use the topology, as shown in Figures 4. The bottleneck link speed is 20Mbps with delay 100ms. All the peripheral links have 10Mbps speed and 2ms delay in order to avoid the multi-hop behavior [11]. The buffer at SW1 for all experiments is 500 packets except when otherwise mentioned. All the 20 application senders send MPEG traffic at constant bit rate with 16 frames per group of picture (GOP). Each GOP has one I frame, three P frames, and 12 B frames with one packet per frame. P frames depend on preceding I frames while B frames depend on preceding and following non B frames. The aggregate rate for each sender is 1.2Mbps, except when otherwise stated. The results we show in this session use an assigned deadline of 3 seconds. We have also performed a series of tests with deadlines of 5, 7, and 9 seconds, and GOP of 3, 6, 12 frames. Since the results are qualitatively similar, we only present the case where the deadline is 3 seconds, and GOP is 16 frames. The rate control mechanism is RAP.

### 4.3.1 Impact of Buffer Size on Packet Dropping

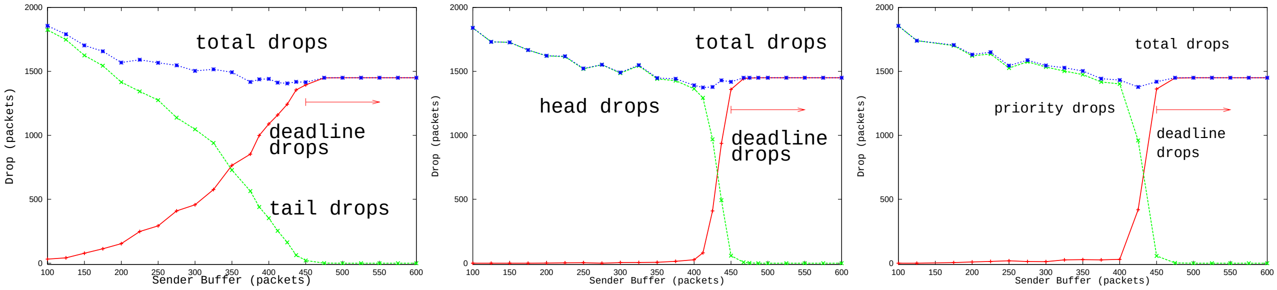


Figure 9: Drops at sender: (a) Tail (b) Head (c) Priority

When there is a mismatch between the application sending rate and the connection sending rate, the buffer at the sender starts to get backlogged. Once the sender buffer is full, we consider three mechanisms for handling incoming packets: (a) head drop, (b) tail drop, and (c) priority drop. If the sender buffer is small, the packet dropping mechanisms described above will kick in. On the other hand, if the sender buffer is large, then packets that violate their deadline will be discarded via deadline drop. Figure 9 shows the threshold value after which deadline drop starts to dominate the total number of dropped packets. It turns out that the threshold value is approximately equal to the “deadline bandwidth” product, which is 450 packets in the example. Note that the total number of packets dropped is the same, as seen from Figure 9 – it is only the component dropping mechanism that contributes to the drops that changes as the buffer size increases.

### 4.3.2 Priority versus Head versus Tail Drop

For the same example, Figures 9 show the impact of three candidate packet dropping mechanisms – head drop, tail drop, and priority drop – as a function of sender buffer size. For large buffer sizes, the dropping mechanism does not matter because deadline drops predominate. However, for small buffer sizes, we see two important effects: (a) from Figure 9.a, tail drop causes more deadline violations because it deletes packets from the tail of the queue rather than the head, where is where the deadline violations are taking place; (b) head drop and priority drop show similar dropping patterns in Figure 9.b/c, but looking at Figure 10 shows that priority drop significantly improves the perceived goodput at the receiver. This is because priority drop preferentially protects higher priority frames, on which subsequent lower priority frames may depend.

In Figure 10, the useful frames for head/tail and deadline drops are similar. With priority drop being part of the goodput control module, it increases the utility when buffer is smaller than 450. Note that deadline drop itself does not necessary increase the quality at the receiver. The reason is that when deadline drop really kicks in, it basically deletes all packets violating deadline without knowing the priority and dependency of the frames. This motivates us to include the anticipated deadline drop in order to improve the quality when buffer is greater than 450(as shown on the right-top of Figure 10 marked as goodput control). The arguments are simular to congestion avoidence case, we need to use anticipated deadline drop to avoid goodput control module to always operate at the rigion that only deadline drop is in effect.

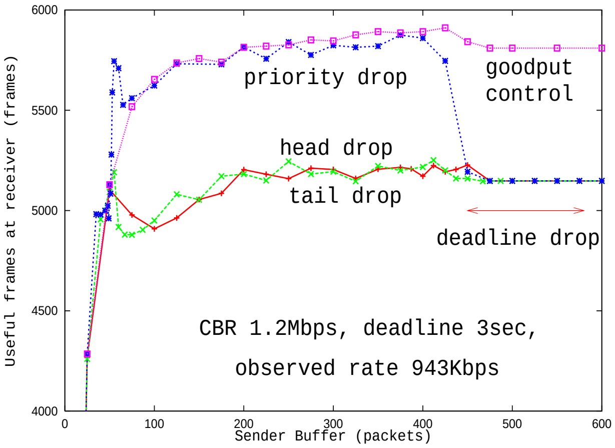


Figure 10: Plot of useful frames at receiver versus sender buffer size. CBR rate is 1.2Mbps and fair share on link is 1Mbps.

### 4.3.3 Impact of Deadline Dropping

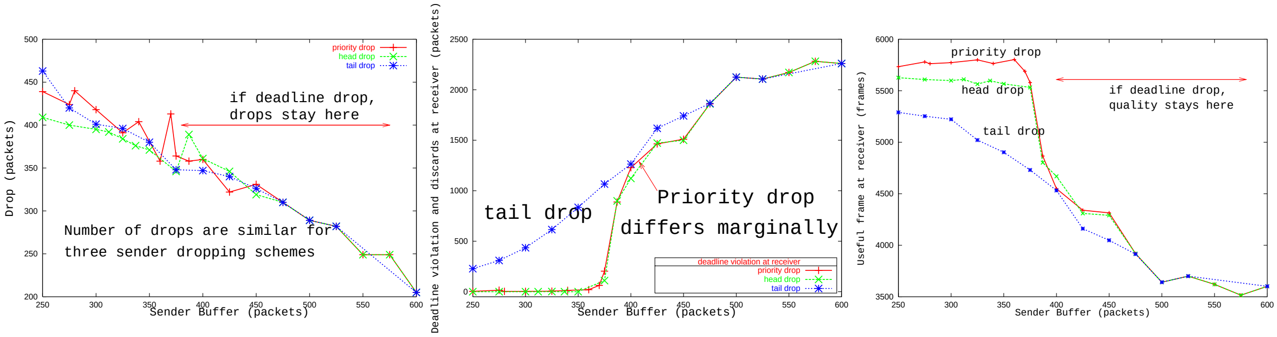


Figure 11: (a) Sender : all dropping mechanisms have similar behavior. (b) Receiver : deadline violation discards 5 times more packets than necessary. Tail drop does not remove packets that are closer to their deadlines. (c) Deadline drop will keep useful frame stay at 5700 frames for tail drop and after buffer is bigger than 375.

In this section, we disable the deadline drop at the sender and show how many packets that will miss deadline at the receiver. In Figure 11.a, it shows the drops in the sender while the sum of Figure 11.b and c are the total amount of packets that reach the receiver. As the sender buffer size increases, a rate mismatch between the application and the sender may cause the sender buffer to fill up. Consequently, packets may spend a long time in the sender buffer and violate their deadline. Clearly these packets should be dropped since they are anyway useless at the receiver, and may further delay following packets. This effect is illustrated in Figure 11.a. When the buffer is smaller than 375 packets, all drops still come from buffer overflow. However, because of the absence of deadline dropping, the sender sends packets that are discarded at the receiver when they arrive, and the receiver actually discards 5 times more packets than necessary, as shown in Figure 11.b. Consequently, the goodput drops sharply as the buffer size increases, as shown in Figure 11.c. Of course, with deadline and anticapted deadline dropping, the perceived quality will stay high regardless of the buffer size.

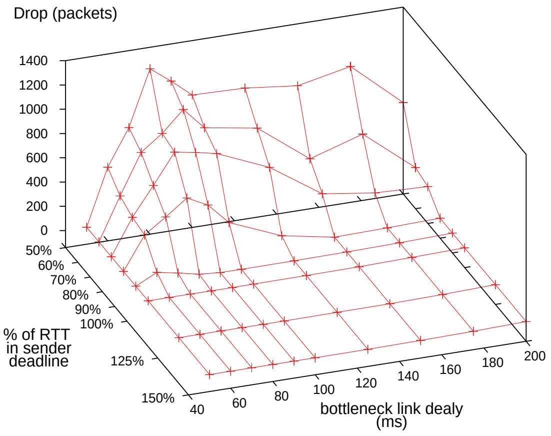


Figure 12: Deadline Violation at Receiver versus RTT Estimation and Bottleneck Link Delay.

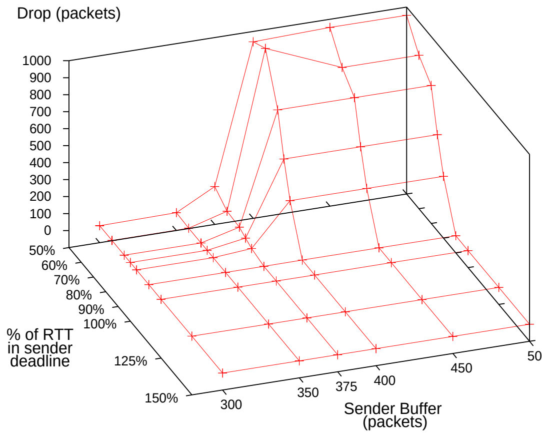
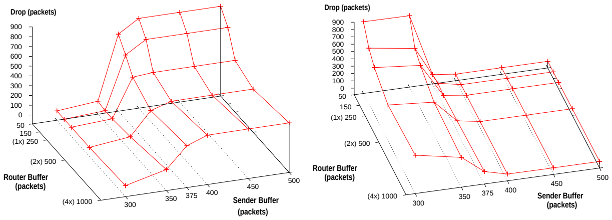


Figure 13: Deadline violation at receiver v.s. RTT estimation and sender buffer size.

#### 4.3.4 Impact of Round Trip Time Estimation on Deadline Drops

When the application hands in a packet to the transport with the assigned deadline $V$, the effective deadline at sender is $V$ - *forward latency*. If the link is symmetric and the estimation is accurate, the forward path latency is half the round trip time. However, we cannot assume symmetric paths, and it is impossible to estimate one way latencies using only end-to-end feedback (without making restrictive assumptions such as synchronized clocks, etc.). To be conservative, as shown in Figure 13 and 12, using 90 percent of the srtt to calculate the effective deadline can avoid almost all of the deadline violations at receiver. Of course, the results of this experiment are heavily impacted by our specific offer load and test scenario. In our deadline estimation mechanisms, we have chosen to be conservative and drop packets more aggressively as packets near their deadline, since optimistically letting packets go through can not only lead to delay violations at the receiver, but also propagate the delay violations to subsequent packets in the stream. This is consistent with the result shown in Section 4.3.3, where the sender conservatively drops more packets using priority and the the receiver perceives improved quality. In fact, using one srtt seems to be a good engineering choice.

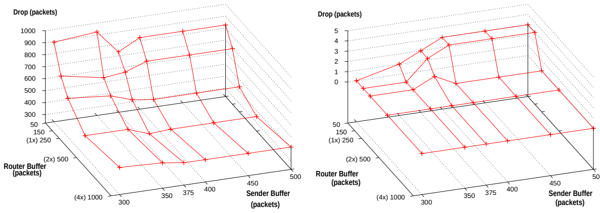
#### 4.3.5 Impact of the Queue Size in the Routers on End-to-End Goodput Control



(a) Deadline Drop at Sender v.s. Router Buffer Sizes

(b) Priority Drop at Sender v.s. Router Buffer Sizes

Figure 14: Deadline and priority drop v.s. router buffer size



(a) Total Drop at Sender v.s. Router Buffer sizes

(b) Deadline Violation at Receiver v.s. Router Buffer sizes

Figure 15: Total number of drops at sender and receiver versus router buffer size

We now explore the impact of router buffer sizes on goodput control. Of course, our topology is very simple, which limits the conclusions that we can reach from the experiment. When the router buffer is very small, buffer contention will reduce the congestion window, and more packets are dropped at the sender. In [5], the authors choose the router buffer space to be four times the delay-bandwidth product of the bottleneck link. We simulate from 0.2 to 4 times of the delay-bandwidth product and conclude that goodput control is not particularly sensitive to the size of the buffer in the routers. In addition, we also change the bottleneck link delay from 10ms to 200ms (not shown in this paper), and the results confirm that goodput control is not very sensitive to the link delay as well. This is because packets are spaced out at the sender by the rate control mechanism, and the rate control ensures that the sending rate is matched to the link capacity.

Figure 14.a and b demonstrate the drops because of the deadline and priority drops, and Figure 15.a shows the their sum. In Figure 15.b, with the buffer size half of the delay-bandwidth product, receiver only sees three packets miss their deadline. This is because we select conservative deadline policy at the sender side, as as mentioned in Section 4.3.4

The interesting observation is that goodput control can improve the performance while at the same time reducing the programming complexity of applications. However, we caution the reader against assuming that these improvements come for free: goodput control mechanisms incur computational overhead during three phases: $O(p)$ overhead during packet enqueueing at the send queue for $p$ priority levels, $O(1)$ overhead during packet dequeueing at the send queue, and $O(np)$ overhead for the anticipated deadline recomputation when the connection parameters change by a threshold value. Our ongoing work is looking into two aspects: (a) studying the impact of the threshold value, and (b) reducing the computation overhead.

## 5 Related Work and Summary

In related literature, there have typically been two other ways of supporting such streams: (a) the application implements all the smarts for rate adaptation, deadline management, priority-based transmission, etc. on top of a UDP datagram transport protocol, and (b) the application uses a real-time transport protocol such as RTP or RAP over UDP; the transport provides connection-level rate adaptation and possibly deadline dropping, while the application provides the mechanisms for application-level rate adaptation, and priority-based transmission. In this paper, we consider the third approach, in which the application specifies the QoS policies for its frames/packets, and the transport layer provides mechanisms for goodput control and rate control in order to maximize the application goodput through QoS-aware packet dropping mechanisms at the sender buffer.

In the first approach, the application implements all the smarts for supporting the multimedia stream, and uses UDP as the transport. Specifically, the application must determine the available sending rate, manage deadlines, send only the highest priority packets that can be sustained at the available rate, and send complete frames (or throw away partially received frames). The advantage of this approach is that the application has complete control over what is sent over the network. The disadvantages are that the sending rate estimation can become quite inaccurate at the user layer, the mechanisms for adaptation must be replicated for each application, and packets once sent cannot be "recalled" even if they are still queued at the sender (e.g. even if they have violated their deadline). In fact, writing an efficient multimedia application that adapts effectively to network dynamics is a very challenging task. Even popular publicly disseminated network video players that have been used for a few years exhibit several flaws in their adaptation design [13]. While some of the flaws are coding related, many of the flaws - specifically relating to the coarseness of rate adaptation and the violation of deadlines at the receiver - are inherent limitations of this approach.

In the second approach, applications use transport protocols such as RTP [12] or RAP [5], that are designed to support real-time multimedia flows. In this case, the transport protocol performs the connection rate adaptation and allows the application to adapt over longer timescales to long-term rate (which is desirable to prevent the quality fluctuation at the application layer), and buffers packets due to the potential mismatch between the application sending rate and the connection sending rate. Among related work, most multimedia transport protocols provide rate adaptation and buffering, but not priority dropping and deadline dropping. The problem with this approach is that in the absence of deadline dropping and priority dropping, once the application has sent packets for transmission, it cannot “recall” them, and due to the buffering in the transport layer (and coarser rate adaptation in the application), this problem is more pronounced than in the previous approach. Our experience has been that while the rate adaptation mechanisms of RAP in particular, and multimedia transport protocols in general, can improve the throughput of the multimedia stream, they still do not improve the goodput as much because of deadline violations, partial frame delivery (which are discarded at the receiver), and delivery of low priority frames without the deliver high priority frames on which they depend.

The problems with the first two approaches motivate the need for goodput control. Basically, we see from the second approach that it is a good thing to perform rate adaptation at two levels – the transport adapts the connection sending rate to short-term variations in the network resources while the application adapts the application-sending rate only to long-term variations in the network resources. However, in addition to the issue of rate adaptation, which answers the question of “how much to send”, there is also the critical issue of “what to send”. The shortcomings of the first two approaches can be directly traced to the fact that the sender sent packets that were useless at the receiver, thereby wasting the connection capacity, and possibly further delaying other packets. This motivates a close coupling between rate control and goodput control at the transport layer.

An interesting point in the paper is the argument that goodput control can be achieved through QoS-aware packet dropping in the sender buffer. While we did not make the point explicitly in the paper, our ongoing work seeks to formally show that smart packet dropping is necessary and sufficient to achieve goodput control.

(We will explode the tradeoffs between transport layer and middleware centric goodput control mechanisms in the final paper).

## References

[1] I. Kozintsev and K. Ramchandran . Robust Image Transmission over Energy-Constrained Time-Varying Channels Using Multiresolution Joint Source-Channel Coding . IEEE Transactions on Signal Processing, April 1998.

[2] D. Taubman. High Performance Scalable Image Compression with EBCOT. to appear in IEEE Transactions on Image Processing.

[3] R. Rejaie, M. Handley, and D. Estrin. Quality Adaptation for Congestion Controlled Video Playback over the Internet. Proc. ACM SIGCOMM, Cambridge, MA, September 1999.

[4] H. Balakrishnan, H. Rahul, and S. Seshan. An Integrated Congestion Management Architecture of Internet Hosts. Proc. ACM SIGCOMM, Cambridge, MA, September 1999.

[5] R. Rejaie, M. Handely, and D. Estrin RAP: An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet . Proceedings of IEEE Infocom'99, New York, NY, March 1999.

[6] J. Li, S. Ha, and V. Bharghavan. Transport Layer Adaptation for Supporting Multimedia Flows in the Internet. *Proceedings of IEEE INFOCOM '99*, March 1999.

[7] I. Busse, B. Deffner, and H. Schulzrinne. Dynamic QoS control of Multimedia Applications based on RTP. Computer Communications, January 1996.

[8] Steve McCanne and Van Jacobson. vic: A Flexible Framework for Packet Video. *ACM Multimedia*, September 1995.

[9] J. Padhye, V.Firoiu, D.Towsley, and J. Kurose Modeling TCP Throughput: a Simple Model and its Empirical Validation. ACM SIGCOMM, September 1998.

[10] S. Cen, C. Pu, and J. Walpole. Flow and Congestion Control for Internet Streaming Applications. Proceedings Multimedia Computing and Networking, January 1998.

[11] S. Floyd. Connections with Multiple Congested Gateways in Packet Switched Networks.. Computer Communication Review, 21(5):30-47, October 1991.

[12] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. *INTERNET-DRAFT draft-ieft-avt-rtp-new-02.ps*, November 1998.

[13] Shanwei Cen, Calton Pu, Richard Staehli, Crispin Cowan, and Jonathan Walpole. A Distributed Real-time MPEG Video Audio Player. *Fifth International Workshop on Network and Operating System Support of Digital Audio and Video (NOSSDAV)*, April 1995.