

Implementing An Differentiated Services Architecture on IXP1200¹

18-884 Network Design and Evaluation

Final Report

Tung-fai Chan, Jiann-min Ho, Jia-cheng Hu, Igor Pevac

{tungfai, jiannmin, jiacheng, ipevac}@andrew.cmu.edu

Abstract

This project uses Intel® IXP 1200 packet processor to implement Differentiated Services Architecture base on the paper proposed by K. Nichols, V. Jacobson and L. Zhang [1]. We then examine the effectiveness of our implementation to see how the service is improved and what side effects are incurred. Meanwhile, we also put emphasis on the bandwidth broker and the signaling protocol.

1) Introduction

The basic idea of Differentiated Service (DiffServ) is to define the architecture and a set of forwarding behaviors so that the service providers can identify and implement end-to-end services on top of this architecture. This framework pushes complexity to the edge routers to the extent possible and keeps the core router simple. The edge routers check the traffic flow coming into the switch, classify it, mark appropriate bits on it, shape the traffic flow or drop the packet if necessary, and forward to the core network. The core routers just simply examine the bits pattern and transmit the packets as appropriate. The forwarding behavior is predefined in profile through the negotiation between service providers. In this project, we will define two different types of service, 'Premium' and 'Best effort'. Each type will provide different forwarding behavior and bandwidth is the resource being requested and allocated. We will briefly explain these terms in the following.

A Premium service traffic flow is hard-limited to its provisioned peak rate. Traffic flows within the predefined bandwidth will be guaranteed, but bursts over the profile may not be injected into the network promptly. On the other hand, the 'Best effort' service will not be given any assurance, but they will be sent as long as there is bandwidth available.

In our implementation, we will focus on the interaction between routers. We incorporate the concept of 'Bandwidth Broker' and regulate forwarding behaviors based on the result of negotiation between bandwidth brokers. The core routers maintain the minimal knowledge of its neighboring routers' profile.

This design report is organized into sections. Section 2 will describe high-level design in flow-basis; implementations specified to IXP platform are discussed in Section 3; different level of evaluations examine this project in Section 4; Section 5 is the future work and improvement discussion; and the last section will present our conclusion.

2) Design

This section will focus on high-level conceptual design of our DiffServ architecture based on [1]. The design is primarily composed of two parts: Admission Control and Signaling, and Data Transmission. The former focuses on how to setup a reservation flow within same domain or across multiple domains whereas the latter defined data packet transmission after flow reservation has been set up.

¹ IXP 1200 is network processor from Intel® Corporation.

2.1) Admission Control

Admission control is a critical part of any Quality of Service scheme. It is very important to know the limitations at which a particular domain can be reserved in order to assure that the reservations made receive the service promised to them. As part of our design we need to examine admission control for the Premium service class of traffic. Premium class is very strict in the way traffic is sent. It is regulated by bandwidth and shaped to eliminate any bursts. Being such it receives absolute priority in the network. As such it is crucial that there is not an over reservation of this type of traffic, since it will starve the remaining traffic in the network.

The approach that we are taking in our design is the admission control through use of a central authority. This is described as a Bandwidth Broker in [1]. The information that the Bandwidth Broker keeps is the allocation on the leaf nodes of the network (edge routers). By knowing how much of the resources are already reserved it can answer requests form incoming reservations. Once it is determined that a reservation can be made, the Bandwidth Broker sends the specifics of the flow to the corresponding edge router to allocate the resources towards classifying the flow and providing the level of service requested. The state kept is a soft state and is refreshed by the Bandwidth Broker for the specified duration of the reservation.

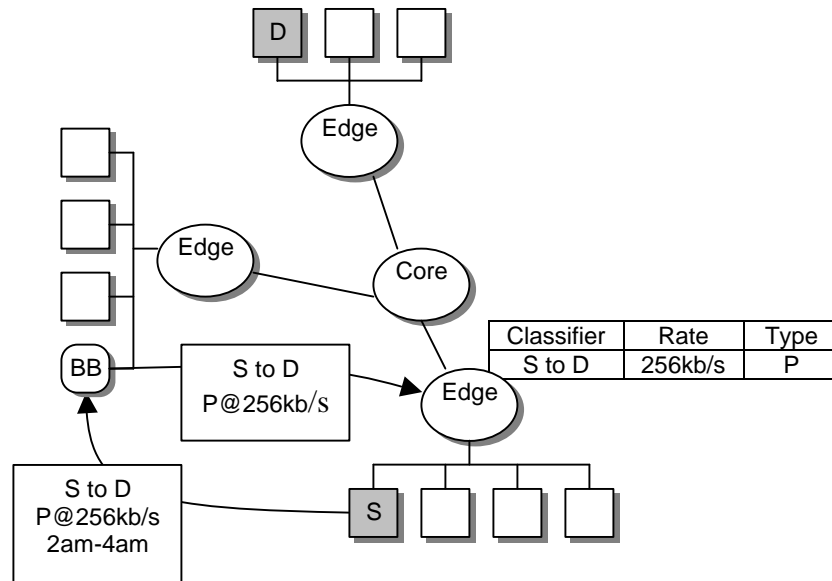


Figure 1 Admission Control for local domain

2.2) Data Transmission

Routers are divided into two categories: Edge and Core, and they perform different tasks in the architecture. According to [2], Edge Routers usually handle fewer flows and thus is where most complexities located. They classify packets into flows, then carry out traffic policing, and marking according to corresponding traffic profile, and then prioritize the packet forwarding. Core Routers, on the other hand, handle high capacity bandwidth pipes and require simpler tasks. They simply according to bit pattern in the packet header and perform traffic aggregation. This section will explain in detail of each role in the architecture.

2.2.1) Edge Router

After the connection setup phase, a traffic profile has been contracted between source and the ingress router. For subsequent traffic entering the domain, the ingress performs following actions before the packets are passed to the forwarding engine at the ingress router as shown in Figure 2.

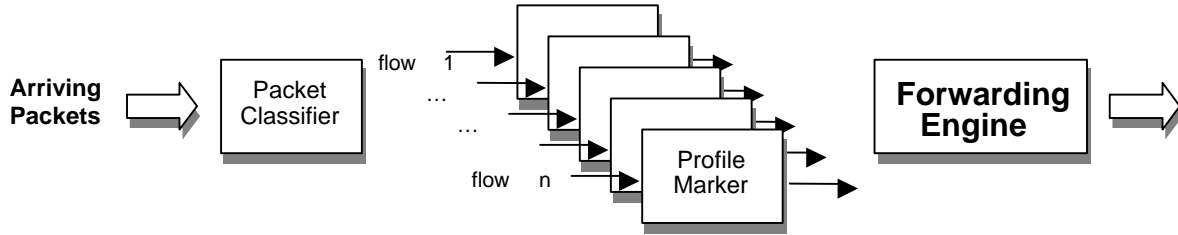


Figure 2 Diagram for Packet Flow in Edge Router

The framework adopts the idea from [1] that designating one bit patterns from the IP header precedence field to indicate the corresponding service type: premium (P-bit), and best effort. For every arriving packet, it must have the P-bit cleared before processed by Packet Classifier. Packets are classified into different flows in which will pass through individual Profile Marker or passed to the Forwarding Engine according to service type. Each marker has been configured from the usage profile for its corresponding flow: service class, rate, and burst size. Packets coming out from markers are injected into the Forwarding Engine in which they are transmitted according to the bit-pattern in the header.

2.2.1.2) Packet Classifier

This component is basically a general classifier, which performs a transport level flow matching, based on the information within the packet header. Packets that match to one of the configured flow profile are forwarding to corresponding Profile Marker, otherwise, to the Forwarding Engine directly. This may be a time-consuming process and keeping it at the ingress router where fewer flow can reduce its workload.

2.2.1.2) Profile Marker

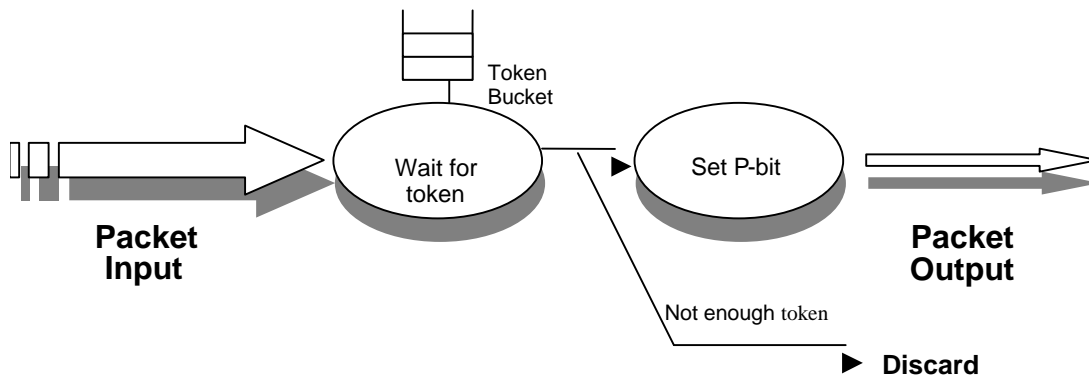


Figure 3 Implement Premium Services by Token Buckers

The marker, according to the flow profile, carries out both policing and marking responsibilities. Figure 3 shows the details of how the marker should do to Premium service types. The marker is implemented by using token bucket in micro-code fills at the flow rate in number of bytes specified in the usage profile.

The incoming packet will have P-bit set and forwarded if a token is present; otherwise, the packets are discarded.

2.2.1.3) Priority Queues

There are two priority queues being maintained: high and low priority. Premium traffics are enqueued into high priority queue while others, essentially Best-Effort traffic, are put into low priority queue. Since Premium traffic will hardly conform to the contract profile and so there should not be queue overflow or starvation problems.

2.2.1.4) Forwarding Engine

This basically employs absolute priority over the two queues. High priority queue with Premium packets must be serviced first before any other traffic, essentially Best-Effort traffic, in another queue.

2.2.2) Core Router

Following the framework described in [2], simplicity is the keep design rationale of the core router for data transmission. During this phase, the core routers carry no per-flow information. Instead, they perform traffic aggregations according to the different service types: Premium, and Best-Effort. All incoming packets into the core router are classified according to the bit-pattern in the packet header. Then, they are enqueued into different priority queue in the Forwarding Engine as shown in Figure 4.

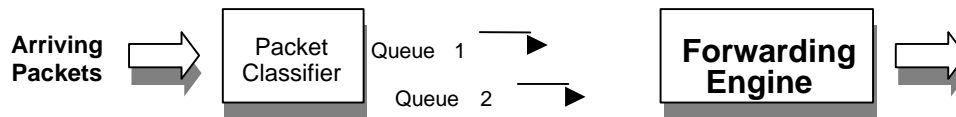


Figure 4 Diagram of Packet Flow in Core Router

2.2.2.1) Packet Classifier

This classifier, unlike the general classifier employed by edges, is in fact a simple bit-pattern classifier. It comprises a simple binary decision based on a particular bit-pattern in the IP header is set or not. This classification is done without look into other header field and this greatly enhances the efficiency and reduces packet overhead at the core router. Since core routers are usually required to handle rather heavy aggregated traffic, this simplicity is essential.

2.2.2.2) Forwarding Engine

The mechanism of the Forwarding Engine at the core routers is essentially the same as edge routers. Packets are transmitted following to the priority level mentioned in previous section.

3) Implementation Details

The implementation discussed in this section is under the context of IXP 1200, the Intel® packet processor platform. The platform mostly consists of two major components: Control Plane, and Data Plane. Control Plane uses C++ as primary programming language whereas Data Plane is implemented by microcode. In

addition, we have an separate entity, Bandwidth Broker, written in Java and run on a separate host. This section is sub-divided into three parts: Bandwidth Broker, Control Plane, and Data Plane.

3.1) Bandwidth Broker.

Currently Bandwidth Broker is implemented as a Java application. Information about the network includes the list of all the edge routers and set of networks they are responsible for. When a request comes in the source and destination are compared to the list of networks. It is determined with edge router is responsible for the source as well as if the destination is part of the domain. Next step in the process is determining if reservation can be supported in the domain. Additionally if the destination is not part of the domain, the Bandwidth Broker of the next domain in line needs to be contacted.

Admission control is simplified currently. There is just a number chosen to indicate the capacity of the network allowed to be used by differentiated services. If the reservation fall under this limit minus the current reservation of the network the flow is admitted. Information is then forwarded to the edge router. Once an acknowledgment is received back, it is forwarded back to the sender making the request.

The Bandwidth Broker also keeps the state of current reservations in the domain. Since the edge routers only keep a soft state, the Bandwidth Broker has to periodically refresh that state. A thread is allocated for this purpose with a timeout interval of half that of the edge router interval.

All communication between the Bandwidth Broker, the edge router and the client is done by UDP. Decision for this is due to the fact that there are only a few messages sent back and forth between them. Thus there is no need to establish TCP connections for just those few messages..

3.2) Control Plane

The responsibilities of the control plane revolve around the reservation table shown in Figure 5. First the control plane is in charge of adding and deleting entries from the table. Entries are added by a request made by the Bandwidth Broker on the behalf of the source. They are likewise removed by a remove message from the Bandwidth Broker or when the timer for the soft state expires. Thus the entries are not permanent guarding against having stale state due to a failure.

The table itself is constructed from eight entries. The first four are filled in by the information sent in a reservation request. Next the data plane in determining the amount of traffic to send for the flow uses the next three entries. Finally the last one is used by the control plane to determine the last time the entree was refreshed by the Bandwidth Broker.

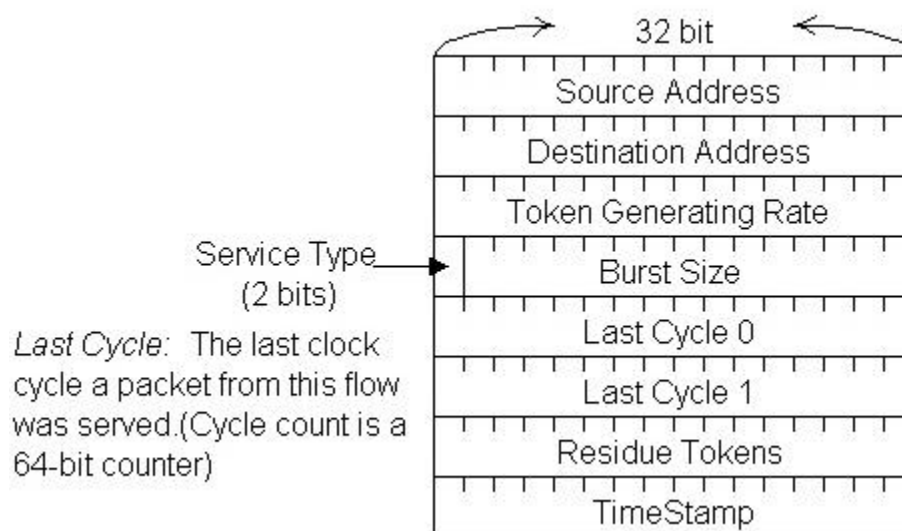


Figure 5 Reservation Table Structure

3.3) Data Plane

This part focuses on the modifications we made on microcode in order to have DiffServ capabilities. First, we would like to discuss briefly about the macros we added in. Then, discuss further on the data packet flow and our token bucket algorithm.

3.3.1) Macros

3.3.1.1) ReservationTable_Lookup_Lock

Input: source address, destination addresss

Output: reservation type, rate, burst, last cycle0, last cycle1, residue tokens, table entry address

Descriptions:

Based on the pass in parameter, source and destination addresses, this macro looks up the reservation table (mentioned in 3.2) by using sequential search. If there is a match, it will look the first longword (32-bits) of the entry and return all information in that table plus the entry address. This only blocks the subsequent packet from same flow. If there is no match, it will return with rate sets to 0x0. Two longwords are used for storing last cycles as 64 bits are used to represent the clock cycle in IXP platform.

Source file: rec_lmatch.uc

3.3.1.2) ReservationTable_Update

Input: table entry address, last cycle0, last cycle1, residue token

Output: N/A

Descriptions:

Write the values, last cycles and residue token, back to the table entry specified by the entry address. Note that this macro will NOT unlock the entry.

Source file: rec_lmatch.uc

3.3.1.3) ReservationTable_Unlock

Input: table entry address

Output: N/A

Descriptions:

Unlock the memory location specified by the entry address.

Source file: rec_lmatch.uc

3.3.1.4) mul

Input: rate, cycle_diff

Output: accum

Descriptions:

In order to be efficient and simple to implementation, there are two major challenges in this multiplication macro. First, the token generating rate, bytes per cycle, is not likely to be integer, so we have to think of a workaround to avoid floating point operation. Second, we try to squeeze all the operands and intermediates inside 32-bit registers, so we don't have use another register to store the carryouts. Under these two limitations, we follow the following steps to do the calculation:

1. Rate is stored in the ReservationTable as binary fraction format with 16-bits right shifted. For example, rate of 0.5 will store as 0x8000 0000 and then right shift to 0x0000 8000.
2. Multiply the lower 16 bits cycle by rate, and store the product in accum. (Since this is a 32 x 16 bits multiplication, the 32-bit register is large enough to accommodate the product)
3. Right-shift the accum by 16 bits, (In essence, discard the lower 16 bits previous product)
4. multiply the higher 16 bits cycle by rate, and store the product in accum. (Again, the accum register should be enough to store the result)

At this point the value stored in accum is the product of the multiplication. After the operation, the precision of the result is compromised, but we acquire the simplicity and efficiency in return. The range of rate can be represented by our floating point arithmetic is from around 2.5KBps to 160MBps.

Source file: rec_lmatch.uc

3.3.1.5) `calc_tokens`

Input: rate, last cycle0, last cycle1, burst, residue tokens

Output: new tokens

Descriptions:

This macro will calculate available tokens for the current flow. IXP 1200 will provide current cycle count by two 32-bit registers. We don't have to worry about the overflow of the clock cycle count, because it will take more than 83 years before overflow happens.

On the other hand, we have to deal with the 64-bit subtraction. The highest bit of the lower 32-bit register is a numeric bit, not sign-bit. Since we don't know any 'unsigned' comparison in microcode (current comparison operation will treat the highest '1' as a minus sign instead of a regular number '1'), we right-shift both operands one bit before doing the comparison. We lost one digit of precision. It should be fine, since it is the least significant bit.

Moreover, in order to avoid long and complex multiplication, we make an assumption that if the difference of current cycle and last cycle is more than 2^{32} , we will automatically assign the maximal value (the 'burst' value) to the new_token. In current design model, the clock frequency is 166MHz, and it will take 25.87 seconds to make the difference of cycles more than 2^{32} . In our opinion, it is long enough for the buffer to accumulate maximum tokens.

Source file: `rec_lmatch.uc`

3.3.1.6) `find_queue`

Input: port

Output: queue ID

Descriptions:

This macro looks up the queue descriptors of the port specified by port ID passed in. It then determines which queue should be serviced according to absolute priority queue scheme. In another word, the only for low priority queue to be serviced is high priority queue is EMPTY and low priority queue is NOT EMPTY.

Source file: `xmit_8x100.uc`

3.3.2) Packet Flow

With QoS capabilities are incorporated, additional processing has to be done on each arrival packet. After route lookup, the TOS field is extracted and the corresponding bit pattern is cleared. The source and destination addresses are taken out from the header for reservation table lookup (3.3.1.1). If it is identified as reserved flow, we then calculate the new tokens available for this flow at this instant of time (3.3.1.5). The token bucket algorithm used here will be discussed in next sub-section. The new tokens value is checked against with current packet size and if it is larger, mark the P-bit on TOS field as a variable, and set queue ID to 0x0 (high priority queue). If there is not enough tokens, the current packet will be discarded. However, if this does not belong to any reserved flow, it will simply set the queue ID to 0x1 (low priority queue). After these processing, the packet will execute the rest of receiving thread as the reference design. Flow diagram is shown in Figure 6.

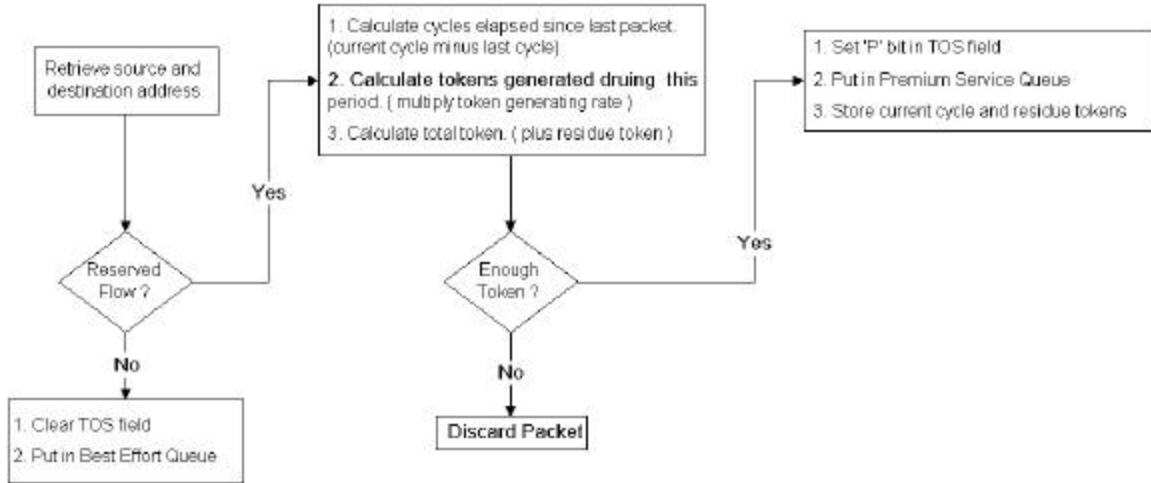


Figure 6 Packet Flow Diagram

3.3.3) Token Bucket Algorithm

The Token Bucket algorithm is implemented in the data plane to enforce the flow control, as shown in Figure 6. In our project, the bandwidth broker will store the reservation information in Reservation Table. Based on the information, the control plane will calculate how many tokens are available for a specific flow. The complete flow of token bucket algorithm is described as the following.

When the packet is classified as reserved flow, we retrieve necessary information from the table to calculate available tokens for this flow. First of all, we will calculate how many clock cycles has elapsed since last time we received a packet from the same flow. We store current cycle in the Reservation Table, so, based on the difference, we know how much time (in clock cycles) has gone since last packet from the same flow. Then we multiply the cycle difference by rate (bytes per cycle) to get how many tokens (in bytes) are generated in this time period (3.3.1.5). The residue tokens from last packet are stored in the Reservation Table and plus it with new generated tokens, we can get total available tokens.

If the available tokens are more the burst limit, the extra tokens will be dropped. If the flow has enough tokens to send this packet, this packet will be sent via Premium Service Queue and its TOS field's 'P' bit will be set. Meanwhile, the available tokens will subtract the packet size, and current time (cycle counts) and residue tokens will be stored in the Reservation Table. On the other hand, if there are not enough tokens to send, the packet will be dropped. According to [1], the packet should be saved in another queue and sent after enough tokens are accumulated, but due to the limitation of time, we will drop it for the time being.

4) Evaluations

This section is going to provide some results and analysis on how our implement performs during simulations. The evaluations are basically divided into 3 levles: cycle count (low), end-to-end delay (medium), and traffic policing (high).

4.1) Clock Cycle Count

In this section, we are going to examine how many extra clock cycles are introduced in our implementation. We work hard to make the microcode more efficient and clock cycle is a very intuitive indicator to show our efforts. Since the time to execute memory access instruction is undetermined, we cannot just count

how many instructions we wrote in our code. It cannot truthfully reflect the overhead we incurred. Plus, we also have to take the time waiting for scheduling between threads within same microengine into consideration. Having those considerations in mind, we wrote the following microcode to count the extra cycles we caused.

4.2.1) Experimental Setup

We time-stamped the clock cycle and stored it in SRAM before proceeding to our code. After exiting our code, we timestamp the cycle clock again and store the difference. Every experiment, we only use one thread to do the counting. If we use more than one thread to do the cycle counting, the extra execution time (caused by our cycle counting code) will inaccurately increase the waiting time during execution.

	Receiving	Transmitting
Actual Cycle counts	245.7319	80.4057

4.2.2) Data Analysis

In our experiments, we gathered the time cycle counts introduced from the receiving and transmitting part. The receiving part is responsible for the Reservation Table search and token calculation and the transmitting part is for the transmit scheduler to decide which queue should be serviced. Most of the cycles are consumed by table access (memory access) and token calculation. On the other hand, the latency in transmission part is essentially contributed by queue descriptors lookup (memory access) to carry out absolute priority scheduling.

4.2) End-to-End Delay

In this section, we are going to analyze how does our implementation affect the end-to-end delay. Although last section has already provide a low level latency in terms of cycle count, this analysis is going to focus towards end-to-end application level latency. System/Network administrators can use this result with low-level latency to obtain a better understanding of how the system performs.

4.2.1) Experimental Setup

We basically implemented two lightweight applications, transmitter, and receiver, running on 2 separate hosts directly connected to the IXP router. The transmitter sends one packet with defined size at a time to receiver and the receiver bounces it back to transmitter. Based on time-stamping sending and receiving echo, the transmitter can obtain the round trip delay between two hosts. The end-to-end delay is half of round trip time. In real world, it is true that one-way delay is NOT equal to half of round trip time because of various issues such as queueing delay, and asymmetric routing. However, in our setup, since the router is directly connected to both hosts, no asymmetric routing will occur; in addition, transmitter will wait for echo before sending additional packet and it eliminates the possibility of queueing delay. We ran this echo experiment for 1 million times and the statistical results (in micro second) are as follow. We pre-set 4 entries on the reservation table.

End-to-End Delay	Min	Average	Lower 10% Average
Unmodified	75.5	84.3298	79.9238
DiffServ: Unreserved	80.5	86.11255	83.3362
DiffServe: Reserved	84.5	88.2391	84.7846

4.2.2) Data Analysis

The 'Min' is the minimum value across 1 million values; 'Average' is the total average; whereas 'Lower 10% Average' is the average value of smallest 10% values. We choose to have lowest 10% data because we believe it has the least operating system scheduling effect since our application is run as user level process and thus better reflect the network effect.

From the table, there are gradual increases across the values. The unmodified project is the original reference design from Intel® and thus we use it as a control experiment. It is well expected that it has the lowest value across different statistical results. With our implementation but for unreserved flow, it has a larger delay than reference design because of searching on reservation table. Currently, the sequential search is used and the delay is largely depending on size of table. If the table size grows, the latency in this case will increase correspondently as it has to go through to the whole table to figure out if it is an unreserved flow. In reserved flow situation, the average case for sequential search is half n (where n is table size). It's true that it is likely to spend less time on searching and memory accesses, however, majority of delay is contributed by token bucket calculation: 64-bits subtraction and multiplication is a rather complicated process such that it has the largest latency. It is possible that for unreserved flow having larger latency than reserved traffic and it all depends on table size and entry locations.

4.3) Traffic Policing

One of the important features brought by edge route is traffic policing and shaping. Premium Service carries out a strict limit on bandwidth consumption in which any excess traffic will be discarded. In another word, all reserved traffics, after go through edge router, must conform with its corresponding rate. This is a curial property for Premium service. As mentioned, Premium traffic has absolute priority over Best Effort traffic, conformed Premium traffic avoids starving other packets indefinitely. We ran two experiments to show if our token bucket implementation can effectively carry out this feature.

4.3.1) Experimental Setup

We ran 2 experiments and both are based on a streaming video. The video file used in both cases is encoded in 300kbps with constant bit rate as real media format². The first experiment configures the reservation table such that only half bandwidth reserved while the second experiment has full reserved bandwidth. Then, base on the statistic provided by RealPlayer®³, we have the following graphs, Figure 7, and Figure 8.

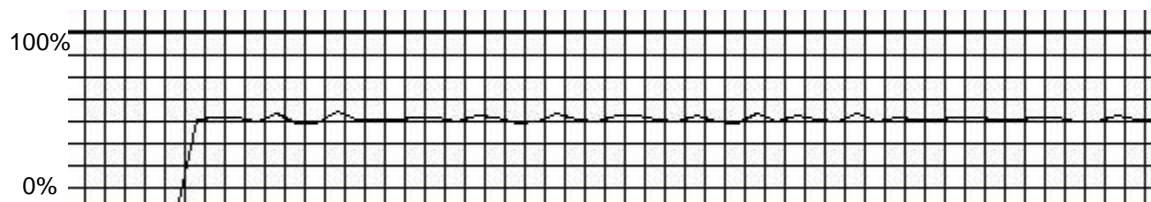


Figure 7 Half Bandwidth Reservation



Figure 8 Full Bandwidth Reservation

² Real Media File Format (extension .rm) is a streaming media format from RealNetwork, Inc.

³ RealPlayer® is a registered trademark of RealNetwork, Inc.

4.3.2) Data Analysis

In both graphs, x-axis is the time while y-axis is the percentage of bandwidth obtained out of required. The required bandwidth is equal to encoding rate, in our case, is 300kps. The first graph clearly shows that the edge router can effectively police the traffic according to its reserved rate. Since half bandwidth is reserved, the graph does show around 50% bandwidth is obtained. In the second experiment, since full bandwidth is reserved and it shows the streaming video can consume most bandwidth as expected. In conclusion, our implementation of token bucket can effectively and accurately police the traffic.

5) Future Works

The architecture we implemented provides the primary framework for quality of services. It is possible to further improve the functionalities and extensibilities. Here are some suggestions:

5.1) Finer Granularity

In our implementation, we use Layer 3 (IP Layer) information such as source and destination addresses pair to distinguish the flow. This setup can only provide a coarse flow and may not satisfy individual application requirement running on the same host. For example, a host running a FTP client and video-on-demand are sharing the same flow, which does not make sense. In order to provide a finer granularity, we can use Layer 4 (Transport Layer) information such as port number to differential the flow. This improvement can support a more specific need for individual application. The drawback is that it increase the workload on Bandwidth Broker and thus may affect the efficiency as it has to maintain more flow per node.

5.2) Assured Services

The architecture has already provide the necessary mechanism and frameworks for implementing Assured Services as described in [1]. Assured service will follow “expected capacity” usage profiles that are statistically provisioned. The assurance that user of such service receives is that such traffic is ‘unlikely’ to be dropped as long as it observes the expected capacity profile. An Assured service traffic flow may exceed its profile, but the excess traffic is not given the same assurance level.

The profile marker for Assured Service can also make use of token bucket algorithm as used by Premium Service. For Assured service, the token bucket depth represents the contracted burst size. When a token is present, packet has its corresponding bit (A-bit) set; otherwise, it is passed to the engine without any bit set. In priority queues, Assured traffic is inserted into low priority queue, which is managed by RIO mechanism from RED.

One of the difficulties for implementing Assured Services is the RED queue management. The management is based on 2 thresholds on average queue length. Since the IXP microcode does not provide any division functionality and using a macro may induce significant amount of latency for such operation and thus reduce the throughput of the router.

5.3) Multiple Domain Reservation

The Bandwidth Broker can extend its functionality to handle reservation across multiple domains. Under multiple domains environments, there are needs to be an agreement between the domains on the link shared exactly how to treat the traffic sent. Once the agreement is made the information need to be entered into the Bandwidth Broker. It should be in the form Domain A agrees to receive X amount of specific class of traffic from Domain B, and likewise Domain be makes a similar agreement. So a reservation across

domain starts in the same way a reservation in a single domain. The sender requests the reservation to its Bandwidth Broker, it in turn makes sure the domain is able to support it, and forwards the reservation to the next domain. The Bandwidth Broker of the second domain looks to see if the flow will exceed the amount of traffic specified between the 2 domains, as well as to see if the second domain is able to accommodate the flow. If it is satisfactory it sends the flow information to the edge router and a replay to the Bandwidth Broker of the first domain. The first domain's Bandwidth Broker sends the flow information to the edge node and a confirmation to the host requesting the flow.

6) Conclusions

Quality of Services is still an on-going research topic in computer science industry. Many research ideas have come out but it is hard to testify the outcomes. Intel® IXP platform provides a great development opportunities to implement and verify new ideas. In this project, we implement and simulate the design idea from [1]. In conclusion, we believe we have met the basic requirement of providing quality of services based on our simulation. However, due to incomplete knowledge of the IXP platform, we have simplified our design and limited our functionalities. We believe given more time and investigation on the platform, we can have a more robust and realistic design as mentioned in Section 5.

References

- [1] Nichols, K., Jacobson, V., and Zhang, L., "A Two-bit Differentiated Services Architecture for the Internet", November 1997.
- [2] Blake, S. et al, "An Architecture for Differentiated Services", RFC 2475, December 1998.
- [3] Jacobson, V., "Differentiated Services Architecture", August 1997.
- [4] Clark, D., and Wroclawski, J., "An Approach to Service Allocation in the Internet", August 1997.

Comments about the course and IXP platform:

We think that IXP platform providing a great tool to implement various research concepts such as Quality of Services, Active Network, etc, which are hard to testify by using standard switch/router nowadays. However, due to limited knowledge and support of this platform, we have a hard time to implement something in large. Besides, the problem created by its development environment also contributed much delays on the progress. We hope that next version of IXP can have these fixes.

18-884 *Network Design and Evaluation* is definitely a good course as it provides us opportunities to implement something that never have chance to do before. We learned that implementation on hardware is far more complex than on simulator used in other courses (microcode code on IXP vs C/C++ on software simulator). We think that after this year's experience on platform, this course will be more interesting and successful in future.