POLITECNICO DI TORINO

III Facoltà di Ingegneria
Corso di Laurea in Telecomunicazioni

Tesi di Laurea

A Simulation Study of Web Traffic over DiffServ Networks

Relatori:
Prof. Marco Ajmone Marsan
Dott. Claudio Casetti
Dott. Marco Mellia

Candidato:
Dario Rossi

Novembre 2001

Contents

List of Tables

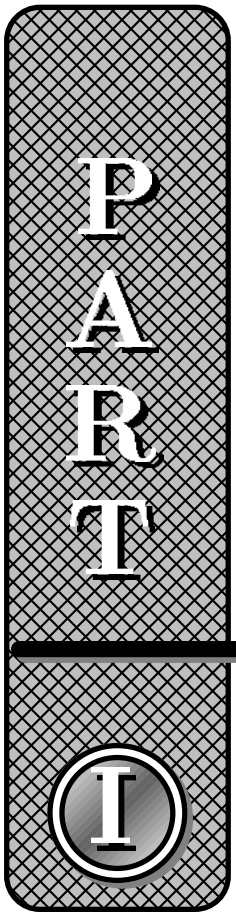
List of Figures

I DiffServ fundamentals

II Long Lived FTP Flows

C *ns* Code – Short Lived HTTP Flows

D Simulation Results – Short Lived HTTP Flows



DiffServ fundamentals

Chapter 1

Quality of Service Overview

Originally designed for research networks, the protocols used in today's Internet have been mainly optimized to provide connectivity: in these networks, the main problem was to reach the destination even if transmission quality was poor. Due to this fact, the Internet and most IP based networks provide today a Best Effort service, i.e. a context where the network does "its best" to transmit information as quickly as possible but does not provide any guarantee on the timeliness or even the actual delivery of this information.

In today's electronic trade context, there is a real demand for a minimum level of performance to be guaranteed to mission critical applications. This was to some extent supported in ATM networks which allow for rate based service categories; in general, with connection oriented networks, information transfers are allotted a guaranteed –yet unique– end to end service, but at the cost of complexity and execution overheads in network nodes.

This demand of performance guarantees reaches now the Internet, which has evolved to an international commercially operated network and has grown, in just over a decade, to over 40,000 sites and 3,000,000 hosts. The startling growth of Internet technology, coupled with the relatively low deployment cost of IP networks, have pushed for an integrating IP-based core – a single network for wireless, data, and voice access; the diverse service requirements, as well as novel traffic characteristics of the emerging Internet applications, imposes the need for providing *different levels of services* other than Best Effort. The *Quality of Service (QoS)*, a collective measure of such a level of the service delivered to a customer, is reflected by the extent of user satisfaction of the network; although also depending on application types and user perception, QoS is usually be measured by packet latency, jitter, packet loss rate and application throughput.

In this chapter we provide a brief excursus on QoS motivations, characteristic and goals, finally introducing two rather different IP QoS mechanisms proposed by the Internet Engineering Task Force (IETF): Integrated Services/RSVP and Differentiated Services.

1.1 The Need For IP QoS

IP [RFC791] networks provide Best Effort data delivery by default, which allows the complexity to stay in the end-hosts, so the network can remain relatively simple. This scales well, as evidenced by the ability of the Internet to support its phenomenal growth. As more hosts are connected, network service demands eventually exceed capacity, but service is not denied: instead it degrades gracefully.

Although the resulting variability in delivery delays (jitter) and packet loss do not adversely affect typical Internet applications –e-mail, file transfer and Web applications– other applications cannot adapt to inconsistent service levels: delivery delays cause problems for applications with real-time requirements, such as those that deliver multimedia, the most demanding of which are two-way applications like telephony. Increasing bandwidth is a necessary first step for accommo-

dating these real-time applications, but it is still not enough to avoid jitter during traffic bursts: even on a relatively unloaded IP network, delivery delays can vary enough to continue to adversely affect real-time applications.

To provide adequate levels of quantitative or qualitative forwarding determinism, IP services must be supplemented; this requires some architectural network models in order to distinguish traffic with strict timing requirements from those that can tolerate delay, jitter and loss. That is what QoS protocols are designed to do: QoS does not create bandwidth, but manages it so it is used more effectively to meet the wide range or application requirements; therefore, the goal of Internet QoS models is to provide some level of predictability and control beyond the current Best Effort service.

On the other hand, it should be said that whether mechanisms are even needed to provide QoS is a debated issue [QOSPIC]: one opinion is that fiber and dense wavelength division multiplexing technologies will make bandwidth so abundant and cheap that high quality of service will be automatically delivered; on the other hand it has been hypothesized that no matter how much bandwidth the Internet can provide, new applications will be created to consume it: therefore mechanisms will still be needed to provide QoS. Here we simply note that even if bandwidth will eventually become abundant and cheap, for now, some simple mechanisms are definitely needed in order to provide QoS on the Internet, and all major router vendors support this view by providing some QoS mechanisms in their products.

1.2 QoS Characteristics

It should be observed that QoS applicability range is not only restricted to the IP networks: although our study focuses on a specific I-QoS mechanism, it is nevertheless important to define and relatively situate these different macroscopic QoS areas. Actually, the several QoS mechanisms and architectural implementations can be divided into three basic groups, which align with the lower three layers of the OSI reference model:

Physical layer
: The physical layer consists of the physical wiring, fiber optics, and the transmission media in the network itself; while the implementation of provisioning diverse physical paths in a network is usually done to provide for backup and redundancy, this can also be used to provide differentiated services if the available paths each have differing characteristics. For example, best-effort traffic could be forwarded by the network layer devices (routers) along the lower-speed path, while higher priority (QoS) traffic could be forwarded along the higher-speed path.

Link layer
: The traffic service differentiation can be provided with specific link layer mechanisms; traditionally this belief in differentiation has been associated with Asynchronous Transfer Mode (ATM): ATM provides a complex subset of traffic-management mechanisms, Virtual Circuit (VC) establishment controls, and various associated QoS parameters for these VCs. The various services, namely Constant Bit Rate (CBR), Variable Bit Rate (VBR), Available Bit Rate (ABR) and Unspecified Bit Rate (UBR), provide various levels of QoS guarantees.

Network and transport layer
: In the global Internet, it is undeniable that the common bearer service is the TCP/IP protocol suite [RFC793]; therefore, IP makes implementation and management of QoS support easier and yields a greater possibility of successfully providing a QoS implementation.

In this latter class, a number of QoS protocols have evolved to satisfy the variety of application needs: the challenge of these IP QoS technologies is to provide differentiated delivery services for individual flows or aggregates without breaking the Net in the process. Therefore, in order to avoid potential problems as QoS protocols are applied to the Internet, the end-to-end principle is still

the primary focus of QoS architects: as a result, the fundamental principle of "Leave complexity at the 'edges' and keep the network 'core' simple" is a central theme among QoS architecture designs; this is not as much a focus for individual QoS protocols, but in how they are used together to enable end-to-end QoS.

We may recall that the quality of service in a network is reflected by the extent of user satisfaction of the network; although also depending on application types and user perception, QoS can usually be measured by packet latency, jitter, packet loss rate and application throughput. Some applications are more stringent about their QoS requirements than others, and for this reason we have two basic types of ability to provide some level of assurance for consistent network data delivery:

Resource reservation
: Network resources are apportioned according to an application's QoS request, and subject to bandwidth management policy.

Prioritization
: Network traffic is classified and network resources apportioned according to bandwidth management policy criteria; to enable QoS, network elements give preferential treatment to traffic identified as having more demanding requirements.

Both these types of QoS can be applied to individual application *flows* or to flow *aggregates*, hence there are two other ways to characterize types of QoS:

Per Flow
: A "flow" is defined as an individual, uni-directional, data stream between two applications, the sender and the receiver, uniquely identified by a 5-tuple including transport protocol, source address, source portnumber, destination address, and destination port number.

Per Aggregate
: An aggregate is simply constituted by two or more flows; typically the flows will have something in common, e.g., any one or more of the 5-tuple parameters, a label or a priority number, or perhaps some authentication information.

It should be pointed out that there are many more possible ways than those presented to characterize QoS; nevertheless, this grade classification introduced a few key aspect of the different services kinds available, allowing finally to discriminate between the two QoS models presented further in this chapter.

1.3 Internet QoS Protocols

The Internet today carries three basic categories of traffic, and any QoS environment must recognize and adjust itself to these three basic categories. The first category is long held adaptive reliable traffic flows, like long TCP sessions, where the end-to-end flow rate is altered by the end points in response to network behavior: the flow rate attempts to optimize itself in an effort to obtain a fair share of the available resources on the end-to-end path. The second category of traffic is that of short duration reliable transactions: since the flows are of very short duration, the rate adaptation does not get established within the lifetime of the flow, sitting completely within the startup phase of the TCP adaptive flow control protocol. The third category of traffic is an externally controlled load unidirectional traffic flow, which is typically a result of compression of a real time audio or video signal; in this case the peak flow rate equals the basic source signal rate and the transportation mechanism is an unreliable traffic flow with a UDP unicast flow model.

In order to provide elevated service quality to these three common traffic flow types, there are three different engineering approaches that may be used. To ensure efficient transmission of long held, high volume TCP flows, the network should offer consistent signaling to the sender regarding the onset of congestion loss within the network. Efficient transmission of short duration, TCP

traffic requires the network to avoid sending advance congestion signals to the flow end-points; since these flows are of short duration and low transfer rate, any such signaling will not achieve any appreciable change in the relative allocation of network resources, whereas it will substantially increase the elapsed time that the flow is held active, eventually resulting in poor delivered service. For efficient transport of the UDP unicast traffic the network should allow the source to specify its traffic profile in advance, and should respond with either a commitment to carry such a load, or indicate that it does not have available resources to meet the requirements.

As a consequence, it should be noted that difficultly a single transport or network layer mechanism will provide the capabilities for differentiated services for all flow types, and that a QoS network will deploy a number of mechanisms to meet the requirements in this area. Therefore the specific applications, network topology and policy dictate which type of QoS is most appropriate for individual flows or aggregates; to accommodate the need for these different types of QoS, there are a number of different QoS protocols and algorithms, among which we may cite:

ReSerVation Protocol (RSVP)
: Provides the necessary signaling mechanism to enable network resource reservation; although typically used on a per-flow basis, RSVP is also used to reserve resources for aggregates.

Integrated Services (IntServ)
: Network resources are subject to bandwidth management policy, and are reserved via a signalling protocol (e.g. RSVP) according to application's specific QoS requests.

Differentiated Services (DiffServ)
: Provides a coarse and simple way to *categorize and prioritize* network traffic flow aggregates.

Multi Protocol Labeling Switching (MPLS)
: Provides bandwidth management for aggregates via network routing control according to labels in encapsulating packet headers.

Subnet Bandwidth Management (SBM)
: Enables categorization and prioritization at Layer 2, that is the data-link layer in the OSI model, on shared and switched IEEE 802 networks.

Finally, we provide a brief description of two interesting and rather different QoS models: IntServ and DiffServ. Since the simulation study conduced is based on the latter model, a deeper and exhaustive description will be the subject of Ch. 2; nevertheless, some introductory considerations are necessary to define both DiffServ goals and macroscopic behavior, in order to facilitate its understanding.

1.4 Integrated Services and RSVP

Integrated Services (IntServ) [RFC1633] are implemented by four components: the signaling protocol, the admission control routine, the classifier and the packet scheduler. Applications requiring guaranteed service or controlled-load service must set up the paths and reserve resources before transmitting their data. The admission control routines decides whether a request for resources can be granted: when a router receives a packet, the classifier performs a multi-field (MF) classification and put the packet in a specific queue based on the classification result; the packet scheduler will then schedule the packet accordingly to meet its QoS requirements.

The IntServ/RSVP represents a fundamental change to the traditional Internet architecture, which is founded on the concept that all flow-related state information should be in the end systems. Two service classes in addition to best effort service are proposed:

Guaranteed Service: provides mathematically provable upper bounds for queuing delays for each packet, and it is intended for extremely intolerant applications .

Controlled Load Service: emulates lightly loaded network even during congestion taking advantage of statistical multiplexing, which may sometimes lead to overflows and service degradation; controlled load is suited for adaptive and elastic applications requiring reliable and enhanced best effort service.

The philosophy of this model is that, in order to provide special QoS for specific user, routers must be able to reserve resources: this requires therefore to handle flow-specific state. IntServ is usually coupled with RSVP [RFC2205][RFC2208], which is part of a larger effort to enhance the current Internet architecture with support for Quality of Service flows. The RSVP protocol is used by a host to request specific QoS from the network for particular application flows – but can be also used by routers to deliver QoS requests to all nodes along the path of the flow and to establish and maintain state to provide the requested service. RSVP requests will generally result in resources being reserved in each node along the data path; briefly:

- the sender sends a PATH message to the receiver, specifying the traffic characteristics
- every intermediate router along the path forwards the PATH message to the next hop determined by the routing protocol
- upon receiving a PATH message, the receiver responds with a RESV message to request resources for the flow
- every intermediate router along the path can reject or accept the request of the RESV message
- if the request is rejected, the router will send an error message to the receiver, and the signaling process will terminate
- if otherwise the request is accepted, link bandwidth and buffer space are allocated for the flow, and the related flow state information will be installed in the router

If they could be fully deployed in the Internet, IntServ and RSVP would provide excellent support for real-time and loss sensitive traffic. Recently there has been, however, some doubts about IntServ architecture deployment due to the following problems:

- the amount of state information increases proportionally with the number of flows; this places a huge storage and processing overhead on the backbone routers and therefore, this architecture does not scale well in the Internet core
- the requirement on routers is high, since all routers must support RSVP, admission control, MF classification and packet scheduling
- ubiquitous deployment is required for guaranteed service, whereas incremental deployment of controlled-load service is possible by deploying RSVP functionality at the bottleneck nodes of a domain and tunneling the RSVP messages over other part of the domain.
- may not be feasible for data applications such as WWW browsing, where the duration of a typical flow is only a few packets: the overhead caused by signalling could easily deteriorate the network performance perceived by the applications, especially if latency is concerned.

1.5 Differentiated Services

Because of the difficulty in implementing and deploying Integrated Services and RSVP, Differentiated Services is introduced in [RFC2474][RFC2475]; moreover, interoperability between the two models is possible using the framework defined in [RFC2998]. This section gives a big picture of Differentiated Services, describing the requirements that addressed its development and introducing the ideas behind model, as well as the components required for its implementation; finally, a comparison of the IntServ versus DiffServ and a list of the principal DiffServ terms and acronyms –given for the ease of the reader– complete this overview.

1.5.1 Model Properties

The history of the Internet has been one of continuous growth in the number of hosts, the number and variety of applications, and the capacity of the network infrastructure. Since this growth is expected to continue for the foreseeable future, there is a number of requirements that were identified and addressed in this scalable QoS architecture. Summarizing, DiffServ should:

- accommodate a wide variety of services and provisioning policies, extending end-to-end or within a particular set of network
- allow decoupling of the service from the particular application in use
- work with existing applications, without the need for application programming interface changes or host software modifications
- decouple traffic conditioning and service provisioning functions from forwarding behaviors implemented within the core network nodes
- not depend on hop-by-hop application signaling
- require only a small set of forwarding behaviors (whose implementation complexity does not dominate the cost of a network device, and which will not introduce bottlenecks for future high-speed system implementations)
- avoid per-microflow or per-customer state but utilize only aggregated classification state within core network nodes
- permit simple packet classification implementations in core network nodes
- permit reasonable interoperability with non-DS-compliant network nodes
- accommodate incremental deployment

1.5.2 Architecture Overview

Differentiated Services are intended to provide a framework and building blocks to enable deployment of scalable service discrimination in the Internet.

The aim is to speed deployment by separating the architecture into two major components: the (fairly well-understood) behavior in the **forwarding path** and the (more complex and still emerging) background policy and **allocation component** that configures parameters used in the forwarding path. This is somehow analogous to original design of the Internet, where the decision was made to separate the *forwarding* and *routing* components: packet forwarding is the relatively simple task that needs to be performed on a per-packet basis as quickly as possible, whereas routing tables are maintained as a background process to the forwarding task; further, routing is the more complex task and it has continued to evolve over the past 20 years.

In the packet **forwarding path** differentiated services are realized by mapping the *DS Code-Point (DSCP)* contained in a field of the IP packet header to a particular forwarding treatment, or *Per-Hop Behaviors (PHB)*, at each network node along its path. Per-Hop Behaviors and mechanisms to select them on a per-packet basis, can be deployed in network nodes today and it is this aspect of the differentiated services architecture that is being addressed first.

Specifically, IPv4 header contains a Type Of Service (TOS) byte, renamed by DiffServ as differentiated services field *(DS field)* that holds the DSCP value: packets are selected with respect to their DSCP value and assigned to different *Behavior Aggregate (BA)*, representing specific traffic classes forwarded according to the correspondant PHB. PHBs are expected to be implemented by employing a range of *Packet Scheduling*and *Active Queue Management (AQM)* disciplines on both *edge* and *core* devices; moreover, in order to enforce specific delivery requirements, PHBs may require some *Traffic Conditioning (TC)* (i.e monitoring, policing, and shaping) to be performed

on the traffic at the network boundaries. Additionally, DSCP *Marking* is performed by traffic conditioners at network boundaries, including the edges of the network, first-hop router, source host and administrative boundaries.

Packets transiting in a *DS domain*, that is a contiguous set of nodes implementing a common set of PHBs and TCs, are firstly classified at the *boundary* via a *MF Classifier*, then marked and possibly conditioned, whereas within the *core* of the domain, packets need only to be steered to a particular BA depending on their DSCP via a *BA Classifier*.

Services end-to-end are realized through concatenation of edge-to-edge intra-domain services implemented using PHBs and TCs: the wide deployment of traffic conditioners is therefore important to enable useful services implementation, though their actual use may evolve over time. Currently, two rather different kind of services may be implemented using different PHBs:

- *Assured Forwarding (AF) PHB*: provides realiable and timely delivery
- *Expedited Forwarding (EF) PHB*: provides reliable, low delay and low jitter delivery

DiffServ is extended across domain boundaries by establishing a *Service Level Agreement (SLA)* between the peering domains: the PHB configuration is specified in the *Service Level Specification (SLS)* part of the SLA, whereas packet classification, re-marking and conditioning rules may be specified in the *Traffic Conditioning Agreement (TCA)* part. The extent of traffic conditioning required is dependent on the specifics of the service offering, and may range from simple codepoint re-marking to complex policing and shaping operations.

The **allocation component** of the DiffServ model, thus the configuration of network elements and the kinds of rules that may be applied to the use of resources, is much less well-understood: as a note, it can be said that dynamic SLAs configuration may use the *Bandwidth Broker (BB)* "oracles", or even use RSVP [DSRSVP], to request for services on demand. Nevertheless, it is possible to deploy useful differentiated services in networks by using simple policies and static SLA configurations, which may be negotiated on a regular basis, e.g. monthly and yearly. Actually, there are a number of ways to compose per-hop behaviors and traffic conditioners to create services: in the process, additional experience is gained that will guide more complex policies and allocations.

1.5.3 DiffServ vs IntServ

Although the DiffServ architecture can be contrasted with a wide range of existing QoS models (e.g. relative priority marking, service marking, label switching, ...) we focus on its comparison with the described Integrated Services/RSVP approach. The two solutions, each achieving a different flavour of QoS, are not interchangeable: this is a consequence of IntServ's scalability problems and DiffServ's problems in the allocation/accounting of the resources needed by applications.

The IntServ/RSVP model relies upon traditional datagram forwarding, but allows sources and receivers to exchange signaling messages – which establish additional packet classification and forwarding state on each node along the path between them. In the absence of state aggregation, the amount of state on each node scales in proportion to the number of concurrent reservations, which can be potentially large on high-speed links. Conversely, there are only a limited number of service classes indicated by the DS field; since resources are allocated in the granularity of class, the amount of state information is proportional to the number of classes rather than the number of flows: DiffServ is therefore more scalable.

Moreover, sophisticated MF classification, marking, policing and shaping operations are only needed at the *edge* of a DiffServ domain: ISP *core* routers need only to implement behavior aggregate BA classification. Conversely, IntServ requirement on routers is high, since *all* routers must support RSVP, admission control, MF classification and packet scheduling; in addition, IntServ also requires application support for the RSVP signaling protocol. However, the requirement for hop-by-hop signaling may be avoided by utilizing only static classification and forwarding policies, which are implemented in each node along a network path; nevertheless, the state requirements

for this variant are potentially worse than those encountered when RSVP is used, especially in backbone nodes: the number of static policies that might be applicable at a node over time may be larger than the number of active sender-receiver sessions that might have installed reservation state on a node.

Finally, IntServ need ubiquitous deployment for guaranteed service, and RSVP signalling is still needed at the boundaries for incremental deployment of controlled load service. In the DiffServ model, incremental deployment is possible in the current Internet without any additional requirements, at least for AF PHB based services: non-DS-compliant routers simply ignore the DS fields of the packets and give all packets best effort service; even in that case, AF packets are less likely to be dropped by DS-capable routers, and the performance of AF traffic might still be better with respect to common Best Effort traffic performance.

1.5.4 DiffServ Terminology

The following list items are mainly reported in [RFC2475]; however, some minor modifications were made, in order both to stress the most relevant references and to include the later developed DS items.

Behavior Aggregate (BA)
: a collection of packets with the same DS codepoint crossing a link in a particular direction

Classifier
: an entity which selects packets based on the content of packet headers according to defined rules; specifically, BA classifier selects packets based only on the contents of the DS field, whereas a multi-field (MF) classifier selects packets based on the content of some arbitrary number of header fields (typically some combination of source address, destination address, DS field, protocol ID, source port and destination port)

DS Boundary Node
: a DS node that connects one DS domain to a node either in another DS domain or in a domain that is not DS-capable: the link between these edge nodes is said to be a Boundary link

DS-Capable
: capable of implementing differentiated services as described in this architecture; usually used in reference to a domain consisting of DS-compliant nodes

DS Codepoint (DSCP)
: a specific value of the DSCP portion of the DS field, used to select a PHB

DS-Compliant
: enabled to support differentiated services functions and behaviors as defined in [RFC2474] and [RFC2475], usually used in reference to a node or device

DS Domain
: a DS-capable domain; a contiguous set of nodes which operate with a common set of service provisioning policies and PHB definitions

DS Egress Node
: a DS boundary node in its role in handling traffic as it leaves a DS domain

DS Ingress Node
: a DS boundary node in its role in handling traffic as it enters a DS domain

DS Interior Node
: a DS node that is not a DS boundary node

DS Field
the IPv4 header TOS octet or the IPv6 Traffic Class octet when interpreted in conformance with the definition given in [RFC2474]: the bits of the DSCP field encode the DS codepoint, while the remaining bits are currently unused

DS Region
a set of contiguous DS domains which can offer differentiated services over paths across those DS domains

Downstream DS Domain
the DS domain downstream of traffic flow on a boundary link

Dropper
a device that performs dropping, that is the process of discarding packets within a traffic stream in accordance with the state of a corresponding meter enforcing a traffic profile

Legacy Node
a node which implements IPv4 Precedence as defined in [RFC791][RFC1812] but which is otherwise not DS-compliant

Marker
a device that performs marking, that is, the process of setting the DS codepoint in a packet based on defined rules; a marker can also perform packets pre-marking (set the DS codepoint of a packet prior to entry into a downstream DS domain) and re-marking (change the DS codepoint of a packet, usually performed by a marker in accordance with a TCA)

Meter
a device that performs metering, that is, the process of measuring the temporal properties (e.g., rate) of a traffic stream selected by a classifier; the instantaneous state of this process may be used to affect the operation of a marker, shaper, or dropper, and/or may be used for accounting and measurement purposes.

Microflow
a single instance of an application-to-application flow of packets which is identified by source address, source port, destination address, destination port and protocol id.

Per-Domain Behavior (PDB)
the expected treatment that an identifiable or target group of packets will receive from edge-to-edge of a DS domain; a particular PHB or, if applicable, list of PHBs, and traffic conditioning requirements are associated with each PDB

Per-Hop-Behavior (PHB)
the externally observable forwarding behavior applied at a DS-compliant node to a DS behavior aggregate

PHB group
a set of one (a single PHB is a special case of a PHB group) or more PHBs that can only be meaningfully specified and implemented simultaneously, due to a common constraint applying to all PHBs in the set such as a queue servicing or queue management policy; a PHB group provides a service building block that allows a set of related forwarding behaviors to be specified together (e.g., four dropping priorities)

Service
the overall treatment of a defined subset of a customer's traffic within a DS domain or end-to-end

Service Level Agreement (SLA)
: a service contract between a customer and a service provider that specifies the forwarding service a customer should receive; a customer may be a user organization (source domain) or another DS domain (upstream domain); a SLA may include traffic conditioning rules which constitute a TCA in whole or in part

Service Level Specification (SLS)
: is a set of parameters and their values which together define the service offered to a traffic stream by a DS domain; it is expected to include specific values or bounds for PDB parameters.

Service Provisioning Policy
: a policy which defines how traffic conditioners are configured on DS boundary nodes and how traffic streams are mapped to DS behavior aggregates to achieve a range of services

Shaper
: a device that performs shaping, that is the process to alter the temporal characteristics of the stream (i.e. delay packets) and bring it into compliance with a traffic profile

Traffic Aggregate (TA)
: a collection of packets with a codepoint that maps to the same PHB, usually in a DS domain or some subset of a DS domain: this generalizes the concept of BA from a link to a network

Traffic Conditioner (TC)
: an entity which performs traffic conditioning functions and which may contain meters, markers, droppers, and shapers: traffic conditioners are typically deployed in DS boundary nodes only; a traffic conditioner may re-mark a traffic stream or may discard or shape packets control functions performed to enforce rules specified in a TCA, including metering, marking, shaping, and policing

Traffic Conditioning Agreement (TCA)
: an agreement specifying classifier rules and any corresponding traffic profiles and metering, marking, discarding and/or shaping rules which are to apply to the traffic streams selected by the classifier; a TCA encompasses all of the traffic conditioning rules explicitly specified within a SLA along with all of the rules implicit from the relevant service requirements and/or from a DS domain's service provisioning policy

Traffic Profile
: a description of the temporal properties of a traffic stream such as rate and burst size

Chapter 2

DiffServ Architecture

This chapter deeply describes the details of the DiffServ QoS model, earlier overviewed in Sec. 1.5. Organization is as follows: rigorous definition of the DS field and the DSCP is given in Sec. 2.1, whereas Sec. 2.2 focuses on DS domain topology issues; general semantic of Per-Hop Behaviors, as well as possible classifiers, schedulers and active queue management mechanisms needed for their implementation, are subject of Sec. 2.3; traffic conditioning is developed in Sec. 2.4, illustrating some of the proposed building blocks; the original 2-bit DiffServ architectural proposal is outlined in Sec. 2.5, in order to both explain the resource allocation topic and to introduce the motivations that led to the actually defined specific forwarding treatments; these are the Expedited Forwarding PHB (Sec. 2.6) and, the Assured Forwarding PHB Group (Sec. 2.7): since simulative study will focus on the latter, special attention will be given to it, reporting examples of services that may be build on his top.

2.1 DiffServ Field

A replacement IP packet header field, called the *Differentiated Services Field*, intended to supersede the existing definitions of the IPv4 TOS octet [RFC1349] and the IPv6 Traffic Class octet [RFC2460], is presented in Fig. 2.1.

To select the PHB a packet experiences at each node, only six bits of the DS field are used as a *DS CodePoint (DSCP)*, while the two-bit currently unused (CU) field, reserved for Local or Experimental Use, is ignored.

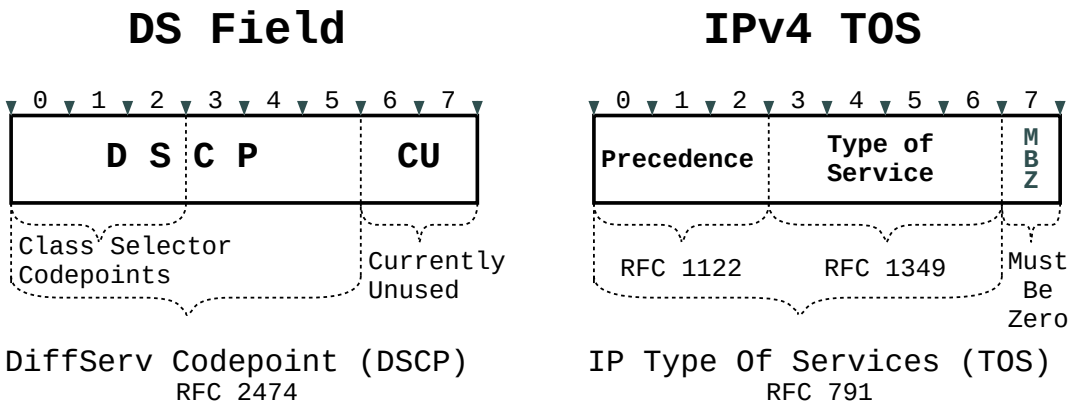


Figure 2.1: Differentiated Services Field Structure

The DSCP field (which we will refer to as `xxxxxx`, where `x` may equal `0` or `1`) within the DS

field is capable of conveying 64 distinct codepoints, whose space is divided, for assignment and management purposes, into three *Pools*:

1 : (`xxxxx0`) 32 *recommended* codepoints individuating Standards Action

2 : (`xxxx11`) 16 codepoints to be reserved for Experimental or Local Use (EXP/LU)

3 : (`xxxx01`) 16 codepoints initially available for EXP/LU, but which should be utilized for standardized assignments if Pool 1 is ever exhausted.

PHBs selection mechanism needs a matching against the entire 6-bit DSCP field, thus treating the value of the field as the index of a configurable mapping table: different DSCP(s) can be used for a PHB, either in addition to or in place of the recommended one; being the total space of codepoints larger than the recommended DSCP space (Pool 1), a DSCP→PHB mapping table may contain both 1→1 and N→1 mappings.

2.1.1 Backward Compatibility Issues

The DS field will have a limited backward compatibility with deployed uses of the IP Precedence Field of the IPv4 TOS octet, while no attempt in that direction is made toward the DTR (or TOS) bits.

The notion of Precedence, introduced in [RFC791], was defined as an independent measure of the datagram importance; Tab. 2.1, confirming the validity of the IP Precedence field use, reports the specific original assignment of the priorities, which stated as merely historical in [RFC1122]. The lowest *Routine* priority is supported in DiffServ architecture by the Default PHB, corresponding to a specific Class Selector Codepoint.

`000`	Routine	`100`	Flash Override
`001`	Priority	`101`	CRITIC/EPC
`010`	Immediate	`110`	Internetwork Control
`011`	Flash	`111`	Network Control

Table 2.1: IP Precedence Values and Meanings

2.1.1.1 Default PHB

A *Default PHB*, whose codepoint is `000000`, will be available in every DS-compliant node: the common Best Effort forwarding behavior.

When no other agreements are in place or where a codepoint is not mapped to a standardized or locally used PHB, it is assumed that packets belong to this aggregate. A reasonable policy for constructing services would have to ensure that the BE aggregate was not *starved* due to resources pre-emption: this could be enforced by reserving some minimal buffers and bandwidth resources in every domain node.

2.1.1.2 The Class Selector Codepoints

The Class Selector Codepoints is the DiffServ approach used to preserve backward compatibility with IP Precedence field without unduly limiting future flexibility. The Class Selector Codepoints set defines eight codepoints `xxx000`, drawn from Pool 1, that must be mapped *not to specific* PHBs, but to PHBs that meet *at least* a minimum requirements set. The DSCP relative order in that set is ranked basing on DSCP numerical value: this ranking must yields to two independently forwarded classes of traffic, reflecting the priority order in timely forwarding packets probability; moreover PHBs selected by codepoints `11x000` have a preferential forwarding treatment with respect to the default PHB, in order to preserve common usage of IP Precedence.

2.2 DiffServ Topology

The DiffServ architecture is based on a simple model that provides classification and possibly conditioning of the traffic mainly at the *boundaries* of the network; this is done in order to rank the traffic in different behavior aggregates, identified by a single DS codepoint: within the *core* of the network, packets are forwarded according to the per-hop behavior associated with the DSCP. This section introduces, define and differentiate the DiffServ routers, a key element for the architecture deployment, further illustrating their relationship and functions when inserted in a DiffServ network.

2.2.1 DiffServ Domain

A DS Domain is a contiguous set of DS nodes implementing a common set of PHBs; normally it consists of one or more networks under the same administration, responsible for ensuring adequate resources provisioning. A first important differentiation of nodes belonging to the same domain is whether the node is in the interior or on the boundary of the network; the latter case may be further discriminated based on traffic direction, as Fig. 2.2 depicts; anyway, since network nodes can be unaware of DiffServ architecture, we firstly need to describe and discriminate such routers in order to examine their effect and relationship with DS compliant ones.

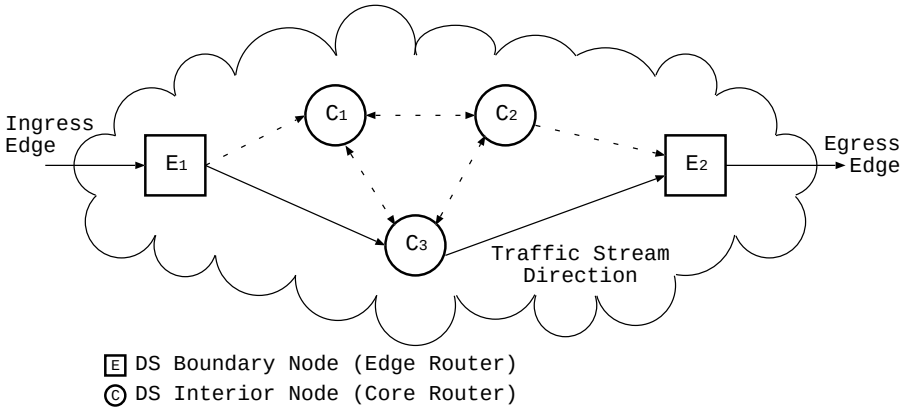


Figure 2.2: DiffServ Domain Synoptic

2.2.1.1 Non-DiffServ Router

A node is said to be *Non DS Compliant* when either it does not interpret the DS field as specified in Sec. 2.1, or it does implement only partly some or all the standardized PHBs.

A special case within non DS compliant nodes is represented by *DS Legacy* node, which implements IPv4 Precedence classification and forwarding but which is otherwise non DS compliant. This means that a legacy node can only support a small group of PHBs: precedence values are compatible by intention with the Class Selector Codepoints (see Sec. 2.1.1.2) and specularly the precedence forwarding behaviors comply with the Class Selector PHB requirements. Furthermore, a legacy node may interpret the TOS field of the TOS byte, but won't be nevertheless capable to fully support PHBs selection via DSCP discrimination.

Nodes which are non DS compliant and are not legacy nodes, (which will be further indicated as "strictly non DS compliant") will thus uniquely be able to deploy the common Best Effort forwarding treatment, and may exhibit unpredictable behaviors for packets marked other than with the null codepoint associated to the Default PHB.

2.2.1.2 DiffServ Edge and Core Router

A DS domain has a well-defined boundary consisting of *DS Edge* nodes interconnecting the DS domain to other DS or non-DS-capable domains; these routers are also identified as DS Boundary nodes. Their tasks are mainly to classify and possibly condition ingress traffic to ensure that packets crossing the domain are appropriately marked to select a PHB from one of the PHB groups supported within that domain. These switches consists of several building blocks, whose description will be further developed in the next sections, each of which accomplishes a different forwarding path task: an edge router must deploy traffic classification, conditioning, packets buffering, buffer management and queues servicing as the synoptic of Fig. 2.3 depicts.

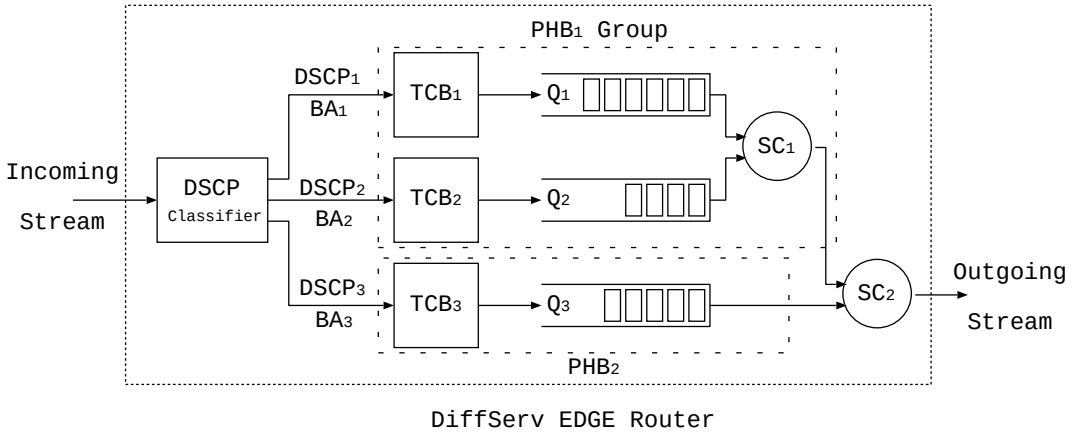


Figure 2.3: DiffServ Edge Router Synopsis

Nodes within the DS domain are called either *DS Core* (or *DS Interior* router), as they are connected to other interior or boundary nodes within the same DS domain. Their task consists basically on selecting the appropriate forwarding behavior for packets based on their DS codepoint, mapping that value to one of the supported PHBs; additionally, core routers may be able to perform limited traffic conditioning functions, such as DS codepoint re-marking. A scheme of the functional blocks deployed in a core router could thus be derived as a simplification of the edge router scheme of Fig. 2.3.

2.2.1.3 DiffServ Ingress and Egress Router

DS Edge routers act both as ingress and egress for different directions of traffic: traffic enters a DS domain at a *DS Ingress* node and leaves it at a *DS Egress*.

The ingress edge is responsible for ensuring that the traffic entering the DS domain conforms to TCA between it and the upstream domain; an egress boundary may either perform traffic conditioning functions or act as an interior node depending on the TCA established between the peering domains.

2.2.2 DiffServ Region

A *DS Region* is a set of one or more contiguous DS domains, capable of supporting differentiated services along paths which span these domains. Fig. 2.4 schemes three basic DS regions, each of which is uniquely constituted by only two peering domains: the upstream DS domain A and one within the tree B, C, D shown downstream domains. DS regions can then be extended to include more than two domains: in this case, the considerations further developed in this sections will hold for every peering domains pair.

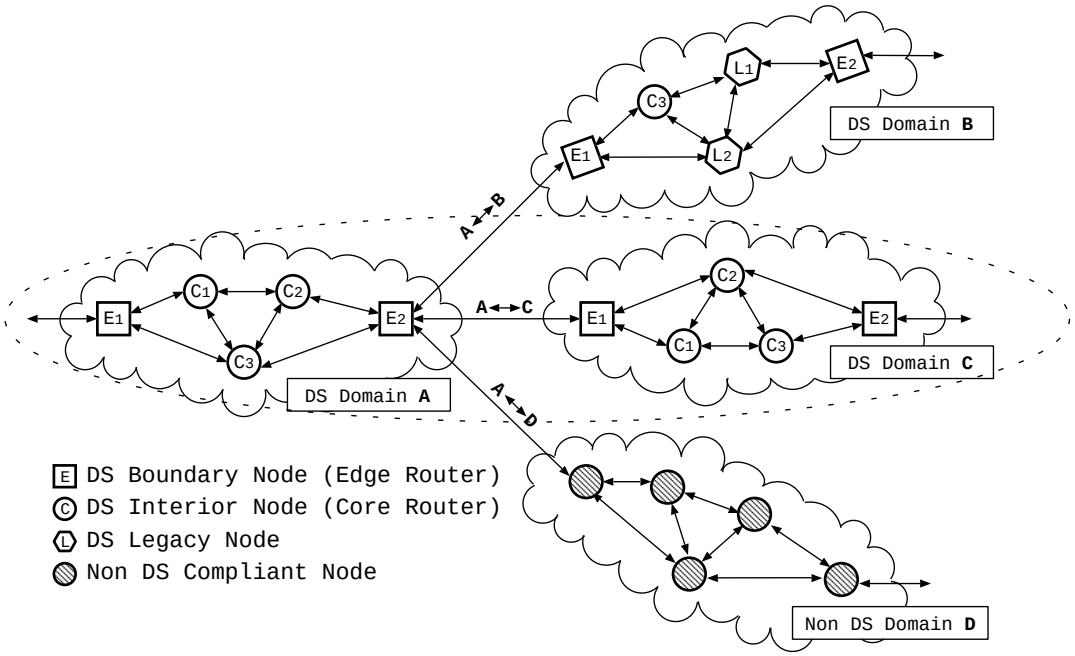


Figure 2.4: DiffServ Region Synopsis

The DS region evidenced in the figure is constituted by A and C DS domains, and fully deploys services differentiation. The peering domains within $A \leftrightarrow C$ region, whether internally supporting different PHB groups, mapping and service provisioning policy, need to determine a transit traffic conditioning agreement via an SLA definition; the edge devices of both the domains have to perform DSCP remarking as well as traffic conditioning, in the need to ensure a statistical assurance level. Whether both the domains decide to adopt a common service provisioning policy and to support a common set of PHB groups and codepoint mappings, then traffic conditioning is no longer needed between those DS domains: an edge router connecting these domains will act as an interior nodes for both traffic directions.

The $A \leftrightarrow B$ region contains legacy nodes: this entails constraints on the per-hop behaviors supported, that is, the region should restrict itself to use only the Class Selector Codepoints (or otherwise packets should be re-marked to a DSCP of that set at the A egress); this may result in an acceptable alternative whether the particular precedence implementation in the legacy nodes provides forwarding behaviors compatible with the services offered. Even though a DS legacy node of B domain is substituted with a strictly non DS compliant one, this may not result in service degradation on high-speed lightly loaded links; however, the lack of PHB forwarding in a node will surely compromise at least some DiffServ capability in more realistic network situations.

In the $A \leftrightarrow D$ region, the non DS capable D domain is assumed not to deploy traffic conditioning functions on its boundary; this is a critical situation, as the flows traversing this region will have a compromised assurance of consistent services delivering. These domains may negotiate an agreement, which might be monitored by traffic sampling, governing egress A traffic re-marking rules; moreover, traffic flowing from the domain D should be conditioned at the A domain ingress. If there is no knowledge of the traffic management capabilities of the downstream domain and no agreement is in place, then the A domain may choose to re-mark packets DSCP to zero: the DiffServ traffic will thus be uniformly treated with BE service.

2.3 Per-Hop Behaviors

A *Per-Hop Behavior (PHB)* is defined in [RFC2475] as "a description of the externally observable forwarding behavior of a DS node applied to a particular DS behavior aggregate". The PHB is the means by which a node allocates resources to different *Behavior Aggregates (BA)*, and it is on top of this basic hop-by-hop resource allocation mechanism that useful differentiated services may be constructed.

Packets transiting in a node receive a different forwarding treatment based on their DSCP: according to Sec. 2.1, there are only 64 possible DSCP values, while there is no such limit on the number of PHBs, whose selection is implemented in routers by means of packets *Classifier*. PHBs are implemented in nodes by means of two classes of routing algorithms, *Scheduling* and *Queue Management*, closely related but addressing rather different performances issues: scheduling algorithms determine which packet to send next and are primarily used to manage the allocation of bandwidth among flows, while the latter algorithm's class manages the length of packet queues by dropping packets when necessary or appropriate.

PHBs are specified in terms of behavior characteristics relevant to service provisioning policies, and not in terms of specific implementation mechanisms. Such behavior may be either expressed in terms of *resource priority* (e.g., buffer, bandwidth) relative to other PHBs or relative observable *traffic characteristics* (e.g., delay, loss; if the router offers several logical queues on the interface, delay and drop preferences are orthogonal: this means that high delay priority PHB can have either low or high drop preference).

These PHBs may grouped –a single PHB defined in isolation is a special case of a PHB group– when consistent and the relationship between PHBs within that group may be expressed in terms of absolute or relative priority, even though this is not required. Moreover, it is likely that more than one PHB group may be simultaneously implemented on a node and utilized within a domain; these groups will usually share a common constraint applying to each PHB within the group, such as a packet scheduling or buffer management policy: in general, a wide variety of mechanisms may be suitable for PHB implementation.

2.3.1 Traffic Classification

The packet classification policy identifies the specific differentiated service (including the Best Effort service) that each subset of traffic will receive within the DS domain, mapping then these subsets to one or more behavior aggregates by DS codepoint re-marking.

Packet classifiers select packets in a traffic stream based on the content of some portion of the packet header. Two type of classifiers are defined: the *BA (Behavior Aggregate) Classifier*, which classifies packets based on the DS codepoint only, and the *MF (Multi-Field) Classifier*, which effectuates packets selection according to a combination of several IP header fields besides the DSCP (chosen among source address, destination address, protocol ID, source port and destination port numbers, and other information such as incoming interface).

Classifiers, whose configuration needs to be done according to the TCA, are thus used to steer packets matching some specified rule to a specific traffic conditioner element (see Sec. 2.4) for further processing, as shown earlier in Fig. 2.3. An important remark about MF classifier is that, in the event of upstream packet fragmentation, they may incorrectly classify packet subsequent to the first whether they examine the contents of transport-layer header fields. A partial solution to this problem would need to maintain fragmentation state informations; however, it should be mentioned that this approach suffers of two majors lack in the case upstream fragment re-ordering or divergent routing paths.

2.3.2 Packet Scheduling

In general, router architectures may be classified into two main categories based on whether packets are buffered at the *inputs* or the *outputs* of the switch. In an input buffered switch, the scheduling algorithm has to make decisions on two levels: scheduling transmissions among the input ports

transmitting to a common output port and scheduling a packet from the chosen input port. In an output buffered switch, the scheduler only has to pick a packet from the output buffer: this section focuses mainly on the output scheduling, since the majority of the current routers are based on this paradigm.

A simplified model of the output buffer of a DiffServ capable router is presented in Fig. 2.5. Each output has a number of logical queues, and the router maps (classifies) each incoming packet into one of these queues depending on the each packet's DSCP.

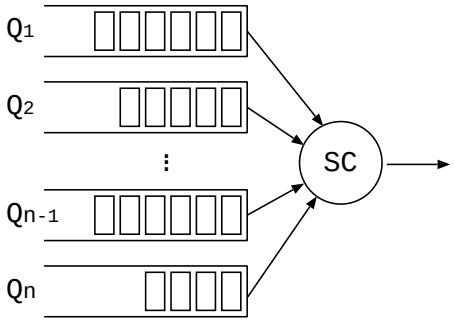


Figure 2.5: Packet Scheduler

The queues are then served according to a particular scheduling algorithm: as some of the queues get better service than others, packets belonging to a BA mapped to a high-priority queue usually experience less delay than the other packets. The main difference compared to IntServ and ATM, is that there is no separate queue for each connection, but only a certain number of them: this means that the connections sharing the same PHB value are not isolated from each other, but can affect each other's performance.

In general, schedulers can be characterized as *work-conserving*, if they never idle when a packet is queued in the buffer, or *non-work-conserving*, if, e.g., they postpone the transmission of a packet expecting a higher-priority packet to arrive soon, even though they are currently idle. Work-conserving schedulers have lower average delays and are by far more popular in commercial implementations.

Another classification of schedulers is based on their internal structure: they are either *sorted-priority* or *frame-based*. In a sorted-priority scheduler there is a global variable associated with the output, usually called "virtual time", updated each time a packet arrives in –or gets serviced by– the scheduler; packets are associated with a timestamp, measured by the virtual time, and are sent out based on their timestamps, which thus determine the order of transmission. In a frame-based scheduler, time is split into frames of fixed or variable size: each logical queue gets serviced for the duration of one frame at a time, or less, if the queue has not enough packets. The frames are allocated to the queues usually in a round-robin fashion. Frame-based schedulers are simple to implement, but their worst-case delay and jitter is greater than in sorted-priority schedulers.

2.3.2.1 Strict Priority

Strict Priority queueing is a simple scheduling algorithm. The queues are arranged in strict priority order, and a particular queue gets service only if there are no packets in the higher priority queues. Priority queueing can guarantee small delay for the highest class, but the other classes face a possible starvation, whether the higher classes use all the available bandwidth.

2.3.2.2 Weighted Fair Queuing and Virtual Clock

Representative examples of work-conserving priority-based schedulers are Weighted Fair Queuing (WFQ) and Virtual Clock (VC). If the weights (the rates) in WFQ (in VC) corresponding to the

individual queues are equal, the algorithms divide the capacity of the output link emulating a time-division multiplexer (TDM): that is, they send the packets out almost in the same order that TDM would. If the weights are not equal, the queues share the capacity accordingly; if any of the queues has not enough packets to send out, the other queues share its portion proportionally to their weights.

2.3.2.3 Weighted Round Robin

Weighted Round Robin (WRR) is a work-conserving frame-based scheduler; WRR serves each queue in a Round Robin fashion, and for each turn, a number of bits corresponding to the queue's weight is extracted from that queue: thus the link capacity is divided according to the weights, as in WFQ. In the worst-case situation, a packet arrives to a queue just after the queues turn: the maximum queuing delay will be thus the sum of the weights of all other queues, if they have also enough packets. In that sense, WRR is not as ideal as WFQ, but it can be easily implemented via one unique software loop: this may become a deciding factor, if the link speeds increase faster than the pure processing power does.

The DiffServ routers used in all the network simulations scenarios presented in this study use a simple Round Robin scheduling.

2.3.3 Active Queue Management

The Internet architecture is based on a connectionless end-to-end packet service using the IP protocol, which brings advantages of robustness and flexibility at the cost of difficulty to provide good service under heavy load: the "Internet Meltdown" phenomenon, firstly observed during the early growth of the middle 80s, made necessary to develop some form of responsiveness to the congestion signals (e.g. packet drops).

The TCP congestion avoidance mechanism [RFC2581] although necessary, is nevertheless insufficient: this basic limit on the effectiveness of the control accomplished by network edges necessitates the application of some form of control, i.e., Queue Management, also in the routers. This section outlines the traditional *Drop Tail* Queue Management technique and its draw-backs, before describing the widely deployed *Random Early Detection (RED)* Active Queue Management (AQM) mechanism and its extensions.

2.3.3.1 Drop Tail

Drop Tail algorithm is simple: it *drop* the packet that arrived most recently (thus the one on the *tail* of the queue) when the queue is full (i.e. when a number of packet corresponding to the maximum queue's length have already been buffered); subsequent incoming packets will be evidently discarded until the queue length decreases, thus when a buffered packet have been transmitted. This method has two major disadvantages, as [RFC2309] reports:

- *Lock-Out*: in some situations, as a result of global synchronization, a single connection or a few flows can monopolize buffer space, preventing other connection from getting their packets to be queued
- *Full-Queues*: Drop Tail discipline allows queues to maintain a full (or almost full) status for long time periods: this is consequence of the fact that congestion signaling (via packet drop) is effectuated only when the maximum queue length is reached. An important goal is to reduce the steady-state queue size: queue limits should reflect the size of bursts that have to be absorbed, and not the steady state queues we want maintained in the network

The lock out syndrome has two easy solutions, *Random Drop On Full* and *Drop Front On Full*, whose behavior can be inferred directly from their names; however, none of these approaches solves the full queues problem.

2.3.3.2 Random Early Detection

The solution to the full queues problem is known in general for flows that are reactive to congestion notification; this can be done by dropping a packet (or by marking a packet by setting its congestion bit) *before* the queue becomes full: such a pro-active approach is the base of AQM. The advantages brought to responsive flows by adoptation of such a mechanism are basicly to reduce the number of packets dropped in the router (due to a greater bursts absorbing capability as a result of keeping the average queue size small), to reduce the delay seen by the flows and finally to prevent full queues problem by ensuring that there will almost always be a buffer available for an incoming packet.

The RED Gateway proposed in [RED] consists of two separate algorithms: one for computing the average queue size, determining the degree of burstiness that will be allowed in the gateway queue, and one for calculating the packet-dropping probability, reflecting the current level of congestion. The average queue size estimation is the means by which RED avoid over-reactions to bursts, and instead rects to longer-term trends; each time a packet is received, it can be evaluated either on a *Time Sliding Window* base (where the window length constant determine the worth of the past history, thus the weight of the past against the present) as defined in [EXPALL], or by an *Exponentially Weighted* moving average approach coupled with a low-pass filter as implemented in [RED].

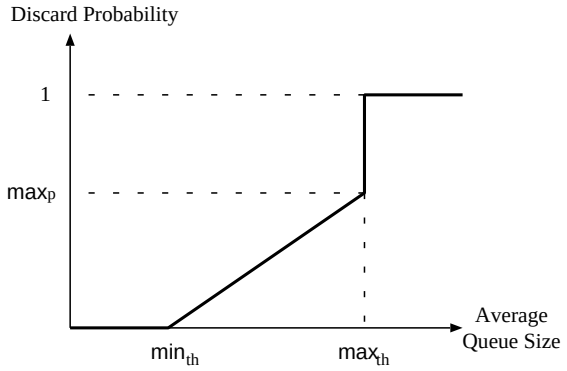


Figure 2.6: RED Linear Increasing Probability Function

The average queue size avg is then compared with two thresholds (min_{th}, max_{th}) to determine whether to drop packets: if the average queue is less that the minimum threshold, no drop action will occur; if the average is between the minimum and the maximum thresholds, an *early drop* test will be performed as later described, otherwise the packet will incur in a forced or *late drop* action, explicit indication of persistent congestion. It should be noted for the latter case that, being the average weighted, possibly no forced drop will take place even when the instantaneous queue is quite large.

The early drop action depends on several factors; an initial drop probability P_b is calculated as a linear increasing function depicted in Fig. 2.6, where the x axis depicts the measured buffer occupancy (which can be expressed in packets, or in bytes, or as a rate) and the y axis depicts the discard probability: as the average varies from min_{th} to max_{th}, P_b varies linearly from 0 to max_p, thus $P_b = max_p \cdot (avg - min_{th})/(max_{th} - minth)$

The final drop probability P_a is a function of the initially calculated P_b and of the count of the number of packets enqueued since last packet drop, as $P_a = P_b/(1 - count \cdot P_b)$: this is done with the intent to ensure that the gateway does not wait too long before dropping a packet.

As reported earlier, RED gateways are able to measure the queue size either in packets or in bytes: the latter option means, for example, that a large FTP packet would be more likely to me

dropped than a small TELNET one; to enhance this ability, the initial probability P_b must be scaled by a factor computed as packet size over maximum packet size ratio.

2.3.3.3 Multi-Level Random Early Detection

MRED is a generic term used to describe any scheme where each drop probability, for packets with different drop probabilities –which are assigned different colors– need to be calculated independently [MRED]. This is achieved by maintaining multiple sets of RED thresholds, one for each drop precedence, and the average queue used in the drop decision can be calculated using a number of different schemes.

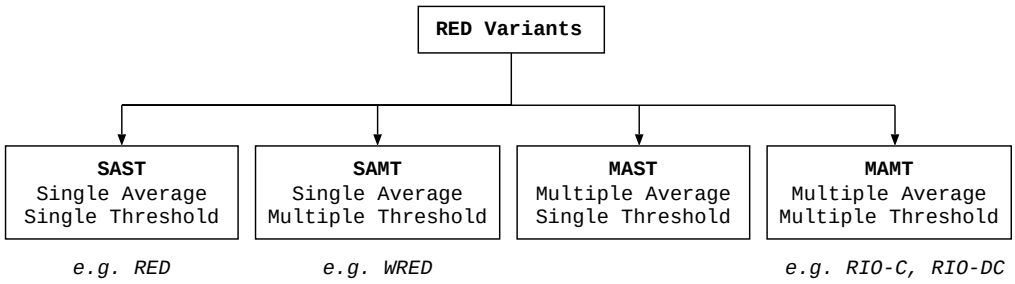


Figure 2.7: Different RED Based Classes

A classification of the possible MRED variants, based on different average calculation assumptions and thresholds set number, is schemed in Fig. 2.7. When multiple thresholds sets are maintained for different drop precedences, thus for different packet colors, another classification applies. Indicating with $\mathrm{I}_{col} \equiv (\min_{th}, \max_{th})$, where $col \in$ (Green,Yellow,Red), whether $\mathrm{I}_{Green} \equiv \mathrm{I}_{Yellow} \equiv \mathrm{I}_{Red}$ the MRED is said to use an *Overlapped* set of parameters; if $\forall i, j \in$ (Green,Yellow,Red) $\mathrm{I}_i \cap \mathrm{I}_j = \emptyset$ then the parameter set is said to be *Staggered*, and *Partially Overlapped* otherwise; finally, the middle case of the staggered and partially overlapped sets, where thus the min_{th} of a color equals the max_{th} of the immediately lower drop precedence color, is said to be *Boundary Staggered*. The overlapped and boudary staggered cases are depicted in Fig. 2.8.

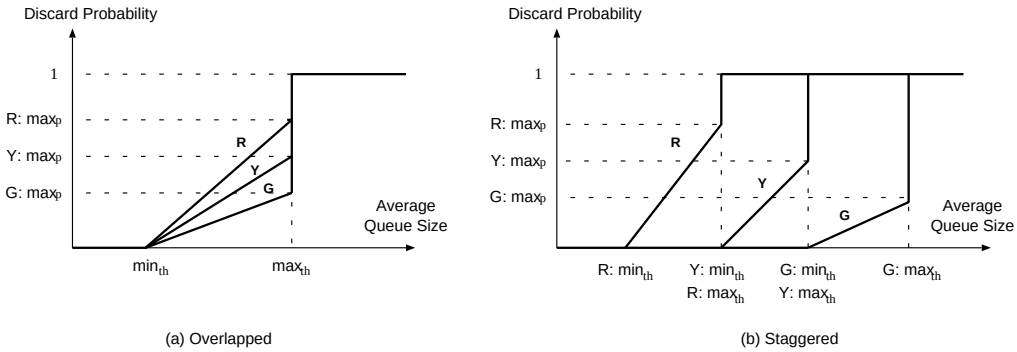


Figure 2.8: MRED Parameter Setting

Weighted RED is a significant example of SAMT RED variant: for any arrival or departure of green, yellow or red packets, WRED calculates *a single average queue* that includes arriving packets of all colors.

A widely deployed MAMT modification of RED is RED with In/Out (RIO), which uses the same mechanism as in RED but is configured with two sets of parameters: one for In marked

packets, that is packet of a stream compliant to specified policy, and one for Out marked ones. RIO average queue can be calculated either in a *Coupled* (RIO-C) or *De-Coupled* (RIO-DC) fashion: the latter calculates the average queues size of In and Out packets independently, while the former compute the In queue average using the number of In packets queued and the Out queue average using the total number of In and Out packets queued. In the RIO scheme, In and Out packets can be mapped respectively to DP0 and DP1 drop precedences; RIO algorithm can be nevertheless easily extended to support three packet colors (GRIO). In GRIO-C, the average queue length of a color can be expressed by adding its average to the average queues of colors corresponding to lower drop precedences: this entails that drop probability for packets with higher drop precedence is dependent on the buffer occupancy of packets having lower drop precedence. In GRIO-DC, the average queue length of a color is calculated using only the number of packets of that color in the queue: this appears to be the most suitable RED variant whether the operator wishes to assign weights to different colored packet within the same queue. Based on empirical study, it has been observed in [MRED] that

- for ON-OFF traffic, RIO is better than WRED in protecting packets marked for treatment with lower drop precedence
- for short-lived flows, RIO achieves higher transactional rates than WRED
- for bulk transfer, RIO and WRED achieve comparable long-term throughput

The DiffServ routers used in all the network simulations scenarios presented here use either GRIO-C or RIO-C active queue management techniques.

2.3.4 Per-Domain Behaviors

A PHB, as defined earlier, is the forwarding treatment experienced by packets having the same DSCP, aggregated in a BA, *as they cross a DS node*. The *Per-Domain Behavior (PDB)* [RFC3140] extends this definition to describe the behavior experienced by a particular set of packets *as they cross a DS domain*; this set of packets, whose DSCPs map to the same PHB within a DS domain, will be further identified as *Traffic Aggregate (TA)*, evident generalization of the behavior aggregate concept.

Each PDB has measurable and quantifiable attributes that can be used to describe what happens to its TA; these obviously depend on traffic classification, conditioning and forwarding treatment experienced by packets inside the domain, but can also depend on the entering traffic loads and, more important, on the domain topology.

A PDB is, from a top-level point of view, where the forwarding path and the control plane interact; there is anyway a distinction between the definition of a PDB and a service specified in SLA: the former is a technical building block whose element's configuration might be taken from a customer-visible SLS; rather, the measurable attributes of a PDB are expected to be among the parameters cited in that SLS: the PDB used by an ISP is thus not expected to be visible to customers any more than the specific PHBs employed would be.

The usefulness of PDB definition lies in the deployment of edge-to-edge, intra-domain QoS and further enables the composition of end-to-end, cross-domain services by building on the PDB characteristics without regard to the particular PHBs used. To date, is not far clear whether DiffServ scaling properties only result if the PHB definition gives rise to a particular type of invariance under aggregation. Specifically, different streams within a TA merge and split as they traverse the domain: if the properties of a PDB hold regardless of temporal characteristics changes, then that PDB scales; PDBs that are invariant to –or that show simple relationships with– network size will be reasonably needed for building scalable end-to-end QoS.

2.4 Traffic Conditioning

Traffic conditioning is a means to enforce the assurance that traffic entering the DS domain conforms to the rules specified in the TCA derived from an intra-domain SLA. Traffic conditioning is usually done at the boundary nodes and it consists, basically, of the following primitives:

- metering
- marking
- policing
- shaping

A traffic stream is selected by a classifier (see Sec. 2.3.1), which steers the packets to a logical instance of a traffic conditioner: a meter then is used (where appropriate) to measure the traffic stream against a traffic profile. The state of the meter with respect to a particular packet –whether thus it is *in* or *out* of profile– may be used to affect a marking, dropping, or shaping action. Fig. 2.9 shows a classic block diagram of a classifier and traffic conditioner.

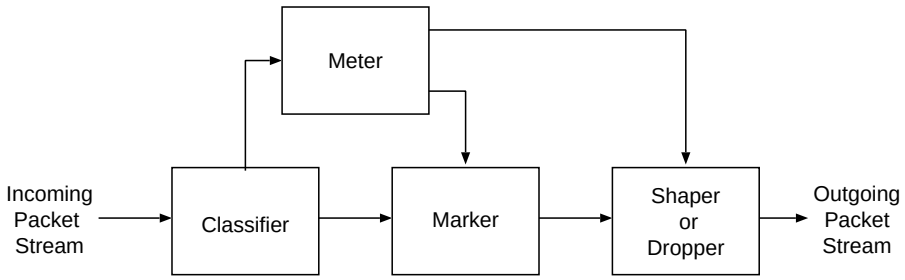


Figure 2.9: Traffic Conditioner Logical Block Synoptic

It should be pointed out that a traffic conditioner may not necessarily contain all four element (i.e. in the case where no traffic profile is in effect, packets may only pass through a classifier and a marker) and that their functional relationship –thus their links in the logical block synoptic– may differ from those depicted depending on the peculiar traffic conditioner's implementation: a different example is given in Sec. 2.4.4.3.

2.4.1 Marking

Packet markers set the DS field of a packet to a particular codepoint, thus determining the marked packet's treatment in the core routers by adding it to a particular DS behavior aggregate; when the marker changes the codepoint in a packet it is said to have "re-marked" that packet. Marking can be performed by either the application or the operating system or finally an edge router.

2.4.2 Policing

Droppers discard some or all of the packets in a traffic stream in order to bring the stream into compliance with a traffic profile. This process, who doesn't cause the stream to alter in time as shaping does, is know as "policing" the stream. A simple policer is implemented using the *token bucket* algorithm, which characterizes the packet stream with two parameters: average rate (token rate) and burst size (bucket depth). Note that a dropper can be implemented as a special case of a shaper by setting the shaper buffer size to zero (or a few) packets.

2.4.3 Shaping

Shaper's task is to bring the traffic stream into compliance with a traffic profile, determined by some traffic properties such as bit rate and burst size. Shaping is often based on the *leaky bucket* algorithm, which has the result of delaying non-conforming packets. A shaper usually has a finite-size buffer, and packets may be discarded if there is not sufficient buffer space to hold the delayed packets, thus being used as policing element.

2.4.4 Traffic Conditioner Examples

To date, the research in the DiffServ area has produced a relevant number of different traffic conditioners. This section firstly introduce two of them, the *Single Rate Three Color Marker* and its companion *Two Rates Three Color Marker*, both token bucket based, developed by Heinan and Guerin in [RFC2697] and [RFC2698] respectively; the purpose of these marking schemes is to provide a low drop probability to a minimum part of the traffic whereas the excess will have a larger drop probability.

Since simulations shown that such markers performances with TCP traffic was not always satisfactory, several researchers outlined that problem solution lies in either increasing the burst size, or shaping the traffic such that a part of the burstiness is removed. The former solution would implies, to efficiently support bursty traffic, additional resources (such as buffer space), while the major disadvantage of the latter is that the traffic would experiment additional delay in the shaper's buffer. Examples of the latter mechanism, that will be briefly described, are represented by the *Rate Adaptive Shaper* and the corresponding *Green Rate Adaptive Shaper*, defined by Bonaventure and De Cnodder in [RFC2963].

Between other interesting approaches, a mention must be made about Feroz, Kalyanaraman and Rao and the *TCP-Friendly Marker*, proposed in [FMARK], intended to enhance performances –in assured service context– of legacy TCP applications, affected by bursty loss packet behavior. Andrikopoulos, Wood and Pavlou developed in [FTCON] a *Fair Traffic Contitioner* intended to provide fair bandwidth distribution among responsive and unresponsive flows sharing the same AF class and originating from the same customer network. The *Time Sliding Window Three Color Marker*, proposed by Fang, Seddigh and Nandy in [RFC2859], will be deeper analyzed in Ch. 3.

2.4.4.1 Single Rate Three Color Marker

The Single Rate Three Color Marker (srTCM), defined in [RFC2697], meters an IP packet stream and marks its packets either green, yellow, or red. Marking is based on a Committed Burst Size (CBS) and an Excess Burst Size (EBS), associated to the Committed Information Rate (CIR). The Meter meters each packet and, as Fig. 2.10 depicts, passes the packet and the metering result to the Marker which takes the tagging decisions as follows: a packet is marked green if it doesn't exceed the CBS, yellow if it does exceed the CBS but not the EBS, and red otherwise.

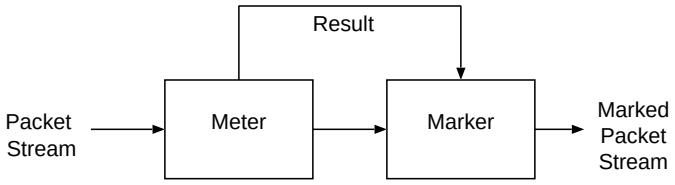


Figure 2.10: Single Rate Three Color Marker Synoptic

The Meter can operate in two modes: in the *Color-Blind mode*, the Meter assumes that the packet stream is uncolored, while in the *Color-Aware* mode the Meter assumes that some preceding entity has pre-colored the incoming packet stream so that each packet is either green, yellow, or

red. The behavior of the Meter is thus specified in terms of its mode and two token buckets, C and E, which both share the common rate CIR; the maximum size of the token bucket C is CBS and the maximum size of the token bucket E is EBS, where both the committed and the excess burst size are measured in bytes and must be configured so that at least one of them is larger than 0. The token buckets C and E are initially full, thus the token count $\mathrm{Tc}(0) = \mathrm{CBS}$ and the token count $\mathrm{Te}(0) = \mathrm{EBS}$. Thereafter, the token counts Tc and Te are updated CIR times per second as follows:

```
if (Tc< CBS)
        Tc ← Tc +1
elseif (Te < EBS)
        Te ← Te +1
endif
```

When a packet of size B bytes arrives at time t, the tagging decisions are taken differently depending whether the marker is color aware or not:

```
if (Color-Blind)
        if (Tc(t) − B ≥ 0)
                PacketColor ← Green
                Tc ← max(0, Tc − B)
        elseif (Te(t) − B ≥ 0)
                PacketColor ← Yellow
                Te ← max(0, Te − B)
        else
                PacketColor ← Red
        endif
elseif (Color-Aware)
        if (PacketColor = Green) and (Tc(t) − B ≥ 0)
                PacketColor ← Green
                Tc ← max(0, Tc − B)
        elseif (PacketColor ∈ {Green,Yellow}) and (Te(t) − B ≥ 0)
                PacketColor ← Yellow
                Te ← max(0, Te − B)
        else
                PacketColor ← Red
        endif
endif
```

This policy have been choosen in order guarantee a deterministic behavior, in the sense that tokens of a given color are always spent on packets of that color; according to the above rules, marking of a packet with a given color requires that there be enough tokens of that color to accomodate packets.

2.4.4.2 Two Rate Three Color Marker

A natural extention to the srTCM is represented by the Two Rate Three Color Marker (trTCM), defined in [RFC2698], which make use of four traffic parameters: a Peak Information Rate (PIR) and its associated Peak Burst Size (PBS) and a Committed Information Rate (CIR) and its associated Committed Burst Size (CBS); like the srTCM, the trTCM can work in one of the two color blind or color aware modes.

The Meter and Marker logical function blocs are depicted in Fig. 2.10; the marking decisions are taken as follows: a packet is marked red if it exceeds the PIR, otherwise it is marked either

yellow or green depending on whether it exceeds or doesn't exceed the CIR; both the PIR and the CIR are measured in bytes of IP packets per second, and the former must be equal to or greater than the latter.

The behavior of the Meter is specified in terms of its mode and two token buckets, P and C, with rates PIR and CIR respectively. The maximum size of the token bucket P is PBS and the maximum size of the token bucket C is CBS, where both the PBS and the CBS and are measured in bytes and must be configured to be greater than 0. The token buckets P and C are initially full, thus the token count $\mathrm{Tp}(0) = \mathrm{PBS}$ and the token count $\mathrm{Tc}(0) = \mathrm{CBS}$. Thereafter, the token count Tp is incremented by one PIR times per second up to PBS and the token count Tc is incremented by one CIR times per second up to CBS. When, at time t, a packet of size B bytes arrives, the following tagging algorithms happens:

```
if (Color-Blind)
        if (Tp(t) − B < 0)
                PacketColor ← Red
        elseif (Tc(t) − B < 0)
                PacketColor ← Yellow
                Tp ← Tp − B
        else
                PacketColor ← Green
                Tc ← Tc − B
                Tp ← Tp − B
        endif
elseif(Color-Aware)
        if (PacketColor = Red) or (Tp(t) − B < 0)
                PacketColor ← Red
        elseif (PacketColor = Yellow) or (Tc(t) − rmB < 0)
                PacketColor ← Yellow
                Tp ← Tp − B
        else
                PacketColor ← Green
                Tc ← Tc − B
                Tp ← Tp − B
        endif
endif
```

Both the trTCM and the srTCM can be used to mark a packet stream in a service where different, decreasing levels of assurances are given to packets by coloring them green, yellow, or red. For example, a service may discard all red packets, because they exceeded both the committed and excess burst sizes, forward yellow packets as best effort, and forward green packets with a low drop probability. Moreover, due to its configuration parameters the srTCM is useful, for example, for ingress policing of a service where only the length, not the peak rate, of the burst determines service eligibility; conversely, the trTCM may be useful for services where a peak rate needs to be enforced separately from a committed rate.

2.4.4.3 Rate Adaptive Shaper

The Rate Adaptive Shaper functional (RAS) blocks synoptic depicted in Fig. 2.11 differs from the one presented in Fig. 2.9 in that the shaper is placed *before* the meter. The main objective of the shaper is to produce at its output a traffic that is less bursty than the input traffic, but avoiding to discard packets in contrast with classical token bucket based shapers. By reducing the burstiness of the traffic, the adaptive shapers increase the percentage of green marked packets and thus the overall goodput performances.

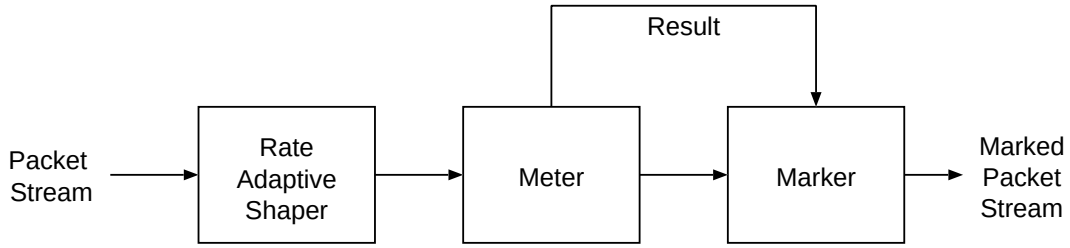


Figure 2.11: Rate Adaptive Shaper Synoptic

The shaper itself consists of a tail-drop FIFO queue which is emptied at a variable rate, function of both the average rate of the incoming traffic and the occupancy of the FIFO queue: when the latter increases, the shaping rate will increase in order to prevent loss and too large delays through the shaper. Two types of rate adaptive shapers are here presented: the single rate rate adaptive shaper (srRAS) will typically be used upstream of a srTCM (see Sec. 2.4.4.1) while the two rates rate adaptive shaper (trRAS) will usually be used upstream of a trTCM (see Sec. 2.4.4.2).

The srRAS is configured by specifying two couples of parameters: the Committed Information Rate (CIR) and the corresponding threshold CIR_{th}, the Maximum Information Rate (MIR) and the corresponding threshold MIR_{th}. To achieve good performances, the CIR is usually set to the same value as the CIR of the downstream srTCM, the MIR is typically set to the shaper output line rate (or with respect to $\text{CIR} \leq \text{MIR} \leq$ line rate), and the thresholds, choosen accordingly to srTCM's CBS and PBS values, must satisfy $\text{CIR}_{th} \leq \text{MIR}_{th} \leq$ buffer size.

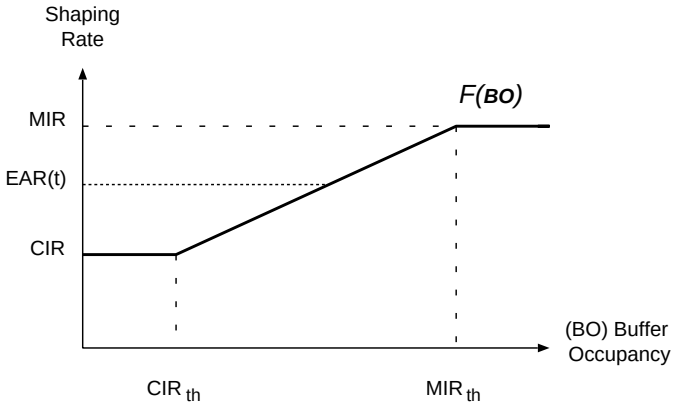


Figure 2.12: Shaping Rate Computation for srRAS

The shaper's Output Rate (OR) is evaluated as the maximum of two factors: the long term Estimated Average Rate (EAR) of the incoming traffic, which can be computed by several means (see Sec. 2.3.3.2), and a function F, which computation is illustrated in Fig. 2.12, of the instantaneous shaper's Buffer Occupancy (BO):

$$\text{OR}(t, \text{BO}) = max(\text{EAR}(t), \text{F}(\text{BO}))$$

The trRAS is configured by specifying three couples of parameters: the Committed Information Rate (CIR) and the corresponding threshold CIR_{th}, the Peak Information Rate (CIR) and the corresponding threshold PIR_{th}, the Maximum Information Rate (MIR) and the corresponding threshold MIR_{th}.

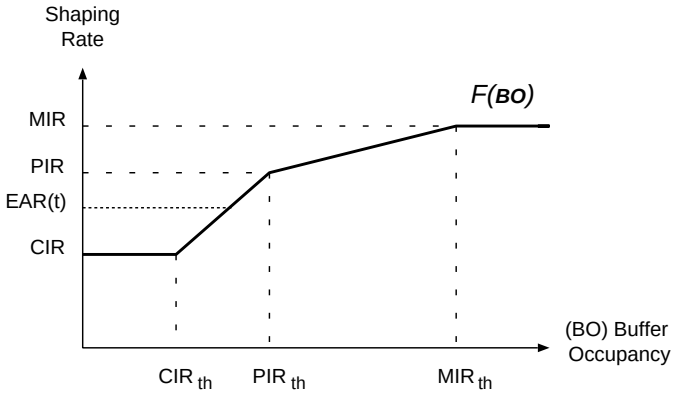


Figure 2.13: Shaping Rate Computation for trRAS

The trRAS behavior is determined by the following equation system, where F(BO) indicated the piecewise linear function depicted in Fig. 2.13:

$$\begin{cases} \text{CIR} \leq \text{PIR} \leq \text{MIR} \leq \text{line rate} \\ \text{CIR}_{th} \leq \text{PIR}_{th} \leq \text{MIR}_{th} \leq \text{buffer size} \\ \text{OR}(t, \text{BO}) = max(\text{EAR}(t), \text{F(BO)}) \end{cases}$$

2.4.4.4 Green Rate Adaptive Shaper

The rate adaptive shapers described in the previous sections are not aware of the status of the meter: a packet may be unnecessarily delayed by such a RAS even if there are sufficient tokens available to color the packet green. This entails that TCP takes more time to increase its congestion window, lowering traffic performances. The Green Rate Adaptive Shaper (G-RAS), whose scheme is shown in Fig. 2.14, solves this problem by coupling the shaper with the meter.

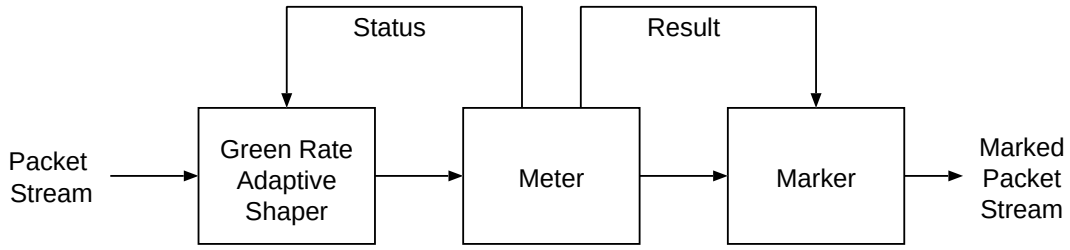


Figure 2.14: Green Rate Adaptive Shaper Synoptic

The two rate adaptive shapers described in section Sec. 2.4.4.3 calculate the time schedule t_1 at which the packet at the head of the shaper's queue is to be released by means of their output shaping rate. By coupling the shaper with the meter, the G-RAS is aware of the earliest time instant t_2 at which this packet would be colored as green by the downstream srTCM: if this time instant is earlier than the release time computed from the shaping rate, then the G-RAS is able to release the packet earlier than the RAS would. When a packet arrives at the head of the queue of the shaper, it will leave this queue not sooner than $min(t_1, t_2)$.

The G-srRAS, whose configuration is identical to those of the srRAS, determines –via its shaping rate– the release time schedule t_1; supposing that a packet of size B bytes is at the head

of the shaper, the second time schedule t_2 would be set to infinity if B > CBS, or otherwise to:

$$t_2 = max\left(t, t + \frac{\mathrm{B} - \mathrm{Tc}(t)}{\mathrm{CIR}}\right)$$

Analogously, for the G-trRAS the earliest time instant t_2 at which a new green token will be available at the downstream trTCM can be expressed as:

$$t_2 = \begin{cases} \infty & (\mathrm{B} > \mathrm{CBS}) \text{ or } (\mathrm{B} > \mathrm{PBS}) \\ max\left(t, t + \frac{\mathrm{B}-\mathrm{Tc}(t)}{\mathrm{CIR}}, t + \frac{\mathrm{B}-\mathrm{Tp}(t)}{\mathrm{PIR}}\right) & \text{otherwise} \end{cases}$$

2.5 A Two-Bit DiffServ Architecture

The two-bit Differentiated Services architecture was firstly described in an Internet draft in november of 1997, predating thus the formation of the IETF's DiffServ Working Group (WG): many of the ideas presented there were key to the work which led, in december of 1998, to [RFC2474][RFC2475]. That original document became a Request For Comment [RFC2638] only in july of 1999; furthermore, it has been submitted in its original form: the forwarding path described there is then "intended as a record of where we were at in late 1997 and not as as indication of future directions", while the resource allocation matter, intended to be a timely proposal, is still subject of on-going research.

This section firstly introduces a few other architectural proposals in order to allow mutual comparison; then, the original services proposals is briefly described –with the purpose to introduce the goals later achieved– in Sec. 2.5.2, before focusing on the Bandwidth Broker resource allocation mechanism.

2.5.1 DiffServ Models Proposals

Before the standardization of a two bit DiffServ architecture –and, most important, the definition of the DS field in the packet header– various proposals were made, each characterized by the peculiar structure of the TOS field used to achieve differential forwarding treatment. Two of these steps of the recent DiffServ history are Simple Differential Services [SIMDS] and Simple Integrated Media Access [SIMA]; it must be said that another interesting alternative model, not discussed here, is represented by the Proportional Differentiated Services architecture [PROPDS].

Simple Differential Services
In this approach, taking into account IP TOS and Precedence, Delay Indication, and Drop Preference, the TOS field is divided into two subfields: a two bit field is given to delay indication (used in mapping the packets to a specific router queue, serviced in a way such as average queuing delay should decrease as the value of delay indication increases) and a three bit field given to drop preference (with the implication that higher value indicates lower dropping probability).

Simple Integrated Media Access
Simple Integrated Media Access (SIMA) proposes a service model assigning three bits of the TOS to be used as a priority, and one real-time (RT) bit to indicate delay tolerance. A customer purchases a Nominal Bit Rate (NBR), and his traffic is measured and compared to NBR at network edges; the priority value is then set as a function of the ratio of measured rate and NBR: the higher the ratio is, the lower the priority marking is; RT bit, which may be set by the application, ensures a shorter averaging period in rate measurement if set. In the core network, the packet priority value is first compared to a threshold, which is a function of buffer occupancy: if the priority is lower

than the threshold the packet is discarded, otherwise RT bit is used to select between two queues. The real-time queue is always relatively short, and it has strict priority over the elastic queue, meaning that RT bit gives support for real-time applications.

2.5.2 Premium and Assured Services

The formal definition of the *Premium Service* and *Assured Service*, as well as the terminology used to describe this "radical departure from the Internet's traditional service, but [...] also a radical departure from traditional QoS architectures which rely on circuit-based models" [RFC2638], evolved so far since their ideation. Therefore, specific technical details will be reported in the description of the standardized Expedit Forwarding PHB (Sec. 2.6) on which top Premium service class may be build, and the Assured Forwarding PHB Group (Sec. 2.7), starting point for Assured service definitions.

The differentiated services model stroke its root from Clark's and Jacobson's talks to the IntServ WG at the Munich IETF Meeting in august 1997: in this context, each explained how to use one bit of the IP header to deliver a new kind of service to packets in the Internet. To allow each packet's service selection "we propose designating two bit-patterns from the IP header precedence field. We leave the explicit designation of these bit-patterns to the standards process thus we use the shorthand notation of denoting each pattern by a bit, one we will call the Premium or P-bit, the other we call the Assurance or A-bit." The idea of forwarding treatment selection based on DSCP mapping enhanced further the possibility to implement, with an unique underlying architecture, additional classes of service, whose proprieties cannot be directly inferred from DSCP as for the SIMA approach.

Premium Service implements a "guaranteed peak bandwidth service with negligible queueing delay that cannot starve best effort traffic and can be allocated in a fairly straightforward fashion"; on the other hand, this service may prove both too restrictive (in its hard limits) and overdesigned (no overallocation) for some applications.

Assured Service implements a "service that has the same delay characteristics as (undropped) best effort packets and the firmness of its guarantee depends on how well individual links are provisioned for bursts of Assured packets"; on the other hand, it permits traffic flows to use any additional available capacity and occasional drops for short congestive periods may be acceptable to many applications.

In a general sense, Premium service denotes packets that are enqueued at a *higher priority* than the ordinary Best Effort queue, while Assured service packets are *treated preferentially* with respect to the dropping probability within the Best Effort queue. Premium and Assured are therefore two very different kinds of services that could be both implemented with a set of very similar mechanism, despite of their rather different policy assumptions: the former creates a service with little jitter and queuing delay and no need for buffer management, while the latter does use active queue management to provide a "better effort" service.

These services are not meant to *replace* Best Effort, but primarily to meet an emerging demand for commercial QoS services that can *share* the network with BE traffic; furthermore, even though an ISP may implement both these two kind of services –as they are not incompatible in a network– this is not required: neither the Assured nor the Premium service are mandatory part of the Differentiated Services architecture.

2.5.3 Bandwidth Broker

A detailed description of the resource allocation and admission control path is very much outside the scope of this dissertation, since our study assumed only static SLA capable domains; however, due to the extreme importance of the argument, we provide an essential description.

A customer contract with a DiffServ network an SLA definition describing, via user traffic profile and parameters, the desired QoS level at which the provider promise to deliver user data; the TCA and SLS part of a SLA contains the technical details of the traffic conditioning, policy and resource provisioning for that particular service. Each administrative domain makes its own decisions on the strategies and protocol used for *internal* resource management to meet QoS: therefore, in order to provide inter-domain services, a first problem is to standardize a model for the SLA.

Moreover, inter-domain SLAs may be static with regular upgrades: in this case, the resources are statically allocated and boundary routers and leaf nodes must be configured with classification, policing and shaping rules regarding the characteristics of the negotiation. On the other hand, dynamic SLAs are essential in order to provide required QoS on demand, and therefore a sort of admission control to make sure that link resources are enough to achieve the agreements is needed. The admission control decision and the configuration of edge node's policy and classification must be co-ordinated through some sort of network manager: *Bandwidth Broker (BB)* is such a resource controller mechanism among the several possible functional structures.

The BB is an agent responsible for managing different levels of service for a local domain through coordination of resources allocation; its name derives from the fact that both AF and EF PHBs use bandwidth as the resource that is being requested and allocated. Every administrative domain will have one main BB and one or more backup BBs, in order not to compromise on demand QoS delivering in case the main BB is down.

The functions of the BB involve host, router and server structure and require inter-BB communications for the effectiveness of admission control decision and resource allocation tasks. BB manages the resources in the domain, according to the SLAs between the domains and the specific QoS required via a *Resource Allocation Request (RAR)*, a protocol message that can be generated either by individual hosts for their local BB or by a BB for a communication with a peering BB.

Many BB architectural projects propose different flexible framework for provisioning a DiffServ network, but none have been standardized yet. In general, there are some necessary components that satisfy the functional BB architecture levels: these are shown in Fig. 2.15.

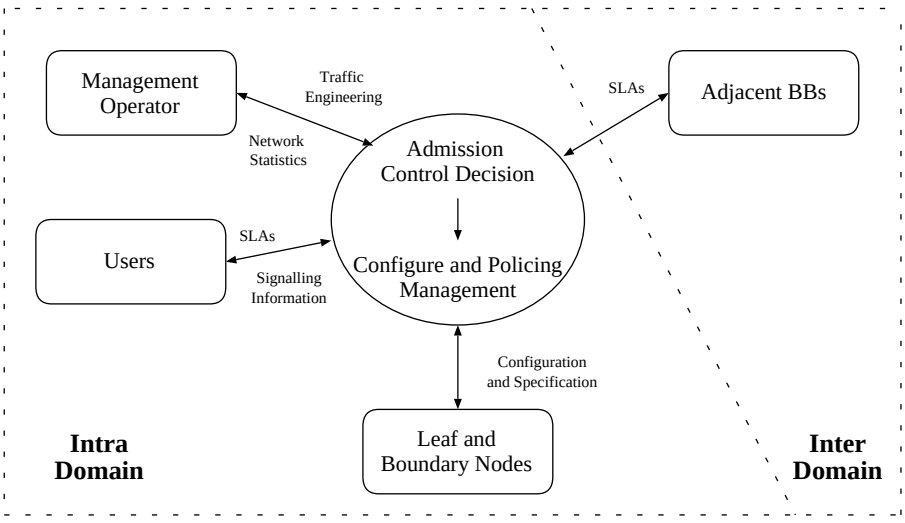


Figure 2.15: Basic Structure of BB Communications and Functions

The BB will therefore provide intra-domain communication interfaces toward the users, leaf and boundary nodes and eventually the management operator. The first interface is carrying the signalling information and the specified SLA of the user, whereas the management operator interface will handle the traffic and network statistics needed to take admission decision in the BB;

finally, the lead and boundary node interface carriers the configuration specifications to deliver the specified QoS: the control decision is taken after performing the policing functions. In addition, the BB will provide inter-domain interface toward the adjacent BBs, carrying the SLA between the peering domains.

2.6 Expedited Forwarding PHB

This section describes the *Expedited Forwarding (EF)* defined in [RFC2598], whose recommended Codepoint drawn from Pool 1 of the codepoint space is `101110`.

The EF PHB can be used to build a low loss, low latency, low jitter, assured bandwidth, end-to-end service through DS domains: such a service, which has also been introduced as Premium service in [RFC2638], appears to the endpoints like a point-to-point connection or a virtual leased line. Loss, latency and jitter phenomena are due to the queues traffic experiences while transiting the network: thus providing a VW service for some TA means ensuring that the aggregate sees almost no queues. This requirement is equivalent to bounding rates such that, at every transit node, the *maximum* arrival rate of the aggregate is less than its *minimum* departure rate, since queues arise in nodes whether the traffic arrival rate exceeds the departure one.

Therefore, EF PHB is defined as a forwarding treatment for a particular DS aggregate whose departure rate from any DS node either equals or exceeds a configurable rate: in this latter case, traffic *must* be discarded. The EF traffic should receive this rate independent of the intensity of any other traffic attempting to transit the node; conversely, whether the EF PHB is implemented by a mechanism that allows unlimited preemption of other traffic (e.g., a priority queue), a means (e.g., a token bucket rate limiter) to limit such damages is mandatory.

2.6.1 Virtual Wire PDB

The *Virtual Wire* (VW) PDB, whose definition is given in [VWPDB], uses the EF PHB to implement a transit behavior intended to mimic –from the point of view of the originating and terminating nodes– the behavior of a hard-wired circuit of some fixed capacity; this is done in a scalable way that doesn't require per-circuit state to exist anywhere but the ingress router adjacent to the originator.

Despite the lack of per-flow state, if the aggregate input rates are appropriately policed and the EF service rates on interior links are appropriately configured, the edge-to-edge service supplied by the domain will be indistinguishable from that supplied by dedicated wires between the endpoints.

Properties of VW PDB make it be suitable for any packetizable traffic that currently uses fixed circuits (e.g., telephony, telephone trunking, broadcast video distribution) and packet traffic that has similar delivery requirements (e.g., VoIP or video conferencing).

2.7 Assured Forwarding PHB Group

Assured Forwarding (AF) PHB group is a means for a provider DS domain to offer different levels of forwarding assurances for IP packets received from a customer DS domain. Packets subjected to AF PHB are forwarded with high probability as long as the aggregate traffic not exceed the subscribed profile; the contracted profile may nevertheless be exceed, with the key understanding that the out-of-profile traffic will not be delivered with a probability as high as those of the in-profile one.

2.7.1 Definition

Within the AF PHB Group, four independent *AF Classes* are defined: each AF class is, in each DS node, allocated a certain amount of buffer space and bandwidth forwarding resources.

No quantifiable timing assurance (e.g. delay or delay variation) can be associated with the forwarding of AF packets; instead, within each AF class IP packets are marked with one of three

possible *Drop Precedence* values, which determines the relative importance of the packet within that class: in case of congestion, packets with a lower drop precedence value are protected from being lost by preferably discarding packets with a higher one. These drop precedence levels, ranked from lower to higher, will be further indicated either with DP0, DP1 and DP2 labels or with Green, Yellow and Red packet *colors*.

	AF1x	AF2x	AF3x	AF4x
AFx1	001010	010010	011010	100010
AFx2	001100	010100	011100	100100
AFx3	001110	010110	011110	100110

Table 2.2: Assured Forwarding PHB Group Recommended DSCP

An IP packet that belongs to AF class i and has drop precedence j is marked with the AF codepoint AFij, where where $i \in [1,4]$ and $j \in [1,3]$; as a result of this definition, Green, Yellow and Red packet of a generic AF x class can also be indicated respectively as AFx1, AFx2 and AFx3. The recommended DSCPs, drawn from pool 1 of the codepoint space, are reported in Tab. 2.2; however DSCP may be remapped, and more AF classes (or levels of drop precedence) can be defined for local or experimental use.

As a final remark, it should be pointed out that the AF PHB group can be used, in conjunction with edge traffic conditioning, to obtain the overall behavior implied by the Class Selector PHBs (see Sec. 2.1.1.2).

2.7.2 Properties

In a DS node, the level of forwarding assurance of an IP packet belonging to a specific class depends on

- the amount of forwarding resources allocated to that class
- the current class load
- the drop precedence of the packet, in case of congestion within the class

A minimum amount of buffer space and bandwidth resources must then be allocated to each implemented AF class in each DS node, and each class should be serviced in a manner to achieve the configured service rate over both small and large time scales. When free network resources are available, either from other AF classes or from other PHB groups, an AF class may receive more forwarding resources than the established minimum; excess resources could either be evenly shared between the AF classes and the Default PHB, or alternatively be devoted to the Default PHB only when all AF demand is met.

Packets in one AF class are forwarded independently from packets in another AF class, that is, DS nodes do not aggregate two or more AF classes together: this enhance the ability of further service differentiation without posing *a priori* constraints on classes inter-dependence other than those entails from the network potential to fulfill the offer. Within each of these classes, packets belonging to the same microflow are not reordered when crossing a node, regardless of their drop precedence, as required in [RFC2474]. Furthermore, in networks where congestion is a rare and brief occurrence, the three drop precedence may yield at only two different loss probability levels, with DP0<DP1=DP2; however, to prevent network misbehavior, all the DSCPs earlier specified must be supported, also in this latter case.

2.7.3 Requirement

Any specific AF implementation, in order to be compliant to the previously defined PHB properties, must be independently configurable for each packet drop precedence and for each AF class: this in order to allow the AF PHB to be used in many different operating environments.

Moreover, within each class, AF PHB implementation must handle short-term bursts by queueing packets, while it must respond to long-term congestion by dropping packets, as required in [RFC2597]; the dropping algorithm must treat all packets having equal precedence level identically, and flows having different short-term burst shapes but identical longer-term rates should be treated fairly: that is, their packets should be discarded with essentially equal probability; finally, in order the system to reach a stable operating point, the level of packet discarded at each drop precedence need to be gradual rather than abrupt.

These constraints entail the need of a queue management mechanism whose algorithm both determines packets drop probability by computing a smoothed congestion level and uses randomness in the drop action. A class of algorithms meeting the requirements is represented by MRED class of AQM techniques described earlier in Sec. 2.3.3.3.

2.7.4 Service Examples

The AF PHB group can be used to implement a series of services whose guarantees focuses mainly on statistical throughput assurance, while neither delay nor jitter can be bounded. This section briefly describes a few examples of assured services that could be implemented by the use of one or more classes within the AF PHB group coupled with the traffic conditioners presented in Sec. 2.4, without entering in the details of the specific proposals.

2.7.4.1 Olympic Service

In this example, given in [RFC2597], the AF PHB group is be used to implement the so-called Olympic service, which consists of three service classes: bronze, silver, and gold. Packets are assigned to these three classes so that packets in the gold class experience lighter load (and thus have greater probability for timely forwarding) than packets assigned to the silver class, and the same kind of relationship exists between the silver and the bronze classes. Moreover, packets within each class are further separated by giving them either low, medium, or high drop precedence.

The bronze, silver, and gold service classes could be mapped in the DS domain to the AF classes AF1x, AF2x, and AF3x. Similarly, low, medium, and high drop precedence may be mapped to AF drop precedence levels AFx1, AFx2, or AFx3.

A possible way to assign the drop precedence level of a packet could be to limit the user traffic of an Olympic service class to a given peak rate and distribute it evenly across each level of drop precedence. This would yield a proportional bandwidth service, which equally apportions available capacity during times of congestion under the assumption that customers with high bandwidth microflows have subscribed to higher peak rates than customers with low bandwidth microflows.

2.7.4.2 Assured Rate PDB

Up to date, no PDB using AF PHB have been standardized yet; nevertheless this is a subject of much on-going study: the *Assured Rate (AR) PDB* [ARPDB] is one of these interesting proposals.

The AR PDB ensures that traffic conforming to a Committed Information Rate (CIR) will incur low drop probability; further, the aggregate will have the opportunity –but no assurance– of obtaining excess bandwidth beyond the CIR. Different SLA models may be used to decide how out of profile traffic should be charged: for example, excess may be charged proportionally, or at a higher rate than assured one, or even be cheaper than the latter. The key parameter of this PDB is evidently the CIR, eventually specified in terms of a Committed Burst Size (CBS) and an averaging time interval T, basing on which the CIR should be measured. In addition to the above, the PDB may have optional extra traffic parameters, either to place further constraints on the

packets to which the assurance applies or to further differentiate packets to which the assurance does not apply.

The AR traffic aggregate can be mapped to use codepoint of a single AF class: green, yellow and red packets will be treated with AFx1, AFx2 and AFx3 PHB respectively.

A first example of its applicability range is represented by *Virtual Private Network* (VPN) like services: the PDB can be utilized to assure a rate for a traffic aggregate between an ingress and an egress within a domain (one-to-one) or from one ingress to few different egress points in the domain (one-to-few). In the former case, is easy for an ISP to demonstrate conformance with the SLS, since all measurements can be performed at a single egress point. Assuming static routing, the links involved in transmitting the aggregate can be identified and capacity in those links can be reserved; an important note is that, sending at full rate, assured rate traffic might use all the service class capacity. In the case of a one-to-few service, measurements need to be performed at all the egress nodes visited by individual flows within the aggregate, in order to determine the cumulative bandwidth of the aggregate as it exits the domain.

In another case of interest, the PDB can be utilized to realize one-to-any assured rate services, where the traffic aggregate originates from one ingress node but whose individual five-tuple flows may exit the domain at any of the egress nodes. The problems addressed by such a kind of service are much more complex than those of the earlier presented VPN-like cases: providing an assured rate with almost no drops can be too expensive, and an admission rule is more difficult to derive. First of all, the links involved with the traffic aggregate vary depending on the destination to which the user is sending at some point in time: reserving capacity in all possible links would imply a hard limit to the amount of assured rate traffic that could be accepted. Furthermore, in this case, traffic aggregate could at most utilize only a small portion of the total capacity, thus resulting in a low efficiency; however, if capacity is not reserved for all possible links, there will be a certain probability that the rate assurances are not met: the one-to-any AR PDB should needs thus to considers the possibility that assured rates commitments are *not met with a certain probability* as additional structural parameter.

Chapter 3

Analysis of Time Sliding Window Three Color Marker

This chapter describes and analyzes the Time Sliding Window Three Color Marker (TSWTCM) experimental protocol, proposed by W. Fang, N. Seddigh and B. Nandy [RFC2859]; this traffic conditioning objetc is primarily designed to be used in the context of the Assured Forwarding PHB.

Markers are logical objects charged to differentiate incoming packets of a traffic stream by tagging (coloring) them. In a DiffServ compliant core router implementing the AF PHB, packets are given different drop precedence level basing upon their color: Green, Yellow and Red marked packets correspond to respectively DP0, DP1 and DP2 growing drop precedence levels.

In DS edge routers, TSWTCM marking is performed comparing the measured throughput of the traffic stream against two thresholds: Committed Target Rate (CTR) and Peak Target Rate (PTR). Packets contributing to sending rate lower than CTR are marked *green* (G); packets contributing to the portion of the rate between CTR and PTR are marked *yellow* (Y); finally, packets causing the rate to exceed PTR are marked *red* (R).

The rest of the chapter is organized as follows: Sec. 3.1 describes TSWTCM functional blocks and algorithms and Sec. 3.2 examine the marker behavior from a probabilistic point of view; Sec. 3.3 reports and discuss simulation results of a MATLAB implementation of the marker and finally Sec. 3.4 gives some conclusive interpretations of the results, including a Service Level Specification (SLS) point of view.

3.1 TSWTCM Definition

The TSWTCM consists of two indipendend components: a *rate estimator* and a *marker*, as shown in Fig. 3.1. The former provides an estimate of the running average bandwidth, taking into account burstiness and smoothing out its estimate to approximate the longer-term measured stream sending rate; the latter uses the estimated rate information to probabilistically associate packets with either G, Y or R color, then translating it into a DS field marking with the DS codepoint correspondent to that particular DP level.

3.1.1 Rate Estimator

The Rate Estimator provides an estimate of the traffic streams arrival rate, Estimated Average Rate (EAR), which should approximate the running average bandwidth of the traffic stream over a specific time-window. Both time-based (i.e. provided in [EXPALL]) and weight-based (i.e. exponential weighted moving average (EWMA)) history approaches are allowed; in the latter case, the estimator should be able to translate the weights into time periods; focusing on rate

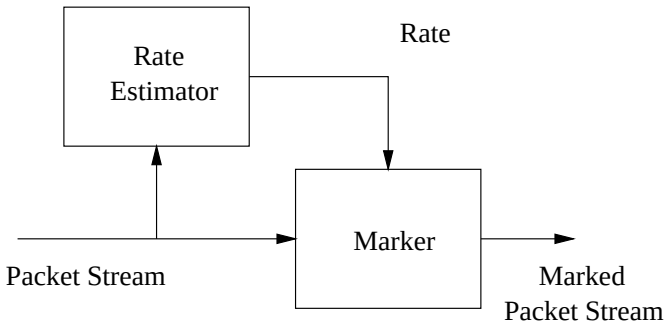


Figure 3.1: TSWTCM Block Diagram

estimator algorithms is outside of the scope of this document, due to simulation assumptions in Sec 3.3.

3.1.2 Marker

The Marker determines the color of a packet based on simple control theory principles of proportionally regulated feedback control; the marker algorithm is presented in Figure 3.2.

Using a probabilistic function in the marker is beneficial to TCP flows as it reduces the likelihood of dropping multiple packets within a TCP window. The marker also works correctly with UDP traffic, i.e., it associates the appropriate portion of the UDP packets with yellow or red color marking if such flows transmit at a sustained level above the CTR.

avgrate $\leftarrow$ EAR()
if (avgrate $\leq$ CTR)
    packet $\leftarrow$ green
elseif (avgrate $\leq$ PTR)
    $P_0 \leftarrow$ (avgrate - CTR)/avgrate
    with P_0, packet $\leftarrow$ yellow
    with $(1 - P_0)$, packet $\leftarrow$ green
else
    $P_1 \leftarrow$ (avgrate - PTR)/avgrate
    $P_2 \leftarrow$ (PTR - CTR)/avgrate
    with P_1, packet $\leftarrow$ red
    with P_2, packet $\leftarrow$ yellow
    with $(1 - (P_1 + P_2))$, packet $\leftarrow$ green
endif

Figure 3.2: TSWTCM Marker Algorithm

3.2 Probabilistic behavior

The TSWTCM marker is configured by assigning values to its two traffic parameters: CTR, also know as Committed Information Rate (CIR), and PTR, also know as Peak Information Rate (PIR), with respec to CTR $\leq$ PTR $\leq$ line rate. If the inequality strictly holds, packets color will be either green, yellow or red, with a probability depending on EAR value; given the couple (CTR,PTR), the drop level probability of each packet within a flow depends on EAR.

Packets are marked green as long as EAR < CTR; when EAR ∈ (CTR,PTR], packets are yellow with probability P_0 (where P_0 is the fraction of packets contributing to the measured rate beyond the CTR) and green with probability $(1-P_0)$. If EAR ∈ (PTR,linerate] red packet marking arises: in this case P_1 represent the fraction of packets contributing to the measured rate beyond the PTR, P_2 the fraction of packets contributing to the measured rate beyond the CTR but below PTR.

Some examples of marker algorithm's GYR probability outcomes for different (CTR,PTR) couples are plotted in Fig. 3.3, where abscissas axis represents EAR, which value is assigned to avgrate in the algorithm.

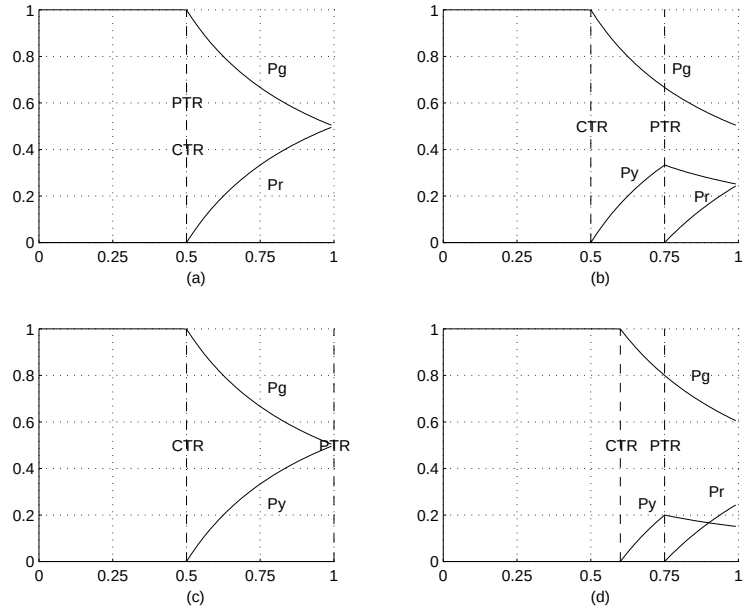


Figure 3.3: Marker Outcomes for a given (CTR,PTR)

It can be noticed by comparing Fig. 3.3(b) to Fig. 3.3(d) that, depending on (CTR,PTR), P_r can exceeds P_y over a certain EAR; as it will be shown in the next sections, this is undesirable and the couple (CTR,PTR) should be chosen in order to avoid it.

The TSWTCM can be configured so that it essentially operates with a single rate, thus becoming a Two Color Marker.

- If the PTR is set to the same value as the CTR then all packets will be colored either green or red, as shown in Fig. 3.3(a).
- If the PTR is set to link speed and the CTR is set below the PTR then all packets will be colored either green or yellow, as in Fig. 3.3(c).

In the following, we will assume that inequality holds but not strictly, that is CTR ≤ PTR; furthermore, we will assume to operate on a link with a line rate of 1 Mbps, and CTR will assume values not lower than 0.5 Mpbs.

Another graphical way to represent marker algorithm behavior is presented in Fig. 3.4; as for each EAR we have $P_r + P_y + P_g \equiv 1$, we can express yellow probability as the sum of green and yellow, thus $\overline{P_y} = P_g + P_y$, and similarly $\overline{P_r} = P_g + P_y + P_r \equiv 1$. The superposed white, gray and black slices of the plot represent respectively green, yellow and red packet marking areas.

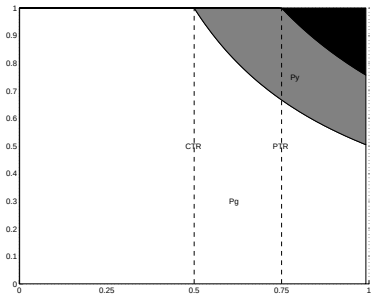


Figure 3.4: Superposed Marker Outcomes

If we reduce the choice of (CTR,PTR) in setting a PTR for a given CTR, we can then plot coloring probability outcomes as function of the EAR for different PTRs. From Fig. 3.5 we can observe that the probability that a packet will be green colored will decrease as EAR is increasing, but is fixed respective to PTR; thus PTR determine the fraction of yellow and red packets of the residual probability after CTR, and so the green portion, has been set.

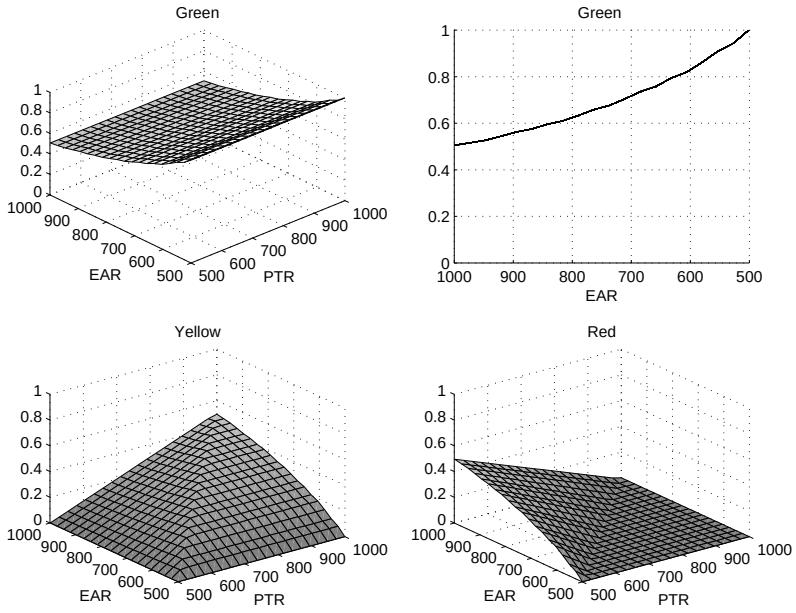


Figure 3.5: Marker Outcomes for a given CTR

3.3 Simulation

3.3.1 Simulation Assumptions

We are now interested in observing TSWTCM behavior from close up, making some assumptions to consider the whole network around the marker (the rate estimator algorithm too) as a black-

box. That is, instead of defining network nodes, node's propriety, links and applications agents such as it would be done in an network simulation, we'll go deep in MATLAB simulation. Key parameters to the TSWTCM analysis are:

CTR: it ranges from 500 Kbps to 1000 Kbps; as a notation of graphic plots, its value will be expressed either as a *real number* $r \in [0.5, 1]$ in which case the omitted measure unit will be $[r]$=Mbps, or as an *integer* n, with $[n]$=Kbps.

PTR: besides what has been said for CTR, PTR won't be lower than CTR nor higher than the line rate B, fixed to the value of 1 Mbps.

Furthermore, two function will be input to the TSWTCM: the Estimated Average Rate EAR(x) and the Free Bandwidth BF(x). The former is the rate of a packet stream, considered as an unique macroscopic DiffServ enhanced flow, which packets have ideally to be forwarded according to a generic PHB of AF service class. Since the TSWTCM rate estimator could implement either a time-based sampling or arrival-based sampling, then the independent variable x of EAR(x) can be expressed in terms of either time or packet arrival samples, provided that its estimate takes in account burstiness and gives a smoothed approximation of the long term measured sending rate. Four different EAR function, plotted in Fig. 3.6, will be used for simulation purposes. These function are modeled as result of the overall combination of different network agents with different behaviors, attempting to recreate some scenarios similar to the real network traffic; simple linear increasing behavior and on-off burstiness-like behavior are also taken into account.

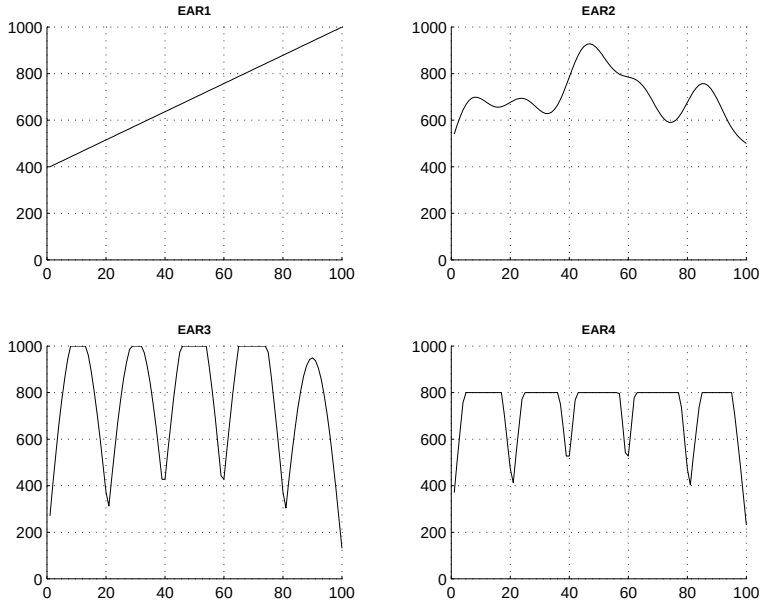


Figure 3.6: EAR Functions

The Free Bandwidth function BF(x), where the independent x variable measure unit is chosen accordingly to those of CTR(x), represents the available link capacity seen by the marker; its purpose is essentially to take into account the network load due to other privileged and non privileged traffic besides the DiffServ AF PHB traffic. Different functions, plotted in Fig. 3.7, model various macroscopic behaviors of different traffic aggregates; in the simplest case, all the bandwidth is available to the incoming DiffServ stream, while the others take into account shaped traffic streams, accordingly to considerations similar to those stated for EAR functions.

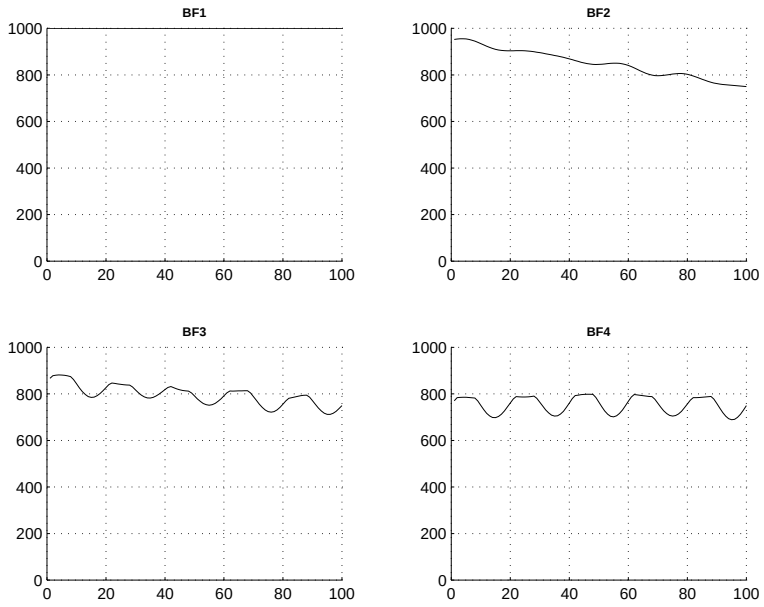


Figure 3.7: BF Functions

The combination of EAR and BF functions produces 16 different scenarios, including both over-provisioned and under-provisioned cases; the simplest scene, build on combination of the linear increasing average incoming rate EAR_1 and the constant free bandwidth BF_1 set to linerate B, is analyzed in Sec. 3.3.2 with the purpose of introducing the TSWTCM setting matter, while Sec. 3.3.3 analyzes the case in which EAR_2 and BF_2 are input to the TSWTCM algorithm.

3.3.2 A Simple Scene

The first-step of the analysis of red, yellow and red coloring probability, shown in Fig. 3.8, is handled by a two levels iterative inspection: for each CTR sample, the algorithm is run at each PTR sample above CTR, to cover all the possible configurations. For each of the above (CTR,PTR) couples the TSWTCM marker algorithm, with inputs functions EAR_1 and constant $BF_1 \equiv$B shown in Fig. 3.8 (d), is run 10 times; during each run, marked output packets are counted and the final number of green, yellow and red packets is mapped to range $[0, 1]$ by normalization over totally sent packets number; the results are then averaged over all the runs to eliminate probabilistic abruptness in the data set.

Lets observe the plots for different marking colors, which correspond to different drop precedences in a DiffServ context. First thing to notice is that the parameters couples for which PTR<CTR have their output set to dummy value of 0 for *any* marking color. Second, each time that PTR equals CTR, the algorithm doesn't color any packet as yellow: as expected from previous results shown in Fig. 3.3 (a), packets will be marked either red (with a peak value for the smallest CTR value of 500Kb) or green (which portion is equal to 1 for CTRs above 800 Kb); similarly, if PTR is set to line rate, no packets will be red marked, according to Fig. 3.3 (c).

3.3.2.1 A Criterion for (CTR,PTR) choice

To further investigate the plot results, we need to define a *purpose* and an algorithmic *criterion* to reach it. Knowing the EAR-dependent marking outcomes the main purpose will be, given any

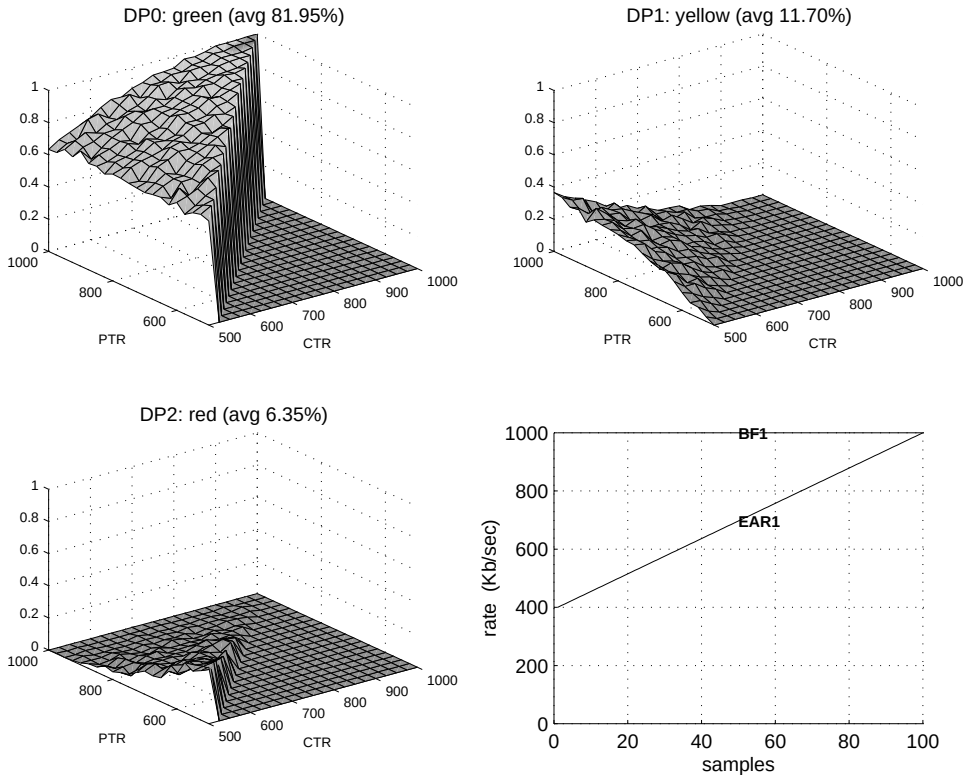


Figure 3.8: GYR Marked Fractions, Scene (EAR_1,BF_1)

CTR, to find the *optimal* respective PTR_{opt}; the criterion, whose decision is based upon the GYR fraction issues for any (CTR,PTR) pairs, should first of all define what we mean by optimal, which is slightly complicated.

It would be suitable to have a reasonably high green marking probability, but being this factor only influenced by CTR, by its lonely application we would define a *set* of optimal PTRs instead of an unique optimal PTR value. Furthermore the choice of CTR is extremely important, as in the case of green marked packet overgrowth a problem would arise: if, by unreasonable augmentation of CTR, we maximize green outcomes regardless of EAR, too many packets above the contracted target rate (or even above the peak committed rate) would be marked as green, with the result of possible in-profile (i.e., green marked and belove CTR) packet dropping in case of long term congestion; green packet dropping will start after all red and yellow packets have been dropped, but since yellow and red coloring probability have been indiscriminately lowered, the amount of red and yellow packet in a stream could be insufficient to protect green packets against dropping.

Moreover, red versus yellow overcoming should be avoided, since it lead to a resources waste and eventually (in the case of multiple packets drop of a single flow within a TCP window) in slowering the transfer process. If we imagine to overlap red and yellow plots, we notice that their intersection will give the PTR lower bound for which red marked fraction won't be higher than the yellow one. For each CTR, the PTRs greather than this bound are scanned to find a local maximum of the green fraction thus deciding which is preferable in the (EAR_1,BF_1) context.

Finally, depending on the peculiar scene individuated by EAR and BF, we may need to refine and smooth out the optimized PTRs values obtained by the application of this criterion.
In Fig. 3.9 the abscissas axes always represent CTR range; Fig. 3.9 (a) plots the optimal PTR_{opt}

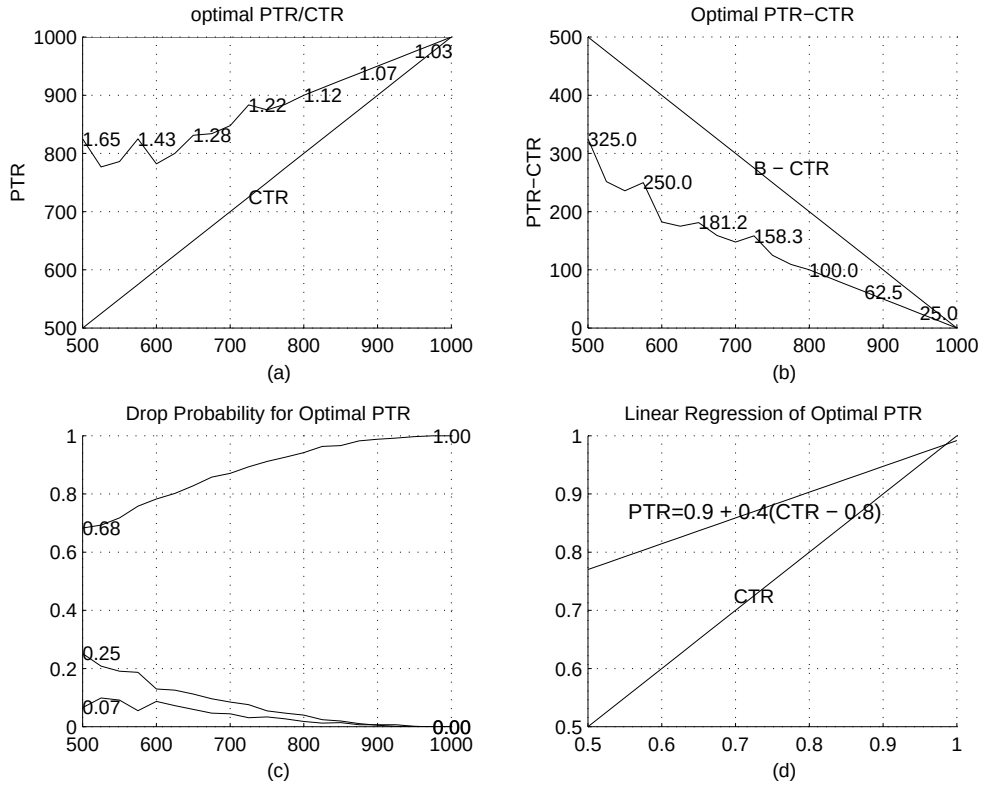


Figure 3.9: Attempting to Optimize PTR choice for given CTR

and tag some samples with the correspondent PTR_{opt}/CTR ratio; Fig. 3.9 (b) expresses PTR_{opt} in terms of the difference PTR_{opt}-CTR, which values are listed for some samples; Fig. 3.9 (c) plots the coloring proportions when TSWTCM is configured with the proper PTR_{opt} for each CTR; Fig. 3.9 (d) approximates granularity simply by linear regression of the samples: its parameters values are $\overline{\mathrm{CTR}} = 750\mathrm{Kbps}$, $\overline{\mathrm{PTR}}_{opt} = 881\mathrm{Kbps}$, $b = 0.443$; so the linear smooth of the data set can be expressed as:

$$\begin{aligned} \mathrm{PTR}_{opt} &= \overline{\mathrm{PTR}}_{opt} - b(CommitedTargetRate - \overline{\mathrm{CTR}}) \\ &= 881 - 0.443(CommitedTargetRate - 750) \\ &= 548.5 - 0.443 \cdot CommitedTargetRate \end{aligned}$$

where both $CommitedTargetRate$ and PTR_{opt} are expressed in Kbps. The simplicity of the data set fitting function isn't a limit yet, since a straight line well approximate the optimization couples; moreover, it allows easy quick comparisons with other scene's optimizations issues.

3.3.2.2 Another Criterion

We can anyway define more optimization criteria different from the previous one in a simple way by logically combining some test blocks. First of all, we redraw the outcome fractions in a merged representation shown in Fig. 3.10, where for each (CTR,PTR) the red, yellow and green marking outcomes outcomes are re-normalized, weighted and mixed together; as for each drop precedence level (DP0,DP1,DP2) the lower probability correspond to a darker color (respectively green,yellow

and red) tone, this leads to a color blended visual representation, which has the only purpose of immediateness in both implementing and verifying the new criterion. In this memo, however, this would not be so evident, since plot palettes have been remapped to grayscale, loosing the beautifulness of the color vision and the its peculiar helpfulness.

Each functional test block should represent a well defined *quasi*-atomic criterion, in order to provide wide flexibility in building the overall criterion scheme; useful test blocks could be, i.e., incremental ratio test at various thresholds, inequality tests, combined differences tests, etc. The new optimization algorithm, expressed in english words, would sound: *"a (CTR,PTR) couple is better than another if it leads to obtaining more green packets (avoiding green packets over-marking) provided that red packets are not overcoming yellow ones, or if it leads to a reasonably low green fraction loss in front of a more advantageous fractioning of red and yellow packets, especially if yellow (red) is (not) growing or its (un) growing ratio is above a certain threshold"*

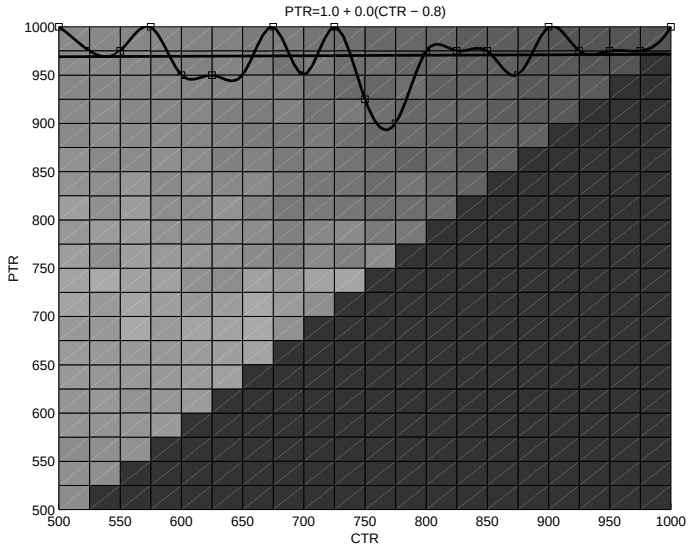


Figure 3.10: Another PTR optimization Criterion, Scene (EAR_1,BF_1)

Results for this second criterion, plotted over the merged representation in Fig. 3.10, are really different to previous optimization issues. The optimization issues should be interpreted as follow: in a well over-provisioned network case as this one, it would be advantageous to permit a higher green packet marking rate, since almost no packets are likely to be dropped; anyway, if the overall flow rate would approach the line rate, the TSWTCM should be setted to a safer CTR. The linear regression parameters set $\overline{\mathrm{CTR}} = 750\mathrm{Kbps}$, $\overline{\mathrm{PTR}}_{opt} = 970.24\mathrm{Kbps}$, $b = 5.2 \cdot 10^{-3}$ is reported to allow comparison with previous criterion:

$$\begin{aligned} \mathrm{PTR}_{opt} &= \overline{\mathrm{PTR}}_{opt} - b(CommitedTargetRate - \overline{\mathrm{CTR}}) \\ &= 970.24 - 5.2 \cdot 10^{-3}(CommitedTargetRate - 750) \\ &\simeq 966.3 \end{aligned}$$

3.3.3 A More Realistic Scene

Let examine another scene, were EAR_2 and BF_2 are shown in Fig. 3.11 (d), together with TSWTCM outcomes.

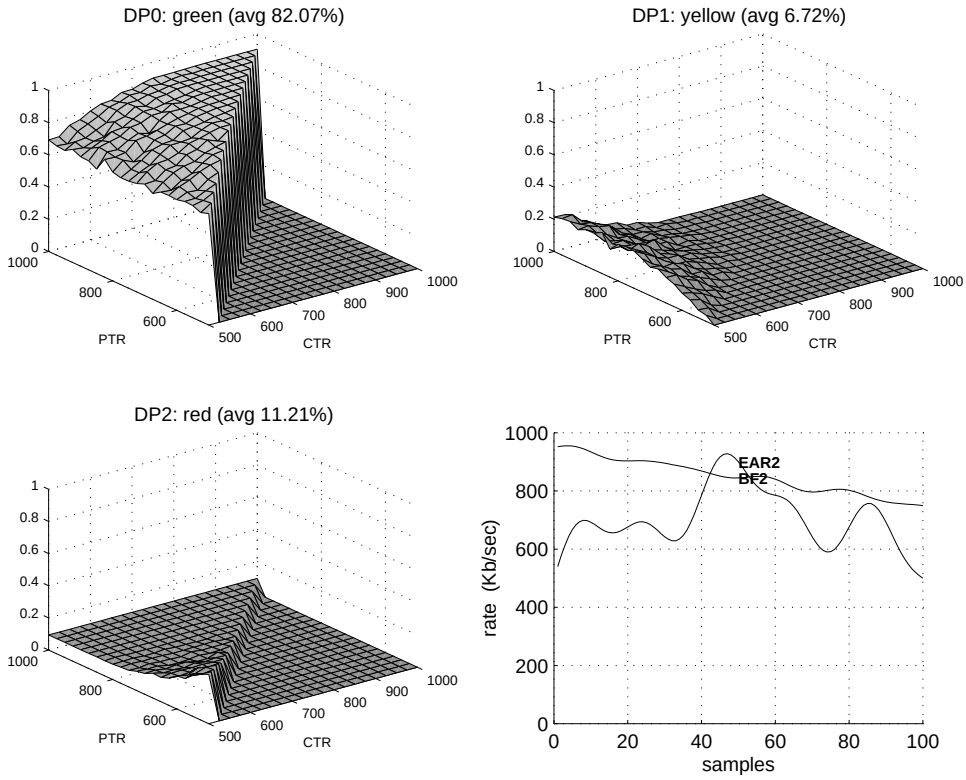


Figure 3.11: GYR Marked Fraction, Scene (EAR_2,BF_2)

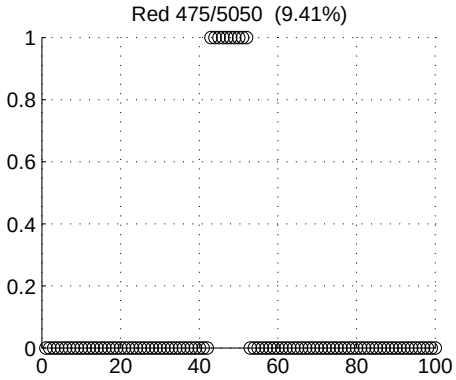


Figure 3.12: Best Case Red Packet Marking

3.3.3.1 Straight Line Criterion

By observing Fig. 3.13 (c), we gather that, opposite to previous case, marked fractions reach their asymptotic values for CTR over ∼800 Kbps; furthermore, red marking can't be avoided nor decreased belove ∼10%, even if PTR is set to B; this is due to due to short-term congestion phenomenon, and a better case for red marking than those shown on Fig. 3.12 can *not* be obtained.

Linear regression has parameters $\overline{\mathrm{CTR}} = 750\mathrm{Kbps}$, $\overline{\mathrm{PTR}}_{opt} = 871.4\mathrm{Kbps}$, $b = 0.536$, lead-

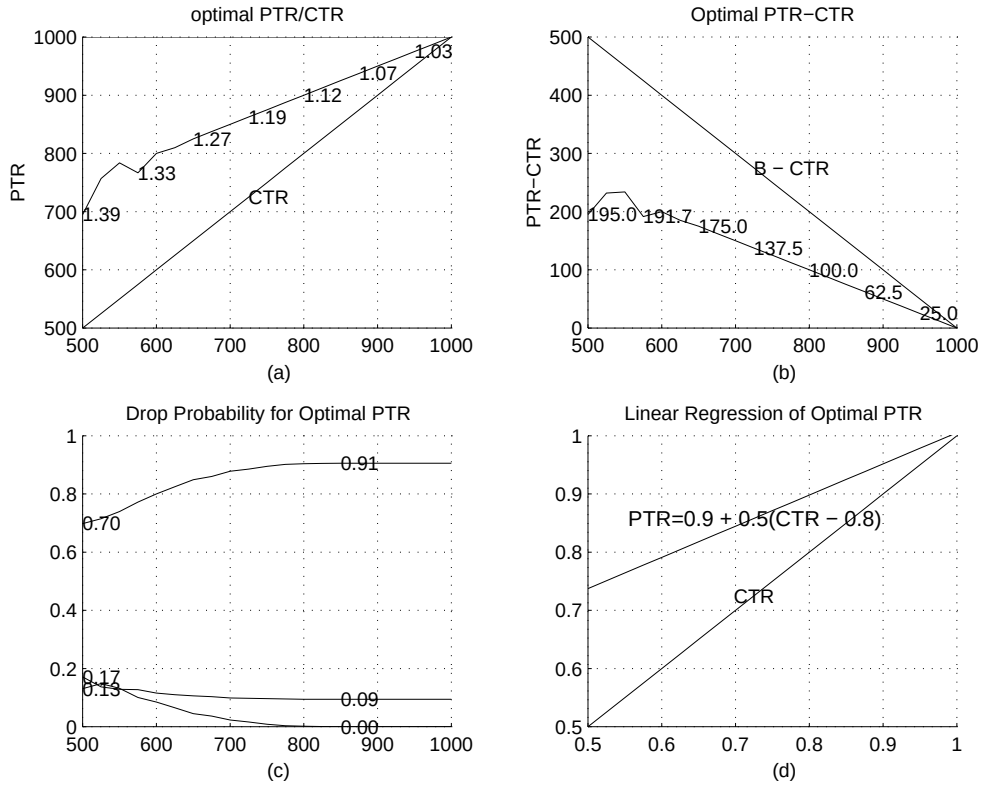


Figure 3.13: Static Optimal PTR choice for a given CTR

ing to $\mathrm{PTR}_{opt} = 871.4 - 0.536(CommitedTargetRate - 750) = 469.6 - 0.536 \cdot CommitedTargetRate$

	μ	σ^2	
B	1000	0	Kb/s
$\overline{\mathrm{CTR}}$	750	0	Kb/s
$\overline{\mathrm{PTR}}_{opt}$	878	11	Kb/s
b	0.487	0.015	

Table 3.1: Statistic Mean and Standard Deviation

Tab. 3.1 summarizes results of all the examined scenarios. The first two entry of the table are reported to remember that these indications on parameter choice were gathered belove a restricted CTR range: from the half of to the whole line rate, fixed to B $\equiv$ 1 Mbps. Value inspection let us write $\overline{\mathrm{PTR}}_{opt} \simeq 1/2 \cdot (B + \overline{\mathrm{CTR}})$.

3.3.3.2 Logical Blocks Criterion

From the adaptive criterion we can't get as easily as for the previous case a "rule of thumb" for the optimal PTR choice; linear regression parameter are, for completeness: $\overline{\mathrm{CTR}} = 750$Kbps, $\overline{\mathrm{PTR}}_{opt} = 870.24$Kbps, $b = 0.22$. Comparing Fig. 3.14 to Fig.3.10 we see how wide is its issue range, which also depends on the network load feedback.
About its behavior on this short-term under-provisioned situation it can be noticed that

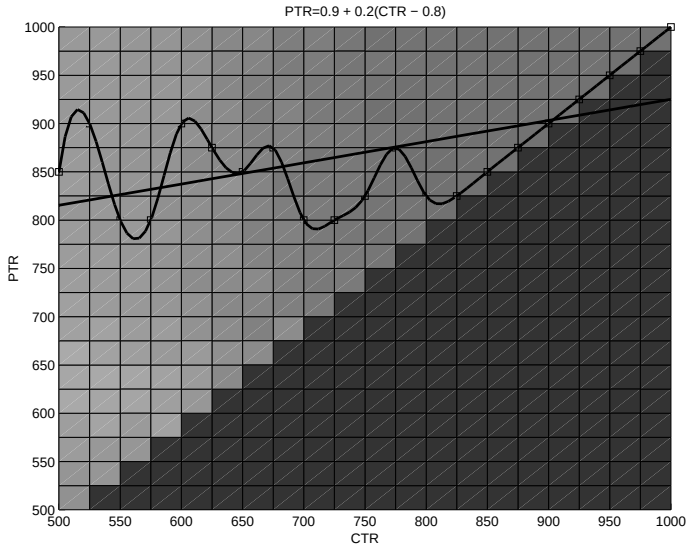


Figure 3.14: Merged Representation

- for rates below ∼800 Kbps, PTR_{opt} present wide oscillations; this may be due to EAR shape, imprecise fine tuning of the logical block thresholds, and cubic spline interpolation of the coarse data samples.

- for EAR rates above ∼800 Kbps, PTR_{opt} is set by the algorithm to its possible lowest value, that is, CTR; this is a matter of importance to prevent as possible in-profile packets drops from being marked other than green: in a more realistic network simulation using AQM techniques (e.g. MRED), green buffer would ensure best protection and thus higher forwarding probablilty with respect to any higher drop precedence colors.

Different combinations of atomic blocks (which results are not reported) were tried, as well as different configurations for the same combination. Oppositely to the previous algorithm, this one doesn't lead to stable and easily recognizable TSWTCM parameters issues; its thresholds configuration is a delicate problem, as most of the optimization decisions are expressed as tests, and even if the results are averaged, the criterion output cannot be easily fitted nor compared with a simple function.

An ISP could monitor system performances over a certain period of time to determine, i.e. via the first criterion, whether its settings are still valuables or not; in the latter case it may be necessary to renew the SLS (see Sec. 3.4). Alternatively, an ISP might decide to change its internal marker settings accordingly to the real-time traffic situation; this would certainly involve an overhead, which relevance would depend on both the complexity of the setting switcher algorithm and the update rate. Performing such an automatic update can be done in a simple manner, allowing the settings to be chosen from a pre-build set if some specific network event occur. But of course, it could be more sophisticated (and eventually less performant too) if a well-designed algorithm is taking charge of optimal real-time settings computation.

A test blocks like algorithm may be used to accomplish this task, even if this study can't give precise indications about its design.

3.4 Conclusive Considerations

3.4.1 Key Parameters

A lower bound of PTR in function of the EAR can be analytically obtained by choosing a PTR such that $P_1 \leq P_2$ in TSWTCM marker algorithm of Fig. 3.2; provided that avgrate> PTR, it results

$$\text{PTR} \geq 1/2 \cdot (\text{avgrate} + \text{CTR}) \tag{3.1}$$

From Eq. 3.1, the lower PTR bound mean $\mathbb{E}[\text{PTR}]$ can be derived and then compared with the $\overline{\text{PTR}}_{opt}$ obtained via the first criterion:

$$\mathbb{E}[\text{PTR}] \geq 1/2 \cdot (\mathbb{E}[\text{avgrate}] + \mathbb{E}[\text{CTR}]) \tag{3.2}$$

$$\overline{\text{PTR}}_{opt} \simeq 1/2 \cdot (\text{B} + \mathbb{E}[\text{CTR}]) \tag{3.3}$$

It should be noted about $\overline{\text{PTR}}_{opt}$ that

- it reflects the worst-case $\mathbb{E}[\text{avgrate}] = \text{B}$ where the node incoming traffic stream always reach, on average, the available node bandwidth
- red packet marking will effectively not overcome yellow coloring
- even thought it has been derived from a particular case, its validity can be extended to wider CTR ranges than those here examined and B $\neq$ 1 Mbps
- similarly, it can be hypotesized that extending the optimization algorithm parameters range will not invalidate its behavior, as its $\overline{\text{PTR}}_{opt}$ estimate is coherent with what stated in Eq. 3.2.

Furthermore, it may be helpful to express the marking probability bound for EAR=B in terms of CTR,PTR,B as presented in Fig. 3.15.

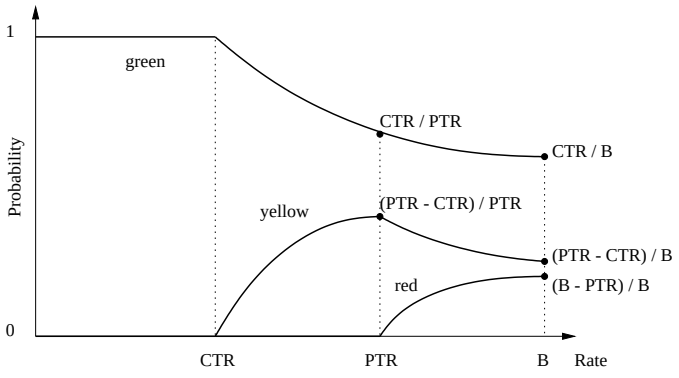


Figure 3.15: Marking Probability Parameters

On a node with a fixed capacity B, the (CTR,PTR) parameters could then be expressed as a function of the requested lowest green probability G_B (achieved for EAR=B) and the corresponding Y_B, R_B probabilities:

$$\begin{cases} \text{CTR/B} = G_B \\ \text{PTR/B} = G_B + Y_B = 1 - R_B \end{cases}$$

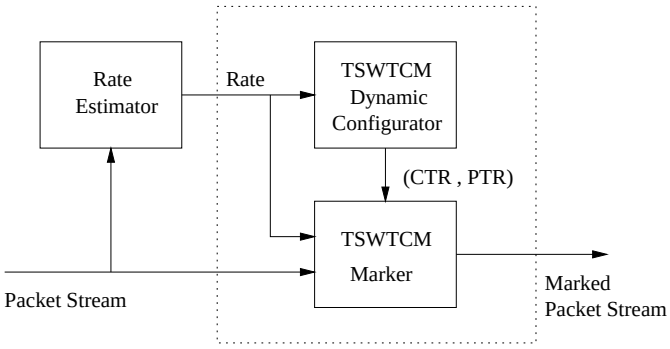


Figure 3.16: Dynamic TSWTCM Block Diagram

From Eq. 3.1, we get the basis for an interesting starting point: as the TSWTCM is aware of the EAR information, its parameter's setting could be dynamically other than statically (as for the first optimization criterion) done.
The the Dynamic Configuration block of Fig. 3.16 could be implemented in numerous ways:

- it could change TSWTCM settings either on each n incoming EAR values or based on a configurable time interval
- the configuration could be either based on look up tables or computations
- the configuration algorithm could be either hardware or software
- a logical blocks PTR optimization criterion could be used

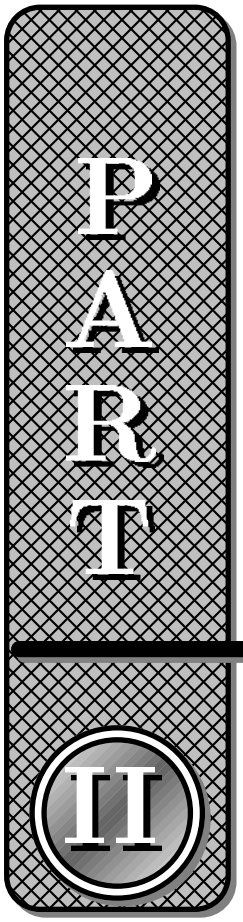
If the configuration is implemented in arrival-based fashion, a criterion to detect short-term arising congestion can be provided based on EAR and EAR incremental ratio; otherwise, the configuration rate should be reasonable in order to avoid introducing excessive overhead, thus obtaining worse network performances.

In a case of well over-provisioned network it would be advantageous to permit an higher green marking rate, by growing CTR value, as long as the overall flow rate is not reaching nor approaching the line rate; when this happens, TSWTCM parameters should be setted to a safer configuration, either presetted or dynamically calculated. This dynamic configurable TSWTCM could become somehow similar to the Green Rate Adaptive Shaper G-RAS [RFC2963] (developed to support srTCM[RFC2697] and trTCM [RFC2698] markers) by providing a feedback mechanism with a RAS (which output would be the packet stream input of Fig. 3.16). However this area of interest is outside the scope of this document.

3.4.2 SLS Considerations

The TSWTCM can work with both sender-based service level agreements and receiver-based service level agreements, and there are no restrictions on traffic stream type: it can be used to meter and mark individual TCP flows, aggregated TCP flows, aggregates with both TCP and UDP flows. Used in conjunction with a service of the AF PHB class an ISP can provide decreasing levels of bandwidth assurance for packets reaching (or originated from) customer sites.

With sufficient over-provisioning, customers are assured of mostly achieving their CTR; sending rates beyond the CTR will have lesser assurance of being achieved and finally rates beyond the PTR have the least chance, due to high drop probability of red packets. It must be pointed out that allowing CTR to vary by dynamic configuration, implies the Service Provider to charge a tiered level of service based on the final achieved rate. However, even for fixed (CTR,PTR) settings this would be probably the unique way to define an SLS tariff, by virtue of probabilistic TSWTCM behavior.



Long Lived FTP Flows

Chapter 4

Long Lived Flows Simulation Introduction

Industry attention has recently been focused on providing differentiated level of services to users on IP networks: a necessary step toward the deployment of a DS model in the Internet is clearly represented by the understanding of the achievable service performances, compared to common Best Effort ones, via a performance analysis sensitivity to common network parameters. Furthermore, although the IETF DiffServ WG is not exploring service related issues, it is however necessary to examine, evaluate and understand the end-to-end Quality of Services that DS model offers, further trying to assess the quantitative assurance which can be given in a SLA contract.

The evaluation of Differentiated Service presented in this dissertation has been provided through simulations performed with VINT Network Simulator *ns* v2.1b8a [NS]: this release of the simulator implements DiffServ specific support, thanks to the efforts of Nortel Network's Open IP Group. Simulation scripts have been implemented using a set of OTcl classes, developed to enhance script flexibility and coherence and to simplify code maintenance task; in the following, we will focus on network scenarios' study rather than their actual implementation in the simulator, leaving thus apart any simulator-specific consideration: an exhaustive description of the *ns* code and library developed is provided in Appendix A.

This chapters provides the necessary introduction to the analysis of simulation results presented in Sec. 5. Considerations developed focuses then on simulations planning and goals, deeply analyzing the relationship between the monitored variables and the corresponding performance parameters, and introducing the notation used in result analysis.

4.1 Simulation Scenarios

This section focuses on the description of the network topology used in all simulations; then the monitored performance parameters are enumerated, and finally the key scenario parameters, as well as their known effects on performances, are briefly described.

4.1.1 Network Topology

The topology used by this first simulations set, shown in Fig. 4.1, is a bottleneck link representing a simple but complete DiffServ domain: it derives from the classic Dumbbell topology, differing from the latter in that traffic offered by every source node has a common destination node. The DS domain is build by two DS edge routers connected by a single DS core node: source nodes and destination node are "external" to the domain.

All the sources produce continuously backlogged flows, generated by an emulated File Transfer Protocol (FTP) application using as transport layer the standard Reno version of the Transmission Control Protocol (TCP). Network supports two kind of services: the common Best Effort behavior,

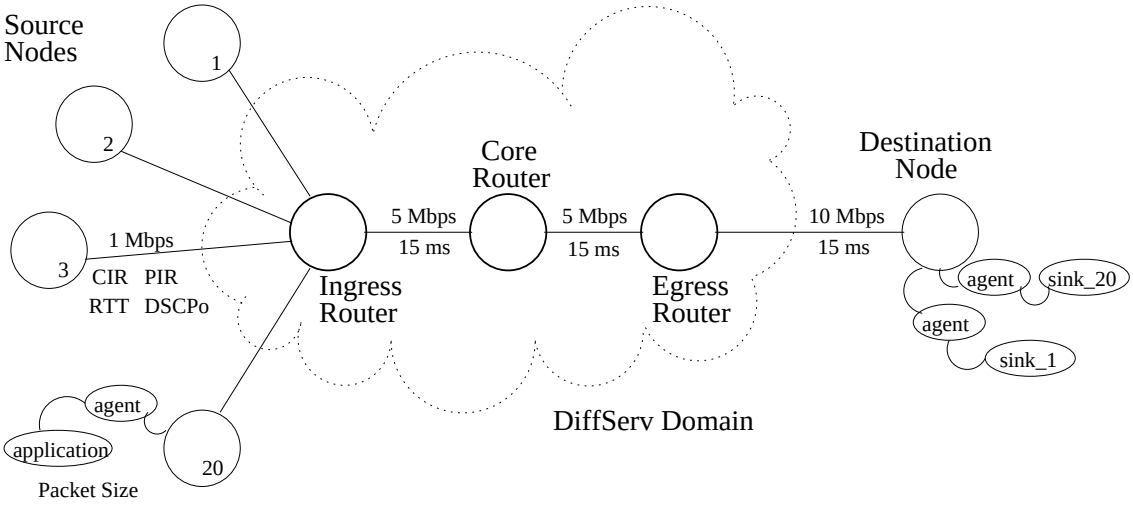


Figure 4.1: Long Lived Flows Simulation Network Topology Scheme

and, build on the top of an AF PHB group's class, the AR PDB (Sec. 2.7.4.2); in the following we will refer to Assured Rate service flows indistinctly as DS, AF or AR flows, being AR the unique service offered by the DS domain other than BE. The idea behind AR is to give the customer the assurance of a minimum throughput, even during periods of congestion, while allowing him to consume more bandwidth when network load is low: thus an AR connection should achieve a throughput equal to the subscribed rate, plus some share of the remaining bandwidth gained by competing with all the active BE and AR connections. AR service level specification may be expressed uniquely in terms of the reserved bandwidth; this kind of service is appealing in its apparent ease of deployment, but at the same time little measurement and analysis has been done to determine its range of applicability and whether it scales to the entire internet.

Every source node is able to generate one or more unidirectional long lived flows; ACKs are never lost on the backward path: as a consequence the performance of the communications will be better than in a real situation where ACKs can be lost. FTP packets are marked at the source with the DSCP corresponding to the contracted service: DS packets are initially marked with the AFx1 DSCP, indicating lowest drop precedence (i.e. their are green colored), but may be re-marked at the ingress router depending on their conformance to the traffic profile; BE packets are marked with the Default DSCP and receive the same treatment as packets whose DSCP is AFx3 (i.e. as highest drop precedence DS red packets). Moreover, every source node have his own set of parameters (i.e. its own SLA defined by means of AIR and PIR, the size of FTP packets, link Round Trip Time, ...) and flow starting time is randomly distributed within the first second of simulation time. The traffic produced by these infinite flows have been modeled to be able to fully utilize the bottleneck link during the whole simulation duration, which is set to 450 seconds. Indicating with RTT the global Round Trip Time and with B the bottleneck bandwidth, simulation results proved that the average cwnd value $\overline{\text{cwnd}}$, (i.e. TCP congestion window average value expressed in packets) satisfies the condition:

$$\overline{\text{cwnd}} \cdot \text{PacketSize} \geq B \cdot \text{RTT} \tag{4.1}$$

Since traffic flows are unidirectional, the two edge routers act either as ingress or egress with respect to the DS terminology. There are 20 source nodes that enter the DS domain using 1 Mbps links; core links, whose line rate is 5 Mbps act as network bottleneck, where thus drops are likely to occur. Traffic is classified, conditioned and re-marked at the ingress edge by a Time Sliding Window Three Color Marker, whose configuration is based upon each source's SLS. Then, in the

core node, packets might be re-marked and receive different forwarding treatment basing on their DSCP; finally, the link capacity between the egress and the destination node is set to 10 Mbps and, since there is no restrictive SLA in place, packets are not likely to be further polished.

4.1.2 Performance Parameters

DS performance evaluation is a matter of importance in service definition: Assured Service has been the subject of recent discussion, but solid information on its applicability to the internet has been lacking. Firstly, a set of parameters that may be negotiated in a SLA definition must be individuated; then, in order to determine whether a SLA contract can provide quantitatively measurable assurance of any form, these parameter's behaviors must be analyzed in terms of network factors affecting (and possibly compromising) the ability to fulfill the contract.

In order to quantify and compare results we use both operator and user metric sets, representing different performance point of view: the overall service performance is a combination of these metrics:

- **Average Link Utilization**
- **Packet Loss Rate**
- **Average Throughput**

Using an AR-like service, the primary parameter used in both performance analysis and Service Level Specification part of the SLA is evidently the throughput: detailed considerations on this metric are further developed in Sec. 4.2. However, though an AR service can be specified uniquely in terms of the reserved bandwidth, drop probability may be among the SLA constitutive parameters in the context of a generic AF service. Furthermore, the operator can set buffer resources to ensure high link utilization; however high drop rates are undesired for supporting customers' interactive applications. Finally, drop results cannot be disregarded with respect to any consideration on RED effectiveness.

All the performance variables are collected per-connection (or per-flow in case of multiple micro flows originating from a single source node); their analysis will be conduced both per-connection and aggregated, reporting statistics such as mean, standard deviation and correlation with the parameters of study.

4.1.3 Impact of Network Scenario

TCP flow performance is influenced by many connection parameters, such as, for example link round trip time, packet size, number of flows within an aggregate, etc.; the effects of these parameters on Best Effort service are partly known: it would be interesting to test whether they are modified by use of DS. To the previous list, we add some DS specific parameters, either directly derived from the SLA or strictly tied to the DS model (e.g. needed in edge devices configuration): earlier study have been focused on such parameters, but there is no clear agreement on how they affect network performance. All the scenario parameters differently affect TCP behavior, mainly resulting in per-flow performances degradation and unfairness; the factors here investigated are:

Round Trip Time (RTT)
: Since TCP utilizes a self-clocked sliding window based mechanism, any bandwidth guarantee is a function of RTT: the smaller the RTT, the quicker a TCP source will recover from a loss; unless otherwise stated, total RTT equals 100 ms for all flows.

Packet Size
: Thanks to the packet quantization used in TCP cwnd algorithms, connections using bigger packets are advantaged with respect to those using smaller packets; by default packets are 1000 bytes.

Commited Information Rate (CIR)
The size of the target rate involves problems of compromised service delivering possibility as well as unfairness between flows with different contracted rates. Target rate will be deeply discussed in Sec. 4.2.

μ-Flows Number Within an Aggregate
Generally, aggregates composed by an higher number of flows opportunistically get performance improvement; by default μ-Flow=1.

Different simulation scenarios, in the following called "suites", are build starting from the topology earlier described: each suite focuses on the study of DS flow reaction to different values of specific scenario parameter, being the remaining factors identically set for any flow. For each different scenario, variation of the following parameters over a coarse set of values further differentiates and completes each simulation suite:

Peak Information Rate (PIR)
Used to tune the TSWTCM behavior, it can assume any value that satisfies CIR$\leq$PIR$\leq$B: simulation were conduced with PIR set to either the lower bound, or the upper one, or finally the optimal setting found earlier in Sec. 3.4.

Number of DS Connections
Due to considerations developed in next section, it will vary among $\{0, 5, 10, 15, 20\}$; the two extreme cases entail that network is crossed, respectively, by either BE or DS flows only, since the total number of DS and BE connections is constant and equal to 20.

Aggregate SLA
Percentage of the total aggregated DS bandwidth varies among $\{0\%, 25\%, 50\%, 75\%, 85\%, 95\%, 100\%\}$, where the lowest and highest values identify respectively BE or DS only traffic cases; coupled with DS connection number, this entails different cases of per-flow fractioning of the SLA.

The performance parameter described in previous section are analyzed in terms of the former impact factor; moreover, in order to give complete performance evaluation, the mutual influence between BE and DS forwarding treatments is also taken into account.

4.2 Throughput Performance Evaluation

In order to analyze the throughput performance, we need to define a few target-related parameters, and then to to expose some of the existing relationship between them: comparison of achieved throughput with these parameters, mainly in the form of a ratio of the former over each of the latter, will be intensively used in the analysis of the simulation results.

In the following, we assume that the SLA contracted by any connection is merely identified by connection's CIR; furthermore, these two acronyms may be used indistinctly whether this doesn't lead to ambiguity: that is, being CIR the only parameter explicitly used in the service level specification, we may refer to a SLA sum to indicate the sum of the corresponding CIRs.

Lets now define $\rho_{DS}\%$ as the bandwidth percentage reserved to DS flows and the corresponding normalized offered traffic ρ_{DS}; we will indicate with N_{DS} and N_{BE} the number of DS and BE active connections, with $N = N_{DS} + N_{BE} \equiv 20$. In every suite, apart from target rate one, the reserved bandwidth is equally shared between all the DS flows; indicating with Trg $_i$ the i-th flow's *Target Rate*, we have:

$$Trg_i(\rho_{DS}, N_{DS}) = \begin{cases} \rho_{DS} \cdot B/N_{DS} & \text{DS flow} \\ 0 & \text{BE flow} \end{cases} \tag{4.2}$$

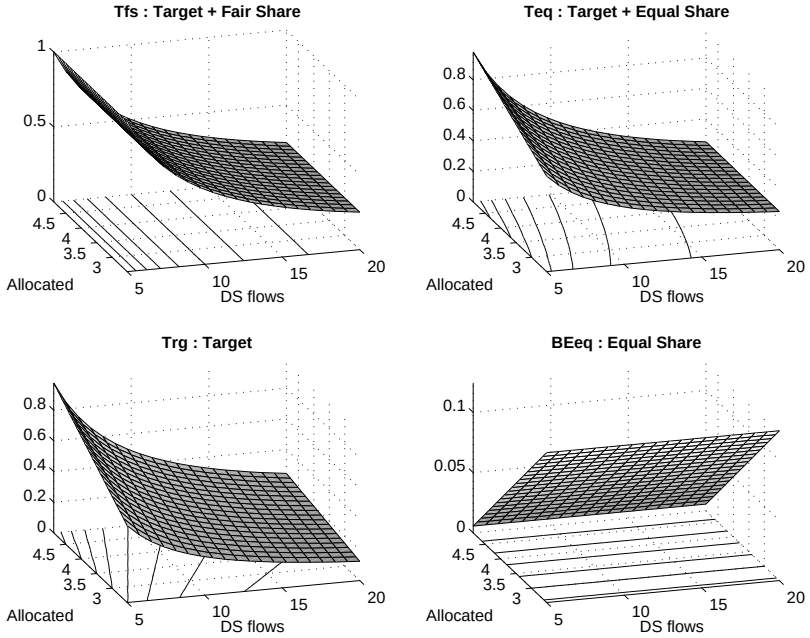


Figure 4.2: Tridimensional Plot of Target Parameters

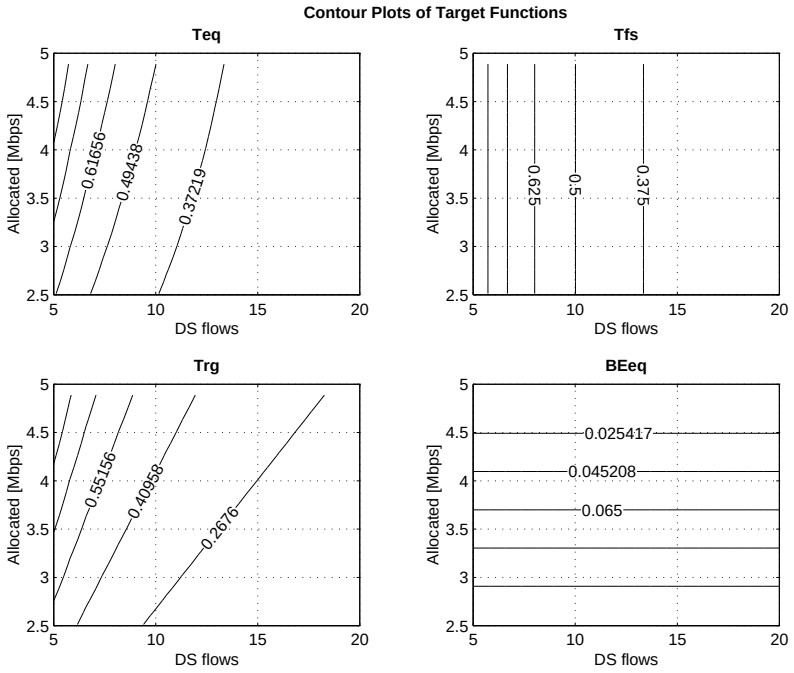


Figure 4.3: Contour Plot of Target Parameters

Eq. 4.2 states that BE flows have null target rate, and DS services bandwidth is equally distributed over all DS flows. Target rate is a key parameter in the sense that whether the achieved throughput is lower than Trg, the service assurance is evidently compromised: whether this case were not unlikely to occur, then it would be difficult to provide quantifiable SLS definition.

Though the goal of AS is to assure a minimum throughput to a connection, AS flows are enabled to get even better performance when the network conditions permits. In severe congestion, where there is not enough bandwidth to satisfy all the contracts, assured connections would compete against each other for the available bandwidth, possibly depriving BE connections of any bandwidth: this raise some concern about the fairness of the AS against BE service. In order to quantify this fairness, and also in the need to provide BE performance evaluations, a target-related parameter must be defined; being $B \cdot (1 - \rho_{DS})$ the bandwidth sharable by DS and BE flows, we can define the excess Equal Share (Eq) quantity as:

$$\mathrm{Eq}(\rho_{DS}, N) = B \cdot \frac{1 - \rho_{DS}}{N} \tag{4.3}$$

Eq. 4.3 defines the amount of excess bandwidth that each flow should get when the excess were equally distributed among all flows, including Best Effort ones; excess share is proportional to the total number of flow present but does not depend on N_{DS}. By adding the Eq rate amount to each flow's target Trg, we obtain the *Target Equal Share* (Teq) parameter, which can be explicitly expressed as:

$$Teq_i(\rho_{DS}, N_{DS}, N) = \begin{cases} B \cdot \left[\rho_{DS}/N_{DS} + (1 - \rho_{DS})/N\right] & \text{DS flow} \\ B \cdot (1 - \rho_{DS})/N \equiv \mathrm{Eq} & \text{BE flow} \end{cases} \tag{4.4}$$

However, depending on each specific context, different definition of fairness are possible: the former parameter, while allowing comparison of *DS versus BE* fairness, does not indeed represent a valid fairness parameter *among DS* flows; in fact, we may assume that a customer with an higher target rate considers "fair" to receive an higher share of the excess bandwidth. If this higher share is decided to be proportional to the purchased target, this is equivalent to attribute *all* the available bandwidth to the DS aggregate, since BE flows have zero target rate. Then, to analyze the DS fairness among flows with different values of the "suite" parameter, other than computing a fairness index for every simulation, we define the commonly used *Target Fair Share Rate* (Tfs) as follows:

$$Tfs_i(\rho_{DS}, N_{DS}, N) = Trg_i \cdot \left(1 + B \cdot \frac{1 - \rho_{DS}}{\sum_{j=1}^{N} Trg_j}\right) \tag{4.5}$$

Fig. 4.2 shows the plots of Trg, Eq, Teq and Tfs parameters as a function of both the AR aggregate reserved rate $B \cdot \rho_{DS}$ (expressed in Mbps) and the number of connection within the aggregate N_{DS} (as if it were a continue variable, even though it is not), while their contour plots are depicted in Fig. 4.3. The ratio of Throughput (Thr) over each of these parameters gives different normalized indications on network performances:

Thr/Trg : asserts either service assurance effectiveness (≥ 1) or failure in DS flow assured rate delivering (< 1)

Thr/Teq : used to quantify DS versus BE fairness, allows comparison of relative service sensitivity to factors affecting TCP behavior

Thr/Tfs : quantifies the relative fairness among DS flows, neglecting BE performance considerations

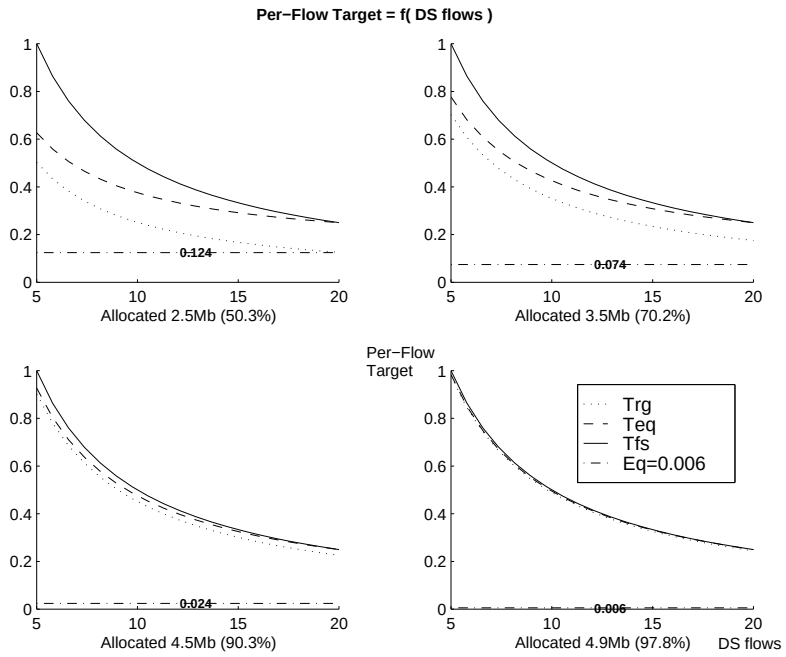


Figure 4.4: Target vs DS Flows Number

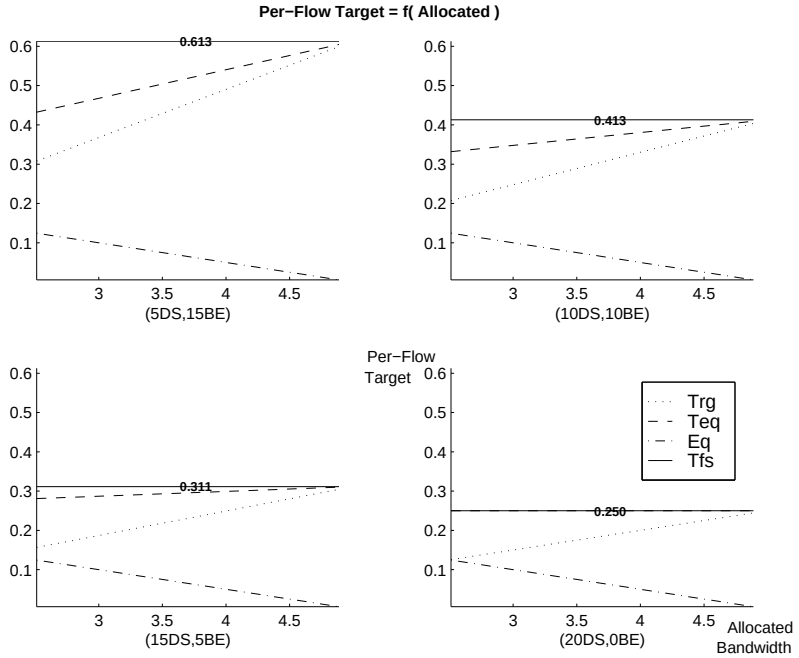


Figure 4.5: Target vs DS Reserved Bandwidth

Since each of these denominators achieve much different values for different N_{DS} and $\rho_{DS}\%$ values, we should describe further some of their properties and goals; from tridimensional parameter plot it might be gathered that Teq curve is lower bounded by Teq and upper bounded by Tfs; moreover, Eq (i.e. BE's Teq) and Tfs are orthogonal: Tfs doesn't depend on DS allocated bandwidth but only on DS flow number, whereas Eq doesn't depend on DS/BE flow ratio but only in DS+BE flow number. This can be thought as a different "strength" of the parameters, indicating that meaningful comparison should reflect a different quantitative interpretations of each ratio results.

A bidimensional representation of Teq, Trg, Tfs, Eq curves is given in Fig. 4.4 and Fig. 4.5: slice of the tridimensional curves are given, respectively, as functions of N_{DS} for a set of different $\rho_{DS} \cdot B$ values and as functions of $\rho_{DS} \cdot B$ for a set of different N_{DS} values. Notice that Teq and Tfs relative difference decreases as long as N_{DS} or ρ_{DS} increases. Moreover, there are two extreme cases of the target distribution among flows, where thus all the flows are serviced by either DS or BE forwarding treatment. Whether $N_{DS} = 20$ we have Teq≡Tfs, since there would be no reason to allocate less than the whole bottleneck to AR service; it must be noticed that this case is artificial, in the sense that, being all the available bandwidth allocated to AF service, only a few connection reach their target rate.

Conversely, when $N_{DS} = 0$, network is crossed uniquely by BE flows: in this case, performance fairness entails bandwidth to be equally shared among BE flows, whose throughput should achieve Eq=B/20, while Tfs=Trg=0. These two extreme cases are however useful, in order to compare each service sensitivity to a specific network factor, providing an immediate idea of factor's strength in performance affection.

Whether the contemporary use of Thr/Teq and Thr/Tfs ratios as measure of fairness seems contradictory, further considerations are needed to prove that is not. Fairness among DS flows does not need Thr/Tfs to be equal to 1 for all flows, since this would entail total BE starvation; moreover it should not be used as an *absolute* performance meter: whether the contracted SLA is much lower than the bottleneck, Thr/Tfs will reach values much lower than 1 even though the achieved throughput is above the flow target. Rather, it should be used as a *relative* performance meter: whether two flows reach an equal value of Thr/Tfs no matter their Trg, then these flow performances is fair with respect to each flow Trg.

TARGET		DiffServ Allocated Bandwidth (Mbps)					
PARAMS		25%	50%	65%	75%	85%	95%
5 DS, 15 BE	Trg	0.250	0.500	0.650	0.750	0.850	0.950
	Teq	0.438	0.625	0.738	0.812	0.887	0.962
	Tfs †	1.000					
10 DS, 10 BE	Trg	0.125	0.250	0.325	0.375	0.425	0.475
	Teq	0.312	0.375	0.412	0.438	0.462	0.487
	Tfs †	0.500					
15 DS, 5 BE	Trg	0.083	0.167	0.217	0.250	0.283	0.317
	Teq	0.271	0.292	0.304	0.312	0.321	0.329
	Tfs †	0.333					
	Eq ††	0.188	0.125	0.087	0.062	0.037	0.013

† *Tfs doesn't depend on DS allocated bandwidth but only on DS flow number*

†† *Eq doesn't depend on DS/BE flow ratio but only in DS+BE flow number*

Table 4.1: Target Performances Parameters

The number N_{DS} of connection will be evidently an integer, and will be furthermore less or

equal to the total flow number N, thus $N_{DS} \in [0, N] \subset \mathbb{N}$, while $\rho_{DS} \in [0, 1] \subset \mathbb{R}$. Simulation were run only for a few of the possible (N_{DS},ρ_{DS}%) couples, chosen for their peculiar properties: it can be noticed that when (N_{DS},ρ_{DS}%)$\in$ $(5, 25\%), (10, 50\%), (15, 75\%)$, whether the reserved bandwidth is equally shared among DS flows, then every flow will have the same Trg=250 Kbps, while Teq and Tfs will be different in the three cases. This is an interesting subset of the (N_{DS},ρ_{DS}%) space that will be analyzed by simulation; however, a significantly larger set of (N_{DS},ρ_{DS}%) couples will be investigated in order to determine how these two factors can further affect or mitigate the effect of each analyzed factor.

The complete simulation setup is shown in Tab. 4.1, reporting flow's target and target related parameter earlier analyzed; this will be used as reference when reporting the measured throughput and its performance measures Thr/Trg, Thr/Teq, Thr/Tfs. It should be recalled that the above values are used by all the simulation suite apart from Target Rate suite, where the reserved bandwidth is uniformly distributed among flows rather than equally shared; further details will be given describing target suite in Sec. 5.4.

4.3 Notation Definition

All the simulations were performed using a common set of scenario parameters, and efforts have been made in order to use a coherent, common scheme in the presentation of the results: the set of naming rules described below have been defined to the help of the reader.

Suite Parameter

Each simulation suite examines how performance are affected from parameter X, being others parameters equally set for any flow. Since each flow pair will differ only on X factor value, the notation Φ_x will be used to indicate "the flow(s) whose parameter X has x value". Moreover, we may order Φ_x and Φ_y according to the respective parameter values, thus Φ_x $>\Phi_y$ whether $x > y$; extending the analogy we may write "the lowest Φ" or "the highest Φ" of a set according to numerical order of parameter value within the set; similarly, when describing performances of a set of flows whose parameter value satisfy $x < y < z$, Φ_y will be the "intermediate Φ" of that set.

Simulation Identification

Each simulation has a different N_{DS} number of AR connections to which a different bottleneck percentage ρ_{DS}% is assigned; we use the notation n@b% where $n =N_{DS}$ and $b =\rho_{DS}$%. With present knowledge, we may say thus that target rate assigned to flows in any of 5@25%, 10@50%, 15@75% is Trg=250 Kbps. Furthermore, simulation results are grouped as shown in according to the set properties: for example, CIR group's DS flows have equal Trg, ALLOC group's DS aggregates are reserved half of the line rate, etc. This naming notation is defined in the need to fraction the results exposition, allowing to focus separately on different influence of Trg, ρ_{DS}%, N_{DS} parameters on performance analysis.

CIR	5@25%	10@50%	15@75%
ALLOC	5@50%	10@50%	15@50%
HIGH-a	5@95%	10@95%	15@95%
FLOW	10@25%	10@50%	10@75%
HIGH-f	15@75%	15@85%	15@95%
DS Only	20@100%		
BE Only	0@0%		

Table 4.2: Simulation Group Naming Notation

Table Properties

Each table reports bandwidth and drop performance results, grouped according to the pre-

viously defined simulation groups, both per-flow and aggregated: in the former case, flow samples are chosen according to their suite parameter X values, while the DS aggregate measures are reported as average (μ) and standard deviation (σ). Finally, the BE and DS sum (Σ) gives useful indications on bottleneck performances: the most significant are the link's throughput (Thr) evaluated over all the sources, and its value normalized over the line rate (Thr/Tfs); to be noticed that the Thr/Tfs parameter reported in column Σ represents therefore the overall link utilization η: its value is *emphasized* in tables to the ease of the reader.

Figure Properties

A figure may be composed of several individual subplots, each of which reports one (or more) n@b% simulation case(s): whether both the x and y axes are the same for any subplot, then the corresponding labels will be reported

Comparison Notation

In the need to compare the fairness results obtained for different simulation scenarios in a simple fashion involving the least possible number of parameters, we define a Thr/Tfs coefficient of variation basing on the Thr/Tfs σ and μ statistics computed for each different scenario.

Thr/Tfs may actually vary in a wide range but, as previously introduced, we are interested in observing its values on a relative scale rather than an absolute one; this allow us to build a rough per-simulation unfairness parameter by considering its DS aggregate average μ and standard deviation σ. Firstly, we should consider that unfairness is evidently more consistent as long as σ increases being μ fixed. However, the unfairness test needs to be performed among different simulations, each of which can be described in terms of its $i = (N_{DS}, \rho_{DS}\%)$ value pair: μ_i actually fluctuates depending on i values, preventing a direct $\sigma_i \gtrless \sigma_j$ comparison. Rather, the value of σ normalized over μ gives indications on parameter x distribution width that can be directly used in comparison between simulations.

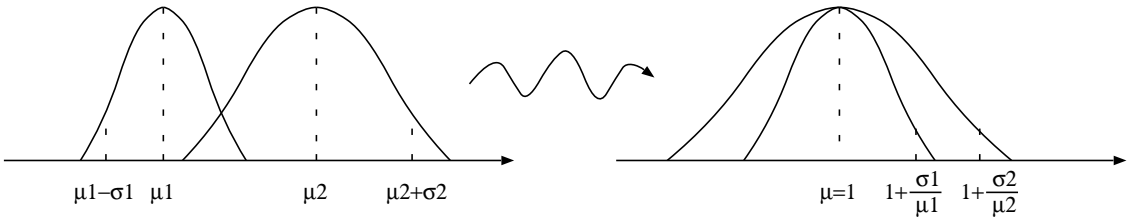


Figure 4.6: Normalization Scheme For Variable Comparison

Fig. 4.6 depicts a scheme of the normalization process where, for the ease of the representation, the two variables are assumed to be normally distributed, thus $x_i = N(\mu_i, \sigma_i)$. Basing on these considerations, the definition of the unfairness index $\mathcal{U}$ for a simulation n@b% is straightforward:

$$\mathcal{U}(N_{DS}, \rho_{DS}\%) = \frac{\sigma(Thr/Tfs)}{\mu(Thr/Tfs)}(N_{DS}, \rho_{DS}\%) \tag{4.6}$$

$\mathcal{U}$, which can be expressed as percentage, represents a rough but immediate evaluation of the unfairness among flows, allowing further to see how the SLA tuning can mitigate TCP dependency on the factors described in Sec. 4.1.3

Chapter 5

Long Lived Flows Simulation Results

This chapter analizes the simulation results of FTP endless flows crossing a DiffServ domain that offers both Best Effor and Assured Rate services. Network performances are examined in terms of the key factors that affect TCP performances: link's round trip time (Sec. 5.1), application's packet size (Sec. 5.2), number of flows within a connection (Sec. 5.3) and flow's target rate size (Sec. 5.4); considerations on active queue management effectiveness are the topic of Sec. 5.7. Then, although differences of DS versus BE services are discussed for each of the previous factors, Sec. 5.5 futher develops and completes the analysis, while the effects on network performances of non-responsive traffic introduction are developed in Sec. 5.6. Finally, Sec. 5.8 concludes this part of the analysis of DiffServ framework, focusing on merits, faults and feasibility of the architecture.

5.1 Round Trip Time Effects

TCP performance is wellknown to be sensitive to a connection's Round Trip Time (RTT): since TCP utilizes a self-clocked sliding window mechanism, any bandwidth guarantee is a function of RTT. The performance difference is entailed from the fact that, with respect to small RTT flows, flows with larger RTT need more time to recover after a packet loss: as a consequence, small RTT connections return to their former cwnd value quicker than bigger RTT ones.

Different RTTs values are distributed among flows: for the generic connection i, the link transmission delay follows $RTT_i = j \cdot 10$ ms where $j \in \{1, 2, \ldots, 10\} \subset \mathbb{N}$. tables report per-flow results for the $j \in \{1, 5, 10\}$ subset, thus for flows whose RTT is either 10 ms or 50 ms or even 100 ms: according the introduced notation, Φ_{10} Φ_{50} Φ_{100} are respectively the smallest, the intermediate and the highest Φ.

A wide range of simulations performed shown that, despite having identical target rates, flows with different RTTs will get different shares of the bandwidth; moreover, depending on the specific SLA applied, it is possible that flows with higher RTT do not even reach their target. The results show also that throughput dependency on RTT is lessened by DS deployment; however, optimizing the network performance by properly configuring the SLA is not straightforward even in the case of a simple network topology.

In the need to examine whether and how RTT effects on DS flows performances are affected by the SLA variation, we can firstly compare a touple of simulations where N_{DS} and ρ_{DS}% are chosen in a way such that DS flows have identical target rate (Trg=250 Kbps) but different Teq, and Tfs decreasing as long as N_{DS} grows, as reported in Tab. 4.1. Tab. 5.1 reports the achieved Thr, Thr/Trg and Thr/Tfs for Φ_{10}, Φ_{50} and Φ_{100}, the DS aggregate average μ and standard deviation σ and the overall link performance including BE traffic ones.

RTT @ CIR		Round Trip Time			DS Overall		Link
		10 ms	50 ms	100 ms	μ	σ	Σ
5 @ 25 %	Thr	0.531	0.332	0.269	0.362	0.095	4.126
	Thr/Trg	2.124	1.327	1.078	1.448	0.378	3.301
	Thr/Tfs	0.531	0.332	0.269	0.362	0.095	*0.825*
10 @ 50 %	Thr	0.454	0.321	0.256	0.323	0.060	4.339
	Thr/Trg	1.814	1.286	1.025	1.293	0.240	1.736
	Thr/Tfs	0.907	0.643	0.512	0.647	0.120	*0.868*
15 @ 75 %	Thr	0.367	0.285	0.241	0.286	0.036	4.573
	Thr/Trg	1.469	1.141	0.962	1.146	0.144	1.219
	Thr/Tfs	1.102	0.856	0.722	0.859	0.108	*0.915*

Table 5.1: RTT: DS Performance, Trg = 250 kbps

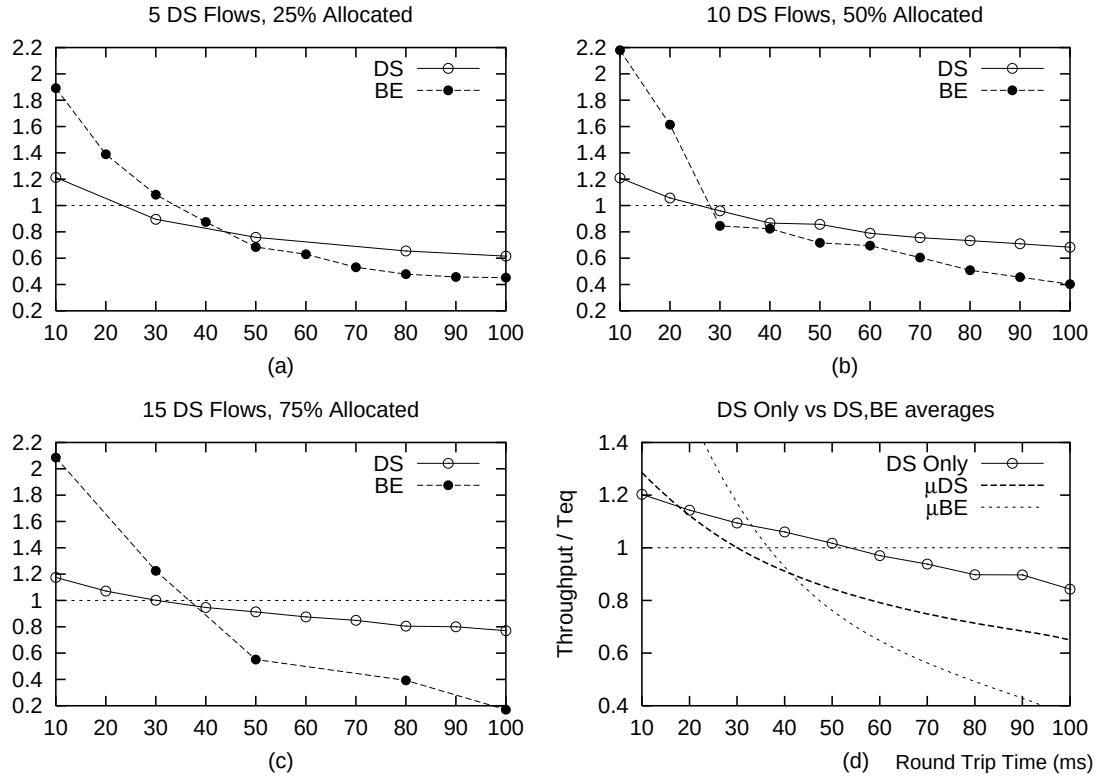


Figure 5.1: RTT: Same Target Rate

At a first glance, it can be seen that both fairness among flows and the link utilization η benefits of the contemporary increase of both N_{DS} and ρ_{DS}%. Specifically at 5@25%, Tfs coincides with each connection's capacity and it is four times bigger than Trg: we can expect that the achieved Thr/Tfs values are well below the unity. Specifically, Thr/Tfs for lowest Φ is the double of highest Φ one and the unfairness reaches $\mathcal{U} = 26\%$; furthermore, only the lowest Φ reaches the Teq threshold, even though the contracted SLA is assured for any flow independently from the connection's RTT; this latter event, coupled with the fact that the majority of the connections use BE service, prevent an utilization of the link higher than $\eta = 82.5\%$. Conversely at 15@75%, the significant DS traffic amount is reflected in an improved link utilization ($\eta = 91.5\%$) and in a smaller throughput variation depending on RTT values ($\mathcal{U} = 12\%$); however, there is a more important remark: large RTT connections *do not reach their target* (actually Φ_{100} reach the 96%), whereas Φ_{50}, although achieving 14% more than their target, obtains only the 85% of the Tfs, and finally Φ_{10} obtains 10% of what considered to be fair.

These tree simulation cases are depicted in Fig. 5.1(a,b,c), plotting the Thr/Teq ratio on the y axis as a function of connections RTT on the x axis, for both BE and DS flows: this lets us examine the effects of DS service on BE connections. Earlier study [AFEVAL] claimed that "having a significant amount of assured traffic not only lowers the dependency on the RTT for AS connections, but also for besteffort connections"; this was explained with the fact that BE "connections with a small RTT send more packets than those with a large RTT, thus having more chances to undergo a packet drop in a certain time interval".

Examining Fig. 5.1, we gather in fact the opposite conclusion: AR service lowers the dependency on the RTT for DS connections, whereas it highers those of BE connections. To understand the reason behind this phenomenon we should observe the per-flow drop probability in both BE and DS traffic cases, reported in Tab. 5.2.

RTT @ CIR		5 @ 25 %		10 @ 50 %		15 @ 75 %	
		DS	BE	DS	BE	DS	BE
RTT=10 ms	Thr	0.531	0.355	0.454	0.272	0.367	0.130
	Drops	4.92%	8.27%	5.81%	10.32%	7.13%	14.66%
	Thr/Trg	2.124	-	1.814	-	1.469	-
	Thr/Teq	1.214	1.891	1.210	2.180	1.175	2.086
RTT=50 ms	Thr	0.332	0.128	0.321	0.090	0.285	0.034
	Drops	2.95%	10.36%	3.82%	15.20%	4.41%	25.08%
	Thr/Trg	1.327	-	1.286	-	1.141	-
	Thr/Teq	0.758	0.684	0.857	0.716	0.913	0.551
RTT=100 ms	Thr	0.269	0.085	0.256	0.050	0.241	0.011
	Drops	1.70%	10.27%	2.11%	17.27%	2.65%	31.42%
	Thr/Trg	1.078	-	1.025	-	0.962	-
	Thr/Teq	0.616	0.453	0.683	0.403	0.770	0.171

Table 5.2: RTT: DS vs BE Per-RTT Performance

Per-flow drop percentages grows, in both BE and DS cases, as long as N_{DS} and ρ_{DS}% increase; however, the overall drop percentage decrease, since when DS flows represent the majority of the connections, their drop percentage is less than those of DS in the reverse situation. There is a major difference is DS and BE per-flow drop probabilities, apart from their amount: DS drops are more likely to occur for lowest Φ, while the opposite happens for BE service.

RTT @ HIGH-f		Round Trip Time			DS Overall		Link
		10 ms	50 ms	100 ms	μ	σ	Σ
15 @ 75 %	Thr	0.367	0.285	0.241	0.286	0.036	4.573
	Thr/Trg	1.469	1.141	0.962	1.146	0.144	1.219
	Thr/Tfs	1.102	0.856	0.722	0.859	0.108	*0.915*
15 @ 85 %	Thr	0.372	0.296	0.257	0.301	0.034	4.652
	Thr/Trg	1.314	1.045	0.907	1.063	0.119	1.095
	Thr/Tfs	1.117	0.888	0.771	0.903	0.101	*0.930*
15 @ 95 %	Thr	0.365	0.315	0.274	0.313	0.029	4.771
	Thr/Trg	1.152	0.994	0.866	0.989	0.092	1.004
	Thr/Tfs	1.095	0.944	0.823	0.940	0.087	*0.954*

Table 5.3: RTT: DS Performance at High Reserved Bandwidth, $N_{DS} = 15$

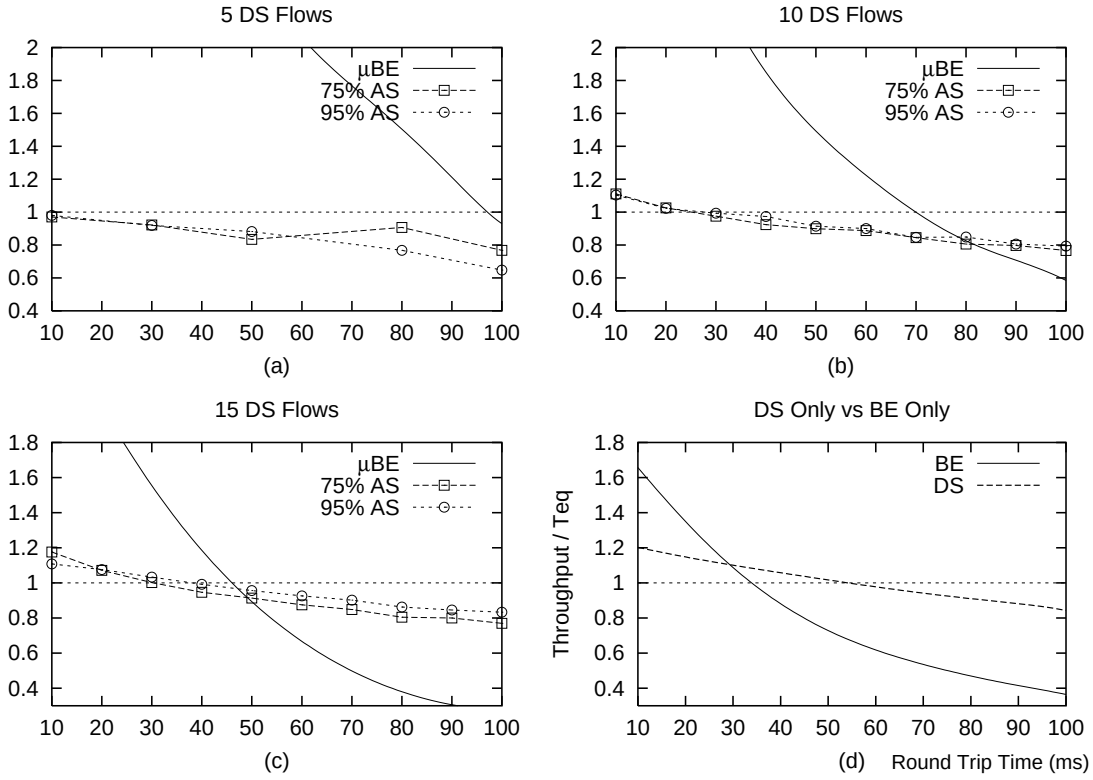


Figure 5.2: RTT: High Allocation

DS drop behavior can be explained as follows: a connection with a small RTT gets more bandwidth by opportunistically exceeding its target rate, thus sending out-of-profile packets that will be marked with an higher drop precedence; since many of those packets will be dropped, the connection will decrease its sending rate; furthermore, the larger number of red packets a DS source sends, the higher the probability that one or more packets will be dropped within a certain time interval: that has the effect of mitigating the gain of connections with a smaller RTT. BE connections send out packets with one uniquely possible DSCP, this entailing that connection advantaged from a small RTT will send opportunistically more packets without any possibility for AQM to penalize that source; since small RTT flows take less time to recover after a loss, these sources are more able to fill the BE queue than longer RTT ones: this, coupled with the narrowness of the BE queue, entails that packets sent from longer RTT flows are more likely to find a full queue and thus to incur in late drop action.

The problems of service guarantees, optimal link utilization and fairness among flows observed in the first set of simulation let us suppose that the choice of a SLA that globally optimize network performance is not a simple task. Moreover, analysis of the results issued from a wider range of simulations leads to the conclusion that such problems cannot be solved together, at least with the deployed AR scheme.

For example, when half of the bottleneck is is reserved to DS flows, the link utilization will be 86% independently from the number N_{DS} of AR flows; however, fairness among flows and service assurance cannot be optimized at the same time, e.g.:

- when $N_{DS} = 5$, Φ_{100} reaches only the 86% of its target but the ratio of Φ_{10}'s Thr/Tfs over Φ_{100} one is less than 1.5, and $\mathcal{U} = 15\%$
- when $N_{DS} = 15$, Φ_{100} achieves 16% more than Trg but Φ_{10}'s Thr/Tfs is nearly the double of Φ_{100} one, and $\mathcal{U} = 21\%$

Whereas system performance dependence on $\rho_{DS}\%$ is not straightforward, we can say that the overall link utilization η benefit of a consistent amount of DS traffic, increasing as long $\rho_{DS}\%$ increases: this is shown in Tab. 5.3 when $N_{DS} = 15$ and the percentage of the reserved bandwidth is either 75%, 85% or 95%. The achieved throughput data reported in the same table shows that in no case the longest flows achieve their target rate, and furthermore that the service is more degraded at higher SLA; as side effect, the unfairness decreases from 12% to 9% as long as $\rho_{DS}\%$ grows from $\rho_{DS} = 75\%$ to $\rho_{DS} = 95\%$. The cases depicted in Fig. 5.2(a,b,c) refer to a high SLA involving a different number of DS connections: although in case (a), where thus $N_{DS} = 5$, the achieved link utilization $\eta = 97\%$ is the best ever, the unfairness is higher than in cases (b) and (c), and furthermore Φ_{100} achieves only the 65% of their Trg. Conversely, when 10 flows are reserved the 25% of the link, its performance is worse and less fair ($\mathcal{U} = 30\%$) than when the same bandwidth amount is reserved to 5 AR flows only. These are evident case of misconfigured system, showing that a proper SLA settings cannot be decoupled from the DS flow number in order to be effective; this might obviously be possible in a real network situation, where the number of connections cannot be exactly determined a priori, and where moreover the actual number of active flows varies with the time.

In order to examine the effects of RTT on TCP performance without having to consider the mutual interactions of BE and DS services, we may consider the two extreme cases where only either DS or BE traffic is flowing on the DS domain. DS flows only case and the opposite BE only one are depicted in Fig. 5.2(d), where it can be gathered the smaller effect of DS on TCP dependency on RTT: where BE and DS trend lines correspond respectively to a linear and an exponential regression and are included to give a rough idea of the different way achieved rate varies with the RTT. However, the DS extreme case is artificial: since the SLA represents the totality of the bottleneck's capacity, DS flows are forced to compete against each other to achieve their target: this entails that only a few flows reach their target.

Finally, Fig. 5.3 depicts on the ordinate axis the throughput achieved rather than its Thr/Teq ratio as a function of the RTT; in order to enhance readability, the figure plots also the Trg and Teq thresholds: since all flows have the same target, thresholds are the same for any flows within a simulation but different depending on both N_{DS} and ρ_{DS}%. Case (a) considers simulations where flows have the same target rate, (b) varies the number of flows to which half of the bottleneck is reserved, (c) varies the bandwidth allocated when DS flows represent half of the total connection number and (d) compares the performances achieved, on average, by DS flows to those achieved in a network solely crossed by DS flows.

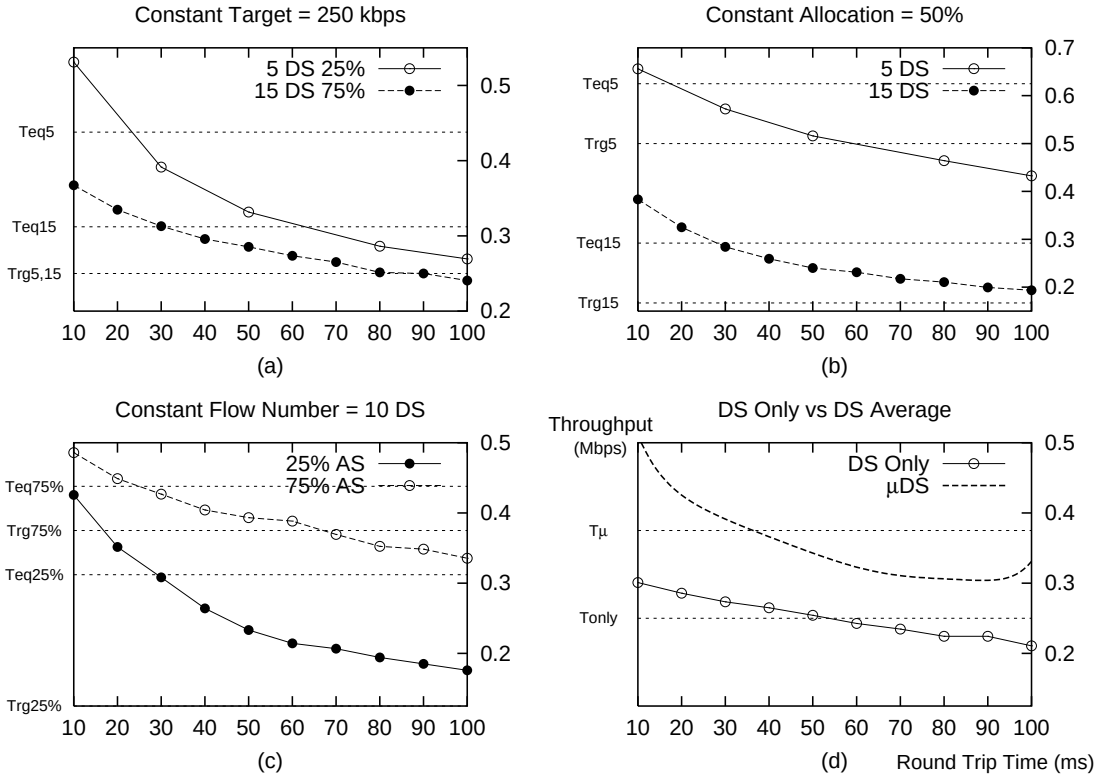


Figure 5.3: RTT: Achieved Throughput Compared to Different Target Thresholds

It can be gathered from (b) that the slope of Thr at a given SLA does not depends on the number of DS flows: this is also evident from comparison of 5@25% in (a) and 10@25% in (c) as well as 15@25% in (a) and 10@75% in (c). Moreover, the number of DS flows steams a consistent service degradation whether few flows are reserved a too large bandwidth (5@25% in (a) and 10@75% in (c)); however, assurance guarantees may also be compromised in situation with improved fairness and otherwise satisfying results (15@75% in (a)). Comparison of two curves in (d) shows that the DS only case, even though service degradation cannot be avoided for certain flows, may anyway be considered a "better" case with respect to performance achieved on average by DS flows.

5.2 Packet Size Effects

Packet size influence on TCP behavior, mainly concerning unfairness in the share of the excess bandwidth, has been noticed in various simulation study. All the analysis agree in that, despite having identical target rate, flows with different packet size will get different share of the excess bandwidth; specifically, connections using bigger packet sizes will profit more of the excess bandwidth with respect to flows sending smaller packets. However, this apparently counter-intuitive result has not been given a possible explanation yet, since justification of the phenomenon is not as immediate as for TCP dependence on RTT. Throughput performances depend on several factors: however we may consider that, being store-and-forward used in the network, propagation time is a key of importance in the achieved result. When the propagation time of the connection is of the same order of the transmission time, increasing the packet size increases the time delay between consecutive ACKs on the backward path. Since TCP's window size at the source is related to the ACK arrival frequency, increasing the packet size would eventually entail a reduced ability for flows in opening their windows. However, whether the transmission time is less than the propagation delay, connections sending bigger packets may achieve an higher throughput, measured in bps, under the same cwnd: this is due to a more advantageous fractioning of data/header per packet.

Lets indicate with Φ_x flows sending packets whose size is x bytes, where $x \in \{256, 512, 768, 1024\}$; although considering constants sizes during transmission is a simplification, this is done for the ease of the analysis, in order to provide a detailed performance comparison over a coarse set of packet sizes. The analysis of simulation results confirms that unfairness disadvantages flows sending smaller packets, and, more critical, shows that shortest packets flows (i.e. Φ_{256}) are likely not to meet their target under certain circumstances.

Tab. 5.4 reports the results of simulations where a constant number of DS flows $N_{DS} = 10$ is reserved a different percentage of the bottleneck link bandwidth: $\rho_{DS}\% \in \{25\%, 50\%, 75\%\}$. At a first glance, it can be seen that incrementing the reserved bandwidth amount produces higher link utilization: $\eta_{10@25\%} = 77\%$ and $\eta_{10@75\%} = 89\%$; at the same time, the coefficient of variation of Thr/Tfs decreases as long as $\rho_{DS}\%$ increases, resulting in an incremented fairness level: $\mathcal{U} = 24\%$ at 10@25% whereas $\mathcal{U} = 8\%$ at 10@75%. Furthermore, service assurance is never compromised, at least for Φ_{512} Φ_{768} Φ_{1024}, and the gain of Φ_{1024} over Φ_{512} is resized from 39% (10@25%) to 12% (10@75%). Since a consistent amount of DS traffic reduces the unfairness among flows with different packet sizes, providing at the same time a better overall link utilization, we may be tempted to consider that, at least for DS flows, performances benefit of the $\rho_{DS}\%$ increase; however, shortest flow performance faces a critical situation when the aggregate SLA increase, as we will show.

The throughput behavior for flows sending packets of different sizes is depicted in Fig. 5.4, where an equal number of DS and BE flows share the bottleneck ($N_{DS} = 10$); the curves report on the ordinate the throughput achieved by Φ_{256} and Φ_{1024} as a function of time, sampled in a Montecarlo fashion, and the Trg, Teq and Eq thresholds are shown for comparison purposes. Case (a) of the figure reports the results of 10@25% simulation: we gather that both DS and BE Φ_{1024} achieve at least the respective target plus equal share, that is Teq and Eq; DS Φ_{256} reaches more than its target rate but, being prevented from higher Φ to profits of the excess bandwidth, does not achieve Teq; similarly, shortest BE Φ_{256} achieves less of Eq, and specifically less of Φ_{1024}'s throughput half. The situation for Φ_{256} becomes more critical as long as $\rho_{DS}\%$ increases: when half of the bottleneck is reserved to 10 DS flows as in case (b), DS Φ_{256} and Φ_{1024} achieve respectively Trg and Teq: shortest DS flows does not even participate to the excess share. Whether the $\rho_{DS}\%$ amount grows over 50% as in cases (c) and (d), DS Φ_{1024} still reaches Teq rate threshold; furthermore, throughput of Φ_{512} and Φ_{768} roughly reach the 90% of Teq when $\rho_{DS} = 75\%$ and the 95% of it when $\rho_{DS} = 85\%$.

SIZE @ FLOW		Packet Size			DS Overall		Link
		512 byte	768 byte	1024 byte	μ	σ	Σ
10 @ 25 %	Thr	0.230	0.283	0.314	0.243	0.058	3.859
	Thr/Trg	1.844	2.267	2.508	1.945	0.465	3.087
	Thr/Tfs	0.461	0.567	0.627	0.486	0.116	*0.772*
10 @ 50 %	Thr	0.305	0.331	0.375	0.315	0.043	4.145
	Thr/Trg	1.219	1.323	1.500	1.260	0.171	1.658
	Thr/Tfs	0.610	0.661	0.750	0.630	0.085	*0.829*
10 @ 75 %	Thr	0.390	0.423	0.437	0.397	0.033	4.460
	Thr/Trg	1.041	1.127	1.166	1.060	0.088	1.189
	Thr/Tfs	0.780	0.845	0.874	0.795	0.066	*0.892*

Table 5.4: Packet Size: DS Performance, $N_{DS} = 10$

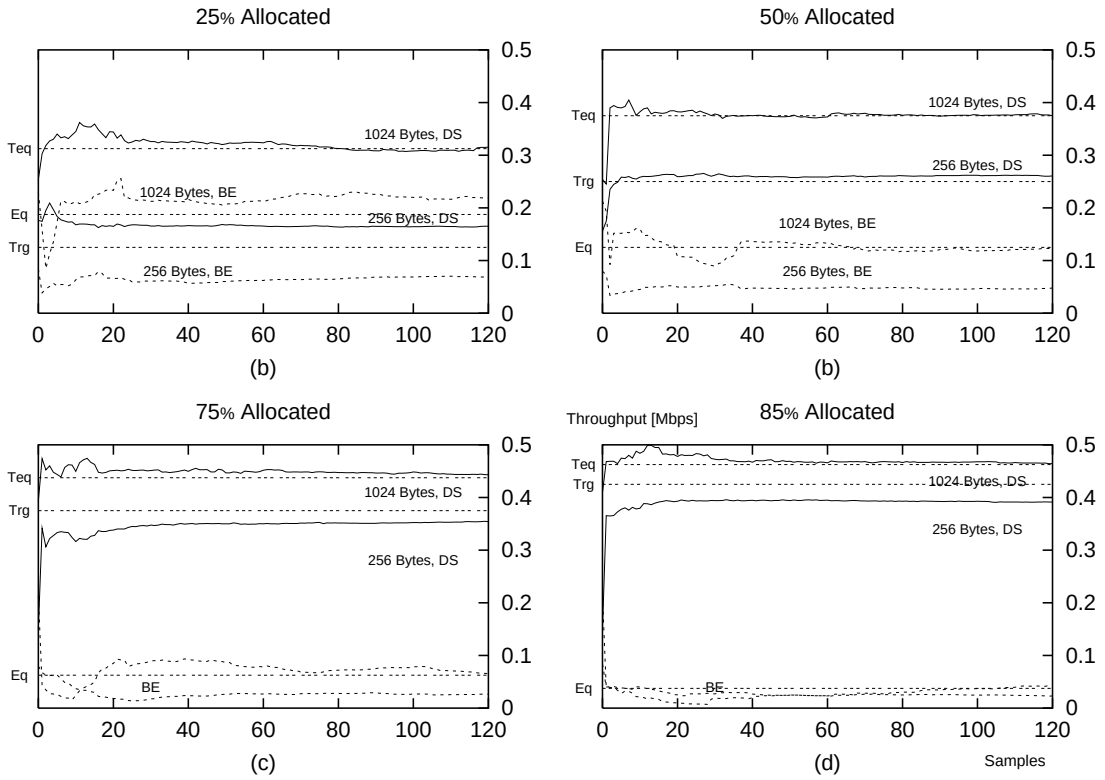


Figure 5.4: Packet Size: Same Flow Number

SIZE @ ALLOC		Packet Size			DS Overall		Link
		512 byte	768 byte	1024 byte	μ	σ	Σ
5 @ 50 %	Thr	0.521	0.566	0.593	0.536	0.040	4.237
	Thr/Trg	1.043	1.131	1.186	1.072	0.079	1.695
	Thr/Tfs	0.521	0.566	0.593	0.536	0.040	*0.847*
10 @ 50 %	Thr	0.305	0.331	0.375	0.315	0.043	4.145
	Thr/Trg	1.219	1.323	1.500	1.260	0.171	1.658
	Thr/Tfs	0.610	0.661	0.750	0.630	0.085	*0.829*
15 @ 50 %	Thr	0.237	0.264	0.286	0.247	0.040	4.129
	Thr/Trg	1.423	1.586	1.717	1.480	0.238	1.652
	Thr/Tfs	0.711	0.793	0.859	0.740	0.119	*0.826*

Table 5.5: Packet Size: DS Performance, $\rho_{DS} = 50\%$

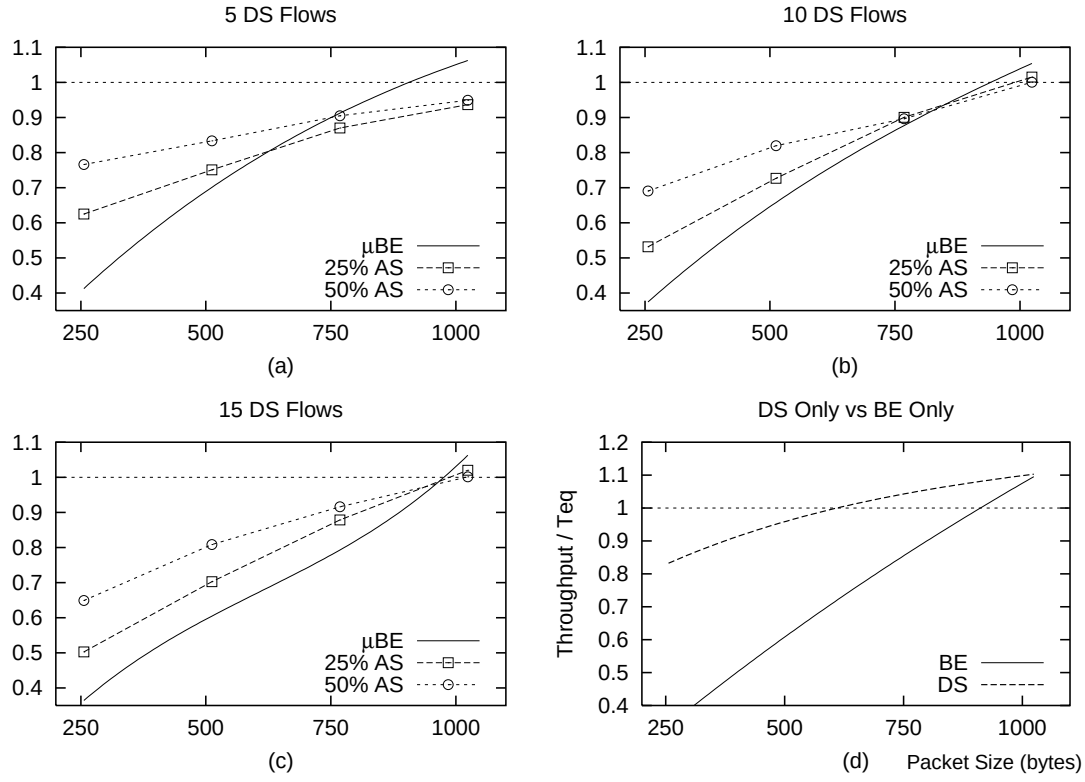


Figure 5.5: Packet Size: Low Allocation

However, the performance improvement for flows sending bigger packets is not without cost : in latter cases, the service cannot provide adequate guarantees for lowest Φ_{256}, whose throughput achieves ∼5% less of Trg. similarly, BE Φ_{1024} obtains better performances than Φ_{256}, achieving in most cases the Eq rate threshold; however, DS flow's throughput is more stable than BE flow's one, in the sense that the latter fluctuates even when network is fully operating: in this order of ideas we may further specify that Φ_{1024} are always able to reach, on average, the Eq rate.

Continuing our analysis, we may expect that varying the number of connections to which a fixed amount of bandwidth is reserved can steam some fairness concerns: especially whether the contracted SLA is narrow with respect to flow number, as long as N_{DS} increases a larger number of big packet Φ will has more opportunity to prevent a larger number of smaller packet Φ to profit of the excess bandwidth; furthermore, as earlier noticed Φ_{256} could even not reach their target. Conversely, N_{DS} increase at a given ρ_{DS}% should not consistently affect the link utilization: being η an aggregate value, any degradation for small packet Φ will be offset by improvement of bigger packets Φ performances.

Simulation results confirm the supposition: Tab. 5.4(a,b,c) reports the result of simulation where half of the bottleneck is reserved to either 5 or 10 or 15 DS flows. As long as N_{DS} increases, the link utilization η slightly (∼2%) decreases: this can be interpreted in reason of the incremented aggressiveness of the increased number of flows competing for a relatively narrower bandwidth. Throughput standard deviation is substantially the same independently from N_{DS}, whereas its average halves between $N_{DS} = 5$ and $N_{DS} = 15$: this entails an increased unfairness among flows, doubling from 8% to 16% in the same N_{DS} range. The considerations developed for $\rho_{DS} = 50\%$ hold whether a lower SLA is applied to the aggregate; furthermore, the described effects are even more noticeable. Fig. 5.5 depicts on the ordinate the achieved Thr/Teq as function of packet size; either a quarter or a half of the available line rate is reserved to $N_{DS} = 5$, $N_{DS} = 10$ and $N_{DS} = 15$ flows in (a), (b) and (c) case respectively. As long as N_{DS} increases Φ_{1024} gain on other Φ grows, achieving Teq whereas others Φ do not; furthermore, the slope of DS Thr/Teq becomes similar to BE Thr/Eq: this means that in case (c) packet size effects are no longer lessened and that the minimum level of lowest Φ protection is no longer effective.

SIZE @ DS vs BE		SIZE			Overall		
		512 byte	768 byte	1024 byte	μ	σ	Σ
BE Only	Thr	0.163	0.213	0.281	0.185	0.071	3.693
	Drops	8.83%	9.25%	9.37%	8.96%	0.47%	9.10%
	Thr/Teq	0.651	0.853	1.123	0.739	0.284	0.739
DS Only	Thr	0.249	0.263	0.275	0.249	0.026	4.980
	Drops	3.90%	6.00%	8.26%	4.99%	2.29%	5.22%
	Thr/Teq†	0.997	1.053	1.101	0.996	0.102	0.996

† *Teq≡Trg being all the linerate allocated to DiffServ flows*

Table 5.6: Packet Size: DS-Only vs BE-Only Performance

TCP flows using DiffServ are however less sensible than those serviced as BE to the packet size factor: this can be gathered from observation of Fig. 5.5(d), which plots the achieved Thr/Teq on the ordinate axis as a function of the packet size when network is offering either DS or BE service only. The small throughput dependence on packed size derives mainly from the introduction of different levels of drop precedence; Tab. 5.6 reports the performance achieved in both DS and BE only traffic cases: DS Φ_{512} drops are less than the half of Φ_{1024} drops, whereas the drop probability is substantially the same independently from packet size in BE service case. As a consequence, throughput achieved by Φ_{1024} is 1.7 times greater than those achieved by Φ_{512} in BE case and

only 1.1 times greater in DS case; however, the enhanced per flow fairness does not prevent service assurance to be compromised in case of short flows: Φ_{512} reach, on average, the 99% of their target and Φ_{256} only the 83%.

AQM gateway in all the simulation used only two levels of drop precedence: in-profile DS green packets were protected by preferentially dropping out-of-profile DS red packets and BE packets. It may be wandered whether the deployment of other AQM parameter sets could improve protection of short DS flows. Therefore, we decided to give the best protection possible to DS packets, by marking them either green (AFx1) or yellow (AFx2) and using highest drop precedence codepoint (AFx3), that is red color, only for BE packets; this can be done by setting in the PIR to equal the line rate in the TSWTCM, working as usual in color aware mode.

Although such a marking scheme sensibly improves DS performances, the gain is however much more consistent for flows sending bigger packets: this can be gathered by comparison of results achieved by DS flows in (a) and (c) respectively to those achieved in (b) and (d); as a result, flows sending small packets are not assured their contracted rate, and unfairness among flows achieves even stronger values. Furthermore, interpretation of the results achieved at higher reservation –where strong penalty occur only for Φ_{256}– lets suppose that AQM does not completely solve the common drop tail synchronization side-effect.

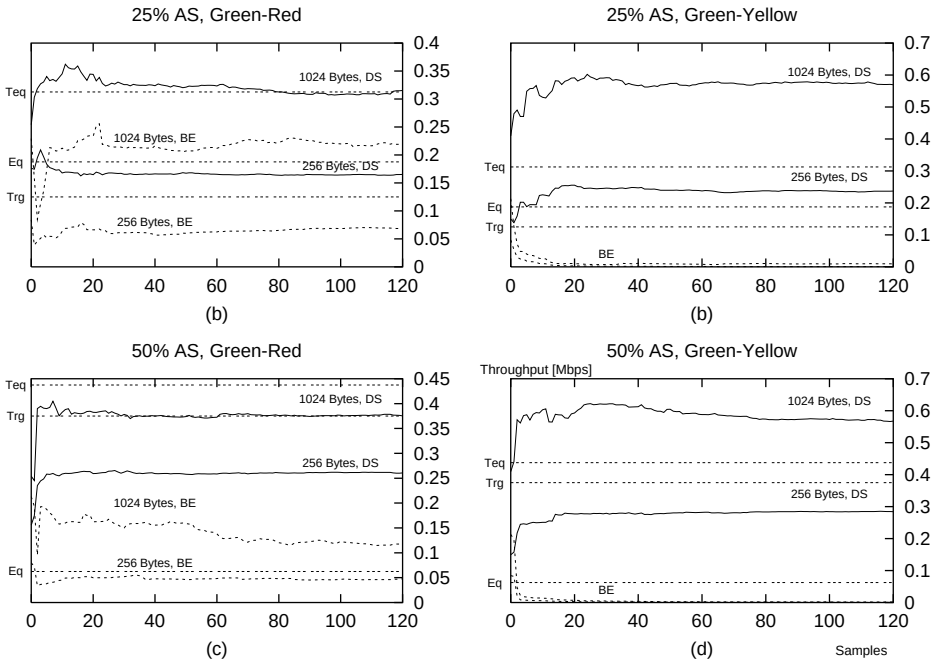


Figure 5.6: Packet Size: Different TSWTCM Configuration

Another critical remark can be described observing Fig. 5.6, plotting on the ordinate axis the DS and BE throughput of Φ_{256} and Φ_{1024} as a function of the time samples; in cases (a) and (c) the TSWTCM has PIR=CIR, thus DS packets are either green or red marked, whereas in (b) and (c) TSWTCM has PIR=B and DS packets are either green or yellow. Comparison of (a) to (b) shows that, even when only the small line rate fraction $\rho_{DS} = 25\%$ is allocated, BE flows face to starvation: the use of a staggered MRED parameter set ensures that all red packets are dropped before yellow packet drop begins. Moreover, TSWTCM marking is not deterministic; this means that an out of profile packet has a probability, decreasing as long as the flow rate increases, to be marked green: that is, to be forwarded as if it were in profile; this means that not only DS flows are allowed to send packets beyond the contracted rate, but also that such packets may be

given the best possible protection: some BE packets will be then dropped to protect unnecessary out-of-profile "lucky" DS packets.

Apart from TSWTCM setting, different MRED parameter sets were tried: however, none of these were useful in order to correct the unfairness and synchronization related problems; due to active queue management technique understanding importance, deeper considerations on its effectiveness are further developed in Sec. 5.7.

5.3 μ-Flows Number Effects

In a Differentiated Serviced Internet, service contracts will not necessarily be on a per-end user basis; that is, one common scenario will be the case where various business houses will contract a target rate with a service provider, and thus service agreement will normally be contracted on aggregated traffic. In such a context, source flows originating from any company would then compete against each other for the aggregate target rate. It is therefore important to investigate what happens when the number of micro flows competing for a particular target rate are different; an important consideration is that the number of flows within the aggregate will impact on the service performance: the aggregate with large number of flows will get more share of the excess bandwidth. This raises important fairness concerns, since it is likely that some organization will have thousands of flows in a target aggregate whereas others will have just hundreds of flows.

Simulations were performed however on the same scenario as before, considering thus a small number of micro flows within each connections: specifically, each aggregate will be constituted by $n = 2^i$ flows where $i \in \{0, 1, 2, 3\} \subset \mathbb{N}$. In this circumstance, the total number of DS flows is actually greater than those of connections: N_{DS} may thus represent the number of SLAs that the provider contracted with different peering domains. However, each of these domains is constituted by one of the source nodes of the common scenario topology, and each of these aggregate SLAs are equal independently from N_{DS}. Results that refer to the overall link utilization η may be directly compared only between simulations of μ-flow suite, since the higher number of flows per connection steams apparently better performances with respect to other simulation suites.

The four special cases represented by 5@25%, 10@25%, 15@75% and 20@100% allow then to test the ability for an ISP to provide the same SLA contract to an increasing number of companies: the fairness concern is therefore dependent of both the number n of active micro flows within the aggregate Φ_n and the number N_{DS} of these traffic aggregates for the same aggregate target size. Tab. 5.7 reports the results for the cases where either 5, 10 or 15 companies contract 250 Kbps of Assured Rate with the DS provider: at a first glance, it can be seen that service assurance is never compromised; rather, the spread of the difference in the share of the excess rate strongly depends on the μ-flow number. The results of these cases are depicted, for BE and DS aggregates, in Fig. 5.7(a,b,c): the ordinate axis represents the Thr/Teq achieved by the aggregate as a function of the μ-flow number; plots may also be interpreted as the average aggregate Thr/Teq that a company received depending on the number of flows within its aggregate.

When only five companies are reserved 1/4 of the domain capacity (5@25%), there is a large amount of excess bandwidth available to the share of both DS and BE services. In this case, BE aggregates with the highest number of micro flows perform even better than DS ones, and none of the other BE aggregate face to starvation; however, BE Φ_8 achieved throughput is 8 times greater than Φ_1 one: the gain of high Φ aggregate is roughly linear on the number of active micro flows. DiffServ mitigates, with respect to BE, the Φ_8 aggregate gain which is however still proportional to μ-flow number, as Fig. 5.7(a) shows; furthermore, the throughput achieved by Φ_8 is more than the double of Φ_1 one, and the unfairness achieves $\mathcal{U} = 28\%$. Whether a larger number of companies contract the same SLA, this is beneficial to both the link utilization and the fairness among aggregates. The slope of the curves in Fig. 5.7(b,c) confirm that TCP throughput dependence on μ-flow number is lessened for both DS and BE services; although the μ-flow number effects are quantitatively halved in the best case, however fairness is far from being reached either when $N_{DS} = 10$ ($\mathcal{U} = 21\%$) or $N_{DS} = 15$ ($\mathcal{U} = 21\%$).

μFLOW @ CIR		Micro-Flow Number			DS Overall		Link
		1	4	8	μ	σ	Σ
5 @ 25 %	Thr	0.332	0.524	0.711	0.476	0.132	4.812
	Thr/Trg	1.326	2.095	2.843	1.905	0.529	3.850
	Thr/Tfs	0.332	0.524	0.711	0.476	0.132	*0.962*
10 @ 50 %	Thr	0.305	0.429	0.531	0.398	0.085	4.824
	Thr/Trg	1.219	1.715	2.125	1.591	0.338	1.930
	Thr/Tfs	0.609	0.857	1.063	0.796	0.169	*0.965*
15 @ 75 %	Thr	0.271	0.347	0.404	0.332	0.049	5.051
	Thr/Trg	1.086	1.386	1.615	1.326	0.195	1.347
	Thr/Tfs	0.814	1.040	1.211	0.995	0.146	*1.010*

Table 5.7: μ-Flow: DS Performance, Trg = 250 kbps

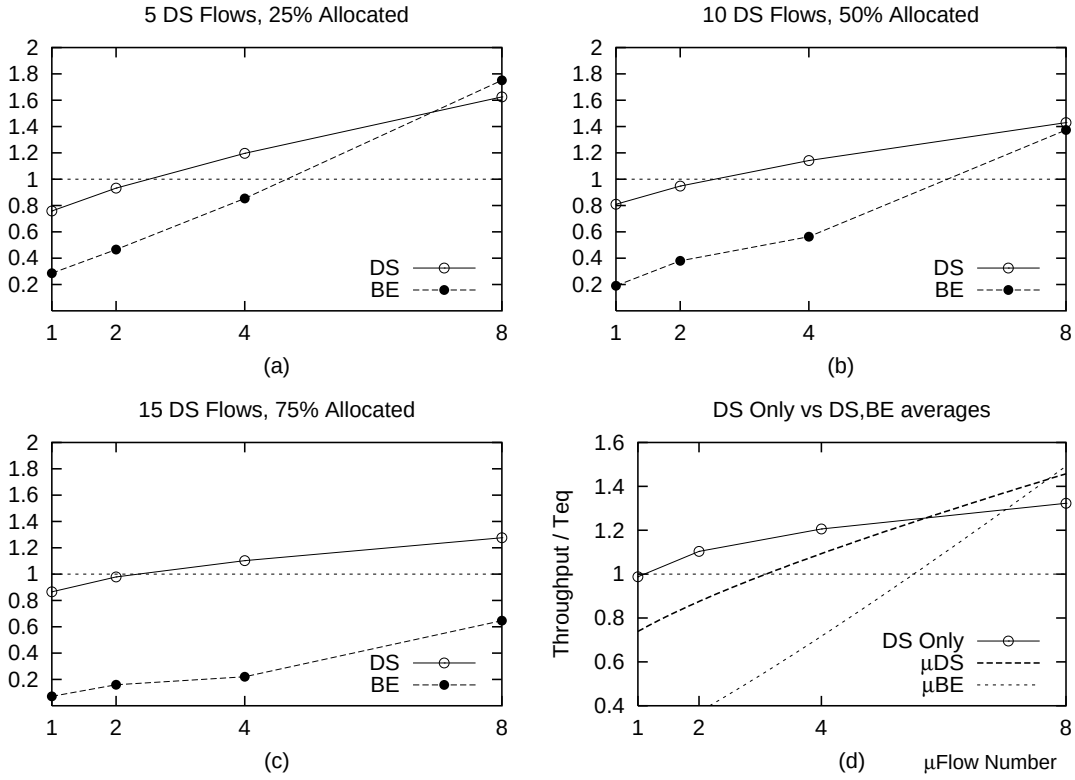


Figure 5.7: μ-Flow: Same Target

Furthermore, the benefits to BE brought by DS steam undesirable effects: considering 10@50%, it can be seen that fairness improvement concerns mainly aggregates with a small number of μ-flow. Specifically, BE Φ_1 achieves 1/5 of the Thr/Eq, and 1/4 of DS Φ_1 Thr/Teq, and similar conclusions can be gathered for both Φ_2 and Φ_4 BE aggregates; conversely, BE Φ_8 achieve 40% more than their equal share rate and Thr/Teq performances are as good as for DS Φ_8. The latter consideration is no longer true when 3/4 of the line rate are reserved to 15 companies; in this case BE Φ_8 gain over lower Φ is consistently reduced, but the increase of the BE fairness is offset by the fact that DS aggregate monopolize the network: BE aggregate with a low number of μ-flow are starving, as depicted in Fig. 5.7(c).

Fig. 5.7(d) depicts the case where whole bandwidth is reserved to DS flows, superimposing the DS and BE behavioral trend; the plotted curves represent the aggregated Thr/Teq for different μ-flow number. It is evident that, Φ_1 being the exception, each DS μ-flow aggregate's throughput will achieve a value higher than Teq: these results may seem contradictory, since none of the aggregates is really penalized with respect to their target; however, this has a straightforward explanation. Firstly, it must be considered that Teq is greater than Trg only for μDS and μBE, whereas Teq=Trg for the DS only case, since, being all the bandwidth reserved, there is no excess to be shared. Then, it should not be forget that throughput can exceed the line when measured *at the ingress* of the network; rather, the throughput sampled at network egress –that is, goodput– is reduced by the drop amount flows experience as long they cross the domain, and its value is lower than 1. This is further confirmed by the highest achieved drop percentage among all suites: in the DS only case, total packet drop percentage in the μ-flow suite is 14.29% whereas it is 5.22% and 4.91% respectively in packet size and RTT suites. Finally, we should notice that unfairness achieved in 20@100% case is $\mathcal{U} = 10\%$, and that Thr dependence on μ-flow is no longer linear: this behavior is rather different from the cases where the domain offers both services type.

Several simulation have been performed for different amount of per-aggregate assured bandwidth. Lets consider firstly the cases where an increasing number N_{DS} of DS aggregates is reserved the same line rate percentage ρ_{DS}%: this entails that the N_{DS} equal SLA, and thus each aggregate target rate, decreases as long as N_{DS} increases; in such cases, both the link utilization η and the unfairness among flows $\mathcal{U}$ increases as long as the SLAs decreases.

Reasons of η performances worsening may be identified with several simultaneous factors; firstly, a reduced number of BE flows will participate to the share of the $(1 - \rho_{DS}) \cdot B$ excess rate: due to consistent BE unfairness, only aggregates with a relatively high number of μ-flow will get consistent throughput performances. Then, it can be argued that the increased number of DS aggregates participating to the excess will be more penalized, as long as their SLA decrease: in fact, an increasing number of red marked DS packets will have lesser forwarding probability and will more easily incur in drop action.

To explain the unfairness increase as long as N_{DS} increases, it must be firstly said that $\mathcal{U}$ increase is partly due to the fact that Thr performance gain is less noticeable for lower μ-flow aggregates. Furthermore, the drop percentage growth as long as N_{DS} increase is more consistent for small aggregates than for bigger ones: this results in the growth of the overall drop percentage. Finally, in case of drops, Φ_1 cwnd take a certain amount of time time to return to its former size; meanwhile, since Φ_8 flows are less likely to simultaneously occur in drop, their average cwnd will be greater of Φ_1 one: this allows Φ_8 to better profit of both the assigned SLA and the excess bandwidth.

Conversely, as long as the size of each aggregate's target increases being N_{DS} constant, opposite results can be gathered: both fairness and link utilization increases. Throughput performances at different bottleneck allocations are reported in Tab. 5.8 for $N_{DS} = 10$, whereas Fig. 5.8 covers the cases where N_{DS} is either 5 or 15, plotting Thr/Teq as a function of μ-flow. An important observation is that the relationship of Thr/Teq results achieved for $N_{DS} = 5$ and $N_{DS} = 15$ is "symmetrical" depending on ρ_{DS}%: for example, in cases (a) and (b) the Thr/Teq difference is more noticeable for Φ_1 than for Φ_8, whereas the opposite happens in cases (c) and (d).

μFLOW @ FLOW		Micro-Flow Number			DS Overall		Link
		1	4	8	μ	σ	Σ
10 @ 25 %	Thr	0.205	0.344	0.505	0.319	0.105	4.709
	Thr/Trg	1.641	2.750	4.044	2.555	0.844	3.767
	Thr/Tfs	0.410	0.687	1.011	0.639	0.211	*0.942*
10 @ 50 %	Thr	0.305	0.429	0.531	0.398	0.085	4.824
	Thr/Trg	1.219	1.715	2.125	1.591	0.338	1.930
	Thr/Tfs	0.609	0.857	1.063	0.796	0.169	*0.965*
10 @ 75 %	Thr	0.387	0.494	0.561	0.465	0.063	4.892
	Thr/Trg	1.033	1.316	1.496	1.241	0.169	1.305
	Thr/Tfs	0.774	0.987	1.122	0.930	0.126	*0.978*

Table 5.8: μ-Flow: DS Performance, $N_{DS} = 10$

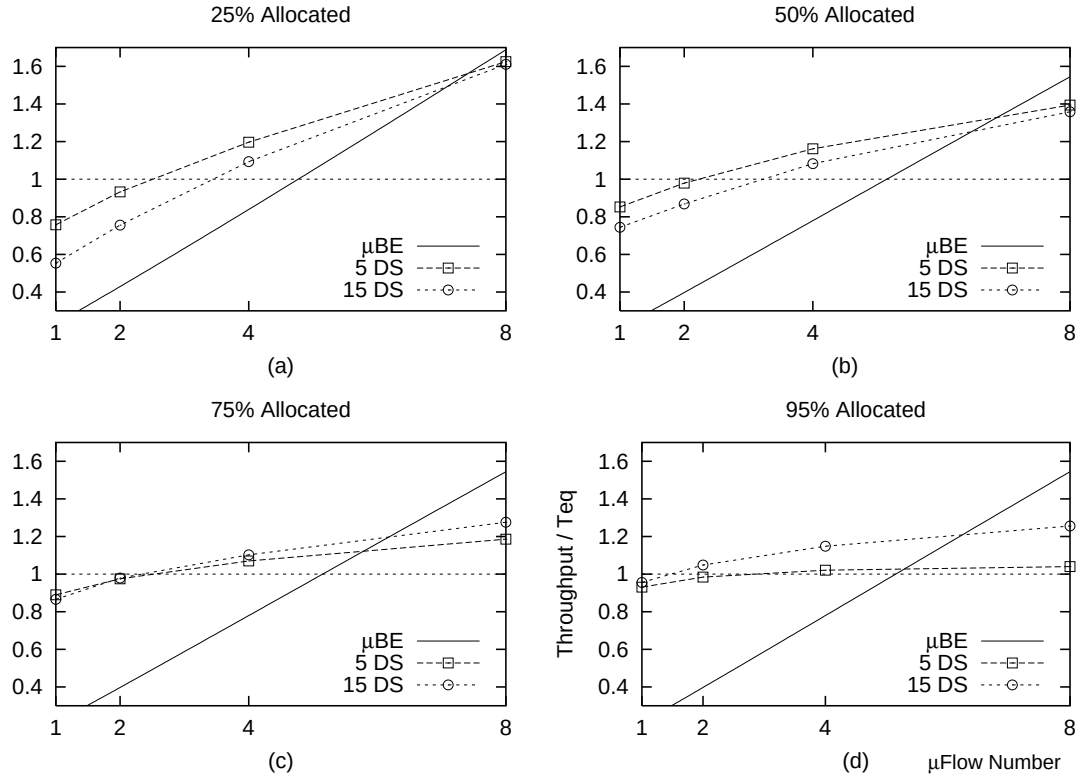


Figure 5.8: μ-Flow: Same Flow Number

Moreover, it should be noticed that, when the totality of the network is reserved to only 5 DS aggregates, unfairness is extremely low ($\mathcal{U} = 4\%$); moreover, the link utilization is not compromised and overall drop percentages are kept under a reasonable amount (5%). As side effects, BE traffic is –as reasonable– starving and the rate assurance is compromised at 6% for DS Φ_1.

μ**FLOW @ DRP**		μFLOW			DS Overall		Link
		1	4	8	*avg*	*sdev*	*tot*
CIR	5 @ 25 %	6.04%	12.43%	15.27%	10.47%	3.14%	18.20%
	10 @ 50 %	7.19%	15.13%	19.34%	12.64%	4.63%	16.94%
	15 @ 75 %	7.42%	16.39%	21.35%	14.29%	5.24%	15.56%
FLOW	10 @ 25 %	10.33%	16.97%	19.99%	15.01%	3.69%	19.43%
	10 @ 50 %	7.19%	15.13%	19.34%	12.64%	4.63%	16.94%
	10 @ 75 %	4.74%	12.84%	17.52%	10.52%	4.80%	12.69%
ALLOC	5 @ 50 %	3.01%	9.15%	12.08%	7.13%	3.15%	15.14%
	10 @ 50 %	7.19%	15.13%	19.34%	12.64%	4.63%	16.94%
	15 @ 50 %	9.91%	18.29%	22.51%	16.37%	4.85%	18.56%

Table 5.9: μ-Flow: Per-Flow, Aggregated and Overall Packet Drop Results

Finally, per-flow and aggregated drop percentages are shown in Tab. 5.9 for the described simulation cases. Quantitatively, per-flow data confirm the earlier developed considerations about N_{DS} and $\rho_{DS}\%$ influence on overall system performances.

Non reported BE drop percentages are always at least the double of DS, and shows almost no dependency on the μ-flow number. Although it can be argued that DS throughput fairness is related to a correspondent consistent drop differentiation depending on μ-flow number, however, any quantitative conclusion is not straightforward. Indicating with $\mathcal{D}\frac{\sigma}{\mu}$ the drop coefficient of variation expressed in per cent (that is, the drop "unfairness"), no satisfying model fitting data reported in Tab. 5.10. This is partially due to the fact that the normalization of σ over μ entails the loss of the spread of these measures; it thus becomes hard to immediately test and compare different marker setting and to find SLAs definition guidelines.

μ**FLOW @ DRP**		$\mathcal{U}$	$\mathcal{D}\frac{\sigma}{\mu}$	η
CIR	5 @ 25 %	27%	29%	96%
	10 @ 50 %	21%	36%	96%
	15 @ 75 %	24%	36%	101%
FLOW	10 @ 25 %	33%	24%	94%
	10 @ 50 %	21%	36%	97%
	10 @ 75 %	13%	45%	98%
ALLOC	5 @ 50 %	17%	44%	99%
	10 @ 50 %	21%	36%	96%
	15 @ 50 %	23%	29%	95%

Table 5.10: μ-Flow: System Metric for Evaluating DiffServ Aggregate Peformances

5.4 Target Rate Size Effects

In this section we explore whether and how well DS long lived TCP flows achieve their target rates, focusing further on how the excess bandwidth is shared among flows having differently sized targets. Results obtained agree with the analysis developed in earlier works [AFEVAL],[BWRES] in that, basically, in an over-provisioned network the excess rate get distributed equally irrespective of the size of the target.

In our simulated network scenarios, the total size of the assured rate depends thus on the DS aggregate SLA, thus on ρ_{DS}%; however, the per-flow fractioning of the DS bandwidth depends on DS flows number N_{DS}: either 5 (when $N_{DS} = 5$) or 10 (when $N_{DS} > 5$) different target sizes are chosen in order to be equally spaced in the interval of possible target sizes; furthermore, when $N_{DS} = 15$, the duplicated target are distributed on the whole set rather than being tightly ranged. These intervals, described in terms of μ and σ, are shown in Tab. 5.11 as well as $\mu - \sigma$ and $\mu + \sigma$ values; in fact, per-flow and thus per-target size results are reported in tables for $\Phi_{\sim(\mu-\sigma)}$, $\Phi_{\sim\mu}$ and $\Phi_{\sim(\mu+\sigma)}$: that is, for flow whose target is the closest to $\mu - \sigma$, μ and $\mu + \sigma$ respectively.

Simulation		σ	$\mu - \sigma$	$\overline{CIR} \equiv \mu$	$\mu + \sigma$
CIR	5 @ 25 %	0.151	0.099	0.250	0.401
	10 @ 50 %	0.131	0.119	0.250	0.381
	15 @ 75 %	0.123	0.127	0.250	0.373
ALLOC	5 @ 50 %	0.302	0.198	0.500	0.802
	10 @ 50 %	0.131	0.119	0.250	0.381
	15 @ 50 %	0.082	0.085	0.167	0.249
FLOW	10 @ 25 %	0.065	0.06	0.125	0.19
	10 @ 50 %	0.131	0.119	0.250	0.381
	10 @ 75 %	0.196	0.179	0.375	0.571
HIGH-a	5 @ 95 %	0.574	0.376	0.950	1.524
	10 @ 95 %	0.248	0.227	0.475	0.723
	15 @ 95 %	0.156	0.161	0.317	0.473
HIGH-f	15 @ 75 %	0.123	0.127	0.250	0.373
	15 @ 85 %	0.140	0.143	0.283	0.423
	15 @ 95 %	0.156	0.161	0.317	0.473

Table 5.11: Target Rates Distribution

We defined the fair excess share to be proportional to the target rate: this equals to suppose that a customer with an higher target rate considers fair to receive an higher share of the excess bandwidth. However, from analysis of the simulation results we gather that that as the target rate increases, the achieved rate increases as well, but not proportionally; rather, excess tend to be equally shared. The explanation is the variation of the congestion window: after the window has closed due to packet losses, the connections with small target rates return to their former window size quicker than those with bigger target rates, thus starting sooner to compete for the excess bandwidth. Small target rate connections profit the most of the time during which the large target rate connections are increasing their window to capture excess bandwidth; this is comparable to the opportunism of small RTT connections at the expense of those with larger RTT.

Rather different results are obtained depending on the aggregated SLA size and on the number of connections that profit of it; due to the fact that target rate is a parameter specific of DiffServ AR service, the performances of BE flows, whose target is identically null, will be mostly disregarded in this suite analysis.

TRG @ ALLOC		Target Rate			DS Overall		Link
		$\sim(\mu-\sigma)$	$\sim\mu$	$\sim(\mu+\sigma)$	μ	σ	Σ
5 @ 50 %	Trg	0.278	0.463	0.741	0.500	0.302	2.500
	Thr	0.383	0.528	0.753	0.555	0.250	4.314
	Thr/Trg	1.378	1.141	1.017	1.346	0.458	1.726
	Thr/Tfs	0.689	0.570	0.509	0.673	0.229	*0.863*
10 @ 50 %	Trg	0.136	0.273	0.364	0.250	0.131	2.500
	Thr	0.237	0.379	0.453	0.345	0.121	4.390
	Thr/Trg	1.735	1.389	1.246	1.623	0.529	1.756
	Thr/Tfs	0.868	0.695	0.623	0.812	0.265	*0.878*
15 @ 50 %	Trg	0.090	0.181	0.241	0.167	0.082	2.500
	Thr	0.203	0.291	0.344	0.267	0.082	4.480
	Thr/Trg	2.251	1.609	1.429	1.890	0.682	1.792
	Thr/Tfs	1.125	0.805	0.715	0.945	0.341	*0.896*

Table 5.12: Target Rate: DS Performance, $\rho_{DS} = 50\%$

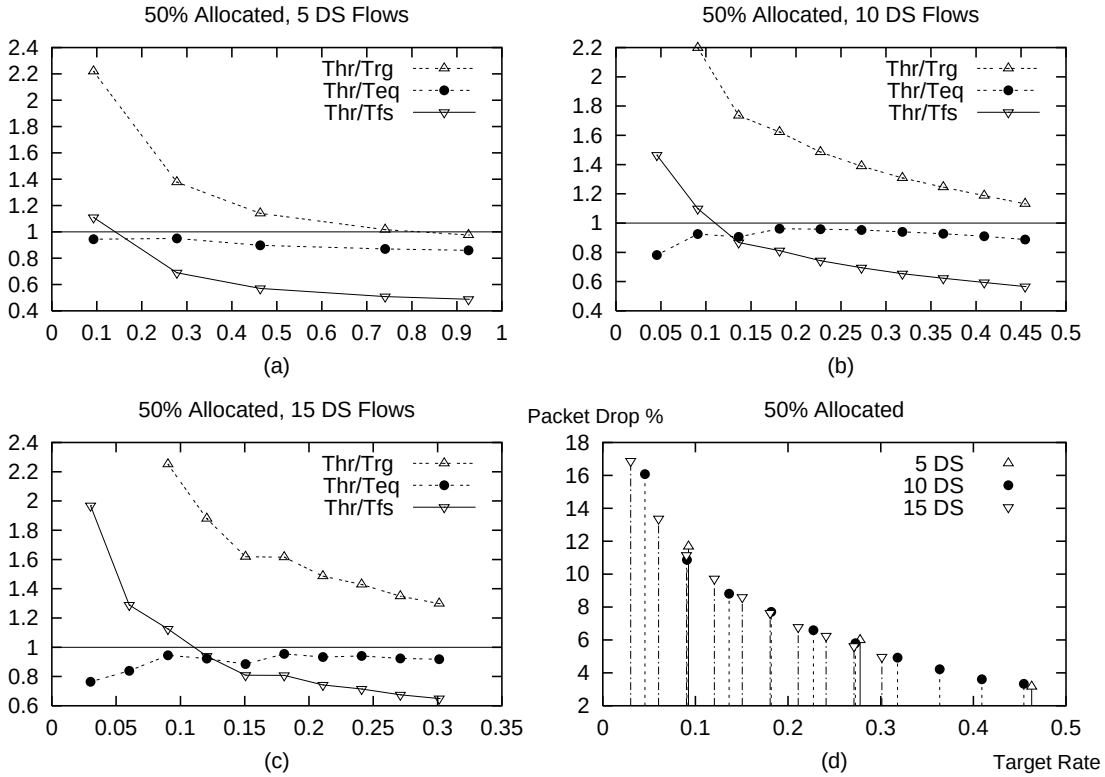


Figure 5.9: Target: Same Allocation

Tab. 5.12 reports the Thr, Thr/Trg and Thr/Tfs data collected when half of the line rate is allocated to a different number of flows $N_{DS} \in \{5, 10, 15\}$; these scenarios are further investigated in Fig. 5.9(a,b,c), reporting DS flows' throughput performances normalized over Trg, Tfs and Teq as a function of Trg, whereas Fig. 5.9(d) depicts the loss rate as a function of the Trg. Although Trg size is quite different depending on N_{DS}, however $\text{Trg}_{\sim(\mu+\sigma)}/\text{Trg}_{\sim(\mu-\sigma)} = 2,67$ is constant over the whole simulation set, so that per-flow performance comparison is done using a common scale among different runs.

Due to the considerations earlier developed, we find that as long as N_{DS} increases from 5 to 15, there is a minimal (0.2 dB) link utilization gain, and unfairness increase is limited to $\sim$3% ($\mathcal{U}_{\sim 1550} = 36\%$); however, flow with the lowest target gains more than 10% with respect to highest target flows and moreover when 5@50%, service assurance is compromised for biggest target flow, as Thr/Trg< 1 in Fig. 5.9(a) indicates. What happened is that RED tries to correct the opportunism of smallest Trg via packet drop: as (d) shows, small Trg flows achieve much higher drop rate than bigger Trg flows. This is nevertheless insufficient to protect bigger Trg flow in some cases; however, the main effect resulting from RED deployment is that the achieved Thr tends to equal Teq rather than Tfs for any flow: this can be inferred from the fact that Thr/Teq is, approximatively, a straight line. It should be noticed that shortest Φ are extremely more advantaged with respect to any other Φ: e.g., Thr/Tfs in (c) shows a piecewise linear behavior and shortest Φ gain on $\Phi_{\sim\mu}$ is 3.9 dB, whereas $\Phi_{\sim\mu}$ gain on longest $\Phi_{\sim\mu}$ is only 1.2 dB Furthermore, whether the unfairness is calculated disregarding lowest Φ performances we obtain $\mathcal{U}_{15@50\%} = 15\%$ and $\mathcal{U}_{15@50\%} = 24\%$; these data are more significant than the previous obtained, allowing us to conclude that the N_{DS} increase result in stronger service unfairness. This derives from the fact that an higher number of short Φ profit the most of the excess bandwidth; furthermore, at a given aggregated SLA, the number of flows achieving more than Tfs seems to be proportional to N_{DS} whether the per-flow SLAs are linearly distributed. Finally, despite in (c) shortest Φ reaches twice of its Tfs, however the Thr/Teq performance is slightly degraded with respect to bigger Φ; this effect, noticeable also in case (b), happens because shortest Φ drop percentage is $\sim$1.5 times greater with respect to case (a).

When a constant number of DS flows $N_{DS} = 10$ is reserved an increasing bottleneck percentage $\rho_{DS}\% \in \{25\%, 50\%, 75\%\}$, the unfairness sensibly, although not proportionally, decreases from $\mathcal{U} = 61\%$ to $\mathcal{U} = 24\%$. From analysis of data reported in Tab. 5.13, we find that $\Phi_{\sim(\mu+\sigma)}$ achieves a Thr 1.5 times greater with respect to $\Phi_{\sim(\mu-\sigma)}$, roughly the half of $\text{Trg}_{\sim(\mu+\sigma)}/\text{Trg}_{\sim(\mu-\sigma)}$, in 10@25% case, whereas the scale factor grows to 2.1 when 10@75%; significant the link utilization achieve a modest gain (0.5 dB). The bigger Trg flows performances upgrade cannot be explained without considering TCP dynamics: in fact, rather than the constance of $\text{Trg}_{\sim(\mu+\sigma)}/\text{Trg}_{\sim(\mu-\sigma)}$ scale factor, the specific Trg size is a key of importance. We know that, in order to achieve their Trg, flow's cwnd must satisfies Eq. 4.1: this entails for cwnd a direct proportionality to Trg; looking at the spread of Trg, it can be gathered that in certain cases, for $\Phi_{\sim(\mu-\sigma)}$ even a cwnd~ 1 is able to satisfies Eq. 4.1. Therefore, being Tfs proportional to Trg and Trg extremely small, such flows achieve both their Trg and Tfs operating in slow start without even the need of opening their window. In these conditions, $\Phi_{\sim(\mu-\sigma)}$ performances are less sensible to drops in the sense that even cwnd$= 1$ after a drop guarantees the flow to achieve both Trg and Tfs; conversely, when a packet is successfully dequeued, the exponential growth of cwnd in slow start steams an extremely consistent throughput gain for such flows.

Fig. 5.10 plots the Thr (normalized over Trg, Teq and Tfs) achieved by 15 DS flows at different aggregated SLA ($\rho_{DS}\% \in \{75\%, 85\%, 95\%\}$) as a function of Trg. It can be noticed that only partial benefits are brought by SLA increase; from system performances point of view, link utilization and drop percentage benefits of the SLA increase: this is mostly due to the fact that a higher number of green packets better uses the RED buffer with respect to red high drop precedence packets, that are more likely to be dropped due to MRED queue control. However, it must be noticed that when the SLA exceeds a certain amount, higher Trg flows will achieve a throughput actually lower than the contracted rate; this SLA limit is shown case (b) of the figure, where biggest Trg flows achieve a Thr/Trg$= 1$ whereas it is 1.5 for the lowest Trg flows.

TRG @ FLOW		Target Rate			DS Overall		Link
		$\sim(\mu-\sigma)$	$\sim\mu$	$\sim(\mu+\sigma)$	μ	σ	Σ
10 @ 25 %	Trg	0.068	0.114	0.182	0.125	0.065	1.250
	Thr	0.210	0.258	0.320	0.273	0.063	4.144
	Thr/Trg	3.073	2.267	1.762	2.943	1.819	3.315
	Thr/Tfs	0.768	0.567	0.441	0.736	0.455	*0.829*
10 @ 50 %	Trg	0.136	0.273	0.364	0.250	0.131	2.500
	Thr	0.237	0.379	0.453	0.345	0.121	4.390
	Thr/Trg	1.735	1.389	1.246	1.623	0.529	1.756
	Thr/Tfs	0.868	0.695	0.623	0.812	0.265	*0.878*
10 @ 75 %	Trg	0.205	0.409	0.545	0.375	0.196	3.750
	Thr	0.272	0.452	0.574	0.419	0.173	4.690
	Thr/Trg	1.331	1.105	1.053	1.252	0.306	1.251
	Thr/Tfs	0.998	0.829	0.789	0.939	0.230	*0.938*

Table 5.13: Target Rate: DS Performance, $N_{DS} = 10$

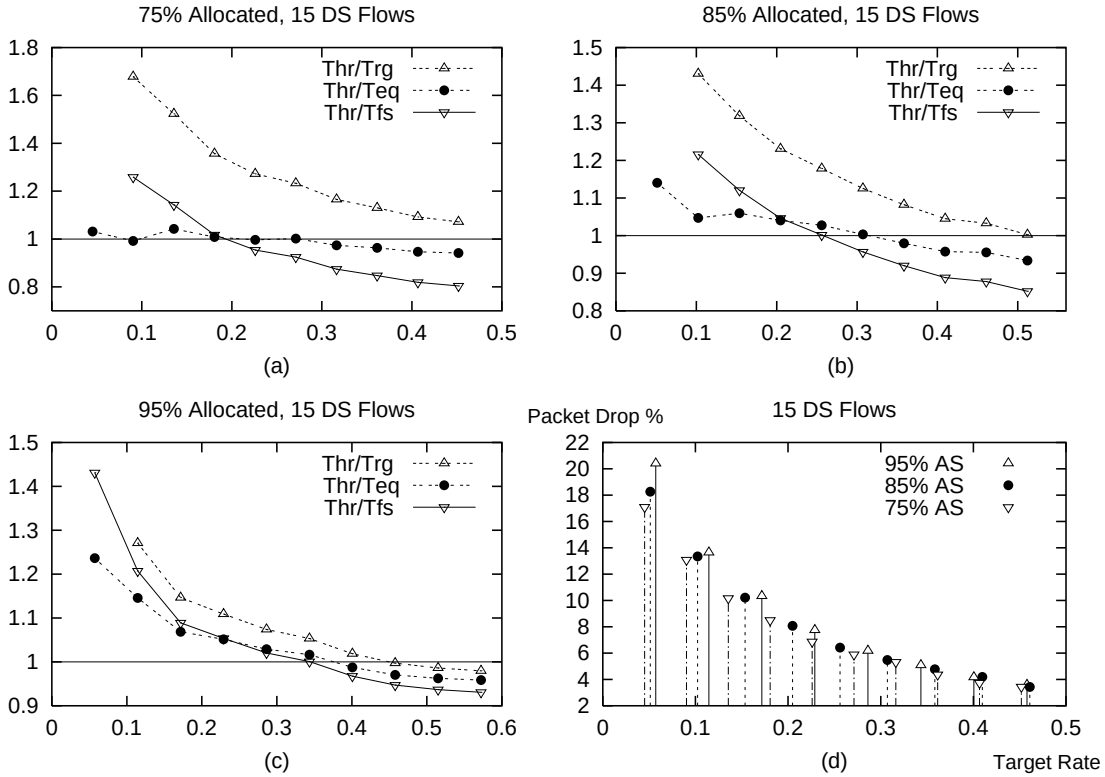


Figure 5.10: Target: Higher Flow Number

Moreover, service degradation for bigger Trg flows happens despite a consistent fairness amount between cases (a) and (c), where $\mathcal{U} = 26\%$ and $\mathcal{U} = 13\%$ respectively: it seems as though RED provides only a partial solution, from the user point of view, toward both service efficiency and fairness problems.

This further confirms that it would be hard to individuate quantitative guidelines for SLA tuning even in the case of ideal network situation where all the network parameters are equal. However, it must be kept in mind that DiffServ architecture was defined with the express intent of providing a *coarse* service differentiation; therefore it is likely to happen that a DS domain may be able to offer, in a real network situation, only a few well-dimensioned different AR amounts.

As final considerations, it can be summarized that in an over-provisioned network customers will be able to achieve their contracted rate as long as the network is properly configured; however, the excess bandwidth will tend to be equally rather than proportionally shared. In an under-provisioned network, paying customers will naturally assume that they will achieve the same ratio of their target rate as all other customers paying in the same class: this equals in fact to a fair service degradation. However, as fairness is far from being reached in any over-provisioned network case, it is likely that the small target Φ opportunism effects will not be lessened; a partial confirmation to this assumption can be gathered by results obtained in some "misconfigured" network cases, where big target Φ does not even reach their rate unlike smaller target ones.

5.5 DiffServ versus Best Effort

In this section we explore how DiffServ traffic interacts with Best Effort traffic; specifically, packet of DS capable flows can assume different drop precedences depending on their compliance to a traffic profile, whereas there is no such packet discrimination for BE packets: DS QoS offering relies on the effectiveness of in-profile green marked packets protection.

Suite		Throughput		Goodput		Drop		L-Drop		E-Drop	
		Mbps	Tot	Mbps	Tx	$\frac{Drp}{Tot}$	Drp	$\frac{LDrp}{Drp}$	LDrp	$\frac{EDrp}{Drp}$	EDrp
DS	RTT	5.03	189239	4.78	179925	5%	9314	89%	8311	11%	1003
	SIZE	4.98	359392	4.72	344841	4%	14551	92%	13384	8%	1167
	μFLOW	5.78	217214	4.95	186175	14%	31039	96%	29786	4%	1253
BE	RTT	3.93	147835	3.64	136789	8%	11046	30%	3333	70%	7713
	SIZE	3.69	225005	3.36	204908	9%	20097	41%	8309	59%	11788
	μFLOW	4.63	173978	3.68	138352	20%	35626	61%	21856	39%	13770
$\frac{DS}{BE}$ Gain	RTT	1.280		1.315		0.843		2.494		0.130	
	SIZE	1.597		1.683		0.724		1.611		0.099	
	μFLOW	1.248		1.346		0.871		1.362		0.091	

Table 5.14: DS Only vs BE Only Scenarios, Ingress Queue Statistics

Conclusive comparison of aggregated system results are reported in Tab. 5.14 for the DS only and BE only cases, showing also the ideal gain achieved by DS packets with respect to BE best case. It is interesting to notice that early drop mechanism is only effective for high drop precedence packets; more generally, when three different drop precedences are used, early drop action has been found effective mainly for yellow marked packets, yet not influencing green packets protection. In fact, it can be argued that DS early drops mostly occur for red or yellow packets, whereas green packet drop is a rare event due to buffer overflow: this means that, with respect to green in-profile packets, RED is mainly working as drop tail. This phenomenon could be avoided increasing the buffer size and modifying the green max_{th}; however, this would be a fake solution to the problem, since the main goal in RED should be a better use of the available resources rather than the need

of their over-provisioning. In order to quantify the strength of the investigated factors on DS TCP flows, Fig. 5.11 provides a normalized representation allowing their direct comparison. The y-axis reports the ratio of the achieved throughput over target rate value, while the x-axis represents, for each different parameter X, the diverse values X_i normalized over that parameter average X_i/X_μ: this is done in order to have a common comparison scale for any network factor. An interesting observation is that in RTT case the throughput degradation is linear with a roughly constant slope, whereas this is not true with respect to the other factors: smallest packet size flows and aggregates constituted by only one flow are much more disadvantaged, resulting in the overall Thr/Trg piecewise linear behavior; finally, flows with a small target rate are consistently advantaged with respect to the trend behavior, and only big target flows are prevented from reaching their target.

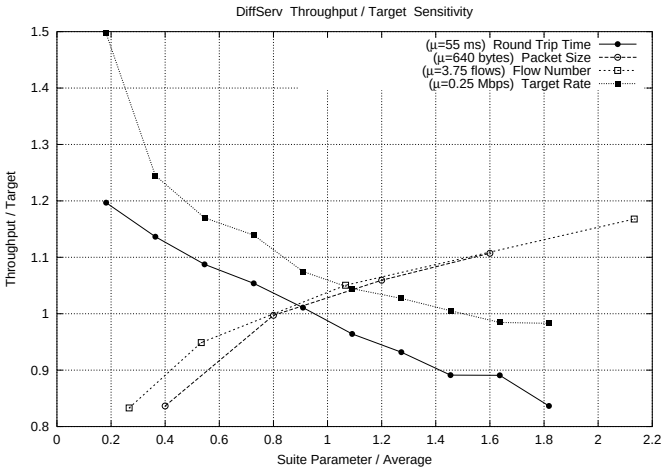


Figure 5.11: DiffServ Sensitivity on Network

Another rough measure of the unfairness determined by each of the former factors is represented by the value of the covariance of throughput with each factors; their values are reported in Tab. 5.15, as well as the respective correlation value ρ for both DS and BE ideal cases. From the latter parameter, which is more readable due to its normalization $\rho \in [-1, 1] \subset \mathcal{R}$ property, it can be gathered that RTT and target unfairness solution may be the most difficult challenge; moreover, whereas the target is known a priori, the RTT can only be estimated in a real network situation. The packet size unfairness may be corrected by simply using a common packet size in the domain; however, DS domains should not disregard the possibility that some companies deliberately decide to use bigger packet sizes to their own profit.

Up to this point, the analysis of DiffServ versus Best Effort performances have been conducted evaluating each service reaction to a specific factor; then, DS and BE throughput were compared to the target plus equal share Teq in order to test whether coexistence of these services is possible within a single AF class. Generally, we noticed that as long as the SLA or the number of DS connections increases, BE connections' rates are decreased; this is reasonable and acceptable since for DS connections to get their subscribed bandwidth, some other connections must lose bandwidth.

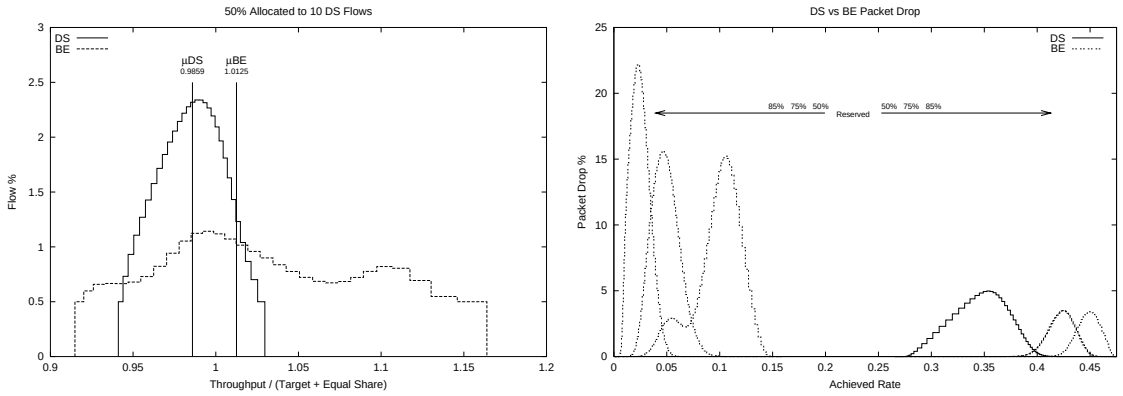
To complete this analysis, our concern is whether in the same network conditions, DS and BE effort connections compete for the excess bandwidth on an equal footing. Therefore, we implemented a scenario where half of the bottleneck is reserved to 10 DS flows, and half is available to the share; all flows are identically configured, thus they all have the same RTT, the same packet size, etc. Simulation were run several times in order to create histograms of the percentage of connections whose average throughput over the run fell into a particular rate range.

Cov(X.Y) Cor(X.Y)		Thr		Thr/Teq†	
		DS	BE	DS	BE
RTT	*cov*	-0.7946	-2.7337	-3.1781	-10.9350
Round Trip Time	*cor*	-0.9895	-0.9311	-0.9895	-0.9311
SIZE	*cov*	6.9927	20.1454	27.9699	80.5798
Packet Size	*cor*	0.9538	0.9924	0.9538	0.9924
μFLOW	*cov*	0.0797	0.4079	0.3186	1.6317
Micro-Flow Number	*cor*	0.9578	0.9893	0.9578	0.9893
TRG	*cov*	0.0156	-	-0.0174	-
Target Rate	*cor*	0.9993	-	-0.8792	-

† *Thr/Teq≡Thr/Trg≡Thr/Tfs in the DS only scenario.*

Table 5.15: DS Only vs BE Only Scenarios, Simulation Parameter Interdependecies

As a first note it must be said that, over all the runs BE connections achieving Teq are twice as many as DS ones; moreover, BE connections achieve a wider rate range than DS. This means that as expected, unfairness is more pronounced: since packets are all marked with the Default DSCP, RED has no possibility of penalizing BE connections exceeding their Eq rate. This is clearly shown in Fig. 5.12(a), which depicts on the ordinate axis the throughput distribution as a function of the achieved rate normalized over Teq. We know already that share distribution is not proportional to the target: comparing the average Thr/Teq references among both BE and DS cases, we notice further that bandwidth share is not even equal. A DS connection sending out of profile packets will experience some drops, just like any BE connection, resulting in its sending window closing and the corresponding decrease of the sending rate; meanwhile, other connections opportunistically increase their rate. Since DS connections send at a higher rate than BE connections do, their window size is larger and therefore, once it has closed, requires more time to get to the original size: during this time, BE flows can open their windows. In other words, a drop causes the window to close not only with respect to the excess bandwidth, but also with respect to the assured bandwidth, as there is only one single window handling packets irrespectively from their profile.



(a) (b)

Figure 5.12: DiffServ vs Best Effort: Throughput and Drop Performances

Finally, it must be said that this effect is noticeable, although lessened, also when the reserved

rate increases above $\rho_{DS} = 50\%$. Fig. 5.12(b) depicts the drop probability as a function of the achieved throughput for different aggregated reservation rates $\rho_{DS}\% \in \{50\%, 75\%, 85\%\}$: as long as $\rho_{DS}\%$ increases, DS flows experience a reduced drop amount; this fact, coupled with the observation that Teq vs Thr difference is less important as long as $\rho_{DS}\%$ approach line rate, result in a lessening the DS versus BE unfairness.

5.6 Non Responsive Traffic Effects

Recently the Internet is becoming identifiable with the Web, and therefore our study focuses on applications using the TCP protocol at the transport layer: long lived FTP flows and short lived HTTP flows detailed simulation results analysis are respectively the subjects of present Ch. and Ch.7.

However, many applications that are candidates for QoS are realtime applications that do not require a reliable transport protocol: usually they run over UDP and are nonresponsive to congestion indication through loss. It may be observed that, in order to fulfill the requirements of a real time application, a service based on EF PHB may be better suited than one implemented via a class of the AF PHB group, since this latter cannot provide neither delay nor jitter bounds.

Nevertheless, since there are no restrictions on the traffic type that may be forwarded using a single class of the AF PHB group, interaction of UDP and TCP sources cannot be disregarded. Recent studies showed that unresponsive sources introduction is among the factors affecting the fair bandwidth distribution for aggregates with equal target rates; in this section, rather than analyzing simulation data, we develop some considerations reported in these studies.

To introduce the problems that arise when unresponsive flows such as UDP share the same AF class with TCP flows, we may refer to [AFEVAL], where the mutual influence of non-responsive and responsive traffic is examined through simulation. One only CBR source, using either DS or BE service, is inserted in a DS domain crossed by an equal number of BE and AF connections carrying unidirectional long lived FTP flows: different simulation are performed with several values for the CBR rate. The traffic conditioning block implemented is based on a deterministic token bucket algorithm; reasons behind this choice stem from the fact that probabilistic marking is problematic for metering a CBR source: a profile meter using an average rate estimator would allow the source to transmit at a sustained rate higher than the contracted one, whereas token buckets do not permit this.

In the case of non-responsive BE source, simulation results show that all the TCP connections get degraded –with BE connections being pushed toward starvation– as long as the CBR source increases its sending rate. Of course the CBR source experiences increasing loss rate but it still captures a lot of bandwidth, achieving a limit when RED $\max_t h$ of DS out-of-profile packets is constantly reached. In this situation, DS TCP connections alternate slow start phases and congestion avoidance phases, never going through a fast recovery phase: as a result, almost all the packets leaving the hosts are marked with the lowest drop precedence, preventing the CBR source from grabbing more bandwidth.

When the nonresponsive CBR source is made DS capable, the results in terms of achieved rate are almost identical to those obtained in BE case; that is, a nonresponsive source produces the same impact on other connections, whether it is DS capable or not: it will grab most of the bandwidth it needs at the expense of TCP connections. The actual difference lies in the number of packets sent by the CBR source which later get dropped: virtually no packets get dropped when the CBR source transmits at its target rate, while exceeding the $\min_t h$ leads to early dropping a nonnegligible number of in profile packets.

In recent DiffServ IETF discussions [AFSTDY] on whether the AF PHB required two or three drop precedences, it has periodically been suggested that the TCP packets can be protected from non-responsive UDP packets by assigning UDP to a different drop precedence value than TCP. The reason is that congestion sensitive flows reduce their data rate on detecting congestion whereas

congestion insensitive flows keep on sending data as before. Thus, in order to prevent congestion insensitive flows from taking advantage of reduced data rate of congestion sensitive flows in case of congestion, excess congestion insensitive traffic should get much harsher treatment from the network than excess congestion sensitive traffic. Hence, it is important that excess congestion sensitive and insensitive traffic is colored differently so that network can distinguish between them.

Experiments developed in [TCPUDP] extend the study of the contemporary presence of both responsive and non-responsive sources within a single AF class, exploring different combinations of drop precedence mappings for TCP and UDP traffic. Results are collected via an experimental testbed using Netperf tool to generate the competing TCP long lived flows –all having equal target rate– and the UDPBLAST tool to generate the UDP non-responsive traffic; the devices implement the AF PHB using the Multiple-RED (MRED) scheme, operating in coupled mode, and the policer used is the TSWTCM tagger.

An important consideration there developed is that "while there is clearly a need to ensure that responsive TCP flows are protected from non-responsive flows in the same class, we also recognize that certain UDP flows will require the same fair treatment as TCP due to multimedia requirements". This is a key of importance, since the discussion on solving the UDP/TCP fairness issue for the AF PHB appears to have focused on penalizing the UDP flows. Rather, there is a need to ensure that the drop precedence mapping scheme utilized should ensure fairness for both TCP and UDP, where fairness is defined to mean that:

- in an over-provisioned network, both UDP and TCP target rates should be achieved with the in-profile traffic being protected
- in an over-provisioned network, UDP out-of-profile and TCP out-of-profile packets should have a reasonable share of the excess bandwidth; furthermore, neither TCP nor UDP should be denied access to the excess bandwidth.
- in an under-provisioned network, TCP and UDP flows should experience degradation in proportion to their target bandwidth

The experiments do not explore the possibility that packets from a single policy aggregate (i.e. either UDP or TCP) may be marked into three different drop precedences: packets can be marked either as in or out-of-profile, but depending on the particular mapping scheme used, may be assigned any one of the three drop precedence markings for that class, as reported in Tab. 5.16.

Type of	Scenario Index					
Packet	1	2	3	4	5	6
TCP *in*	DP0	DP0	DP0	DP0	DP0	DP0
TCP *out*	DP1	DP1	DP1	DP2	DP1	DP1
UDP *in*	DP0	DP1	DP1	DP1	DP2	DP0
UDP *out*	DP1	DP1†	DP2	DP2	DP2†	DP2

† No distinction is made between UDP *in* and *out* packets

Table 5.16: Different Schemes for TCP and UDP Drop Precedence Mapping

The six scenarios defined represent a subset of "meaningful" mapping policy of the whole permutation possibilities. TCP *in* packets are always assigned DP0 best forwarding probability and, except for case 4, TCP *out* packets are always DP1 marked; different UDP precedence mapping schemes are investigated in order to test whether it exists a mapping that produces overall fairness among TCP and UDP performances. Furthermore, for each scenario experiments are performed with either *totally overlapped* or *staggered* model for the RED parameter settings:

the latter model allows a greater opportunity for lower drop precedence DP0 packets to reach their end destinations than the other model. Tab. 5.17 summarizes the results of the six experiments, translating the three fairness objectives previously introduced into several criteria by which any drop precedence mapping scheme can be evaluated. One relevant observation from the table is that none of the scenarios achieves all three objectives for both TCP and UDP: at most, four out of six criteria are achieved.

Scenario	Over-Provisioned				Under-Provisioned	
	Achieves Target ?		Shares Excess ?		Fair Degradation ?	
	TCP	UDP	TCP	UDP	TCP	UDP
1	$\surd$	$\surd$	χ	$\surd$	χ	$\surd$
2	$\surd$	$\surd$	χ	$\surd$	$\surd$	χ
3	$\surd$	χ	$\surd$	χ	$\surd$	χ
4	$\surd$	$\surd$	χ	$\surd$	$\surd$	χ
5	$\surd$	χ	$\surd$	χ	$\surd$	χ
6	$\surd$	$\surd$	$\surd$	$\surd, \chi$ †	χ	$\surd$
$\surd$ Yes, χ No, † Depends on the RED model used						

Table 5.17: Summary of Test Results from Six Scenarios

The results show that in an over-provisioned network, if TCP is mapped to DP0, it mostly achieves its target rate irrespective of the drop preference to which UDP in-profile packets are mapped. UDP achieves its target rate if its *in* packets are protected by mapping them to DP0 (cases 1 and 6) or whether they are mapped to DP1, with the further constraint that UDP *out* packets have not higher drop precedence than TCP *out* ones (cases 2 and 4). However, the manner in which excess bandwidth is shared remains dependent on the drop preference assigned to both TCP and UDP *out* packets: none of the scenarios are fair to both UDP and TCP. In an under-provisioned network, mapping TCP in-profile to DP0 and UDP in-profile to either DP1 or DP2 causes TCP to suffer lesser degradation from its target bandwidth than UDP (cases 2 to 4). Mapping both UDP and TCP in-profile to DP0 results in unfairness to TCP as it experiences severe degradation from its target bandwidth as opposite to UDP (cases 1 and 6); however, depending on specific MRED thresholds set properties UDP can be allowed (overlapped set) or denied (staggered set) to participate to the excess share.

From these results, it appears as though fairness issues between TCP and UDP in a single AF class remain unsolved. In order to achieve fairness it appears as though TCP and UDP packets must be isolated from each other in terms of their drop precedence; however this is not sufficient: when RED operates in coupled mode, UDP packets in DP2 receive unfairly degraded service because their drop probability is dependent on the buffer occupancy of packets from DP1 and DP0. Therefore, the same mapping experiments where effectuated with decoupled drop decision, that is, for packets of each drop precedence marking is dependent only on the buffer occupancy of packets with their own marking. Although some of the results are beneficial for UDP, this happens, as expectable, at the expense of TCP in-profile traffic. Furthermore, increasing DP2 $\max_t h$, the service rate would increase accordingly irrespective of the actual traffic profile: even decoupling drop decisions will not solve the UDP/TCP fairness problem.

To summarize, the UDP and TCP interaction issues cannot be resolved completely using drop preference mapping, since:

- in an over-provisioned network, UDP and TCP traffic achieve their target rates if the in-profile traffic is protected, but the share of excess bandwidth is dependent on the mapping of out-of-profile packets

- in an under-provisioned network, if TCP and UDP share the same AF class, fair degradation for both traffic types cannot be achieved simply by different drop precedence assignments
- the specific RED model used impacts the way in which TCP and UDP traffic interact: due to the importance of active queue management, considerations are further developed in Sec. 5.7.

It could be argued that the real test of isolation and decoupling can only be done with 4 drop precedences where TCP in and out of profile are mapped to DP0 and DP1 with UDP mapped to DP2 and DP3. It must be said that, whether further study proved the effectiveness of such a solution, the resulting service would no longer belong to the AF PHB; on the other hand, it would be easy to "transpose" the codepoint space in order to distribute four different DP to three independent AF classes.

A better solution that allows UDP and TCP to coexist fairly is to put them in separate queues or AF classes. Such a scheme will not only address the fairness issues but will also restrict the delay and jitter that UDP packets experience in the queue: this should make AF more amenable for real-time services.

5.7 Traffic Conditioning Considerations

DiffServ architecture defines a framework and a set of primitives by means of which rather different kinds of service can be offered, without imposing any constraint to the specific implementation of forwarding path primitives. On the other hand, service efficiency relies on the behavior of these components (such as classifiers, meters, polishers, ...) and on the appropriateness of their configuration; moreover, although active queue management technique is of key importance to QoS building, nevertheless it is far from being properly understood.

The marker used in all the performed simulations uses the TSWTCM window based probabilistic tagging algorithm; a coarse set of configuration schemes were investigated, using either two or three drop precedences: this yields a different use of the MRED queue, in the sense that early drop action depends on each virtual RED queue parameter set.

PIR≡CIR
: The marker tags DS packets either as green or red, whereas BE packets are always red colored, being null the BE flow committed rate; this choice should give a reasonable forwarding assurance to DS green packets, allowing further BE to compete with DS out of profile traffic to the share of the excess bandwidth.

PIR≡Linerate=B
: The marker tags DS packets either as green or yellow, whereas BE packets are mapped to the red queue; due to the strictness of red queue thresholds with respect to green and yellow ones, this choice is supposed to mostly prevent BE traffic from getting excess rate share, contrarily to the previous case.

PIR≡PIR$_{opt}$ $= 1/2 \cdot (B + \overline{\mathbf{CIR}})$
: The marker tags packets using three different levels of drop precedence, optimizing DS drop performances irrespectively from BE; the optimization criterion ensures that DS packet yellow marking probability is not lower than red marking one when the traffic rate exceeds the peak rate threshold.

With respect to drop system metric performances, results shown in Tab. 5.18 confirm the TSWTCM configuration guidelines derived from the analysis developed in Sec. 3. Throughput (T) and early/late drops (respectively Ed/Ld) statistics are expressed in packets, while goodput metric is given as ratio over throughput (G/T); these parameters are reported in the table both per-codepoint (where `0` stands for BE DSCP and `1x` maps DS packets to DPx drop precedence) and aggregated (Σ), for different amount of ρ_{DS}% reservation.

From observation of the collected data it can be gathered that, when a single AF class is used to implement both AR and BE services, BE traffic yields starvation if the PIR is not set to its CIR lower bound; therefore, analysis of the data considered only this TSWTCM setting case. Moreover, although reaching comparable results with respect to PIR≡B, PIR≡PIR$_{opt}$ leads to achieve an higher DS throughput; this fact, coupled with the observation that the G/T ratio is identical in both cases and furthermore independent from ρ_{DS}%, results in an overall reduced drop amount and thus better link utilization. Finally, it must be noticed that RED effectiveness in green packet protection does not depend from the specific PIR setting: this is expected, since green marking probability only depends on CIR.

DSvsBE		PIR≡CIR				PIR$_{opt}$				PIR≡B			
		T	G/T	Ld	Ed	T	G/T	Ld	Ed	T	G/T	Ld	Ed
50%	Σ	170208	0.91	9022	5467	180303	0.96	2758	2726	176118	0.96	3629	2841
	0	35884	0.81	4479	2341	464	0.34	295	11	603	0.40	342	18
	10	90281	0.99	16	0	92371	0.99	17	0	91693	0.99	17	0
	11	-	-	-	-	86252	0.95	1404	2686	83822	0.92	3270	2823
	12	44043	0.82	4527	3126	1216	0.11	1042	29	-	-	-	-
75%	Σ	177828	0.93	7576	3636	180730	0.96	4302	1878	179879	0.96	4778	1799
	0	18032	0.75	2846	1509	408	0.30	276	9	415	0.30	280	8
	10	133884	0.99	20	0	135425	0.99	21	0	135134	0.99	22	0
	11	-	-	-	-	44502	0.87	3623	1866	44330	0.85	4476	1791
	12	25912	0.73	4710	2127	395	0.02	382	3	-	-	-	-
95%	Σ	180924	0.96	5862	1177	182360	0.96	5037	691	181568	0.96	5281	711
	0	3037	0.61	929	235	391	0.28	276	3	391	0.28	276	3
	10	165703	0.99	21	2	165810	0.99	24	31	164602	0.99	25	25
	11	-	-	-	-	16068	0.66	4646	657	16575	0.65	4980	683
	12	12184	0.52	4912	940	91	0	91	0	-	-	-	-

Table 5.18: Ingress Queue Packet Drop Statistics

It has been noticed in [AFEVAL] that probabilistic marking is problematic for metering a CBR source: since the profile meter uses an average rate estimator, this would allow the source to transmit at a sustained rate higher than the contracted one, whereas token buckets do not permit this. However, when both responsive and non responsive traffic sources share the same AF class, fairness among both types of sources cannot be provided by simple deployment of token bucket based meters (such as srTCM or trTCM). Moreover, the claimed superiority of the token bucket is "due to its permitting transmission of a deterministic burst of IN packets, whereas an average rate estimator probabilistically marks some packets beyond the target rate as IN and some as OUT": therefore, a correctly configured token bucket should allow for the natural burstiness of TCP by marking as IN all the packets within a burst, while with an average rate estimator some packets will be marked OUT giving them drop preference. The above considerations should however be coupled with the fact that marked traffic is input to RED queue management algorithm, which uses randomness in early drop decision: therefore, a deterministic marking does not ensures packets to be forwarded even though they are green marked; moreover, considerations of the token bucket effectiveness on TCP bursty traffic cannot be gathered without consistent analysis of network performances, either via experimental testbeds implementation or intensive simulations.

Implementation of a traffic conditioner not only involves the choice, e.g., of the marking scheme, but stems also the need for investigation of the implemented queue management algorithm behavior. Limiting our consideration to file transfer applications, in the context of a Best Effort network, TCP adjusts its sending rate according to network capacity and traffic load. Recently,

several studies including [TCPMOD][TCPCON], have proposed increasingly accurate models for the performance of TCP as a function of network and endhost conditions. According to the model defined in the former paper, the average throughput of a TCP flow is modeled as a function $T(p, R)$ of p network packet drop probability and of round trip time R. Let us indicate with $p = H(q)$ the packet drop probability law, increasing as long as the average estimated queue size q increase; obviously, the congestion control mechanism of TCP interacts with the RED algorithm, and analytical models have been proposed recently in [AQMSTDY][REDFLU]. Following the latter model, in a simple setting with n identical TCP flows having round trip propagation delay R_0 and traversing one congested link of capacity c, the average throughput of a TCP flow is c/n, and the average queue size q and packet loss probability p are the solutions to the system:

$$\begin{cases} T(p, R_0 + q/c) &= c/n \\ p &= H(q) \end{cases}$$

Denoting by $T^{-1}(p, t)$ the inverse of $T(p, R)$ in R, thus $T^{-1}(p, T(p, R)) = R$ the first equation of the previous system can be rewritten as $q = c \cdot (T^{-1}(p, c/n - R_0)$. Furthermore, whether we denote the queue law as $G(p) = c \cdot (T^{-1}(p, c/n - R_0)$, then the system becomes:

$$\begin{cases} q &= G(p) \\ p &= H(q) \end{cases}$$

In Fig. 5.13, the intersection of the queue law function and RED control function at P represents the average operating point of the traffic: in other words, the TCP traffic experiences on average the loss rate and queuing delay given by P. It also follows that a given traffic set has a unique queue law function, and therefore, for different RED control functions, produces different operating points along that queue law function: we can thus define a range of queue management policies such as delayconservative and dropconservative. Under the dropconservative policy, the traffic has a small loss probability, but a relatively large delay (P_{drop}), whereas under the delayconservative policy, the traffic has a short delay but a relatively large loss probability (P_{delay}); this obviously means that there is a trade-off between drop and delay, which cannot be contemporary optimized.

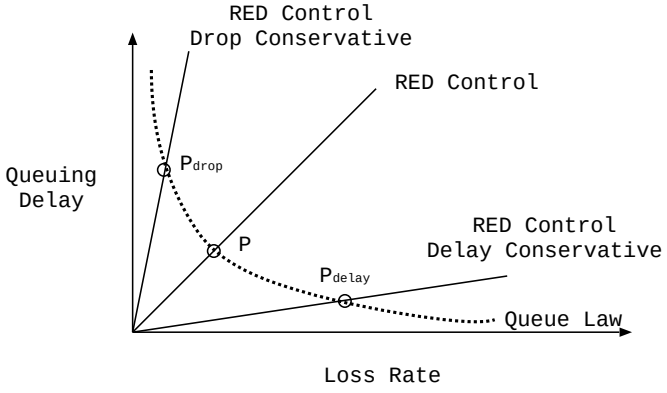


Figure 5.13: Queue Law function and RED Control Function for TCP traffic

Moreover, parameter choice in RED seems to be hard, since the inventors periodically changes their recommendations [RED],[REDLIG]; even more, Van Jacobson recently claimed [REDNOTE] that any parameter setting, as long as the control law is monotone, non-deceasing and covers the full drop rate range will improve system performances.

Although RED have been the subject of much study in the past years, however there is no unanimous concordance on the benefits brought by RED deployment. Moreover, significant number of published RED papers are based on simulation studies: while simulation is a core tool for

network protocol investigation, the traffic generated by most of the simulators is quite different from real network traffic (e.g. our long lived flows simulations used a constant number of infinite greedy TCP sources). Therefore is important to couple simulative approaches with results issued from testbed experiments, where the traffic more closely approximate real network scenarios. An example of such experimental studies is given in [REDNOT]; there, the authors' express concerns to benefit of RED, asking "why would I drop perfectly good packets when there is no obvious reason to do so, and why would I change a really simple mechanism (i.e. tail drop) that works perfectly for a more complex mechanism for which I have no proof it works better"? The results obtained via testbed experiments showed that RED with small buffers does not improve significantly the system performances: in particular, the overall throughput is smaller than with Tail Drop (TD) and the difference in delay is not significant. Summarizing the results:

- TCP performance does not seem to be simple function of max_p setting, and a variation of the dropping interval, thus the difference between max_{th} and min_{th}, does not influence the system performance for TCP as well as for UDP traffic; moreover, TD penalize non-responsive traffic to a greater extent than RED.

- Neither TD nor RED results in fair throughput for all connections being far from solving synchronization problems; that is, RED still allows one or few flows to achieve a throughput many times greater than their target rate, while other connections are prevented from reaching their target.

- Queue length averaging, claimed to be a very important feature of the RED algorithm, since it should avoid the bias against bursty traffic; indeed, results show that RED increases the drops for bursty traffic with respect to TD, and moreover the influence of the weighting factor is minimal or not visible.

- Since RED parameter tuning is already complicated in a single router case, it is obviously more complex in a heterogeneous multi-router and multi-ISP environment: probably, a single misconfigured router can determine the performances of the whole network (e.g. using a setup degrading UDP traffic while others ISPs are interested in increasing UDP throughput).

- nevertheless, RED allows to control the queue size with large buffers: increasing the buffer size, the number TCP packets sent is increased; however, the bigger the buffer the fewer drops occur for UDP traffic: contrary to the expectations, drop rate for UDP is much higher with TD than with RED.

It may be argued that, due to the dynamics of the parameters that influence RED, a static configuration cannot provide better results than TD in the general case; this would be anyway difficult to test, since RED mechanism is far from being clearly understood. While it has been noticed that DS services realized through RED partially correct the TCP dependence on the factors earlier studied (this strike roots from flow penalty via packet drop, when they opportunistically transmit beyond their contracted rate), there is agreement on the fact that RED is not able to provide faultless service guarantees and furthermore that RED behavior is far from being fair.

A possible solution would need to keep per-flow state variables, tracked by performing extensive measurements, and further necessitating of communication among traffic conditioners; this seem to implicitly mean that efficient QoS architecture cannot be implemented without state var tracking: a sort of denial of basic DiffServ principles. In the author's personal opinion, signalling should be used only in dynamic SLA accounting via BB agents, whereas traffic conditioners should be kept stateless, in order not to introduce excessive overhead and not to raise scalability concerns.

Conversely, there are a number of stateless marker proposals intended to solve the fairness problem; however, each of them focuses on mitigating the unfairness due to a specific factor, without being able to contemporary solve all the AQM draw backs. Among these appealing approaches, we may cite the Stateless Prioritized Fair Queueing [SPFQ], the Core Stateless Fair Queueing [CSFQ], a Fair Traffic Conditioner [FTCON] and a TCP-friendly marker [FMARK]: such proposals should be intensively studied to test whether they can yield better and fairer system performances, with respect to RED, in real network situations.

5.8 Conclusions

The first set of simulation results give insight into the workings of an Assured Rate like service, build on the top of the Assured Forwarding PHB group; nevertheless the topologies used is quite simple and only longlived connections were simulated: simulations are thus optimistic in that they reflect only the simplest network dynamics. However, basing on the gained experience and on [AFRES], we may express some interesting final considerations from a Service Level Agreement point of view. In order to understand the kinds of end-to-end services that could be created using the AF PHB group, it is necessary to determine which SLA parameter can have quantitative assurance; three common performance parameter included as a part of a SLA are:

Latency
Guarantees on end-to-end delay bounds for virtual private networks crossing multiple DS domains remains a possibility in the one-to-one or one-to-few cases; however, this require the knowledge of delay pattern for AF traffic within the network, and furthermore every transited domain must know the VPN profile: an intra-domain signalling protocol is clearlyy needed. In the case of one-to-any VPN, the total delay depends on the delays incurred in a variable number of intermediate domain, and cannot therefore be bounded a priori.

Drop Probability
Drop probability control may be possible in a single DS domain, via bilateral agreements between two peering domain: e.g., depending on a committed traffic rate threshold, in- and out-of-profile packets may be assured, respectively, of $x\%$ and $y\%$ maximum drop probability with $x\% < y\%$; however, it is not clear how the end-to-end drop probabilities of aggregate flows crossing multiple domain along their path in the internet can be ensured for a customer.

Throughput
In the previous sections has been shown that, depending on other traffic and specific topologies, a subscriber may get more or less than the contracted rate. Therefore, it is unlikely that the subscription rate can be viewed as a hard contract, in the sense that AF does not allow a strict allocation of bandwidth between several users. A possible solution to this problem would be to include, as a part of the SLA, a parameter quantifying the compromised level of statistical assurance: e.g., target rate x can be sold with the further specification that the service is guaranteed at $y\%$. More precisely, in over-provisioned networks, target rates are achieved but there is unfair share of the excess bandwidth for equal paying customer; moreover, excess will be shared irrespectively from target rates, rather than proportionally to them. Finally, in under-provisioned networks, there is unfair performance degradation for equal paying customers. Therefore, it appears as though SLA contracts should try to avoid specific sharing of the excess bandwidth in an over-provisioned network as well as stating how bandwidth would be distributed in an under-provisioned network.

For the above described difficulties in providing quantitative assurances, services of the Assured Forwarding PHB group have largely been described as Better Effort. Looking at AR in a more general way, it appears it can assure that any packet marked as IN will reach its destination with a higher probability than an unmarked packet as long as all the networks involved in the transmission are adequately provisioned. Moreover, when the total offered load is close to the line rate, system performances on a DS capable domain are effectively Better than Best Effort; with this we means that DS is less sensible, with respect to BE, to the network factors affecting TCP behavior, and furthermore that drop probability and link utilization system metrics benefit of DS deployment.

Altough AF services have been evaluated with different approaches, i.e. they have been implemented with differentt traffic conditioners assuming differentt policies, nevertheless DS performances are still sensibly affected by the studied factors. There is an actual large number of

possible schema and variations of them that may contemporary solve some of the unfairness factors; however, each differentt scheme need intensive simulation and should be further coupled with testbed experiments, in order to consider a more realistic and wider traffic range.

Focusing on AR service, we gathered that DS domain might be able to provide fairness when offering really differentt SLA amounts. Therefore, whether an ISP implement an AR service, it would be likely to offer a few AR amount, e.g. $\rho_{DS}\% \in \{5\%, 10\%, 25\%\}$ further indicated with SLA_5 SLA_{10} SLA_{25}: when the differentt SLAs are known a priori, thus in a static SLA configuration context, each SLA should be differenttly conditioned at the edge. Due to the smaller SLA_5 opportunism, it would be suitable to further pose some constraint on the peak rate that it should be able to reach; e.g., when the marker is either a srTCM or a TSWTCM, SLA_5 may have PIR=CIR, whereas SLA_{25}'s PIR may be set to PIR_{opt} and SLA_{10}'s one may be set halfway between CIR and PIR_{opt}. These examples are not quantitatively meaningful yet, but they show possibly successful scheme in order to achieve a fair, proportional share of the excess rate.

Moreover, packet size unfairness may be solved by adopting a common size for packets crossing the DS domain; in fact DS routers should not reorder packets of a same flow, but no constraints obviously applies to datagram fragmentation. Therefore the possibility that some companies deliberately decide to use bigger packet sizes to their own profit can be prevented by fragmentation of packets exceeding a certain threshold in routers; this would steam however further difficulties in multi-field classifiers, and would have no impact on assuring better performances to packets whose size is well below the threshold. Alternatively, RED queue size may be computed in bytes rather than in packets; however, it should be investigated how this choice would differenttly affect UDP short packets (40 bytes) and TCP longer ones (500-1000 bytes).

The unfairness of TCP throughput performance due to RTT and μ-flow dependence seems to be harder to correct. Specifically, RTT cannot be known a priori and can be only estimated in real networks: this may be solved in the case of VPN-like one-to-one or one-to-few AR services by developing intelligent marking schemes, therefore needing state variable tracking; however, when the SLAs are statically configured, the ISP may decide to handle each company with a differentt marker configuration (e.g. variating the PIR): a proper SLA setting may be found via measurements of performances issued with differentt configuration schema. Moreover, the meter should be aware of the the μ-flow number measure, in order to be fair with respect to differentt companies contracting the same SLA: this would be almost impossible in real network situations where the μ-flow number is highly variable, especially when traffic is partly constituted by HTTP flows.

To introduce important considerations on BE and DS coexistence, it may be useful to underline that "we are motivated to provide services tiers in somewhat the same fashion as the airlines do with first class, business class and coach class. [...] A part of the analogy we want to stress is that best effort traffic, like coach class seats on an airplane, is still expected to make up the bulk of internet traffic" [RFC2638]. It is thus clear that DS traffic must not prevent BE traffic from being forwarded; however, this is quite likely to happen, even whether just a small portion of the DS capable domain bandwidth is reserved: e.g., in some simulation we observed that DS flows gained by competition 85% of the line rate when only 25% of it was reserved. Therefore, BE traffic cannot be simply forwarded as no-profile traffic beside AF protected traffic: rather, DS should map the Class Selector codepoints to some service meeting the requirements.

A possible solution would involve MRED setting tuning; in fact, it has been noticed that RED is able to control the queue when the buffer size is quite large, but this happens at the expense of BE traffic: most of BE drops are due to the fact that early drops actions are possibly taken when only 5 red packets occupy the buffer, independently from the actual buffer size. This has the effect that BE flows have a reduced possibility to open their window, unlike DS flows, even when only a small amount of bandwidth is reserved; therefore, BE cannot be simple mapped to a queue with the same setting as DS red one; however, in order to ensure green packet protection, DS red max_{th} should not be enlarged, whereas it might (or might not, due to MRED setting difficulty) be suitable for BE one.

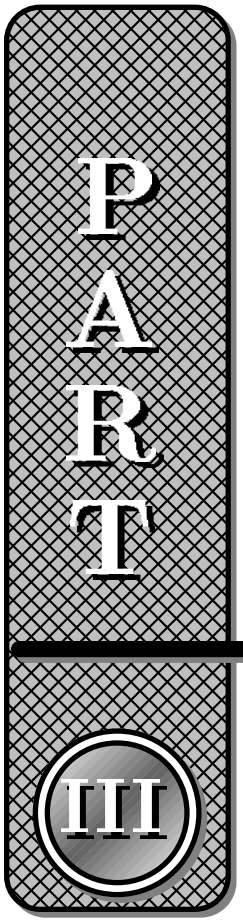
Another possible approach could imply for an AF class to be destined to carry Best Effort traffic, the others classes being available for the Better Effort one; furthermore, it would not be

necessary to reserve all the free bandwidth beyond the contracted one, but just a certain amount of it. The BE meter may be configured such as the aggregated BE bandwidth is equally shared among the number of active BE connections: this would yield better aggregated and fairer per-flow BE performances, assuring that BE get a minimum amount of network resources in the case that DS aggregate are sending high above their contracted rate. Moreover, independent forwarding of BE and DS this may result in benefits to DS aggregates too: DS flow penalty due to red buffer overflow would be no longer influenced by the presence of BE traffic in that queue; rather, DS red drops will be due only to DS opportunism, and so for BE. However, it should be noted that whether the downstream peering domain is non DS capable or DS capable but supporting differentt policy assumptions, then BE traffic mapped to the Best Effort AF queue should be remarked with the default DSCP. Although giving differentt level of protection to BE traffic may seem a counter-sense, indeed it should be noticed that DS architecture relies on the primitive IP support of differentt forwarding priorities via the TOS byte of the packet header. Moreover, Class Selector PHB is required to yield at least to two differentt forwarding precedences: differentt mapping schemes are available (DP0 and DP1, or DP0 and DP2, or DP1 and DP2, or just DP1, ...) to satisfy this exigency. Therefore, it may be worthy to test such approaches via Experimental or Local Use PHB.

	TCP AF	UDP AF	BE AF	Free AF
AFx1	000111	010111	100111	110111
AFx2	001011	011011	101011	111011
AFx3	001111	011111	101111	111111

Table 5.19: Implementing AR PDB via an Experimental AF PHB Group

Up to date, there is only one Per-Domain Behavior whose study is being considered: the Assured Rate PDB; indeed, there are three more AF classes available to three other inexistent services. This may justify the proposal of dedicate one AF class to TCP traffic and another to UDP one; in fact, although service offer should not impose constraint neither on the kind of application nor on the transport protocol used, however it has been proved that TCP versus UDP fairness problem cannot be solved in a straightforward manner whether they share the same AF class. The idea of using two classes depending on the transport layer, seems less insane when considering that its only valid alternative consists in using per-flow state tables, possibly needing intensive signalling; At least, this would not compromise scalability, would be immediately implementable and would not introduce overhead at the edge apart in the MF classifier; furthermore, this would not compromise the ability to re-implement the service whether the research produces an effectively and efficient TCP friendly marker. Tab. 5.19 depicts the draft scheme of the AR PDB via and Experimental AF PHB group including BE, TCP, and UDP service classes; the recommended codepoints, drawn from Pool 2 in conformity with [RFC2474], are given just to illustrate a possible DSCP mapping scheme. However, such a PHB is defined just in order to stress the importance of the considerations developed in this section.



Short Lived HTTP Flows

Chapter 6

HTTP Simulation Introduction

Web traffic represents the most dominant subset of TCP connections on the Internet today: measurement studies have shown that the majority of TCP connections are HTTP flows and many of these connections are quite short-lived, often of the order of a few TCP segments; nevertheless there are currently few models of HTTP traffic, and it is likely the case that Web traffic dynamics are evolving faster than our current ability to measure and model the traffic. Furthermore, although DiffServ architecture performance is the subject of much on-going study, evaluation has largely focused on network-centric measures such as network link utilization on aggregate TCP throughput, as for the long lived flows simulations early discussed. Thus, since Internet performance is becoming synonymous with Web performance, understanding the effects brought by DiffServ architecture deployment on HTTP traffic using a user-centric measure of performance –i.e. HTTP flow's completion time– is a research area as important as unexplored: it is in this perspective that Web-like traffic simulations have been conduced in a DiffServ network.

Organization of this chapter is as follows: firstly, Sec. 6.1 briefly introduces an undocumented *ns*-2.1b8a HTTP traffic model and the motivations that led to the implementation of a new one, whose description is given in Sec. 6.2 with the purpose to introduce and clarify the analysis of simulation results, leaving aside any simulator-specific consideration. The details of its implementation are given in Appendix C, which focuses on model motivations as well as code engineering and both its applicability and effectiveness from system requirements point of view. Then, Sec. 6.3 focuses on the simulated network topology, while simulation-time and time-scale related considerations are developed in Sec. 6.4. Finally, Sec. 6.5 introduces the notation used in results analysis discussion and gives an overview of each specific simulation "suite".

6.1 *ns* HTTP model

Besides Web Cache model, *ns*-2.1b8a implements HTTP traffic based on Mah's empirical flow model developed by at Berkeley University [HTTPMOD], whose short description have only the goal of introducing the alternative approach used.

The model is an application-level description of HTTP protocol usage in terms of its critical elements, based on empirical data distribution extracted from more than 230 hours of network traces; these distributions are used to determine a synthetic workload, generating transfers that need to be run through TCP's algorithms. The model does not generate packet sizes and interarrivals by itself: rather, it captures logically meaningful parameters of Web client behavior:

– HTTP request/reply length
– Number of files per document
– Time between retrieval of two back-to-back documents
– Number of consecutive documents retrieved from any given server

The transfer of a single document, characterized by the *number of files* that it embeds, may need to employ multiple HTTP transactions, each of which requires a separate TCP connection carrying one request and one reply. Between Web page retrievals, the user behavior is modeled based on the "think time" measured; moreover, assuming that users tend to access strings of documents from the same server and that all the components of a Web document come from the same server, the *consecutive document retrievals* distribution determine the number of consecutive pages that a user will retrieve from a single Web server before moving to a new one. Finally, the *server selection distribution* defines the "relative popularity" of each Web server (in terms of how likely it is that a particular server will be accessed for a set of consecutive document retrievals).

The main motivations that led to the implementation of an alternative model can be identified by two crucial aspects:

result readability: in order to sound the impact of DiffServ architecture deployment on short-lived TCP traffic, the alternative model uses a coarse set of flow-lengths, simplifying the analysis focused on traffic treatment disparity among flows of different sizes;

simulation system requirement: short-lived flows simulations need a long time scale in order to reach a reasonable level of confidence on the results obtained; the dynamic creation of sources instance for every flow within the simulation will surely result in poor simulator performance; additionally, a mechanism performing per-flow TCP and DiffServ statistic collection is clearly needed in order to avoid managing of huge trace files.

6.2 HTTP Simulation Model

The HTTP model developed foregoes any of the previous client-based behavioral parameters; instead, a Flow Generation Model determines both a Flow Size and a Flow Scheduling Time for every *HTTP Flow*, which is, from a *ns* perspective, simulated via an FTP application starting, at a given time, to send a certain amount of packets over a TCP source to a TCPSink; an HTTP flow can thus be considered as a mere HTTP datagram transfer, rather than a Web browsing like HTTP connection.

An *HTTP Cloud* is an aggregate of HTTP flows, all having common (source,destination) nodes pair, SLS and initial DSCP, being thus subject to a common traffic conditioning. The HTTP flows belonging to the same cloud are equally distributed among a certain number of *ideal* routers, modeling either a traffic multiplexer (source) or demultiplexer (receiver). The routers are said to be ideal in the sense that their links and buffers characteristics are engineered not to influence the DS domain topology, by neither introducing a further bottleneck element on the end-to-end path, nor additional RTT and undesired drops (see Sec. 6.3).

6.2.1 HTTP Flow Length

The length λ of flow i is determined –through an integer uniform randomized variable x_i used as vector index– from an empiric distribution vector Λ, while the interarrival time ϕ_i is determined according to a Poisson process.

$$\begin{cases} \overline{\lambda} = 13.6\ KB & \text{Average flow length} \\ \overline{\tau} = \rho \cdot B/\overline{\lambda}\ sec & \text{Average interarrival time, where} \\ & \quad B \text{ is the line rate} \\ & \quad \rho \text{ the offered load} \\ \lambda_i = \Lambda[x_i] & \text{Flow } i \text{ length, with} \\ & \quad \Lambda \text{ vector storing the empirical flow length} \\ \tau_{i+1} = \overline{\tau} \cdot \phi_i & \text{Next HTTP flow arrival time, with} \\ & \quad \phi_i \text{ exponentially distributed with average 1} \end{cases}$$

Being the packet size equal to 1 KByte over all simulations performed, the number of packets π_i that must be sent to achieve transmission completion of a flow of length Λ_i bytes, results in the coarse set reported in Tab. 6.1. Although Pareto-based flow length models have been implemented, the following sections only discusses simulation results issued from the use of the previously defined empirical model.

i	Flow Length Vector Λ	Packet To Send π
0	61	1
1	239	1
2	539	1
3	1349	2
4	2739	3
5	4149	5
6	6358	7
7	10910	11
8	19878	20
9	90439	91
avg	13660	13.66

Table 6.1: Coarse Flow Length and Corresponding Packet To Send Set

6.2.2 HTTP Interarrival Time

Packet arrival is determined according to a Poisson process: this model does not match empirically derived models of TCP [HTTPMOD] nor other long-range dependent bursty pattern models reproducing the self-similar nature of web traffic; however, it provides a simple means to investigate the impact of DiffServ on bursty traffic.

The average flow interarrival time $\overline{\tau}_i$ of cloud i depends both on the total normalized offered load ρ of the network and that particular cloud ρ_i fractioning of ρ, with $\sum_k \rho_k = 1$. Being the average flow length $\overline{\lambda}$ expressed in bytes and the bottleneck link rate B expressed in Mbps, the following formula expresses $\overline{\tau}_i$ in μs:

$$\overline{\tau}_i = \frac{\overline{\lambda} \cdot 8}{\rho_i \cdot \rho \cdot B}$$

From previous definition we recall that the offered load from each cloud is $\rho_i \cdot \rho$: in case a single HTTP cloud is used in network traffic simulation, its load would then equal the bottleneck ρ, thus $\rho_1 = 1$.

6.3 HTTP Simulation Scenarios

HTTP Cloud mechanisms can be applied to a wide variety of network topologies. The short-lived TCP traffic simulation performed focuses on a basic network bottleneck, simplification of the one used previous long-lived traffic simulations.

From a DiffServ point of view, the topology used can be identified with the simplest DS Region consisting of two simple DS Domains: the upstream S source domain and the downstream R receiver one, both containing one single node identified by the same letter of the respective domain. The HTTP cloud(s) source/receiver pair are peered respectively to the source node S of the upstream S domain and the receiver node R of the downstream R one.

Packets originated by any cloud sources, in their path form S to R, cross a DiffServ Edge router E, which acts both as egress edge for the traffic leaving the upstream domain from node S and as ingress edge for the traffic entering the downstream domain to reach their final destination through R.

Each cloud generates packets that will be initially marked with just one DS codepoint: its traffic represent a Behavior Aggregate with respect to E, or by extension, a PDB Traffic Aggregate when considered in a domain context. The backward path is only used by TCP acknowledgment packets, and the rules for its configuration are identical to those for the forward path; however, since losses will occur to ACKs, the backward traffic will not be further discussed nor analyzed.

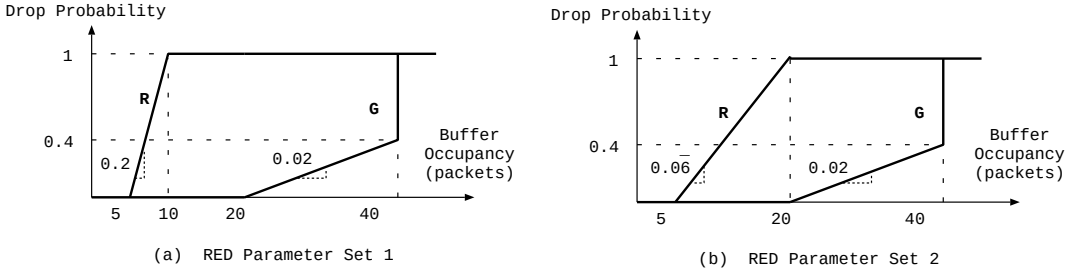


Figure 6.1: Ingress Edge RIO Parameter Set

The E router serves its queues, whose size is fixed to 100 packets, in a Round-Robin fashion and deploys RIO-C active queue management technique, whose RED parameter sets are chosen between the *Staggered* and *Boundary Staggered* ones presented in Fig. 6.1(a) and Fig. 6.1(b) respectively. Being the router queues equally weighted, both the resource provisioning and TCB functions are implemented uniquely by means of a TSWTCM, whose settings must then reflect the traffic agreements imposed by any established SLA.

6.3.1 Single-Cloud Scenarios

A cloud may generate only HTTP traffic: with respect to the previous topology and HTTP traffic model description, a Single-Cloud network scenario could be depicted as in Fig. 6.2, where the links characteristics will be common to all the simulations performed.

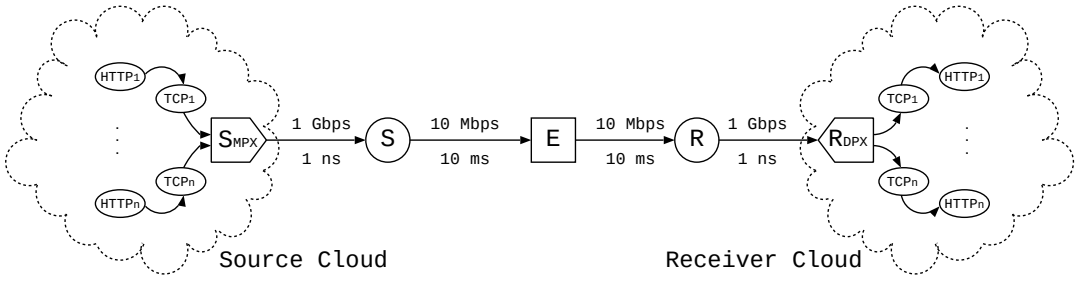


Figure 6.2: Single-Cloud HTTP Traffic Topology

The number n of HTTP sources that will arise in a simulation can't be exactly known *a priori*: it depends on source re-use, thus n means the maximum number of contemporary active

sources during the whole simulation, depending, among others parameters, from interarrival time as well as simulation duration; although a probabilistic bound can be analytically determined (see Appendix C for details), this will not nevertheless affect the simulations, being the sources dynamically created when needed.

A first level of traffic differentiation concerns the use of both long and short-lived TCP traffic within the same simulation run; interaction of HTTP and FTP applications sharing the same DiffServ profile is supplied by adding to a cloud a given number of infinite traffic FTP background sources. The per-cloud FTP sources number N is distributed among the M router of that cloud; for instance, when a simulation instantiates FTP background traffic sources, the FTP parameter set used is $N = 10$ and $M = 1$, unless otherwise specified. The single-cloud topology scheme in presence of long-lived TCP flows is presented in Fig. 6.3.

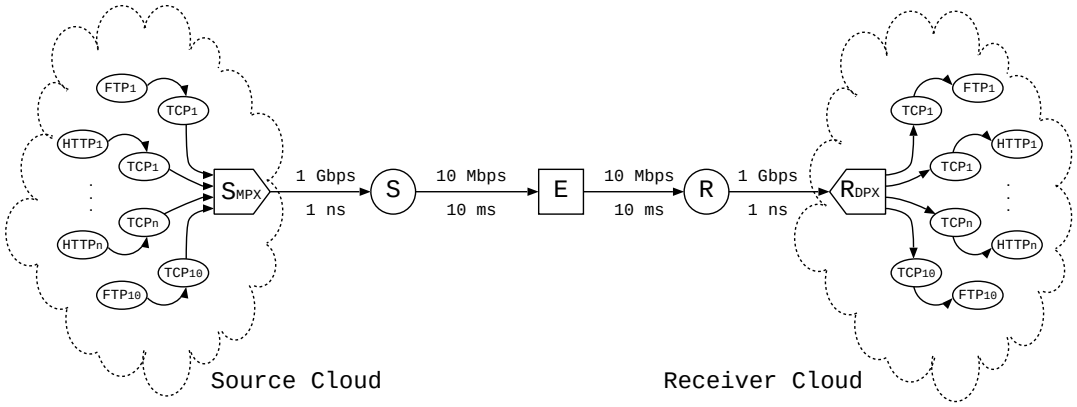


Figure 6.3: Single-Cloud HTTP,FTP Traffic Topology

6.3.2 Two-Cloud Scenarios

Two clouds "objects" are necessary to enhance further traffic differentiation, being each cloud only capable of traffic generation constrained to a fixed source/destination pair, initially marked with an unique DSCP, and committing a common SLA.

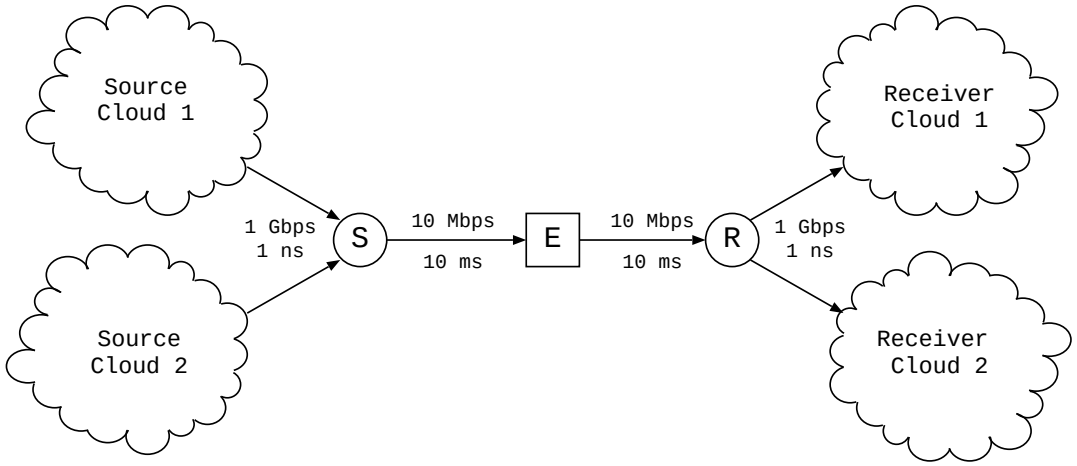


Figure 6.4: Two-Cloud Topology

The Two-Cloud extension of the previous scenario will be basically used to allow mixed DSCPs traffic simulation, in order to compare traditional Best Effort short-lived traffic to DiffServ-enhanced one, either in the presence or in the absence of background infinite TCP traffic defined in the previous section: such a scheme is presented in Fig. 6.4, where each traffic cloud is considered as a black box.

6.4 HTTP Simulation Timing

All the network simulations performed, as well as the results sampling, are based on the considerations developed in this section.

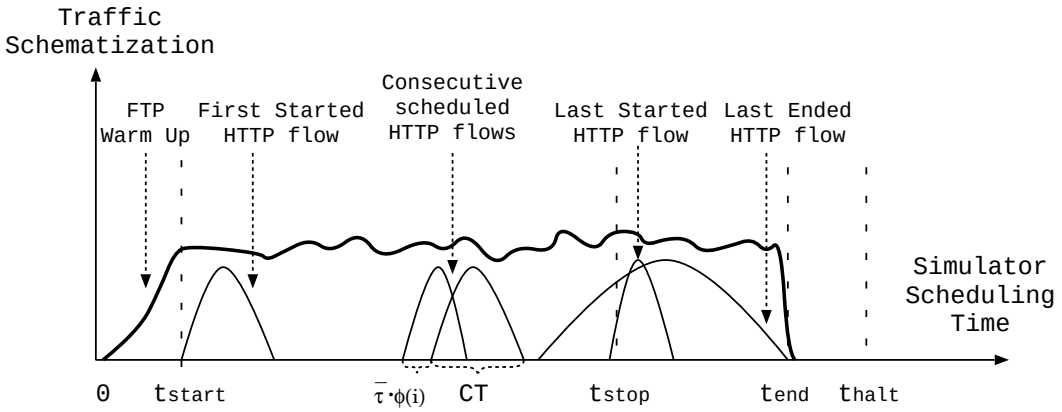


Figure 6.5: Traffic Scheme vs Simulation Time Parameters

Fig. 6.5 depicts a Scheme of the time evolution of simulated traffic, regardless of any meaningful measurable traffic characteristic. The x-axis represents the simulator scheduling time, without any purpose of scale precision; the y-axis may be interpreted as the bandwidth used by the aggregated FTP traffic, while the HTTP traffic "bell" representation is just used to achieve visual differentiation of the schemed traffic.

Each "bell" is used to represent a single HTTP flow, which we will refer to as HTTP flow-bell; in the figure, only a few HTTP flows, crucial to this explanation, are depicted. As previously explained, the height of the HTTP flow-bell, as well as the area that it covers, are meaningless; the intersection of the flow-bell will the x-axis results in two points which represents the starting t_s and ending t_e time instants of the HTTP flow; the amplitude of the bell "base", thus the length of the segment delimited by t_s and t_e, represents the HTTP flow Completion Time (CT). HTTP traffic scheduling begin at t_{start}, and no new flow will be scheduled after t_{stop}, while the scheduling delay between two back-to-back scheduled flows is determined according to the interarrival model. Simulation will run until the minimum between t_{end} and t_{halt}, where these two parameters represent respectively the end time instant of the last ended HTTP flow and an upper bound to simulation duration fixed *a priori*; t_{end} evidently depends, among other factors, on network congestion level and its value can be either lower or greater than those of t_{halt}: the former case is depicted in Fig. 6.5, where thus $t_{start} < t_{stop} < t_{end} < t_{halt}$; if the simulation finishes at t_{halt}, then t_{end} is undefined, as *at least* one of the flows which started sending data before t_{stop}, has *not* completed its transmission.

The FTP sources starting time is close to the time axis origin $t = 0$ s, but for each individual source, in order to avoid synchronization phenomena, a uniform random shift time, whose maximum can be $t_{start}/10$, have been added. FTP traffic provides a network warm-up phase before HTTP traffic begin to be scheduled at t_{start}, and its duration will be no shorter than t_{stop}: FTP sources will stop sending data either at t_{end}, thus at the same time as last ending HTTP, or at

t_{halt}.

The HTTP flows results are collected on a per-flow basis every time that a flow completes its transmission. Computation of per-flow-length and flow-length-independent data averages (and possibly collection of FTP sources data) is performed at two time instants: either (t_{stop},t_{end}) or (t_{stop},t_{halt}), depending on each simulation's specific issues. This is done to ensure a good level of confidence to the simulation results, mainly by observing whether the relative variation of the flow's completion time is kept under a reasonable order; this may turn out to be critical, especially for Best Effort traffic, when total normalized offered load is close to 1, even more whether FTP background sources are present.

In order to achieve a certain degree of confidence with simulation results, the time scale considered involves t_{stop}=600 (minus one) simulated seconds of HTTP traffic starting at $t_{stop} = 1$ s, and the simulation forced-halt instant is set to t_{halt}=900.

6.5 HTTP-Related Notation

The simulations results presented in the next chapter aim to analyzing the performance of mixed Best Effort and DiffServ TCP traffic in the simple network topology earlier described, using TCP Reno at the transport layer. Though these TCP sources produce either short-lived flows or a mix of both short and long-lived flows, the analysis will focus on HTTP flows performance.

The DS region offers a unique DS service, build on top of a one-to-one Assure Rate PDB, implemented using a single class within the AF PHB group; specifically, the SLS part of its SLA is specified uniquely in terms of the aggregated reserved bandwidth: in the following, SLA_x will indicate that the x percent of the network's (i.e. the bottleneck link's) bandwidth B is reserved to the AR traffic, and thus the resources available to the Default PHB represent the $(100-x)$ percent. Since the edge router serves its queues in a Round-Robin fashion, the allocation of the resources satisfying the SLS are guaranteed uniquely by means of the TCB; this consists of a TSWTCM with CIR=PIR=$x \cdot B/100$, which marks packets conforming to the subscribed SLA as Green, Red otherwise. This implementation of the PHBs, although simple, is coherent with the assumption that the traffic type is either AR or BE: flows are granted x and $100 - x$ percent, respectively, of the totally available bandwidth and they are both entitled to participate, when possible, to a share of the excess bandwidth.

The DS cloud's sources initially mark the generated packets with the AFx1 codepoint, which corresponds to the lowest drop precedence level; since only two levels of drop precedences are implemented, packets whose codepoint is AFx2 (or AFx3) have the highest drop precedence. The BE cloud's sources mark their packets uniquely with the Default DSCP, correspondent to the Default PHB, realized by treating BE traffic with the same rules used for the lowest drop precedence packets.

In order to examine different aspects of the AR and BE traffic interaction, several simulations were performed, characterized in terms of:

SLA_x	Indicates that x percent of the bottleneck bandwidth is reserved to DS cloud's traffic
ρ	Total normalized offered load
ρ_{DS}	DiffServ cloud's normalized offered load
ρ_{BE}	Best Effort cloud's normalized offered load, equal to $1 - \rho_{DS}$
Φ	Number of generated flows
ϕ	Maximum number of flows active at once

Several SLA_x were tested bysimulation, and, keeping the same SLA, either ρ or ρ_{DS} were varied, depending on the kind of investigation. Both Φ and ϕ depend on flow interarrival time, determined via total and per-cloud normalized loads; however, while the former can be quite

accurately determined a priori, this task becomes much harder for the latter: a fluid model can only give a rough estimation of the *average* number of contemporary active flows, since transport-layer algorithms, network load and simulated traffic type play an essential role in its upper bound determination.

In the first scenario examined all the rates were allocated to just one DS cloud (SLA_{100}); various simulations were done for different bottleneck load, the traffic being either HTTP or a mix of HTTP and FTP flows. Then, the comparison of Best-Effort-only traffic (SLA_0) to SLA_{100} performances for a given load, permits to introduce some considerations about the effectiveness of active queue management. Further investigation on BE and DS coexistence in a two-cloud scenario were simulated for three different SLAs (SLA_{25},SLA_{50} and SLA_{75}) and various network loads, and the overall performance eventually compared. Finally, in the case of SLA_{50} and for given $\rho \in \{0.85, 0.975\}$, each cloud's load were varied, and the results analyzed in terms of ρ_{DS}.

Flow statistics were collected for each cloud, during the simulation, on a per-flow basis; with respect to the analysis further developed, every HTTP flows is characterized in terms of:

π		Number of packets to send
Π		Number of packets transmitted at the source
Cwnd		Maximum value of TCP congestion window
Drop%	$= \text{Drop}/\Pi \cdot 100$	Dropped over transmitted packets percentage
E-Drop%	$= \text{E-Drop}/\text{Drop} \cdot 100$	Percentage of early drops over total drops
L-Drop%	$= \text{L-Drop}/\text{Drop} \cdot 100$	Percentage of late drops over total drops
Time Out%	$= \text{RTO}/\text{Drop} \cdot 100$	Percentage of drops due to TCP's RTO expiration
Fast Recovery%	$= \text{FR}/\text{Drop} \cdot 100$	Percentage of drops due to fast recovery

Flow properties were collected separately for every flow, then each parameter's average were calculated both on a per-π basis and over all flows independently of their length; finally, as rough comparison of DS and BE performances, the ratio of DS over BE parameter value were computed.

Chapter 7

HTTP Simulation Results

7.1 SLA_{100} Simulations

In these scenarios the network is populated only by DiffServ TCP traffic, to which the whole 10 Mbps bandwidth is allocated; in the first case of our study, the traffic is produced exclusively by HTTP sources, later put side by side with ten background infinite FTP sources. The network load is controlled only on HTTP traffic volume (i.e. $\rho_{DS} \cdot \rho = \rho$), while packets produced by the FTP sources are not subject to any form of load control.

7.1.1 Case 1: HTTP Traffic Only

The HTTP flows performance can be described observing that network behavior can be divided into three macroscopic phases with respect to the network load:

uncongested:
when $\rho \leq 0.7$, all packets transiting in the network are marked green by the TSWTCM, and the growth of flows completion time reflects the uncongested state of the network, as Fig. 7.1(a) shows. From Fig. 7.1(b) it can be gathered that the maximum cwnd value reached during a transmission is, on average, roughly the half of π when $\pi > 2$: this is reasonable because the window growth remains substantially in slow-start phase and the threshold is certainly many times bigger than that window length. Only a few packets are dropped due to sudden packet bursts, and the RED gateway could not be distinguished from a simple drop tail FIFO queue (Fig. 7.2(a,b)). The peak of fast recovery (Fig. 7.2(a)) for flows whose $\pi = 10$ at $\rho = 0.6$ is strongly correlated with their almost null drop percentage at that load: this means that multiple drops within the same window are less likely to occur and thus Reno can complete fast recovery without incurring in retransmission time outs.

lightly congested:
when $0.7 < \rho < 0.9$, the percentage of red packets is kept below $\sim 1\%$, and it is important to notice that the drop percentage increases and its slope abruptly augment. Early drop actions begin to be taken even for flows whose $\pi = 10$: being the load higher, the packets arrival is more regular and the estimated queue average represents a better smoothed estimate of the real queue length that in previous cases. HTTP flows whose length is $\pi = 90$, take advantage of TCP's fast recovery mechanism as long as ρ grows, and their maximum congestion window is resized to $\sim 2/3$ with respect to the previous case (Fig. 7.3(a,b)).

heavily congested:
when $\rho \geq 0.9$, the percentage of red marked packets grows from $\sim 1.5\%$ to $\sim 12\%$:

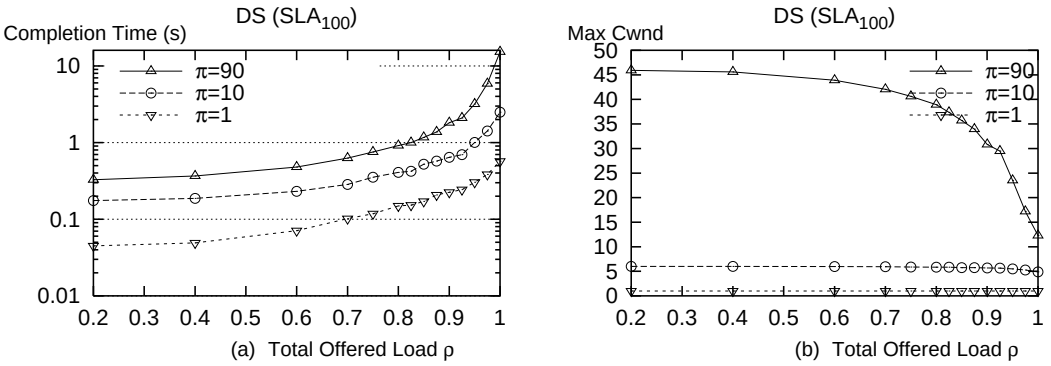


Figure 7.1: Completion Time and Maximum cwnd (HTTP, SLA_{100})

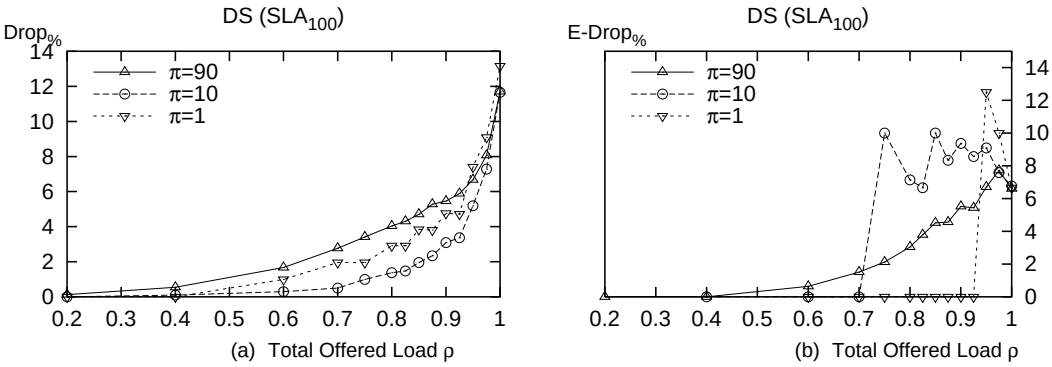


Figure 7.2: Drops and Early Drops Percentages (HTTP, SLA_{100})

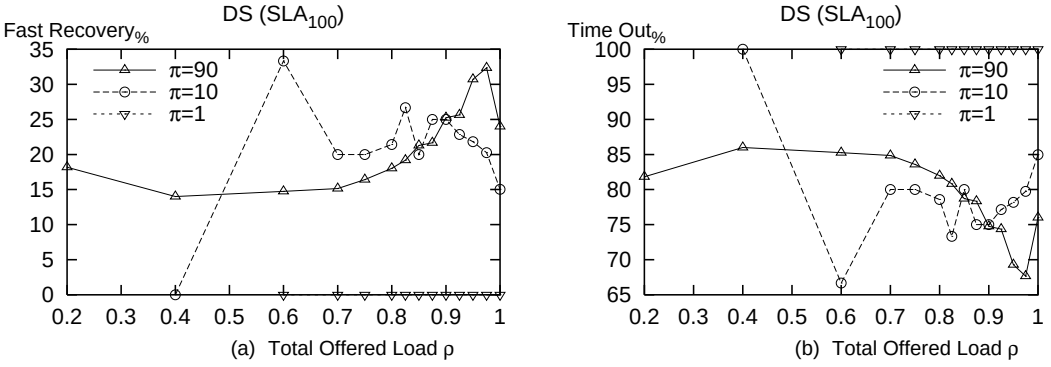


Figure 7.3: Time Out and Fast Recovery Percentages (HTTP, SLA_{100})

the network operates in a critical state, as the sudden growth of drop percentage indicates; flows whose length is $\pi = 10$ experience a fast recovery reduction in favor of longer flows, except when the network load equals the line rate; an important remark is that flow completion time increases independently from flow length, in the sense that its growth is of the same order for any π.

7.1.2 Case 2: HTTP with Background FTP Traffic

The long-lived FTP sources introduced alongside HTTP flows, generate an uninterrupted high background traffic, since the amount of traffic is not subject to any form of control other than those implemented by the TCP mechanism; the expected result is that FTP traffic introduction will push the network in an overload region, thus eliminating the *uncongested* phase and *lightly loaded* phases described in the previous section.

heavily congested:
Simulation results confirm the assumption: even for very low HTTP traffic loads the percentage of web red marked packets is $\sim 2.5\%$, growing to $\sim 5\%$ at $\rho = 0.7$; furthermore when $\rho = 0.2$ the maximum TCP congestion window achieved by the longest HTTP flows is ~ 25 (Fig. 7.4(b)), while without the background traffic, these value were reached only for really high loads ~ 0.95. The flow's completion time is, roughly, scaled by a factor of five (Fig. 7.4(a)) with respect to the case where only HTTP packets travel on the network; furthermore, the FTP introduction does not change the flows relative completion time according to their lengths. One of the more relevant effects that can be noticed is that packet drop occurs even for very low HTTP traffic volumes (Fig. 7.5(a)); furthermore, it can be gathered that the drop probability is largely independent of flow sizes. The drop percentage grows from a minimum of $\sim 5\%$ (which was the heavily congestion state boundary value in the previous case) to twice as much as long as the load increases from ~ 0.2 to ~ 0.85. Fast recovery and RTO are more unfairly per-π distributed, as infinite flows, as well as longer HTTP flows, are more likely to profit of them, as Fig. 7.6(a,b) depicts; this effect, more evident than in the previous case, is mainly due to both TCP Reno behavior (Reno is able to complete a fast recovery only if no multiple drops occur in a window) and to the fact that receiving three duplicated acknowledgement packets within the same cwnd is clearly less likely for smaller cwnd values.

starvation:
When the web traffic load reaches ~ 0.9 the amount of red packets is $\sim 10\%$ growing to $\sim 16\%$ for the highest load; the HTTP traffic experences starvation: its completion time, as well as packet drop probability, drastically grows; specifically, packet drop is almost entirely ($\sim 95\%$) due to buffer overflow as a consequence of RTO ($\sim 90\%$ of cases on average), and its percentage nearly doubles as long as the load approaches the unity. The early drop action is taken for low HTTP traffic volumes, thus when traffic is mainly composed of FTP connections; conversely, as long as web traffic injects in the network bursts of packets, the deployment of RED queue management is no longer effective, as Fig. 7.5(b) shows.

	$\rho = 0.2$		$\rho = 0.7$		$\rho = 0.9$		$\rho = 0.975$	
time	Σ Thr	$\overline{\text{cwnd}}$	Σ Thr	$\overline{\text{cwnd}}$	Σ Thr	$\overline{\text{cwnd}}$	Σ Thr	$\overline{\text{cwnd}}$
t_{stop}	8011.7	8.027	2780.2	6.460	1096.7	4.41	487.0	3.09
t_{end}	8022.4	7.853	2918.4	7.977	2333.3	8.17	3059.3	8.39

Table 7.1: FTP's Overall Throughput and Average Congestion Window

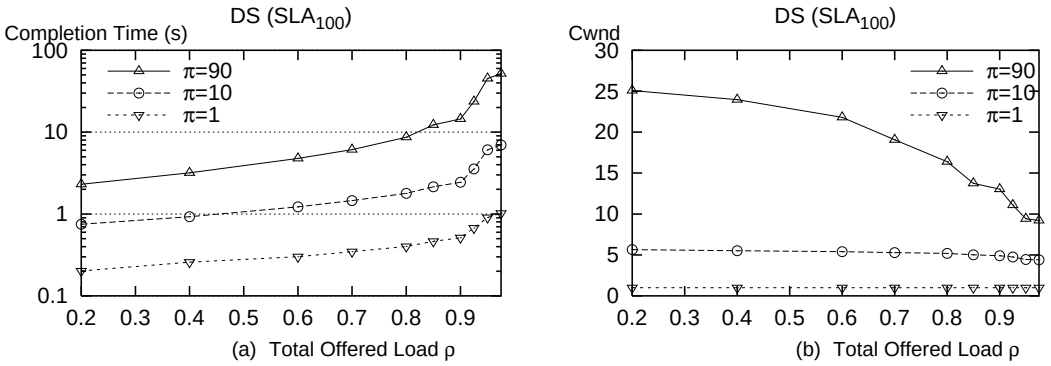


Figure 7.4: Completion Time and Maximum cwnd (HTTP+FTP, SLA_{100})

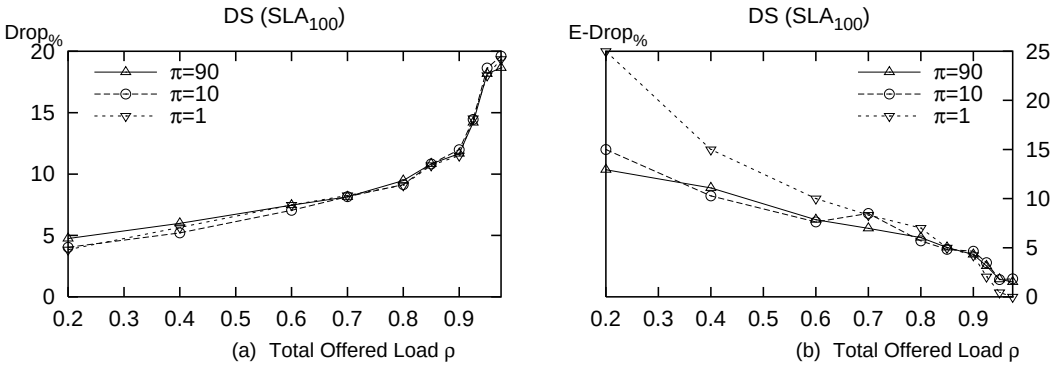


Figure 7.5: Drops and Early Drops Percentages (HTTP+FTP, SLA_{100})

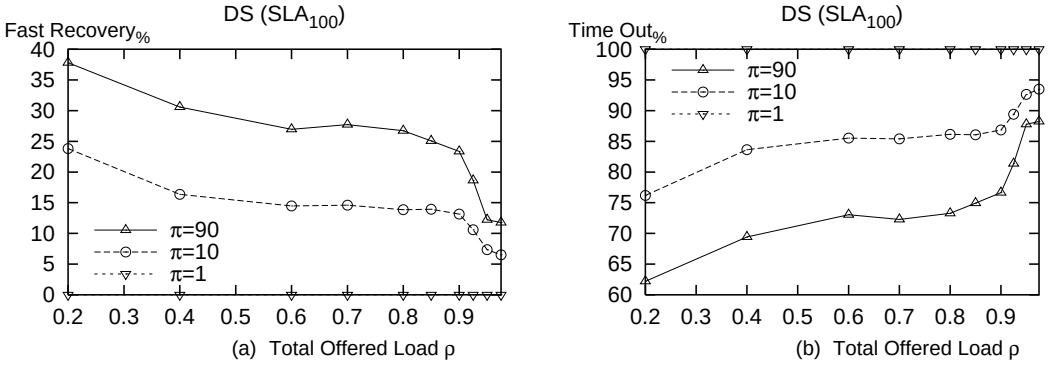


Figure 7.6: Time Out and Fast Recovery Percentages (HTTP+FTP, SLA_{100})

7.1.3 Case 1 vs Case 2 Comparison

The interaction between HTTP and FTP flows can be better explained with the statistics reported in Tab. 7.1; this table reports for the two time instants t_{stop} and t_{end}, the overall throughput of the FTP sources and the congestion window reached by flows on *average* up to that time instant. The traffic produced by the infinite flows when $t \simeq t_{stop}$, thus when the last web flow started, fills up the link bandwidth not used by HTTP packets; furthermore, since no explicit control is performed on the FTP traffic volume, this will make the real network load exceed the DS domain's available capacity.

The FTP performances are really different at the two sampling time instants: this phenomenon stems from the considerations developed earlier in Sec. 6.4. The FTP sources do not stop until the last HTTP active flow completes its transmission: that is, a part of the bandwidth used up to t_{stop} from web packets, becomes available to the infinite flows, which then profit indiscriminately of the available "excess" capacity affecting the active flows' transmissions. For simulation time up to t_{stop}, the overall FTP throughput will decrease as long as web load increases, and this obviously holds even for the average length of TCP's congestion window; when indeed the simulation time exceeds t_{stop} threshold, FTP connections will have the chance to enlarge their window lengths (with the exception of $\rho = 0.2$ where $t_{stop} \simeq t_{end}$) reaching thus a higher throughput. The ratio of the throughput sampled at t_{stop} over the value sampled at simulation's end grows from ~ 1 for low web traffic loads up to exceeding ~ 6 when $\rho = 0.975$; that last value represents anyway an exceptional case, caused from the fact that during the last minute of simulation only two HTTP flows were still active.

	HTTP Only Traffic			HTTP and FTP Traffic		
Web ρ	CT(t_{stop}†)	t_{end}	CT(t_{end})	CT(t_{stop}†)	t_{end}	CT(t_{end})
0.2	0.126	600.0	0.126	0.629	602.0	0.630
0.7	0.233	601.9	0.233	1.328	614.5	1.345
0.9	0.546	604.8	0.549	2.617	699.7	2.683
0.975	1.325	610.8	1.339	6.793	857.5	8.477

† In these and the following simulations $(t_{start}, t_{stop}, t_{halt}) = (1, 600, 900)$

Table 7.2: Simulation Times and Flow Completion Time Average

HTTP performances too are sampled at those two time instants: this is done to ensure a certain degree of confidence to the results issued from simulation. Tab. 7.2 reports the completion time average of HTTP flows when traffic is either solely constituted by web flows or by a mix of short and long-lived flows. In the former case, the maximum completion time relative error is reached when $\rho = 0.975$ and it is $er(CT, \rho) = (CT(t_{end}) - CT(t_{stop})/CT(t_{end}) \simeq 1\%$; in the latter case, FTP presence alters somehow the final sampling results for the reasons earlier explained, especially for very high loads: the uncertainty grows up to $er(CT, \rho = 0.9) \simeq 2.5\%$ and to the bogus value of $er(CT, \rho = 0.975) \simeq 20\%$.

In Tab. 7.3, some of the simulation characteristics are reported for loads at the boundary of the various network congestion phases, taking into account, even in the case of background traffic, exclusively the HTTP datagrams flowing in the network.

It should be noted that the activation of more FTP sources, increasing the congestion state and thus the drop percentage, implies the growth of the number of packets totally transmitted at the source Π for the same amount of π and thus each flow's completion time; furthermore, the statistics reported clearly show that the early drop trend is the opposite in the two cases. The number of flows generated depends both on the network and the cloud load, being the bottleneck size and the average flow length constant: the introduction of FTP flows beside the HTTP traffic

	0.2	0.7	0.9	0.975	DiffServ HTTP Load ρ_{DS}
HTTP Only	0.13	0.23	0.55	1.34	Average Completion Time
	0.09	2.15	4.76	7.88	Average Drop%
	0.00	1.83	6.10	7.75	Average Early Drop%
	151808	539654	710218	801135	Packet Totally Sent Π
	10987	38065	48696	53462	Flows Totally Generated Φ
	13	60	121	219	Max Contemporary Flows ϕ
HTTP, FTP	0.63	1.35	2.68	8.48	Average Completion Time
	4.49	8.07	11.75	18.92	Average Drop%
	15.97	10.20	7.80	5.45	Average Early Drop%
	157908	577167	752300	903649	Packet Totally Sent Π
	10913	38571	49195	53445	Flows Totally Generated Φ
	61	151	308	1218	Max Contemporary Flows ϕ

Table 7.3: HTTP Parameters on Boundary Load Values

does not affect this characteristic of the simulation. The maximum number of flow active at once ϕ is indeed strongly dependent on flow completion time and on the difference $\Pi - \pi$ between the number of packets effectively sent and the number of packets to send determined by the HTTP model; here, the effects of FTP sources are decisive: in presence of FTP, $\phi|_{\rho=0.2}$ equals to $\phi|_{\rho=0.7}$ value when only HTTP datagrams are sent over the domain.

7.2 SLA_{100} vs SLA_0 Simulations

In these single-cloud scenarios, Best Effort and DiffServ traffic have been simulated *separately* and their performances compared for a small set of web loads. If in the SLA_{100} case all the bandwidth is reserved to assured services, then in the SLA_0 case all the line rate is available to best effort packets: the simulated traffic performance reflects thus the specific configuration of each RED buffer composing the MRED one; in fact, being the traffic crossing the network marked with one unique DSCP during the whole transmission, DS and BE performance differences depend exclusively on the specific $(min_{th}, max_{th}, max_p)$ set used in active queue management.

7.2.1 DS vs BE Performances

In an assured service, the DS packet color depends on its conformance to an established SLA; being the SLS specified in terms of reserved bandwidth in the case of the Assured Rate PDB, the traffic volume produced by a web cloud must not exceed the allocated bandwidth in order to be compliant to the specific service profile, and be thus marked green; however during short congestion periods, it is possible for packets to be degraded (i.e. re-marked to a higher drop precedence) before being buffered. For $\rho = 0.85$, if only HTTP traffic is injected into the network, DiffServ packets are almost entirely ($\sim 99.7\%$) marked green and the early drop action will thus be taken according to $(min_{th}, max_{th}, max_p)$ parameters of the AFx1 queue; when indeed FTP sources create background traffic, the percentage of HTTP red marked packets grow to $\sim 6\%$. Furthermore the average green queue length is calculated based exclusively on the number of packets buffered in that queue, so the absence of lower drop precedence traffic will not affect in any way its estimate.

Similarly, BE packets will always be mapped to a queue whose settings are equal to AFx3 (i.e. DS Red packets queue), and the marking cloud not be affected by the presence or the absence of another kind of traffic. However, in RIO-C mode, the average queue length estimate for a specific drop precedence is based on the number of packets with equal or lower drop precedence buffered

in the MRED gateway: here, the absence of DS traffic will have the effect of taking into account only red marked packets in BE queue length computation.

To compare DS versus BE traffic performances, some key parameters are plotted for $\rho = 0.85$ where the x-axis represents the number of packets to send, π, and the y-axis reports the average value of the parameter computed over all flows with the same π; the DS and BE plots are superposed on the same graph, in the two cases of presence and absence of background traffic sources belonging to the same BA, thus subject to the same service.

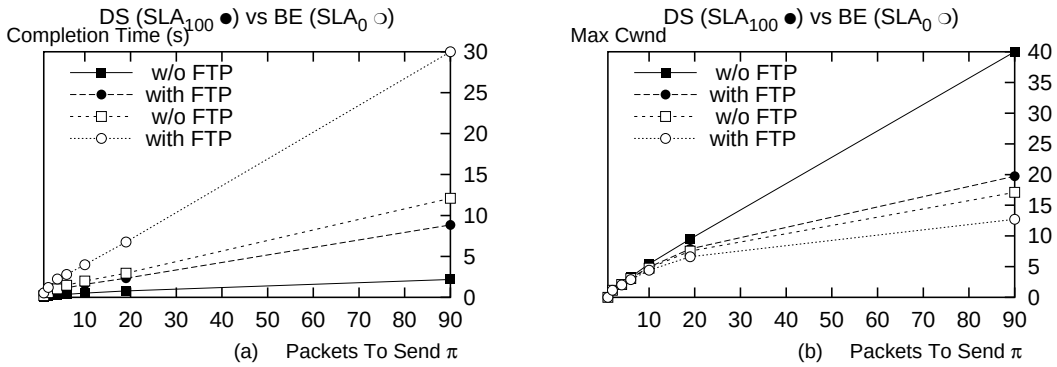


Figure 7.7: Completion Time and Congestion Window (SLA_{100} vs SLA_0)

The flow completion time is depicted in Fig. 7.7(a); as expected, the completion time grows with the introduction of long-lived flows, even though they are sending within the profile. These effects are slightly more relevant for the DS traffic than for BE one: averaging completion time over all flow lengths, DiffServ flows take ~ 2.7 more time to complete the transmission in presence of background FTP traffic, whereas BE average completion time is scaled by a factor of ~ 2.2. However, the BE flows compared to DS ones achieve poorer performance as long as π grows: connection duration is ~ 2.9 times higher when $\pi = 20$ and ~ 3.5 when $\pi = 90$.

The maximum congestion window is depicted in Fig. 7.7(b): for long web flows, the DS cwnd value is halved by FTP sources introduction, while there is no such sudden variation in BE case, since its value is anyway much lower; the cwnd DS/BE ratio for flows whose $\pi = 90$, is lowered from ~ 2.3 to ~ 1.6 by the presence of background traffic.

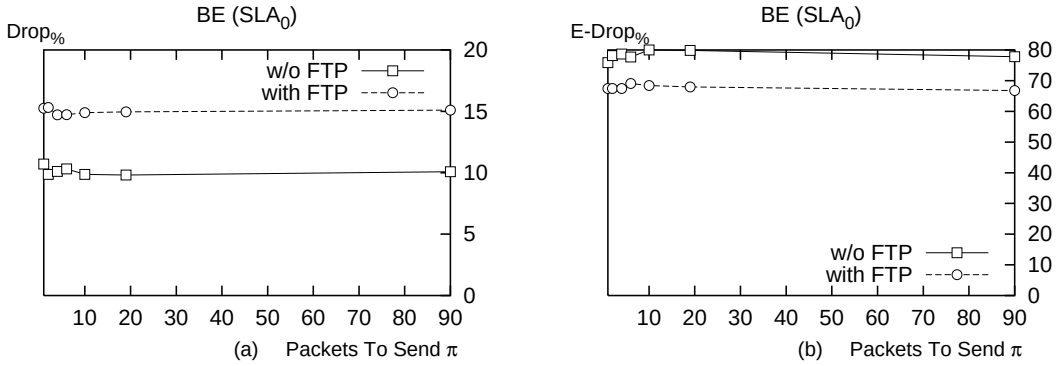


Figure 7.8: Best effort Drop and Early Drop Percentages (SLA_0)

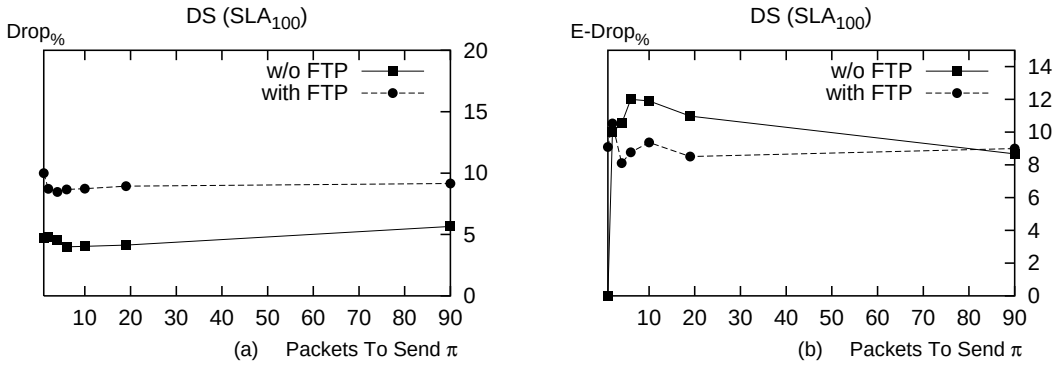


Figure 7.9: DiffServ Drop and Early Drop Percentages (SLA_{100})

The main reason for completion time growth in presence of FTP traffic can be identified with the growth of the packet drop percentage with both traffic aggregates as Fig. 7.8(a) and Fig. 7.9(a) depict; from average computations over all π, it results that the FTP scaling factors amount to ~ 1.5 and ~ 1.7 respectively in the BE and DS cases, and further that the amount of drop in the former case is ~ 1.7 times higher than in the latter. Furthermore, it can be noticed from Fig. 7.8(b) and Fig. 7.9(b) that RED parameters are more gentle toward red packets than toward green ones in early drop distribution since, in the latter case, buffer overflow entails the pre-eminent late drop action.

Time out and fast recovery per-flow length distribution is depicted in Fig. 7.10(a,b) for a higher load ($\rho = 0.975$), where flows whose $\pi <= 6$ cannot benefit of fast recovery mechanism due to the impossibility to reach a sufficiently large cwnd. Due to the early drop distribution, RED mechanism is working substantially in drop tail mode with respect to DS traffic; late drops are in majority, traffic is no longer controlled and the probability to incur in multiple drops within a window increases the probability of fast recovery.

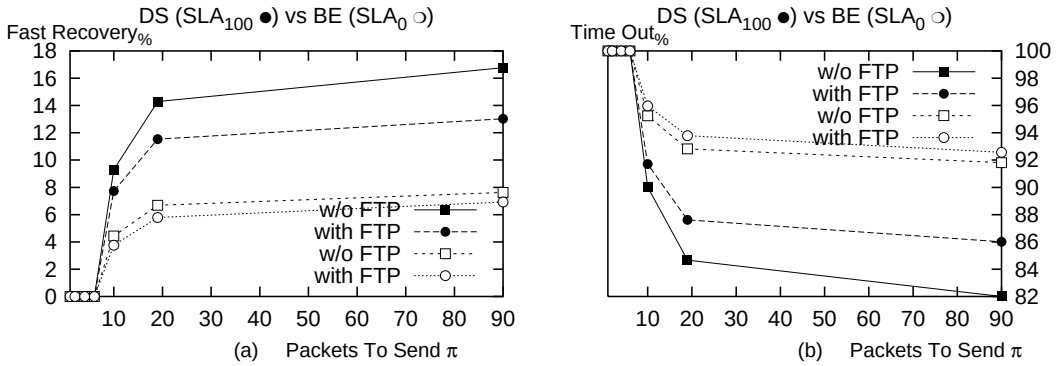


Figure 7.10: Time Out and Fast Recovery $\rho = 0.975$ (SLA_{100} vs SLA_0)

Traffic	SLA$_0$		SLA$_{100}$		
MRED Settings	Boundary Staggered	Staggered	Staggered $scale = 1/2$	Staggered $scale = 1$	Staggered $scale = 2$
Cwnd	4.46	4.05	6.26	7.14	7.30
Completion Time	2.37	2.08	0.74	0.19	0.15
Drop%	11.82	10.04	5.97	0.77	0.12
Early Drop%	28.7	77.4	6.8	14.9	19.2
Late Drop%	71.3	22.6	93.2	85.1	80.8
ϕ	359	379	167	59	39
ρ	0.85	0.85	0.7	0.7	0.7

Table 7.4: Different MRED Settings

7.2.2 RED Settings Considerations

All the previous simulation were performed using the MRED staggered parameter set depicted in Fig. 6.1(a). The reason behind the choise of a staggered parameter set rather than an overlapped one lies in its property of lower drop preference protection: AFx2 marked packet drop begins after all AFx3 packets have been dropped, and similarly, before dropping an AFx1 packet, AFx2 and AFx3 queues will be emptied first. However, since the marker is used in two-color mode, the buffer space between the red $max_{th} = 10$ and green $min_{th} = 20$ is not used to queue yellow packets but only to provide further protection to green ones. In the need to ensure that red and best effort packets are not penalized by too "strict" parameter set, simulation were done configuring MRED with the boundary staggered set of Fig. 6.1(a); the latter set is derived from the first by setting red $max_p = 1$ when red max_{th} coincides with green min_{th}.

Simulation results with these two MRED parameter sets are shown in the first two columns of Tab. 7.4: the boundary staggered set produces a gain on the maximum congestion window reached by HTTP flows on average which is compensated by the growth of drop percentage resulting in poorer completion time performances; furthermore, the ratio between early and late drop is the opposite in the two cases.

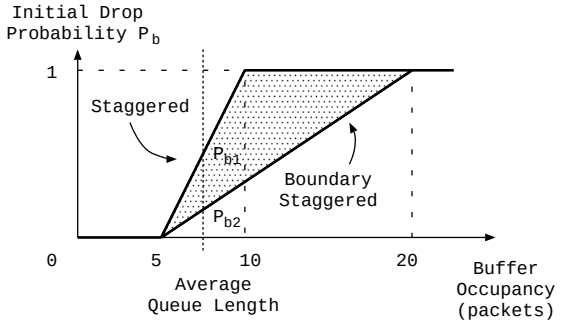


Figure 7.11: Red Color Staggered and Boundary Staggered RED Parameter Set

An intuitive explanation of this phenomenon can be given observing the superposed red-color Staggered and Boundary Staggered parameter set plotted in Fig. 7.11. These piecewise linear functions used to determine the initial early drop probability P_b as a function of the average queue size, resulting in $P_b = max_p \cdot (avg - min_{th})/(max_{th} - min_{th})$. The shaded area would thus represent the increased early drop probability when P_b is used to "mark" packets in the gateway: it would be obvious that, being *avg* constant, a packet would be more likely to incur in early drop

action using the staggered set than the boundary one. However, whether P_b were used as the final early drop probability, the intermarking time *expressed in packet* would be a geometric random variable X, as explained in [RED]; early drop packets may be clustered and the intermarking time ($E[X] = 1/P_b$) may be too long, potentially leading to global synchronization, with several connections reducing their windows at the same time.

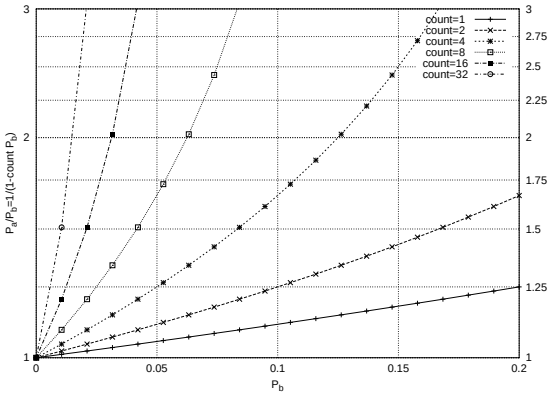


Figure 7.12: Final P_a vs Initial P_b Drop Probability for Different *count* Values

To avoid these problems, the actual drop probability P_a is eventually calculated from the initial one, introducing a count of the arrived packets since last early drop, as $P_a = P_b/(1 - count \cdot P_b)$; in this case the intermarking time X is an uniform random variable $X \in \{1, 2, \ldots, 1/P_b\}$ (assuming for simplicity that $1/P_b \in \mathbb{N}$) with average $E[X] = 1/(2 \cdot P_b) + 1/2$. The $P_a/P_b = 1/(1 - count \cdot P_b)$ ratio is plotted in Fig. 7.12 (where P_b doesn't exceed the maximum recommended value for max_p) for different *count* values: as a result, P_a increases slowly as *count* increases, ensuring that the gateway will not neither wait too long before dropping a packet nor tend to cluster packet drops.

The explanation given above is really simplicistic, as it neglects the latter part of the algorithm; indeed, RED algorithm is very complex, and a quantitative explanation would need a complex probabilistic model. An analytic description of RED is reported in [REDMOD]: a probabilistic model of the gateway is compared with the common drop tail FIFO queue model in terms of the performances impact on bursty (Poisson-based) and smooth traffic; the simplifications made to the effective algorithm include both the assumption that the real queue length (and not its averaged estimate) is input to the algorithm and furthermore that drop decision is taken based on P_b (and not P_a), so that consecutive packets belonging to the same bursts experience the same drop probability. In the need to consider TCP's and specifically HTTP's mechanism interaction with the gateway, a satisfying analytic approach is not clearly feasible.

As parameter tuning in RED remains an inexact science, the unique alternative is to proceed with a trial-and-error experimentation, as for the Scaled Staggered parameter set, derived from the staggered one by uniquely scaling every threshold pair of the set by a constant factor: the max_p parameter and the buffer size are both left unchanged. These simulations, whose results are reported in the last three columns of Tab. 7.4, were run with the intention to increase the early drop percentage to make RED effective on DS HTTP flows, which is not straightforward without augmenting the buffer size. The most important remark is that probably, even in a non critically congested network (ρ ==0.7 and there were no background sources), a single misconfigured router can determine the performances of the whole network: for example, traffic crossing an edge configured with the halved-scale set may not receive the contracted SLA, even thought it could, e.g, if the switch used the doubled-scale one.

A study of RED effects on web traffic performances is presented in [REDWEB]: experiments were conduced on a local testbed, implementing a traffic generator based on Mah's model, in order to empirical evaluate RED behavior across a range of parameter settings and offered loads. Results compared to a drop tail queue show that, contrary to expectations, RED has a minimal effect on

HTTP response times when $\rho < 0.9$; moreover, for loads in this range, response times are not substantially affected by specific RED settings. This is mainly a consequence of the fact that the drop percentage is kept reasonably small when $\rho < 0.5$, and that performance degradation only occurs at loads greater than $\rho = 0.7$, as confirmed by simulations issues described in Sec. 7.1.1. For loads $0.7 < \rho < 0.9$, the performance of the *lightly congested* network decreases monotonically as the load increase, and the effects of RED may yield performances somewhat superior to FIFO. However, response times are quite sensitive to the actual RED parameter settings, and finding an optimal set is non-obvious; specifically, it exists a trade-off between improving completion time performances of short and longer flows: the former benefit of shorter queues (decreasing thus the queuing delay) while the latter of longer ones (since, even though the time spent in the queue by each packet is longer, the reduced rate of drops entails a reduction of RTO). This means that performances of short and longer flows cannot be optimized simultaneously: according to these considerations, our domain's queue size have been set to 100, in order to improve the performances of the largest fraction of the responses (that is, the shorter ones). The most significant performance decrease occurs when the network operates in a *heavily congested* state, and these are the targets where there is more performance to gain; here, RED can be tuned to obtain a significant gain with respect to FIFO, although there exists a trade-off between network utilization and flows completion time: finding a set of parameters suitable for any congestion state is thus not straightforward.

It can be argued from these considerations, that providing adequate link capacity is far more important –for web traffic completion time– that tuning active queue management parameters; furthermore, due to current lack of a widely-accepted analytic model and field-tested engineering guidelines, setting RED parameter remains an extremely complex task.

7.3 SLA_{50} Simulations

In these two-cloud scenarios, Best Effort and DiffServ traffic have been simulated *simultaneously* and their performances are shown across a range of web loads. In the SLA_{50} case, half of the whole bandwidth is reserved to assured services and half is available to best effort flows; the per-cloud normalized offered loads are, in these simulations, equally set to $\rho_{DS} = \rho_{BE} = 1/2$: this means that the amount of traffic sent by each clouds will be, on average, equal to the half of the total network load ρ. Both clouds will thus send, on average, an amount of traffic which does not exceed the SLS established with the domain, except for short periods of congestion due to packet burts: this allow to investigate the network performance degradation as long as ρ grows when the traffic is substantially produced within the profile.

Previous simulations tested the behavior of the network crossed by traffic belonging to either DS or BE service separately, thus giving an abstract best case performance reference; whether the traffic is a mix of both service types, then each BA will be differently forwarded depending on packet marking, and it must be pointed out that the relative influence of the two services is not symmetric. Specifically, BE influence on DS traffic should not entail performance degradation: the average green queue length is computed independently from queues of any other color and furthermore, due to the use of a staggered MRED parameter set, green packets drop will begin only after every red packet have been dropped; conversely, BE traffic performances will suffer of the presence of AR packets, as its average queue calculation is coupled with any other lower drop precedence queue.

DS traffic is expected to incur in a largely reduced amount of drop with respect to BE, due to green marked packet protection: the assured rate packets should thus perceive a less congested network than best effort ones; as a consequence, DS traffic should obtain better response time performances than BE, especially for shorter flows due to MRED settings (according to considerations developed in Sec. 7.2.2).

Network peformances will be analyzed in terms of aggregate flows average characteristics, both for a flow length subset ($\pi \in \{1, 10, 90\}$) against the total offered load ρ and for a load subset ($\rho \in \{0.7, 0.85, 0.975\}$) versus the number of packet to send; flows parameters taken into account are completion time, maximum congestion window, packet drop percentage, early drop percentage

over total drops and eventually fast recovery and retransmission time out percentage over total dropped packets.

For each of these characteristics, separate figures for DS and BE traffic type as well as for DS/BE parameters ratio will be plotted on the same page. In the following, to reference a figure, the notation Fig. $n(X,x)$ will be used, where:

n: is the number of the figure, composed of six separate plots

x: can be either (a), in which case the abscissa axis reports the network load ρ or (b), in which case the abscissas represents the number of packets to send π

X: discriminate between the plotted traffic type (i.e. $X \in \{\text{DS,BE}\}$) and the DS/BE ratio of the parameter values (i.e. $X = \text{DS/BE}$)

Beside DS/BE parameter value ratio, in order to allow comparison of the X parameter within the same traffic class for different loads or flow lengths, two complementary scale factor notation are defined as follows:

$\pi_{scale}(\pi_1, \pi_2)$: measure the ratio of the aggregate Y parameter average value sampled for two different flow lengths at a fixed ρ, thus $\pi_{scale}(\pi_1, \pi_2) = \overline{Y}(\pi_2, \rho)/\overline{Y}(\pi_1, \rho)$

$\rho_{scale}(\rho_1, \rho_2)$: measure the ratio of the aggregate Y parameter average value sampled for two different network loads values for a fixed flow length π, thus $\rho_{scale}(\rho_1, \rho_2) = \overline{Y}(\rho_2, \pi)/\overline{Y}(\rho_1, \pi)$

The X traffic type to which these parameter refer will either be omitted ($\rho_{scale}, \phi_{scale}$) in non ambiguous contexts, or explicitly specified ($\rho_{scaleX}, \pi_{scaleX}$) if this may lead to ambiguity.

Finally, a simple analysis of a specific parameter Y sensitivity with respect to network load ρ is provided via a normalized function, rather than calculating the incremental ratio $\Delta Y/\Delta\rho$ on the coarse load sample set; furthermore, the normalization characteristic of that function allows straightforward sensitivity comparison across a set of different flow lengths, but necessitate to be coupled with real parameter values analysis in order to be meaningful. Lets indicate with $\overline{Y_\pi}$ the parameter Y average value, extended to all simulation flows with the same π value, at a given network load ρ; the sensitivity functions C'_π, computed for flows whose packet length π, is the cumulated probability distribution of $\overline{Y_\pi}$ as long as ρ grows in the range. The C'_π function differs from the standard cumulated probability distribution C_π of Y in that the latter indicates the probability for Y_π parameter to assume any value lower or equal to a specific y_π value at a given load, whereas the former indicates the "strength" of $\overline{Y_\pi}$ values change for different loads; the relative sensitivity, with respect to different flow length, can then be computed as C' distributions difference.

7.3.1 HTTP Traffic Only

Simulation issues confirm that network performances are greatly different depending on the forwarding behavior experienced by packets as they cross the network; this means that, despite the difficulty of fine parameter tuning, RIO technique provides a good level of protection of lower drop precedences packets. However, this is done at the cost of a strong performance degradation for traffic that does not belong to assured services PHB; furthermore, the contemporary deployment of different services introduce an unexpected side effect: drop distribution unfairness among flows of different sizes; this may result in a critical problem, as DiffServ traffic is not likely –by intention– to represent the majority of the Internet traffic.

7.3.1.1 Completion Time and Congestion Window

HTTP flows response time is depicted in Fig. 7.13(a) for different overall offered load: it can be seen, as expected, that network congestion level has a remarkable effect on time performances

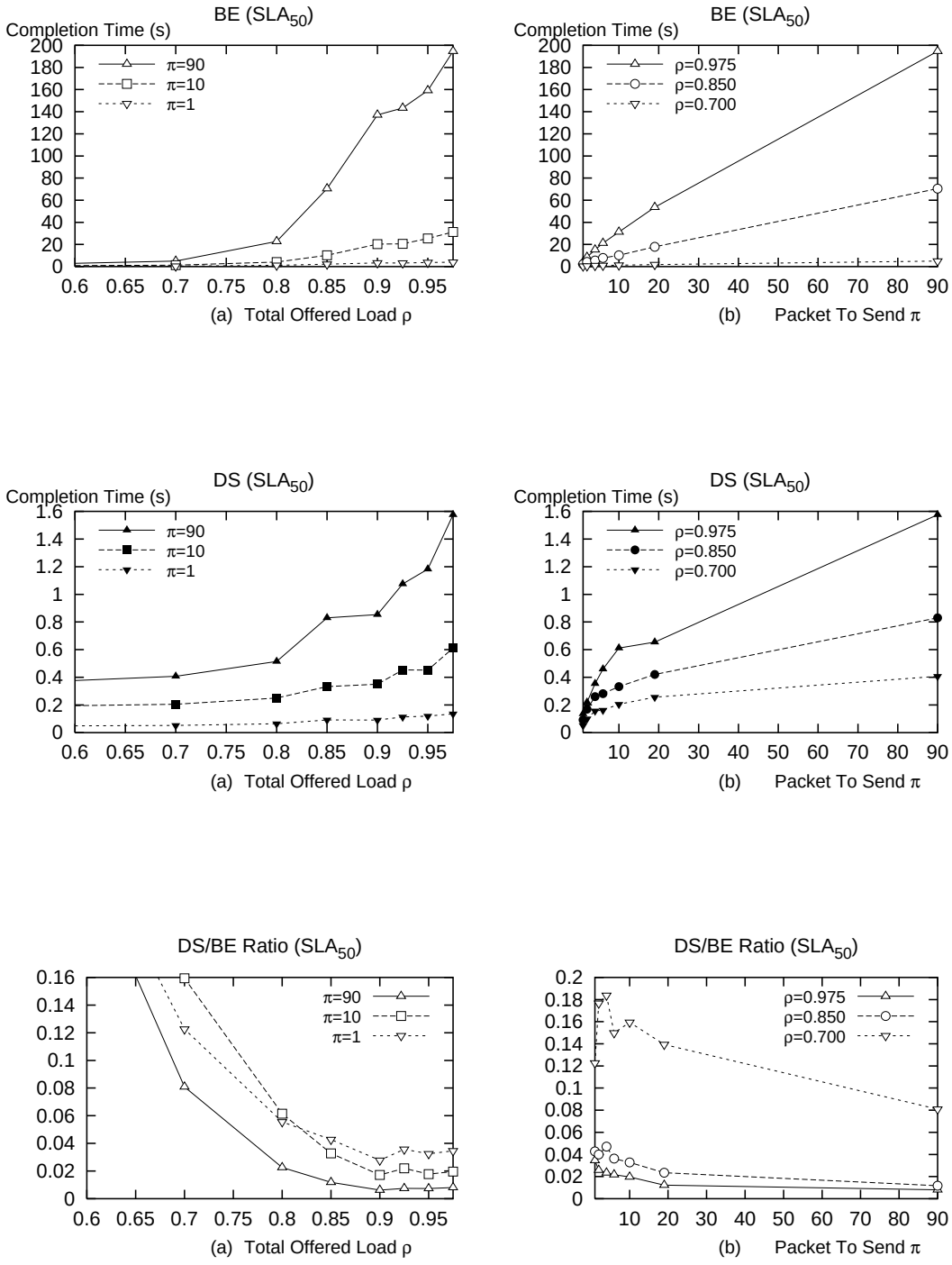


Figure 7.13: Completion Time: DS, BE and DS/BE Ratio (HTTP)

when $\rho \geq 0.7$, and that strong service degradation occurs for $\rho \geq 0.9$. Response time achieved by longer flows is more influenced from network load than short flow performances; specifically, DiffServ completion time scale factor of flows whose $\pi = 10$ is $\rho_{scale}(0.975, 0.8) \simeq \rho_{scale}(0.985, 0.5) = 3$, whereas, when $\pi = 90$, completion time doubles as long as ρ grows from 0.6 to 0.8, and doubles again in the interval $(0.8, 0.975)$ eventually giving $\rho_{scale}(0.985, 0.5) = 5$.

As expected, BE traffic is more influenced by network load than DS flows; moreover, longest 90-π flows are clearly starving on a heavily congested network: their completion time results, on average, $CT_{BE} = 200\ s$, whereas $CT_{DS} = 1.6\ s$ is a reasonable value in such an overloaded network. The performance "gain" experimented by DS flows compared to BE flows may be evaluated as the inverse of the functions depicted in Fig. 7.13(DS,BE,a), since a value smaller than 1 indicates that DS achieved lower response time with respect to BE, thus better time performances.

The slope of response times ratio indicates that the congestion level *perceived* by DS packets is smaller than the actual load: completion time of 10-packets flows is up to ~ 6 times smaller with respect to BE at $\rho = 0.7$ and ~ 25 times smaller if $\rho > 0.9$; the DS/BE gain further increases to 12 and 100, respectively when $\rho = 0.7$ and $\rho > 0.9$, for flows whose $\pi = 90$: this effect strike roots essentially from the fact that long BE flows achieve poorer performances than shorter BE flows, coupled with an slightly significant long flow performance improvement with respect to shorter ones.

ρ	Completion Time Growth Percentiles					
	$\pi = 1$		$\pi = 10$		$\pi = 90$	
	DS	BE	DS	BE	DS	BE
$(0.2, 0.5)$	5.28	0.52	5.43	0.29	4.03	0.09
$(0.5, 0.7)$	5.96	1.93	6.02	0.86	4.73	0.53
$(0.7, 0.8)$	7.34	5.26	7.34	2.73	5.98	2.39
$(0.8, 0.85)$	10.55	9.79	9.78	6.81	9.65	7.39
$(0.85, 0.9)$	10.44	14.91	10.28	13.65	9.92	14.37
$(0.9, 0.925)$	12.96	14.41	13.31	13.80	12.49	15.02
$(0.925, 0.95)$	13.65	16.69	13.28	17.05	13.74	16.69
$(0.95, 0.975)$	15.60	17.93	17.98	20.96	18.30	20.40
$(0.975, 0.985)$	18.23	18.56	16.57	23.84	21.15	23.13

Table 7.5: DS vs BE Response Time Growth Percentiles (HTTP)

An enforcement of the considerations earlier developed comes from the comparison of BE traffic against DS traffic response time percentiles –used to compute the C' distribution– given in Tab. 7.5. A first interesting remark is that severe BE performances degradation begins earlier than for DS flows: the neglectable percentile value of completion time growth, as long as load grows from $\rho = 0.2$ up to the half of the bandwidth, indicates that the completion time achieved on higher loads will be much greater. Furthermore, when $\rho \in (0.5, 0.85)$, BE completion time growth percentile is smaller than DS one, whereas, when the $\rho = 0.85$ threshold is crossed, the BE percentile values are always greater than DS ones, and their increase is abrupt rather than gradual; the latter observations allow us to quantify the stronger dependence of BE on ρ with respect to DS, being thus the proof of the substantial difference between the load perceived by assured and non-assured service traffic. Finally, it should be noted that in the DS case, long flows completion time begin to grow at a rate higher than those of shorter ones for congestion level higher than $\rho = 0.9$.

Continuing with the sensitivity analysis, Fig. 7.14(a) depicts on the ordinate the response time

C' distribution of DS and BE flows sending respectively either 1 or 90 packets, as a function of the ρ abscissas; this can be thought as a normalized parameter indicating how much, in a given load range, the response time are affected by that load increment. Best Effor C' distribution plot is kept below the DS one, as massive BE response time increase begin earlier and it effects are more pronounced, as the slope of the BE curve shows for $\rho > 0.85$.

Moreover, DS short flows completion time growth is smoother and, as shown earlier, DS short flows scale factor is lower than those of longer ones. From Fig. 7.14(b), plotting the C' difference between intermediate ($\pi = 10$) and short ($\pi = 1$) flows as well as between long ($\pi = 90$) and short ($\pi = 10$) ones, we gather that effects of congestion are differently perceived from short and long flows. specificly, in the BE case these effects are more pronounced than in DS case, where there exists no such evident difference between flows sending either 1 or 10 packets. The peak of 10-π flows at $\rho = 0.975$ is steamed by a completion time local minimum, which may invalidate simulation results at that load. DS longest $\pi = 90$ flows still show an unfair treatment with respect to short flows, and their performances are further penalized if heavy congestion occurs.

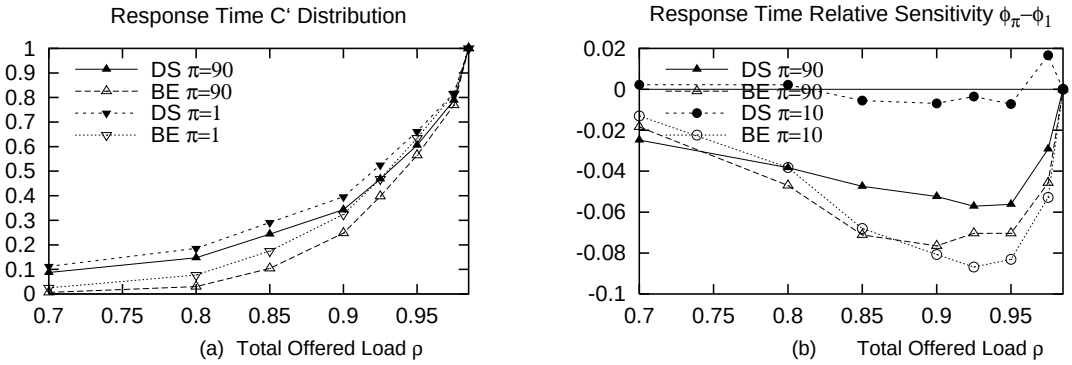


Figure 7.14: Completion Time C' Distribution and Sensitivity Analysis

The maximum congestion window value is depicted on Fig. 7.15; obviously, the advantages of DS against BE packets forwarding are limited to flows carrying enough data to make their window grow and possibly to transmit packets in TCP congestion avoidance phase. DS flows are highly protected from losses even for loads near to the line rate, and 90-π flow window decreases smoothly, as depicted in Fig. 7.15(DS,a), contrarily to the BE case.

Furthermore, the comparison of these results with those of the previous SLA_{100} simulations indicates that DS flows cwnd values benefit of the presence of BE traffic, since the latter is heavily penalized as long as network load grows, as Fig. 7.15(BE,b) illustrates, protecting thus the AR flows from severe performance degradation.

Cwnd of flows whose $\pi = 90$ decrease from ~ 45 to ~ 32 as ρ grows from 0.7 to 0.975 in the DS case, and from ~ 25 to ~ 8 in the BE case; applying the previously defined ρ_{scale} to 90-π flows, DiffServ scale factor $\rho_{scale}(0.7, 0.975) \simeq 1.25$, whereas best effort $\rho_{scale}(0.7, 0.975) \simeq 3.12$ Furthermore, it can be observed from Fig. 7.15(DS,BE,a) that long DS flows $\pi_{scale}(10, 90)$ is $\sim 3-4$ for any load, while this cwnd gain is halved in the BE case: this means that long BE flows are less likely, with respect to DS, to profit of fast recovery, and that they mainly transmit packets in a slow-start phase. This eventually entails that the DS/BE ratio, depicted in Fig. 7.15(DS/BE,a), increases as the load increase, with a maximum value near $\rho = 0.9$: for higher loads, DS flows will reach a cwnd four times bigger than those of BE flows.

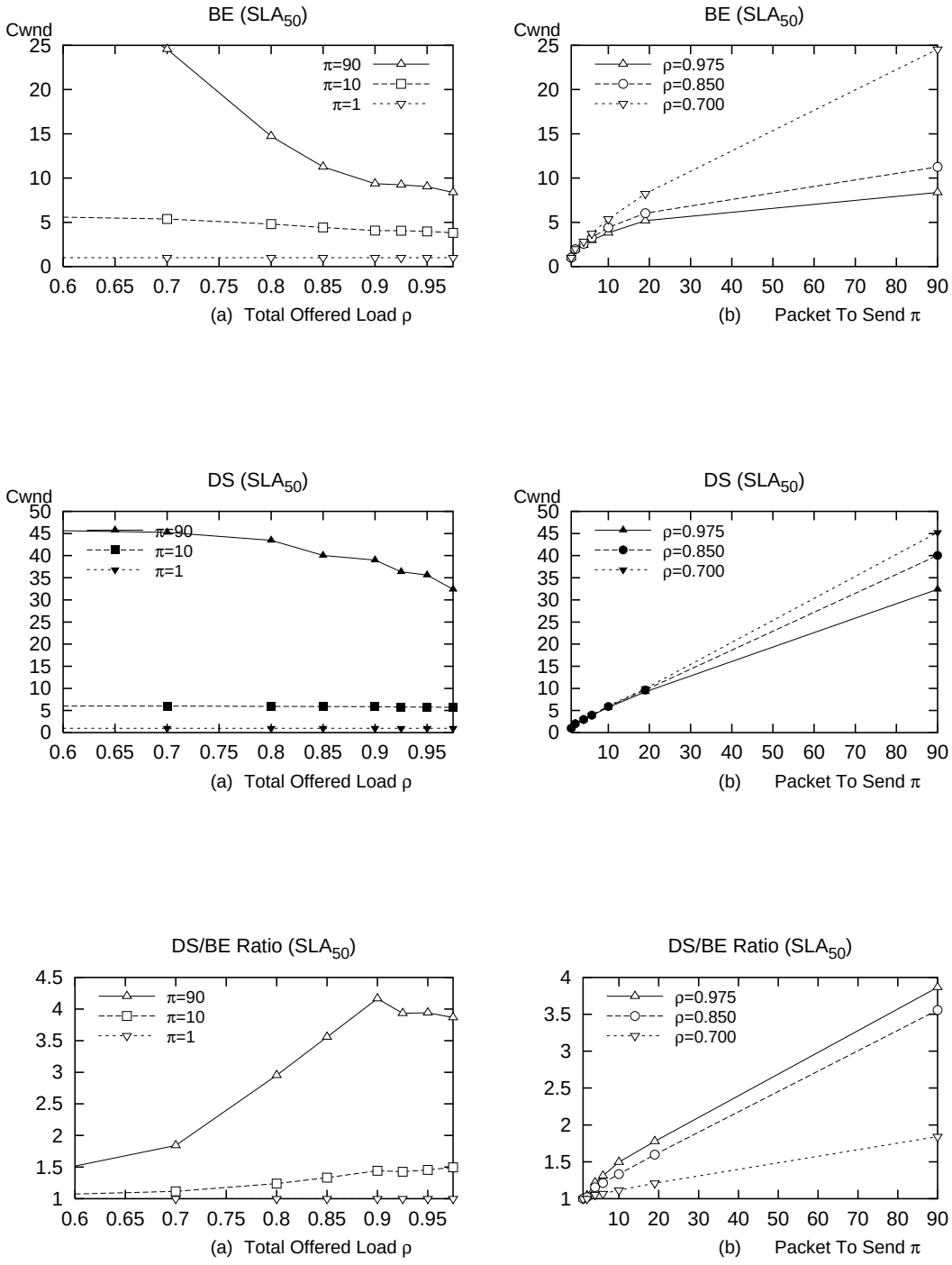


Figure 7.15: Maximum Congestion Window: DS, BE and DS/BE Ratio (HTTP)

7.3.1.2 Packet Drop and Early Drop

DS traffic initially send green marked packets, and since the cloud generate a traffic rate equal, on average, to the subscribed bandwidth, they are likely to maintain AFx1 codepoint marking; however, network may incur in short term congestion due to HTTP traffic burstiness, and packets would be remarked to a lower level of protection, as depicted in Fig. 7.16(a). It can be seen that, as expected, the amount of red marked packets is kept very small ($< 0.1\%$); therefore, DS red marked packet fraction is neglectable in describing HTTP performances in the sense that DS traffic is almost entirely polished with AFx1 MRED settings. Moreover, in the DS case, red packet marking should not depend on flow length: the cumulated red marking probability distribution (Fig. 7.16(b)) validates the assumption; however, it must be said that the latter result, derived for SLA_{50} simulation set, might be representive of a network where the AR service packets are created at a rate higher than the contracted one. While red marked packets fairly distributed over a range of flow lengths, packet drops are not, and the per-flow length drop distribution depends on actual MRED parameter setting: this means that DS green, red and BE packets are likely to receive a different treatment depending to the length of flow to which they belong.

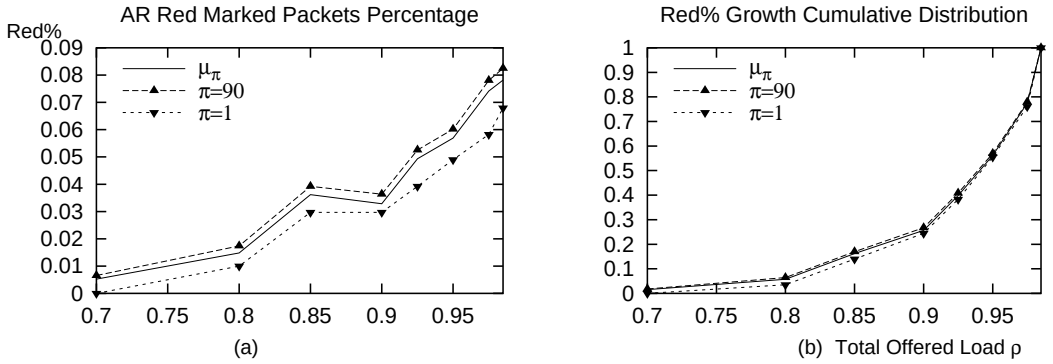


Figure 7.16: DS Red Marked Packet Percentage and Red Marking Cumulated Distribution

DS packets, mostly green marked, will be far protected from drop even at loads near to the line rate: this is shown in Fig. 7.17(DS,a) where it can be seen that DS drop percentage is kept below $\sim 3.5\%$ even when BE traffic faces starvation and drop percentage is ~ 10 times greater. MRED parameter set used is not suitable for short BE transmissions: as depicted in Fig. 7.17(BE,a), BE shortest flows achieve the higher drop percentage for any network load, whereas DS flows sending an unique packet are perfectly protected by drops for any load up to $\rho = 0.8$; moreover, we gather from Fig. 7.17(BE,b) that BE drop percentage actually monotonically decrease as long as the flow size increases. This phenomenon were not present at all in the SLA_0 case (Fig. 7.8) and only minimally noticeable in SLA_{100} (Fig. 7.9); it is thus reasonable to suppose that, due to the relative influence of the services and since is mainly the BE traffic to be affected by this draw back, the reason of this effect strikes its root from MRED parameter settings.

This is a counter intuitive phenomenon, since flows that are just one packet long have non-correlated drop, and needs to be further explained. We know that for flows sending less than 10 packets, the TCP congestion window is essentially controlled by the slow start algorithm during the whole transmission, while longer flows are likely to reach a congestion avoidance phase and are thus more aggressive; moreover, long flows are very likely to benefit of fast recovery algorithm rather than incurring in RTO expiration, as conversely happens for short-lived flows. Looking at Fig. 7.18(BE,b), it can be seen that longer flows are more likely to incur in an early drop action than the shorter ones, as if short flow's packets were without memory: in this case, the benefits brought by RED are hardly distinguishable from common Tail Drop performances.

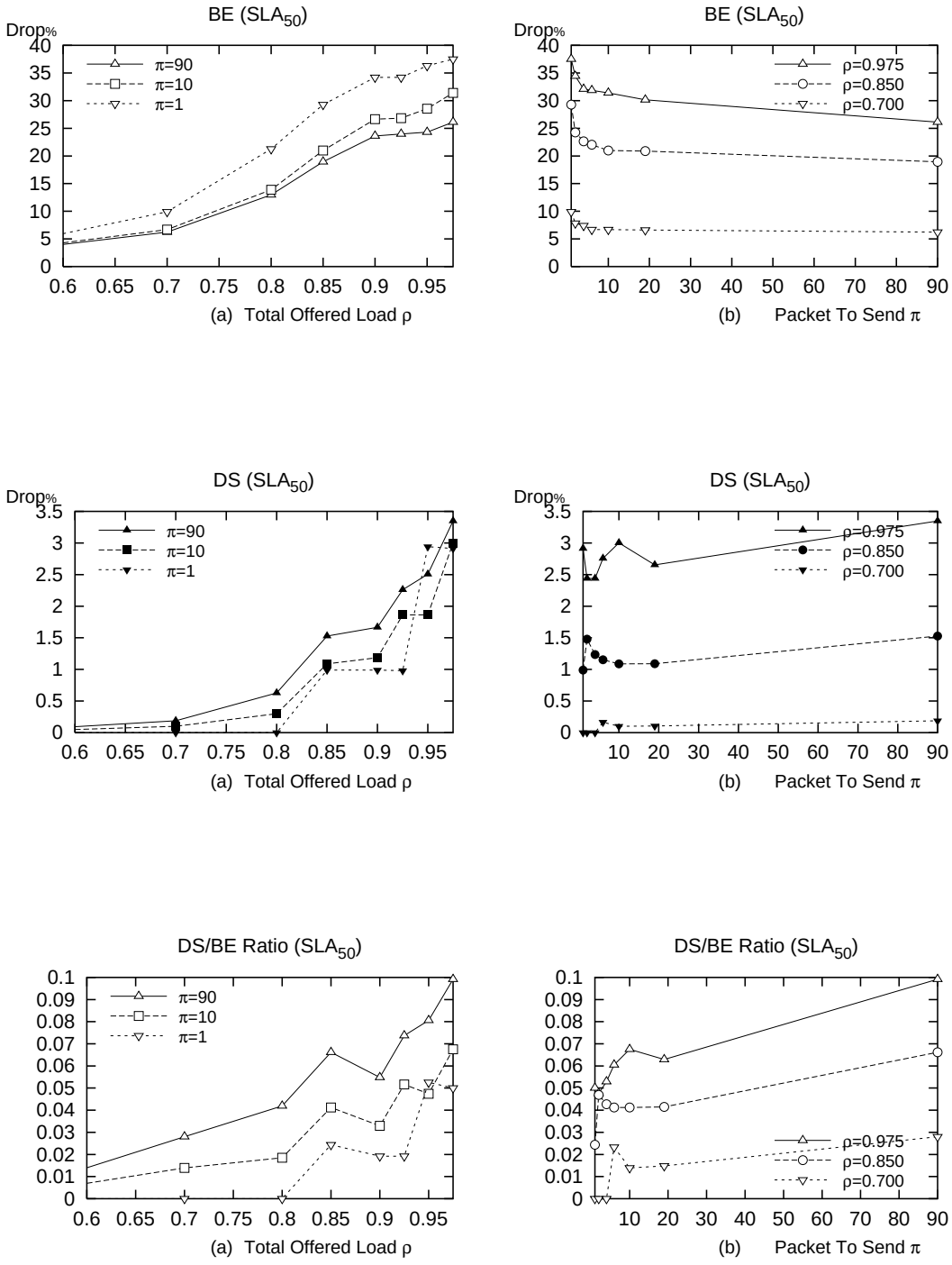


Figure 7.17: Packet Drop Percentage: DS, BE and DS/BE Ratio (HTTP)

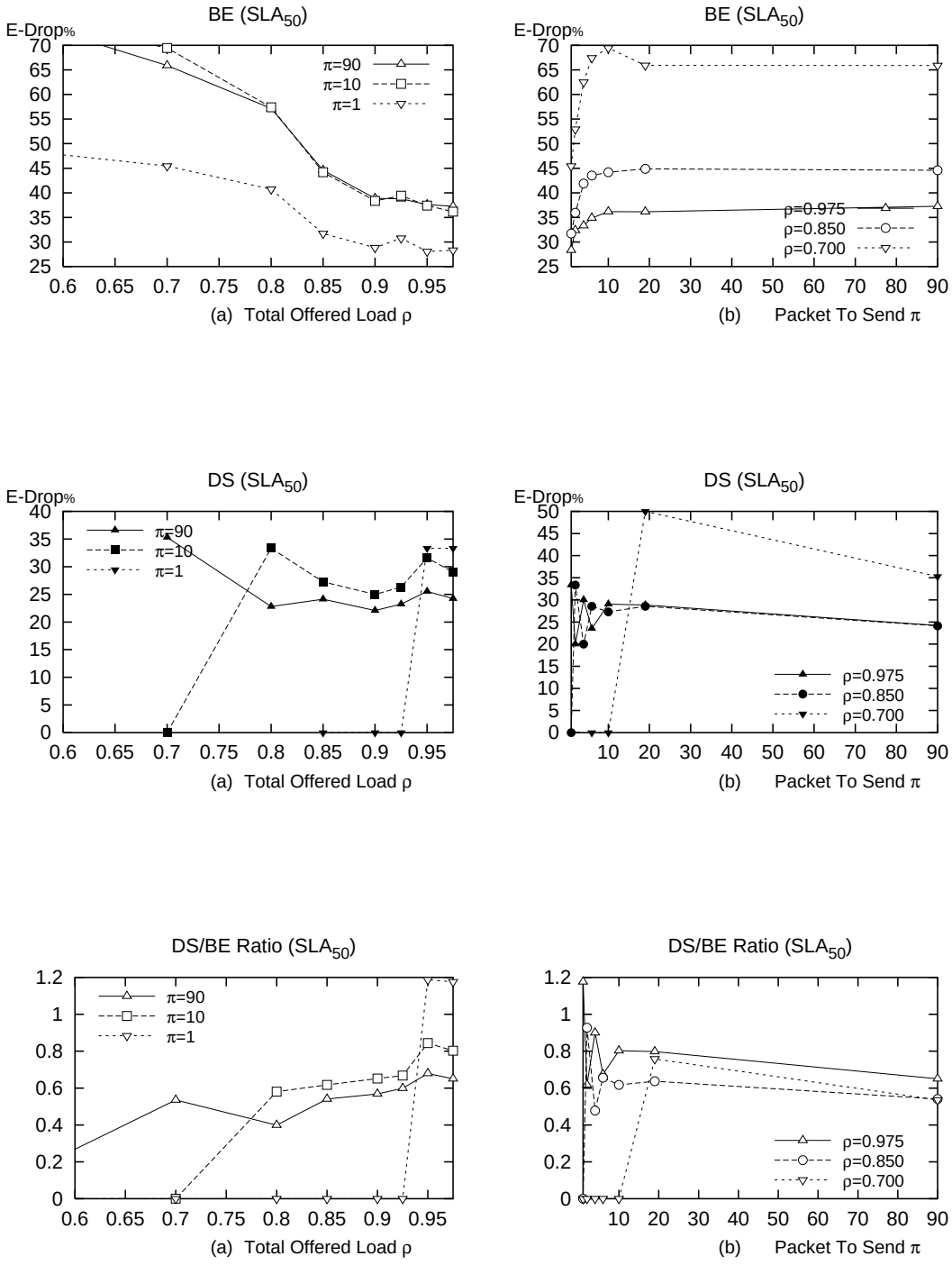


Figure 7.18: Early Drop Percentage: DS, BE and DS/BE Ratio (HTTP)

Moreover, the greater early drop fraction reached by long flows entails that buffer occupancy is mainly due to longer flows, and Fig. 7.18(BE,a) confirms that these occurrence holds for any network state. This said, let's suppose that a 1-π flow arrives at the ingress of a full queue: if the next sent packet will be successfully buffered, its drop percentage would be calculated as the ratio of 1 dropped packet over 2 totally sent packets, giving thus 50%; in the case that a 90-π flow is sending, i.e, 6 packets within the same widow, if the buffer becomes full, at least one of these packets gets "lately" dropped: due to the flow size, the drop percentage won't be influenced as much as in the former case; this observation, coupled with the fact that late drops are less likely to occur than in the former case (i.e. longer flow's packets are more likely to be successfully queued), can be the reason of the unfair drop fractioning depending on flow sizes.

Conversely, in the DS case depicted in Fig. 7.17(DS,b), the long and short flows are substantially affected by the same drop probability; as a consequence, the DS/BE ratio, plotted in Fig. 7.17(DS/BE,b), entails a gain on short flow protection under all network load circumstances. However, the drop percentage dependence on flow size is not straightforward as for BE traffic; this may be due to the limited drop amount, which makes results less significant than in the BE case. Also the DS early drop percentage should be analyzed in this order of ideas; for example, from Fig. 7.18(DS,a) we may notice that short 1-π flows does not incur in early drop for loads lower than $\rho < 0.925$; anyway, rather than addressing this fact to an unfair per-flow size treatment, we should firstly observe that no packet gets dropped when $\rho < 0.7$: it may then be preferable not to interpret these low-confidence results, based on a small scale occurrence set; similar reasons can be used to explain the $\rho = 0.7$ plot of Fig. 7.18(DS,a), where drop level for flows sending up to 10 packets is kept below 0.1%.

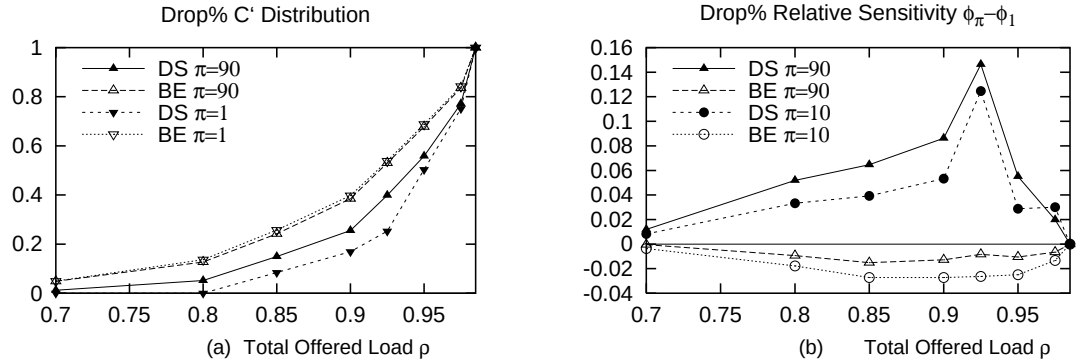


Figure 7.19: Drop Percentage Cumulated Probability and Sensitivity Analysis

The drop C' distribution is plotted in Fig. 7.19(a) for both BE and DS shortest and longest flows; in the BE case, the drop percentage grows independently from flow size as long as load grows: this further confirms that drop unfairness between short and long flows depends on specific RED parameter setting. Conversely, in the DS case, short and long packet drop rate shows rather different behavior with respect to network load; short $1-\pi$ flow's packet drop begin later ($\rho > 0.8$), and its C' is therefore null up to that load value; since at higher loads the drop percentage reaches $\sim 3\%$ independently from flow size, short flow's packet drop rate growth is stronger than those of longer ones. However, there are anomalous and misunderstood results in drop rate behavior: across $(0.925, 0.95)$ load range, drop rate growth with an extremely abrupt slope ($\rho_{scale}(0.95, 0.925) \simeq 3$), whereas $10-\pi$ flow's packets drop percentage does *not* increase within that range ($\rho_{scale}(0.95, 0.925) = 1$).

One of the most important differences between BE and DS cases is that drop percentage shows diametrically opposite behavior with respect to flow size; this can be either gathered from

Fig. 7.17(DS,BE,a) comparison, or by observing Fig. 7.19(b). From the latter plot it can be gathered that our implementation of the Default PHB will treat flows sending respectively 1, 10, 90 packets with decreasing dropping probability, whereas AR service will treat flow of increasing length with increasing dropping probability; however, it must be pointed out that these results have been derived for a special case of AR traffic (since HTTP were the unique flow source application, and, more important, DS traffic were almost entirely green marked) and may not apply to different network scenarios.

There is a first major difference with the SLA_{100} case, where shortest ($\pi = 1$) flow drop percentage were bounded between intermediate ($\pi = 10$) and longest ($\pi = 90$) flow drop results. Furthermore, when the network were crossed by packets belonging to one unique behavior aggregate, there were no such strong drop unfairness depending on flow length. This may result in a critical network performance optimization problem: in addition to the considerations developed in Sec. 7.2.2 (which argued that contemporary optimization of both long and short flows is not possible with an unique set of RED parameters), it should be noticed that it may be also impossible to optimize two different services for a same target (i.e. different AR services conflict). One may want to take advantage of this draw-back to create different coexistent services optimized for either short or long flows: a multi-field classifier may be used to differentiate packet marking basing on their sequence number or, even better, packets may be conditioned near the source to receive either AR-short or AR-long PDB treatment. However, this solution in not reasonable, and in contrast with the basis of scalable DiffServ architectural model; furthermore, the AR PDB definition does not apply to a specific traffic type; finally, the effort to find RED settings –if any– satisfying the specular services requirement is not likely to be profitable.

7.3.1.3 Fast Recovery and Time Out

DS traffic profit more than BE of fast recovery, as expected from the indications gathered from previous SLA_0 and SLA_{100} simulations; moreover, long DS flows outperform the SLA_{100} case, as their are advantaged by the traffic heterogeneity, in the sense that is BE high drop precedence traffic to result further penalized. Fast recovery and time out drop fractioning results, depicted respectively in Fig. 7.20 and Fig. 7.21, will be here briefly discussed only in terms of the former parameter.

Comparison of Fig. 7.20(BE,b) to Fig. 7.20(DS,b) simply confirms that DS traffic is less influenced from load variations, in the sense that its fast recovery probability scale factor is, for both 10 and 90 packets long flows, $\rho_{scale,DS}(0.975, 0.7) \simeq 0.8$, whereas $\rho_{scale,BE}(0.975, 0.7) \simeq 0.2$. Moreover, from Fig. 7.20(DS/BE,a) the DS/BE ratio grows, with respect to load, roughly with the same order for any packet size: this is due to BE that fact recovery percentage decreases proportionally to load (reaching, near $\rho = 0.9$, an asyntotic value which is halved with respect to those reached in SLA_0 case) whereas DS flows tend to keep a constant fast recovery percentage independently from load, as Fig. 7.20(DS,BE,a) depict; here again, the fact that 10-π DS flows begin to profit of fast recovery and fast retransmission algorithms only if $\rho > 0.8$ must be coupled with the corresponding scantiness of drop percentages below that threshold.

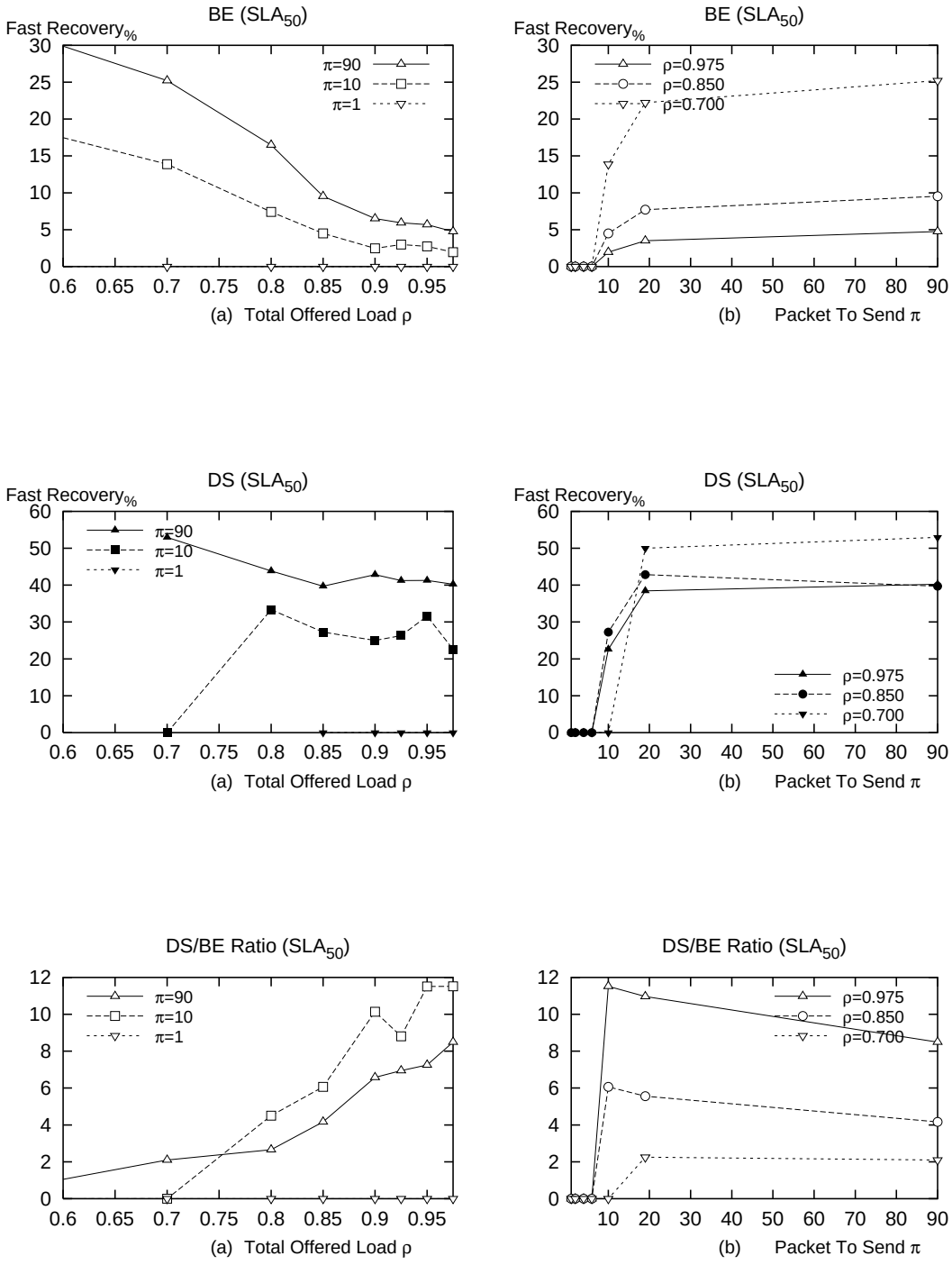


Figure 7.20: Fast Recovery Percentage: DS, BE and DS/BE Ratio (HTTP)

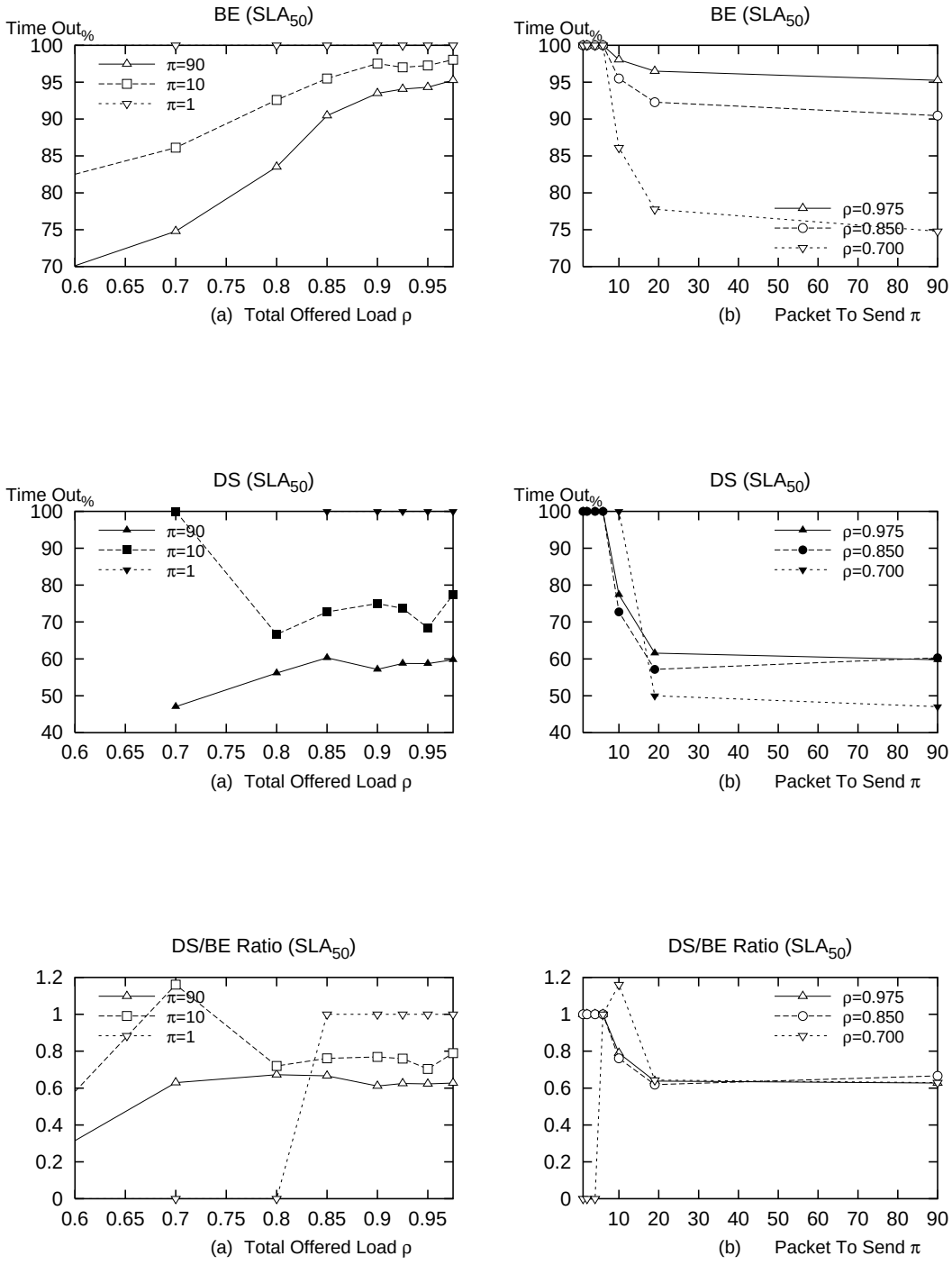


Figure 7.21: Time Out Percentage: DS, BE and DS/BE Ratio (HTTP)

7.3.2 HTTP with Background FTP Traffic

In these two-cloud scenarios, Best Effort and DiffServ cloud contains each, beside short HTTP-like flow sources, 10 FTP background sources generating infinite traffic: the packets produced by these sources are initially marked with the same codepoint that HTTP packets originating from the same cloud.

Since in each cloud HTTP traffic generator produces, on average, a traffic rate equal either to the assured one (in the AR cloud) or to the unsubscribed bandwidth (in the BE cloud), the introduction of FTP will have two main effects: bring overall network load near to the line rate and ensure that traffic produced by each cloud actually exceeds its service level specification; moreover, being the network congestion level equal over all the simulation, the HTTP load indicates the importance of burstiness in congestion. Packets that exceed the limit imposed via SLS are, if possible, downgraded to a lower level of forwarding probability by increasing their drop precedence; specifically, this means that AR packets arriving at a rate higher than the half of the whole bottleneck bandwidth will be red colored by the TSW marker, while no further degradation is possible for highest drop precedence BE packets. In the case of HTTP traffic without other background traffic, it was verified that DS packet remarking isn't influenced by flow size; furthermore, the red marking percentages of HTTP and FTP traffic shown in Tab. 7.6 confirm that packet marking is independent from the specific application type and from the traffic volume produced, thus all flows should experience the same drop rate; however high bandwidth flows will have a higher number of packets dropped, being more likely to incur in an early drop action.

		FTP		HTTP	
ρ	BE% †	kpkt ††	Red%	kpkt ††	Red%
0.2	14.96	465.02	31.07	78.52	31.80
0.4	10.42	354.38	27.10	160.51	27.68
0.6	5.02	236.08	22.41	248.04	22.88
0.7	4.15	190.83	20.91	285.16	21.71
0.8	3.79	147.29	20.73	327.41	21.49
0.85	3.64	130.86	21.24	346.31	21.86
0.925	3.10	106.83	21.33	372.11	21.96
0.95	3.30	99.16	22.59	386.49	23.20
0.975	3.38	84.22	24.35	411.08	24.74

† BE traffic percentage refers to the whole FTP traffic

†† Traffic volume measured in kilo packets

Table 7.6: FTP Traffic Volume and Red Packet Marking Percentage

This means that, globally, DiffServ flows performance should suffer from the presence of FTP traffic, in the sense that out of profile traffic will perceive the real network congestion state rather than the lower one perceived by green in-profile packets; that is, being HTTP flows partly green and partly red colored, each packet of that flow will receive a different forwarding treatment, differently influencing the overall flow's performance. Both HTTP and BE flows are entirely constituted of red marked packets; whether the load offered by HTTP application is low (say $\rho = 2$), then both DS and BE infinite flows will profit of the unused bandwidth, but with a key difference; DS FTP traffic will have the possibility to use part (i.e. the 80%) of its reserved free bandwidth being green marked, and to participate to the share of the unsubscribed free (i.e. again the 80% of the line rate's half) bandwidth with the same drop precedence as for BE packets; whether the unsubscribed bandwidth excess were equally shared between out-of-profile DS packets and BE ones, this would mean that BE FTP traffic would be likely to receive a small fraction of the excess (i.e. 20% of the line rate, whereas 60% would be used by DS FTP sources, meaning that BE would represent the 25% of the whole FTP transmissions).

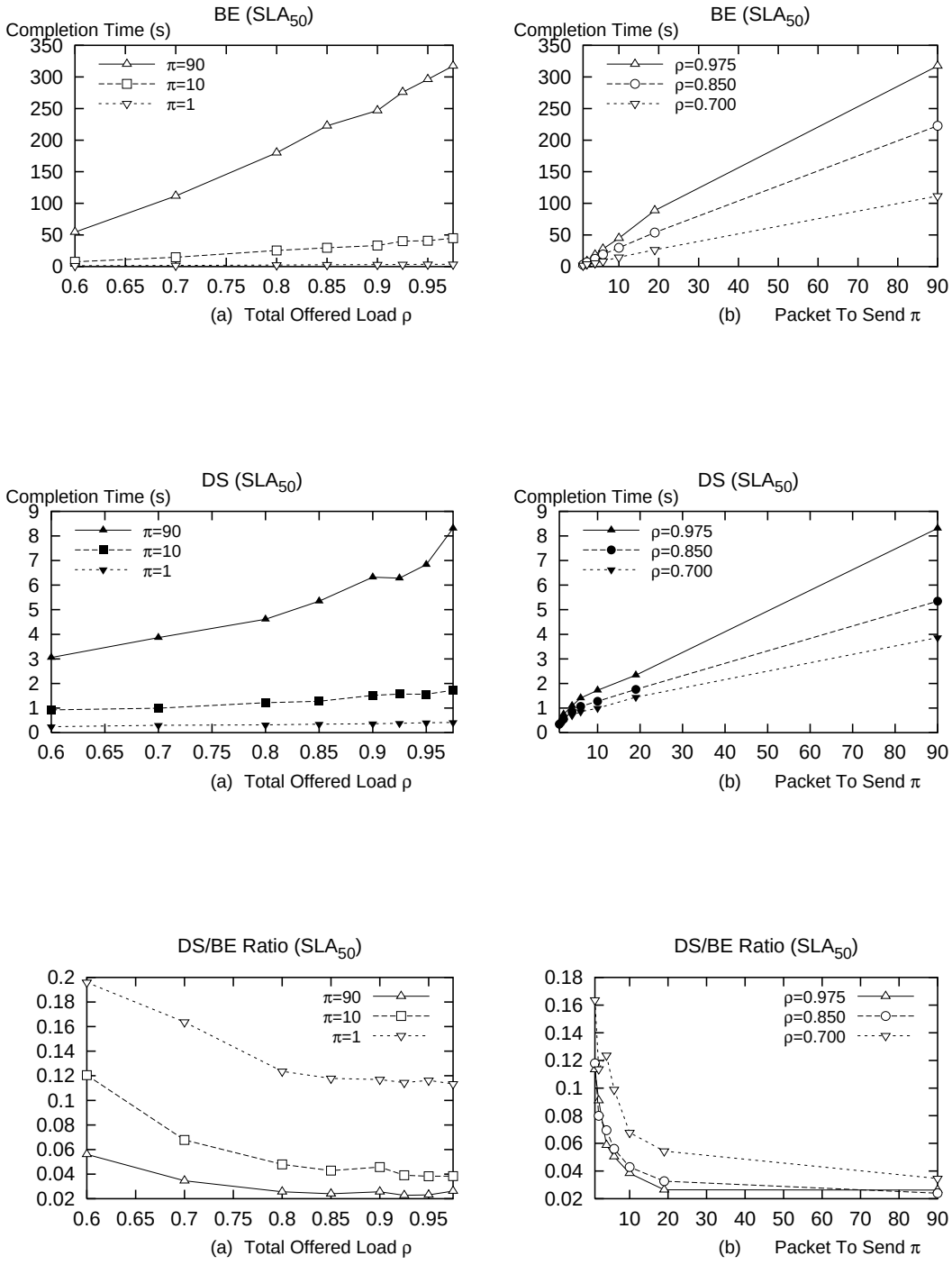


Figure 7.22: Completion Time: DS, BE and DS/BE Ratio (HTTP+FTP)

However, partly due to unresolved unfairness of RED gateway, which may result in synchronization, BE FTP traffic performances are even poorer than what expected: the extremely unfair distribution of FTP traffic volumes is reported in the first column of Tab. 7.6, showing that, for HTTP loads over $\rho = 0.6$, the FTP packets crossing the network are mostly ($\sim 97\%$) originated by DiffServ cloud.

Response time performances of HTTP flows are heavily influenced, even in the DS case, by presence of FTP traffic: completion time growth is linear rather than abrupt and earlier noticeable, as Fig. 7.13(DS,a) shows, at $\rho = 0.6$ long flows complete their transfer in a time that is the double than at $\rho = 0.975$. This is due to several aspects: firstly, no less than one packet out of five arrivals is marked red and then queued; then, network operates close to the linerate, and its congestion is sustained at lower HTTP loads rather than bursty; furthermore, at high HTTP loads, consistent performance degradation let us presume that drop percentage grows consistently and its late drop fractioning grows as well, partly as a consequence of the augmented traffic burstiness.

However, the effect of load on DS traffic appear limited if compared to BE case; Tab. 7.7 reports some ρ_{scale} values, ranged per flow length, that can be used as a simple analysis of completion time sensitivity: it can be seen that DS scaling factor, although depending on flow sizes, is kept below a reasonable value –even when calculated over the extended $I_\rho = (0.975, 0.2)$ load range– whereas BE is not. The same table reports also some π_{scale} factors sampled at different loads: in the DS case, π_{scale} grows linear with the load with a slight advantage of shorter flows. In the BE case π_{scale} factor analysis shows that completion time growth is far more different between short and intermediate BE flows, while the ratio between intermediate and long flows is less sensitive and non-monotonic with respect to load; however, there is no strong quantitative confidence on BE simulation result depicted in Fig. 7.13(BE,a), for reasons that will be further developed.

	$\rho_{scale}(0.975, 0.2)$		$\rho_{scale}(0.975, 0.6)$			$\pi_{scale}(10, 1)$		$\pi_{scale}(90, 10)$	
π	DS	BE	DS	BE	ρ	DS	BE	DS	BE
1	2.68	17.30	1.66	2.89	0.2	3.80	6.18	3.30	7.11
10	3.02	31.33	1.82	5.83	0.7	3.32	8.02	3.88	7.61
90	4.30	44.31	2.72	5.84	0.85	3.67	10.09	4.19	7.49
avg	3.40	34.11	2.17	5.51	0.925	4.17	12.32	4.81	7.06

Table 7.7: DS and BE per-π and per-ρ Scale Factor

The slope of the DS over BE performances with FTP sources decreases slower and shows less variation compared to the HTTP only case of Fig. 7.13(DS/BE,a): this indicates clearly that, as expected, in the former case there isn't a congestion-free load range, this entailing BE starvation to begin earlier. Another main difference is that in this latter case short flow's gain is much lower (thus DS/BE ratio is higher) with respect to longer flow; moreover, from Fig. 7.22(DS/BE,b) we gather that gain exponentially increase as π increases. This happens because short BE 1-π flows suffer, on average, the half of the other flows: that is, from Tab. 7.7, $\rho_{scale}(\pi = 1; 0.975, 0.6) = 1/2 \cdot \mu(\rho_{scale}(\pi; 0.975, 0.6))$ and similarly when $I_\rho = (0.975, 0.2)$.

It must be said that the specific implementation of FTP background traffic causes some unwanted effects, that may interfere with the confidence of BE data analysis; first at all, the completion time distribution changes substantially if sampled at t_{stop} or at t_{end}, as flows finishing after t_{stop} exibit a higher completion time; this can be exacerbated by FTP background sources, since these are still sending data indefinitely in this period of time, thus occupying almost all the available bandwidth; having more possibility, with respect to short lived flows, to grow their cwnd, they may be a cause of excessive performance degradation, preventing HTTP packets to find space in the router buffers. This can be gathered from comparison of Fig. 7.23(a,b), plotting

the average flow completion time cumulative distribution (for $\pi = 10$) as a function of the completion time abscissas; this is especially evident for BE flows at $\rho = 0.7$, in which case, the FTP have limited capacity available up to last HTTP flow is started, whereas after that time instant, only previously started short lived flows are sending packets in the network; then, as long as other HTTP flows completes their transmission, more bandwidth will be available to FTP sources, that are substantially monopolizing the queues.

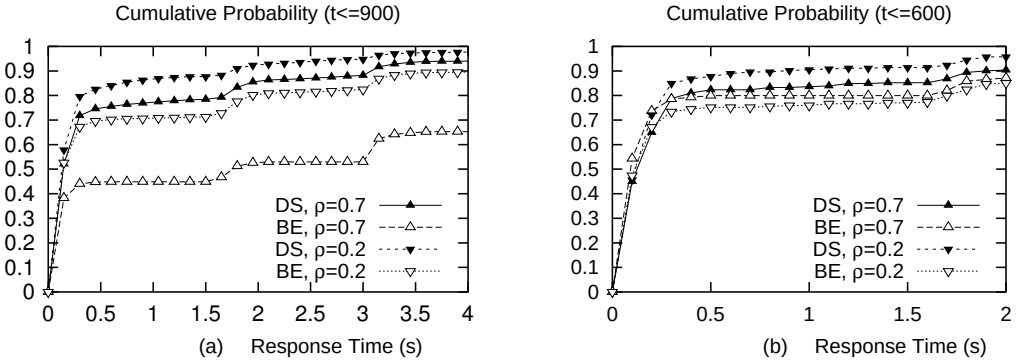


Figure 7.23: Timing Interval Effect on DS Flows Response Time Cumulated Probability

Another unexpected phenomenon arises evidently from observation of Fig. 7.24(a), plotting the DS response time distribution for different load and packet sizes, sampled over the whole simulation interval: intermediate flow's completion time achieves only a coarse set of values; this appears to be sort of strange synchronization-like problem, although each FTP source's starting time is shifted by a different randomized offset, and even if randomness is used in HTTP flow arrival time determination too. Furthermore, Fig. 7.24(b) shows that synchronization happens for shorter flows too: this is again due to FTP queues monopolization, and such spurious data problem should be corrected. A possible solution would be to control FTP traffic load via a configurable upper bound (see Sec. 7.6.2): that way, FTP traffic may be used to slightly exceed the subscribed bandwidth when $t \in (t_{start}, t_{stop})$, and to produce a constant load level as long as HTTP flows achieve completion network resources become unused when $t \in (t_{stop}, t_{end})$.

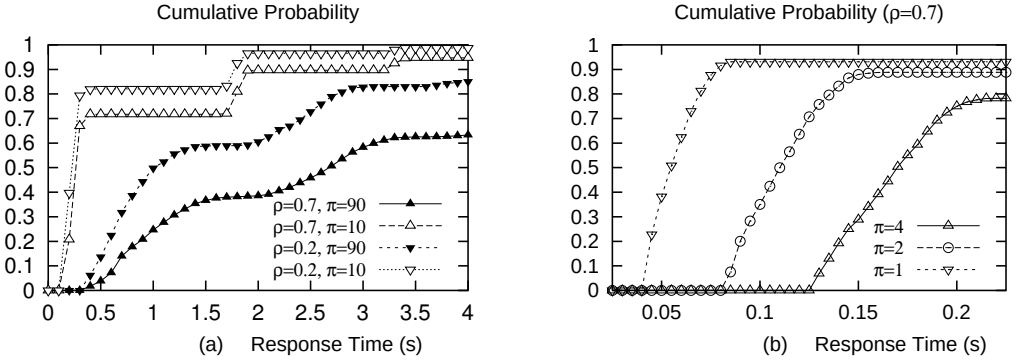


Figure 7.24: FTP effects on DS and BE Flows Response Time Cumulated Probability

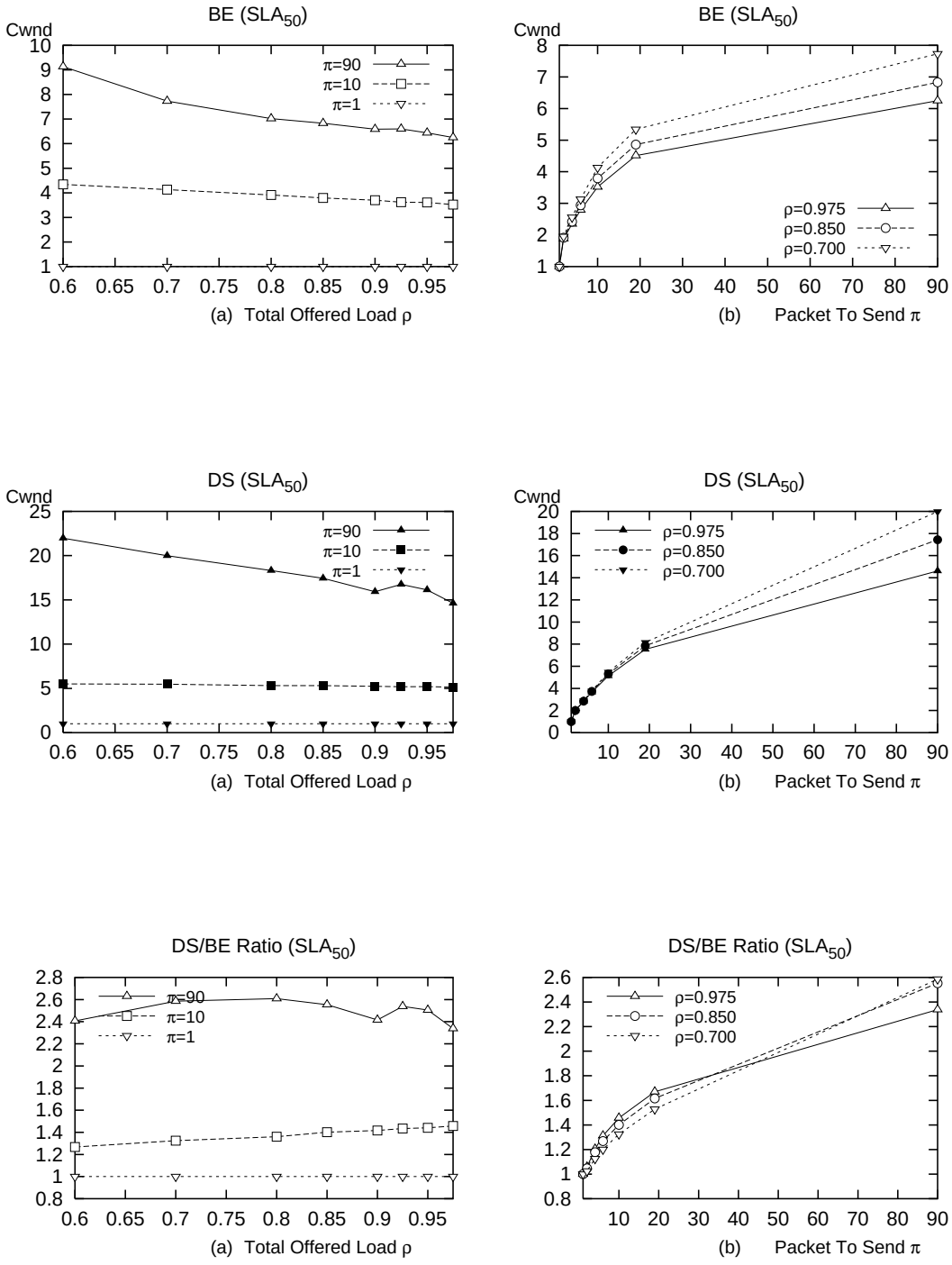


Figure 7.25: Maximum Congestion Window: DS, BE and DS/BE Ratio (HTTP+FTP)

Congestion window slope is similar in both the DS and BE cases of Fig. 7.25(DS,BE,a): this is confirmed in Fig. 7.25(DS/BE,a), showing the roughly constant gain in DS performances; however, DS flows are still better protected than BE ones from drop, thus reaching anyway the half of the cwnd achieved by HTTP only traffic, whereas BE cwnd reaches at best the previous worst case values.

The former observation can be further extended to explain the unfair DS/BE fractioning of the long lived FTP flows traffic volume introduced earlier in first column of Tab. 7.6; it can be argued that long lived FTP flows will have a congestion window gain not lower than those shown by the longest HTTP flows; therefore, DS FTP flows are able to take advantages of fast recovery and fast retransmit technique more than BE flows: DS background sources are less likely than BE ones to leave, due to RTO occurrence, the congestion avoidance controlled phase of their transmission.

As background traffic pushes the network in an overloaded state even for low HTTP traffic loads, drop percentages are higher than in the previous HTTP only simulations, in both DS and BE cases: the main difference is that BE drops are four times greater than DS ones: AR packet drops grow from $\sim 5\%$ (at $\rho = 0.6$) to $\sim 10\%$ (at $\rho = 0.975$), whereas BE drops range from $\sim 20\%$ to $\sim 38\%$; however, both BE and DS drop percentage growth are linear with respect to ρ, roughly doubling their value between $\rho = 0.6$ and $\rho = 0.975$. The explanation to the causes of the consistent growth of DS drop percentage, as depicted in Fig. 7.26(DS,a), is straightforward: at any HTTP load, there is not less than 20% of DS red marked packets, and even at $\rho = 0.2$ a small amount ($\sim 0.15\%$) of dropped green packets due to buffer overflow; since MRED implements a staggered set of parameters, before any green packet gets dropped, every red packet must already have been dropped. BE packet drop is less influenced from flow size, in the sense that the scale factor $\pi_{scale}(90, 1) \simeq 1$ is lower that in the HTTP only case (where $\pi_{scale}(90, 1) \simeq 1.5$), but the longest flow's drop percentage is always lower than those of shorter ones. Conversely, DS shortest $\pi = 1$ flows drop percentage (though no longer the lowest), is bounded, respectively lower and upper, by intermediate and longest flow drops. This effect was already noticed in the SLA_{100} case in the *absence* of FTP traffic, whereas in its *presence* packet drop tended not to be influenced by flow size. In the latter case, since DS were the only traffic type flowing in the network, its drop percentage showed a linear growth when $\rho < 0.8$ (where $\sim 10\%$ packets were dropped) and then an abrupt ($\sim 20\%$ at $\rho = 0.975$) drop growth: if BE traffic is put behind DS, part of the latter drops are "absorbed" by the former traffic class.

ρ		0.7	0.8	0.85	0.9	0.925	0.95	0.975
HTTP	Packets	255392	290718	316894	329585	340556	345684	355011
	E-Drop	10	15	43	36	25	18	18
	L-Drop	0	0	35	23	0	0	43
HTTP, FTP	Packets	562261	562031	562318	562597	562575	562134	562448
	E-Drop	0	0	0	0	0	0	0
	L-Drop	2534	2559	2725	2553	3098	2585	2455

Table 7.8: Green Marked Packets: Transmitted, E-Drops and L-Drops

FTP introduction when SLA_{50} is committed causes also the 0.5% of the green AR packets produced by FTP and HTTP application to incur in buffer drop; this is shown in Tab. 7.8, reporting, in both traffic cases, the number of green packets transmitted at the source, buffer drops and early drops for different network loads. In bursty traffic case green packet early drop occurs, though in a neglectable amount, indicating that RED is not able to manage the queue size: seldom, it may happen that sudden packet bursts cannot be handled by router queues and packets are necessarily dropped.

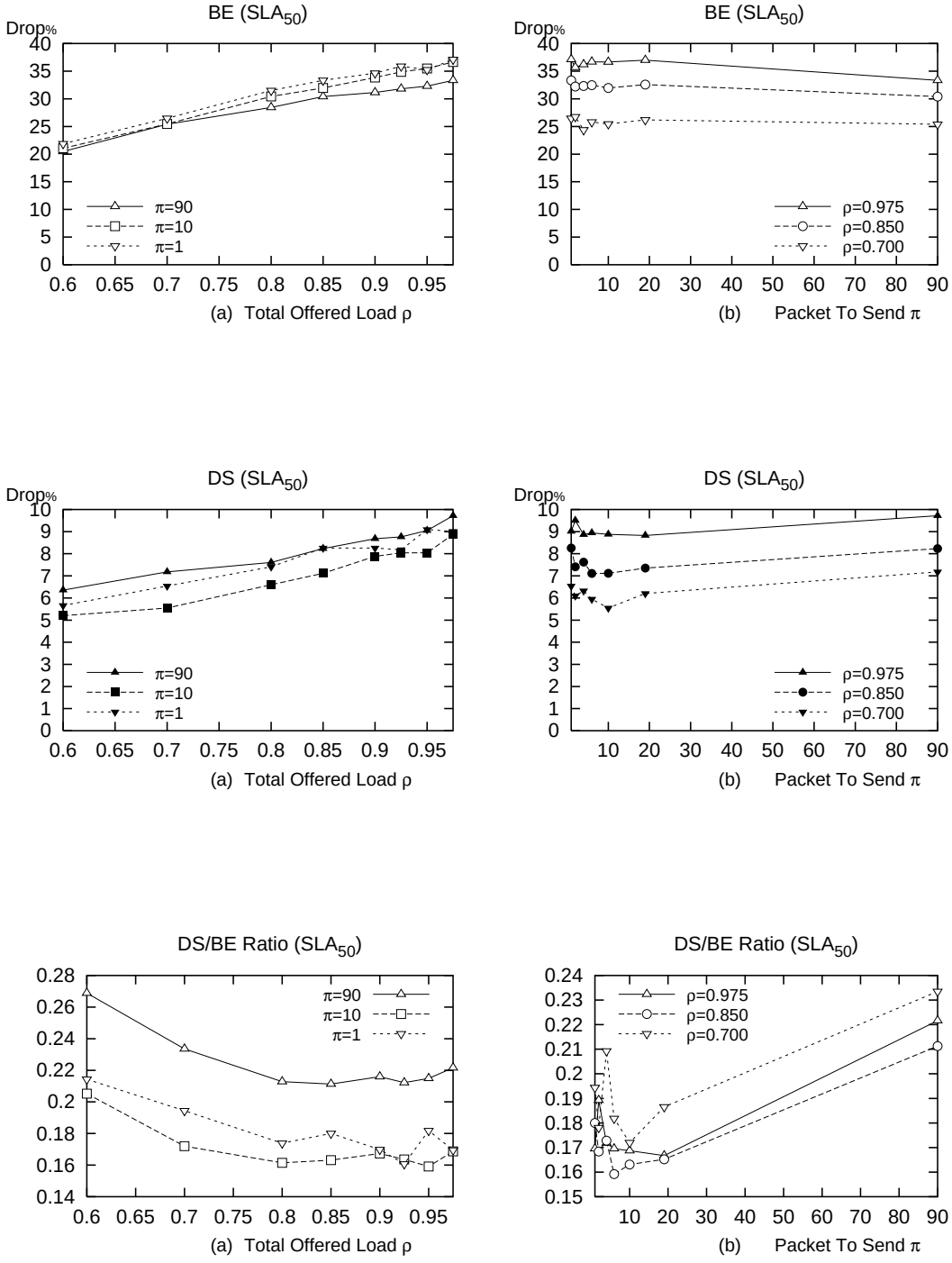


Figure 7.26: Packet Drop Percentage: DS, BE and DS/BE Ratio (HTTP+FTP)

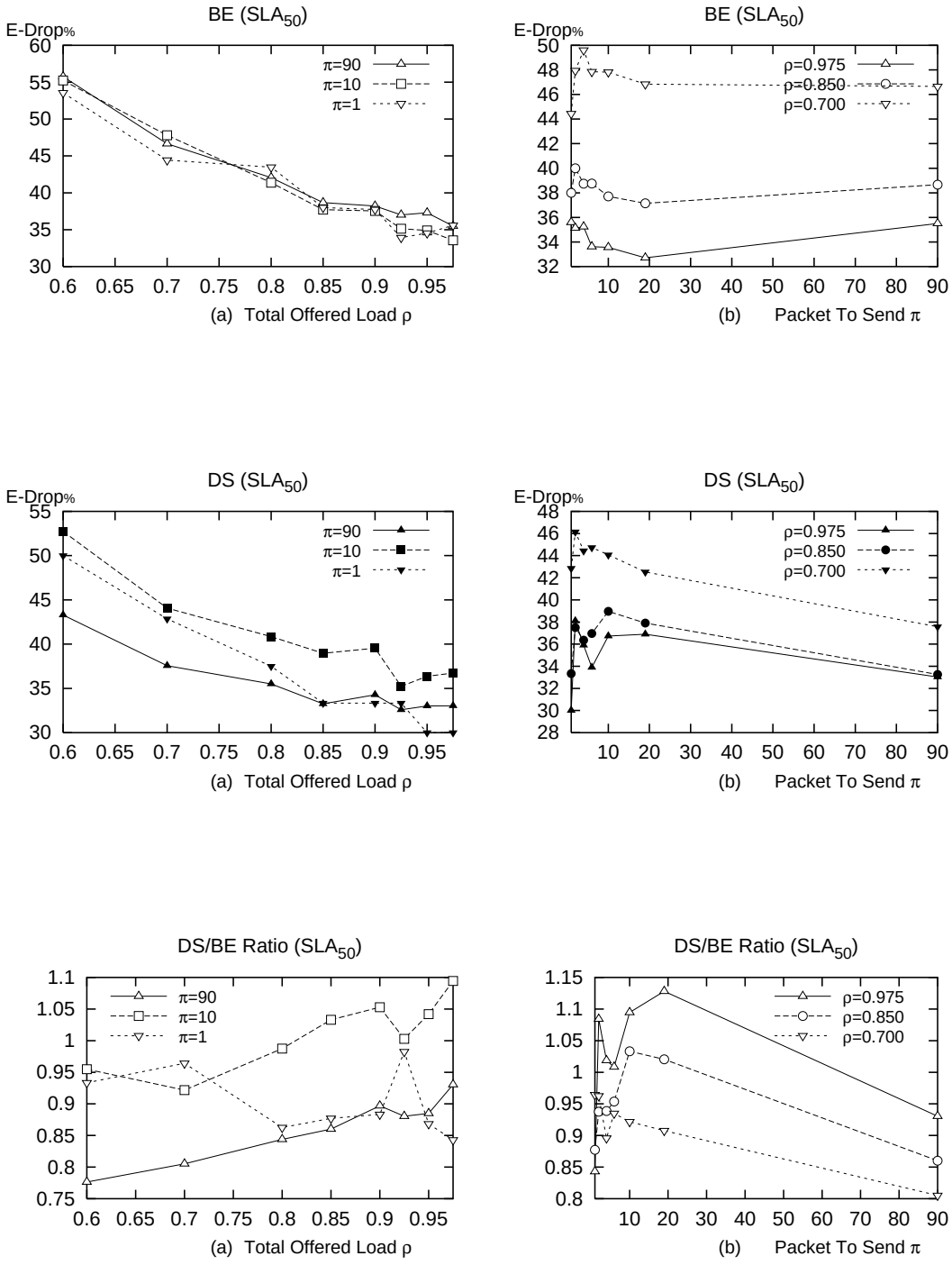


Figure 7.27: Early Drop Percentage: DS, BE and DS/BE Ratio (HTTP+FTP)

This might have been avoided, since RED timely signals incipient congestion based on a the smoothed average queue size; the latter parameter is not representive, when dealing with bursty traffic, of the highly variable instantaneous queue size: though different queue weights (i.e. of the present over the past) may produce better results, this still need further modeling and simulation; unfortunately, this phenomenon is not easy to quantify, since it occurs during short congestion periods, whose dependence on network offered load is not straightforward. The behavior is completely the opposite when infinite responsive sources, beside short lived ones, continuously send packets to the network: that is, green packet drop is always due to buffer overflow, and its drop percentage reach $\sim 0.5\%$ level; here, the effectiveness of RED might be compromised more by its parameter setting (the relationship between queue thresholds and buffer length is a key of importance) rather than to the effectiveness of the average queue length estimator algorithm. Early drop distribution for HTTP traffic is shown in Fig. 7.17: it can be added, to the earlier exposed considerations, that the noticeable amount of AR red marked packets entails a growth of early drop actions with respect to the SLA_{100} case; however, the RED mechanism is clearly more effective for infinite smooth flows than for bursty traffic: as the ρ parameter increase, and so does the burstiness, the percentage of early drop decreases.

			DiffServ				Best Effort			
	ρ	kΦ	ϕ	kΠ	kFR	(FR/Π)%	ϕ	kΠ	kFR	(FR/Π)%
	0.7	19	21	257	0.19	0.07	118	281	3.73	1.33
HTTP	0.9	24	38	341	1.98	0.58	1471	450	5.62	1.25
	0.975	26	72	383	4.27	1.11	2479	500	5.01	1.00
HTTP,	0.7	19	61	286	5.67	1.98	785	352	4.73	1.34
FTP	0.9	24	108	375	8.66	2.31	2363	484	4.35	0.90
	0.975	26	129	415	9.97	2.40	3133	538	4.21	0.78

Table 7.9: Effects of FTP on Fast Recovery and Other HTTP Flows Parameters

Finally, the drops fraction due to fast recovery is plotted, as usual for BE and DS traffic and DS/BE ratio, in Fig. 7.29; evidence is that DS achieves a FR percentage that depends on real network load rather than HTTP load ρ, whereas BE FR probability linear decrease as long as ρ grows, thus giving a DS FR gain proportional to the HTTP load.

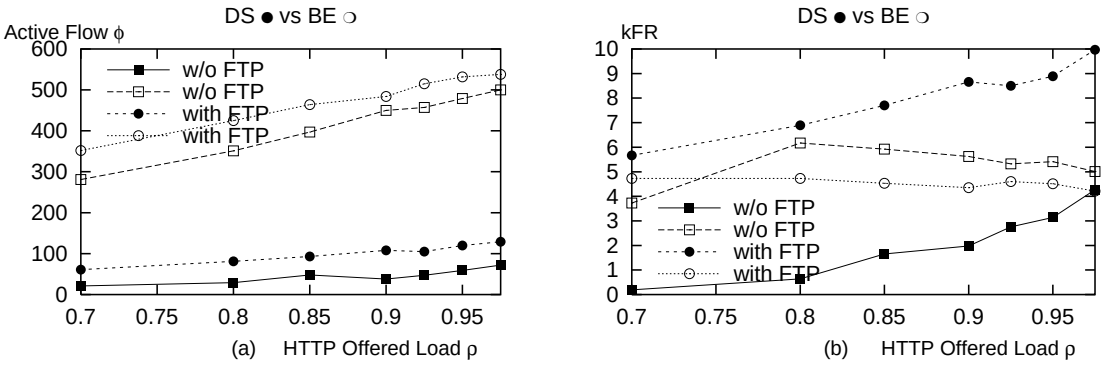


Figure 7.28: Maximum Active Flow and Fast Recovery Percentage

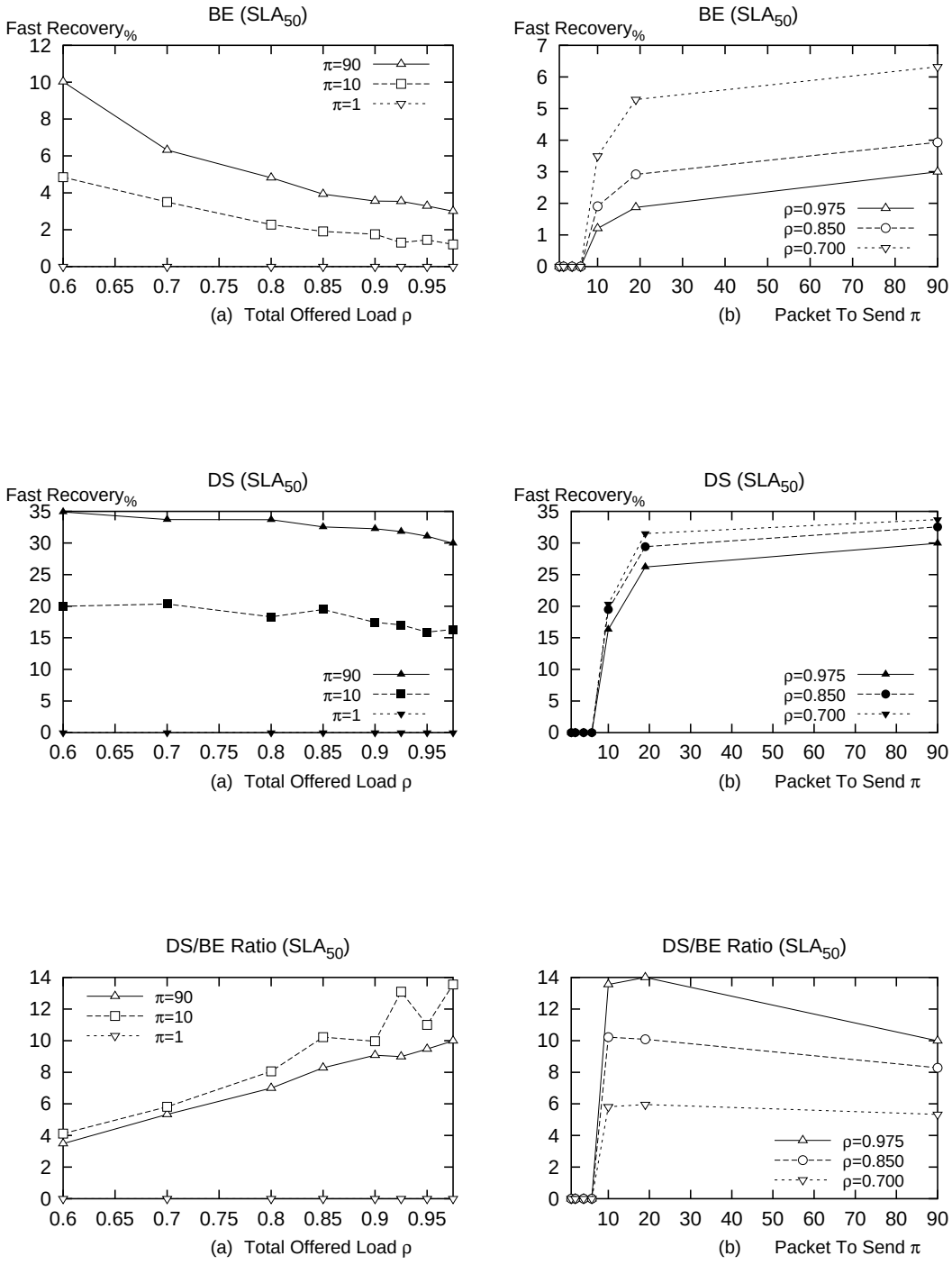


Figure 7.29: Fast Recovery Percentage: DS, BE and DS/BE Ratio (HTTP+FTP)

Tab. 7.9 reports some HTTP key parameter –covering the whole simulation length and expressed in kilo-packets– such as the maximum number of flows active at once ϕ, number of totally created flows during the whole simulation Φ, number of packets created at the source Π, FR number and percentage over Π, for different traffic and service type; first important remark is that BE produces 30% more packets than DS at any load and in any traffic condition, and obviously faces the same number of totally created flows Φ; maximum active flow data are depicted in Fig. 7.28(a) as a function of ρ: both DS and BE ϕ growth is linear in ρ, but the latter traffic type's slope is many (~ 30) times higher.

A possibly interesting remark comes from the comparison of FR data, where this time the percentage is calculated over the total number of packets created whose quantitatively equivalent visual representation, that is, the fast recovery number, is depicted in Fig. 7.28(b): whereas BE goes in fast recovery in a measure that is roughly constant on HTTP load and is little influenced by FTP traffic, DS shows a completely different behavior. Substantially, FR is proportional to ρ and furthermore the FTP effect results in an addictive offset: therefore DS fast recovery percentage is proportional to the drop percentage, as the drop behavior showed very similar aspects.

7.4 SLA_{25-75} Simulations

To extend the analysis of DS and BE interactions, simulation were run in a scenario similar to the previous one, varying the AR service level specification, expressed –as usual– as bottleneck percentage. Thus, SLA_{25} means that 25% of the bottleneck (250 kbps) are sold as assured rate DS service, while the remaining resources are available to BE and out-of-profile DS packets, while the SLA_{75} case mirrors to the former. As a mere notation, SLA_x will be said to be greater (or higher) than SLA_y when $x > y$ and smaller (or lower) otherwise; thus, SLA_{25} is also the lowest, SLA_{50} the intermediate and SLA_{75} the highest.

Full SLA_{25} and SLA_{75} simulation results, with and without background traffic, are reported in Appendix C without further explanation; here, the analysis of results will focus on two parameters: completion time, a user-centric metric, and packet drops, key in AR service and drop level deployment effectiveness. We expect that the different service level agreed upon should have minimal or no effects on DS traffic performances, whereas it could lead to consistent BE performance degradation; performance comparison for different SLAs are done in order to further broaden, validate and detail considerations derived from the single SLA_{50} case: if the results issued from higher and lower SLA cases are compliant with those drawn from the former, then, especially in the DS traffic case, SLA_{50} performance analysis will be used as reference.

7.4.1 Completion Time

Response time performances of Fig. 7.30 are plotted separately for each SLA and traffic type, in order to permit performances comparison for different π-long flows with respect to ρ, as well as different ρ with respect to π.

In the need to compare how performances are affected by the establishment of different SLA, Fig. 7.31(a) superimposes DS completion times, averaged over all flow length, obtained with different SLAs; since response time shows similar value for any size of the assured rate, its growth as a function of ρ il largely independent of SLA. This is reasonable as DS packets are generated at a rate meeting, on average, the contacted profile: as DS flows are almost entirely green marked, their higher forwarding probability is assured via better protection against dropping. An interesting remark is that DS completion time behavior seems, from Fig. 7.31(b) inspection, to be proportional to the root square of the number of packets to send.

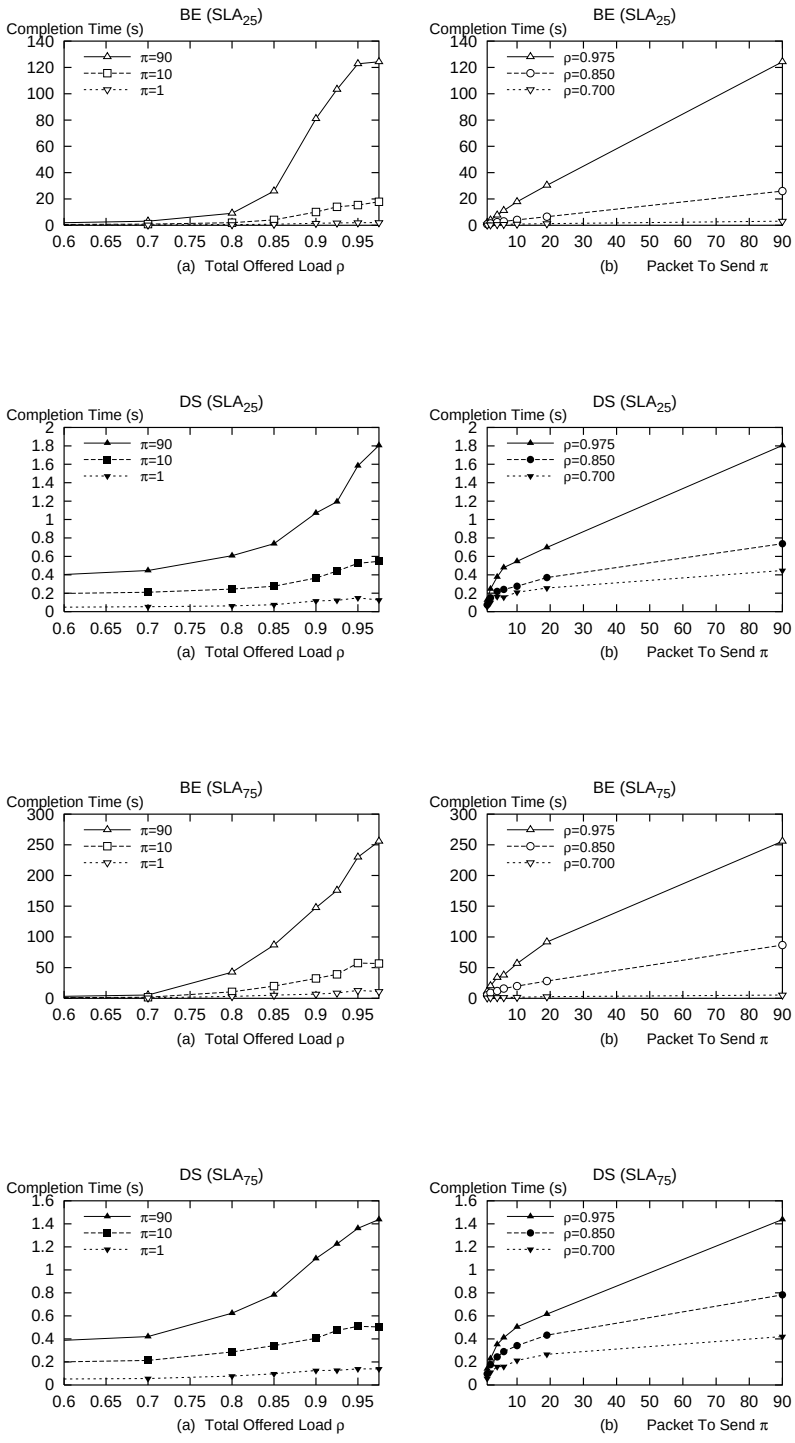


Figure 7.30: DS and BE Completion Times for Different π, ρ and SLA (HTTP)

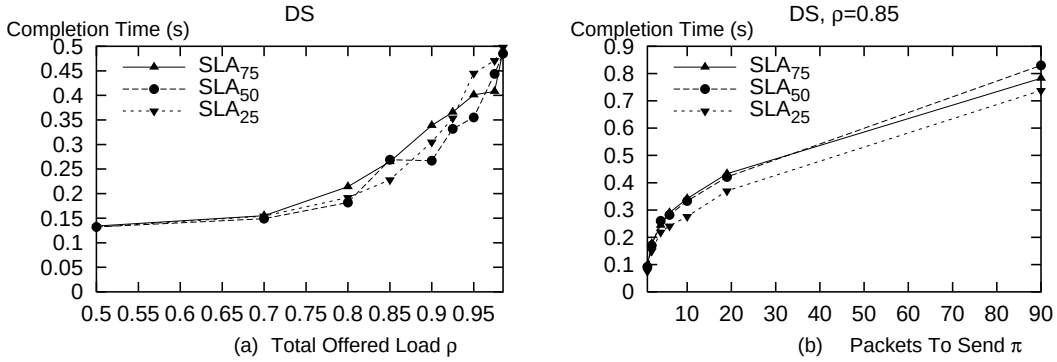


Figure 7.31: DS Completion Time (HTTP, $SLA_{25,50,75}$)

BE flows, as shown in Fig. 7.32(a), suffer more at higher SLA; especially in HTTP only case, DS flows participate to the excess in a measure that mainly depends on traffic burstiness, since the offered load is modeled to equal, on average, the reserved bandwidth; traffic burstiness and congestion are not influenced by the SLA, this meaning that the amount of out-of-profile packets at a same load are likely to be of the same order in any SLA_x case: this competing traffic would be more aggressive, as its weight would be more important, when competing for the share of a relatively narrow bandwidth beyond the assured. Face to π, response time growth is linear when DS load is lower (e.g. SLA_{25}), whereas its behavior is similar to the DS one when the SLA increases, as depicted in Fig. 7.32(b).

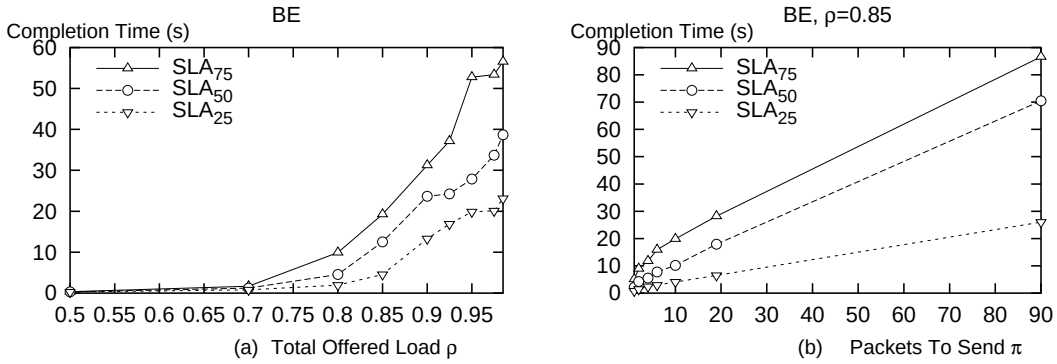


Figure 7.32: BE Completion Time (HTTP, $SLA_{25,50,75}$)

Considering the DS gain over BE as the inverse of DS/BE ratio depicted in Fig. 7.33, the higher gain is achieved, as a consequence of the previous considerations, at the highest SLA, thus where BE performances are more degraded. Faces to π, the gain brought by DS deployment with respect to BE traffic, is unfairly distributed among flows of different length, with increasing longest flows advantaging as long as the SLA decrease: confirming that BE dependence on flow length on is no longer linear but similar to DS behavior, DS over BE ratio is roughly constant on π.

In order to prove the completion time behavioral differences between DS and BE services when SLA_{75} is applied, the data collected have been modeled via simple functions of either ρ or π; models depicted in Fig. 7.34 have been fit using the nonlinear least-squares (NLLS) Marquardt-Levenberg

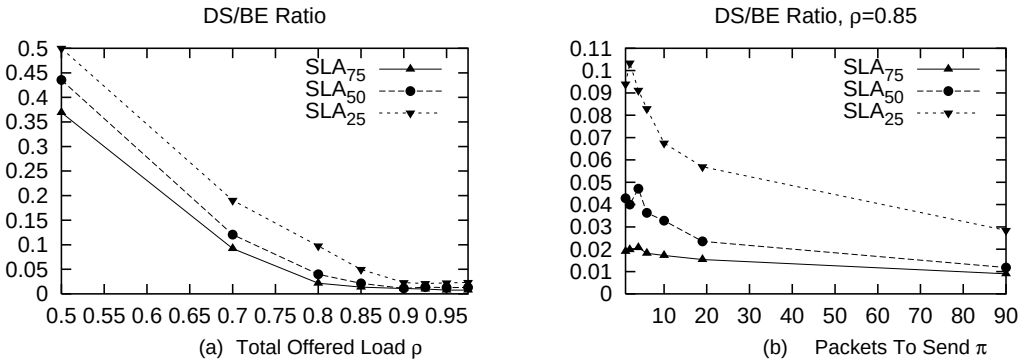


Figure 7.33: DS/BE Completion Time Ratio (HTTP, $SLA_{25,50,75}$)

algorithm, which is not guaranteed to converge to the global optimum (that is the solution with the smallest sum of squared residuals); the analytical approximations of the data, without entering further in details, have been modeled using the following equation set:

$$\begin{cases} \mathrm{CT}_{DS}(\rho) = a \cdot \rho^2 + b \cdot \rho + c & \text{averaged over } \pi \\ \mathrm{CT}_{BE}(\rho) = \begin{cases} a \cdot e^{\rho} + b & \rho < 0.95 \\ a \cdot \rho + b & \rho > 0.95 \end{cases} & \text{averaged over } \pi \\ \\ \mathrm{CT}_{DS}(\pi) = a \cdot \sqrt{\pi} + b & \text{at the fixed load } \rho = 0.85 \\ \mathrm{CT}_{BE}(\pi) = a \cdot \pi + b & \text{at the fixed load } \rho = 0.85 \end{cases}$$

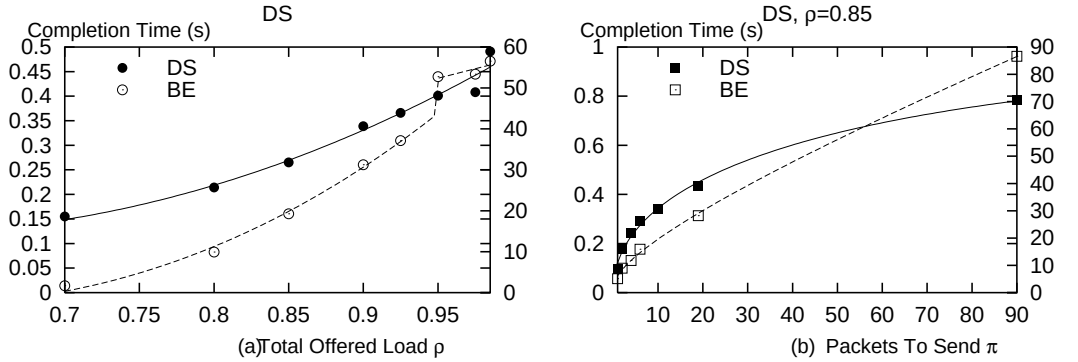


Figure 7.34: Completion Time Data and its Analytical Approximation

While DS flows show almost no dependence on SLA, with anyway a slight performance improvement as long as the assured rate increases, the FTP presence leads DS traffic to a macroscopic behavioral difference: its response time increase, reaching values similar to those earlier achieved in the SLA_{50} case. BE performance is affected by both SLA variation and FTP presence; the latter introduces a time offset and response times is consistently affected even at low HTTP loads, while the rate parameter subscribed to the SLA is mainly responsible for the abrupt slope increase; however, as can be observed in the HTTP only case too, the response time nearly *saturates* (in

the sense that response time growth is no longer exponential but linear) at high loads: this is also underlined by $CT_{BE}(\rho)$ model proposed.

7.4.2 Packet Drop

The use of different levels of drop preference depending on traffic type and stream compliance to a specific profile, ensures that the limited drop amount for DS traffic –as expectable– is largely independent of specific SLAs; anyway, from Fig. 7.35(a) it can be gathered that when SLA_{75} is the chosen agreement, the drop percentage at $\rho > 0.9$ grows more gradually than lower SLAs. FTP sources introduction entail, as already noted in the SLA_{50} case, a more gradual drop growth as ρ grows, and further increments the DS drop percentage of $\sim 7\%$ for any SLA.

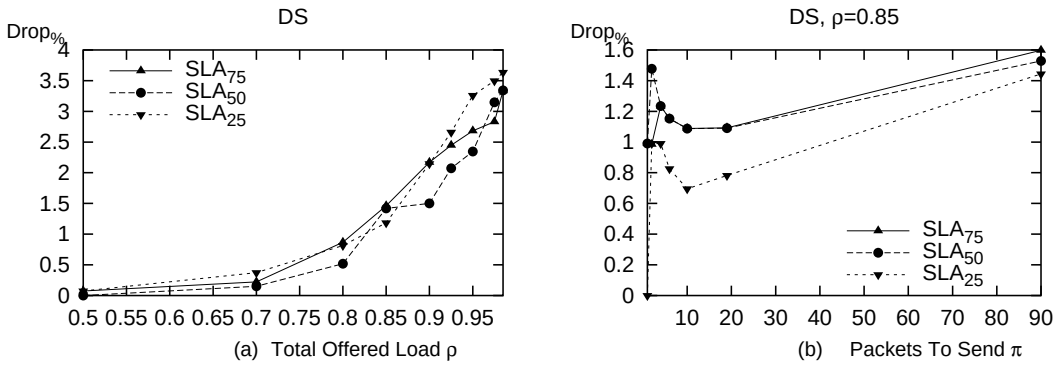


Figure 7.35: DS Packet Drop Percentage (HTTP, $SLA_{25,50,75}$)

BE traffic performances are indeed strongly influenced by the amount of free bandwidth beyond the assured; the BE drop behavior when ρ varies and for different SLAs is depicted in Fig. 7.36(a).

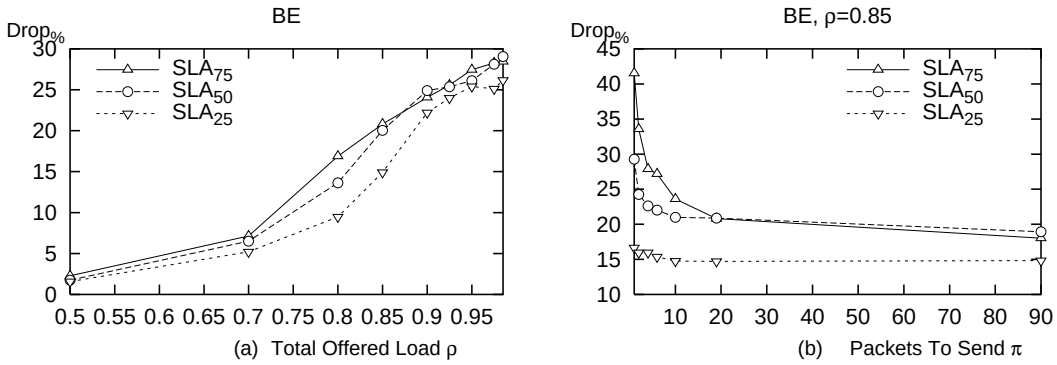


Figure 7.36: BE Packet Drop Percentage (HTTP, $SLA_{25,50,75}$)

When the total HTTP offered load is lower than $\rho = 0.7$, thus when the network is lightly congested, BE traffic shows almost no dependence on the established SLA; this no longer holds if the network experiences a relevant congested state, thus when $\rho \in (0.7, 0.9)$: in this load region drop percentages are considerably higher for higher SLAs (e.g., at $\rho = 0.8$, SLA_{75} drop percentages are twice as much as those achieved with SLA_{25}); furthermore, drop slopes present a concavity

change within the load interval, and it can be gathered that the flexus point is placed at lower ρ for higher SLAs; eventually, when network is heavily congested, drop percentages grow with the same order Fig. 7.36(b) depicts the BE drop percentage as a function of flow size: evidence is that the relative oddity of drop percentages among π strongly increases with the SLA (e.g. with SLA_{75} 1-π flow packet drops are twice as large as 90-π ones).

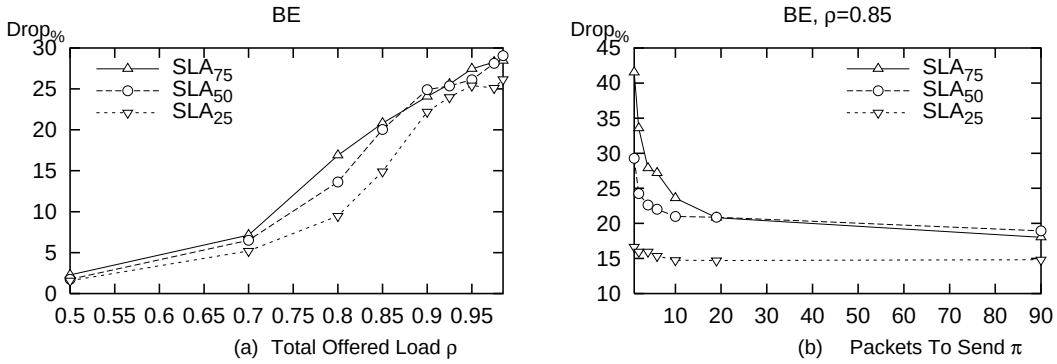


Figure 7.37: BE Packet Drop Percentage (HTTP+FTP, $SLA_{25,50,75}$)

Background FTP traffic introduction partially "corrects" the unfairness among flows (e.g. with SLA_{75} 1-π flow packet drops are ~ 1.25 greater than 90-π ones) at a price of increasing the drop percentage at any load; furthermore the initial drop percentage offsets, that is those introduced at light network congestion, depend on the SLA used growing with the amount of reserved bandwidth; then it can be seen from Fig. 7.37 that, especially for high SLA, drop percentage can be modeled with a linear function –whose slope is basically the same for any SLA– rather than with an exponential one.

Let us now examine the drop performance separately for each SLA, to provide comparison of different π long flows with respect to ρ as well as different ρ with respect to π, as for the earlier SLA_{50} case: interesting considerations can be gathered from relative comparison of each graph depicted in Fig. 7.38. DS packets drop are always kept below $\sim 4\%$, but there is less than 1% difference from SLA_{75} to SLA_{25}; furthermore, in the former case shortest packets drop are bounded between longest and intermediate, whereas in the latter case short flows drop begins at load higher than $\rho = 0.85$.

BE performances are heavily affected by the SLA: comparing SLA_{25} to SLA_{75} it can be gathered, observing per-flow size drop probability as a function of ρ, that the drop distribution oddity is caused by a reduced amount of free bandwidth, thus an increase of the SLA.

Finally, comparing DS SLA_{25} to BE SLA_{75} drop percentage as a function of π, one can observe that in the former case there is more difference when load increase from $\rho = 0.7$ to $\rho = 0.85$ than $\rho = 0.85$ to $\rho = 0.975$, whereas the latter case shows a mirrored behavior: this is a consequence of the relative difference of drop percentage slope as a function of ρ.

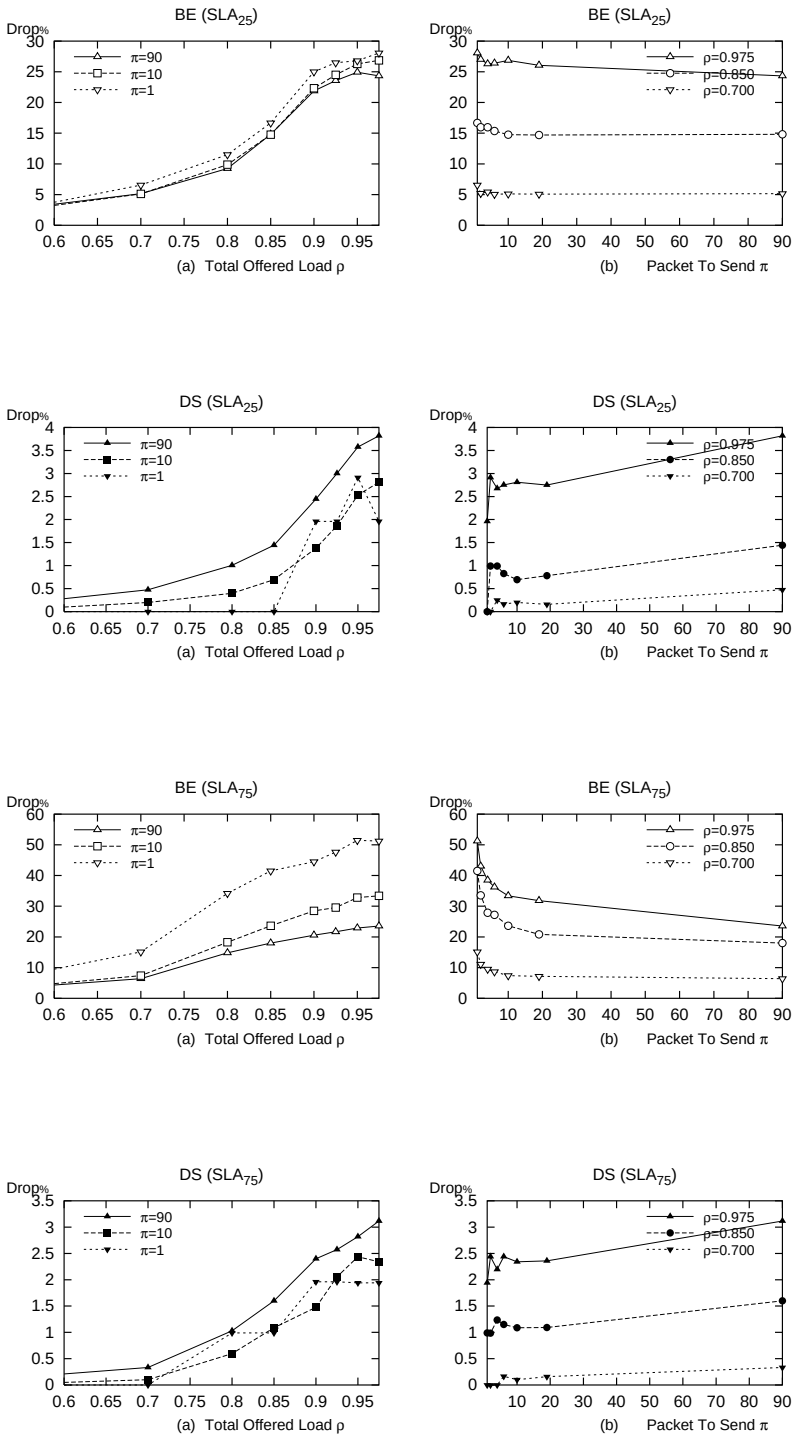


Figure 7.38: DS and BE Packet Drop Percentage for Different π, ρ and SLA (HTTP)

7.4.3 Other Considerations

BE early drop percentage, plotted in Fig. 7.39(a), decreases al long as ρ grows, with lower values for a same ρ as long as SLA increases, while DS performance –not shown here– shows a roughly constant behavior on network load; however, DS early samples are not trustable due to the limited amount of drop. Moreover, from Fig. 7.39(b) it can be seen that, as the SLA increases, the relative difference among flows of different size further advantages longer flows, whereas this phenomenon is almost not noticeable in SLA_{25} case. The larger early drop amount in SLA_{50} and SLA_{75} indicated that buffers are mainly occupied by longer flows, while short flows are mostly late dropped: as a consequence, it can be gathered that a flow sending a few packets is more likely to find no room in queues, and thus to incur in buffer overflow; this thesis is confirmed by the analysis of drop percentages, especially comparing the results issued from the highest and intermediate SLA simulations to those of lowest SLA one, where this trade off was inexistent.

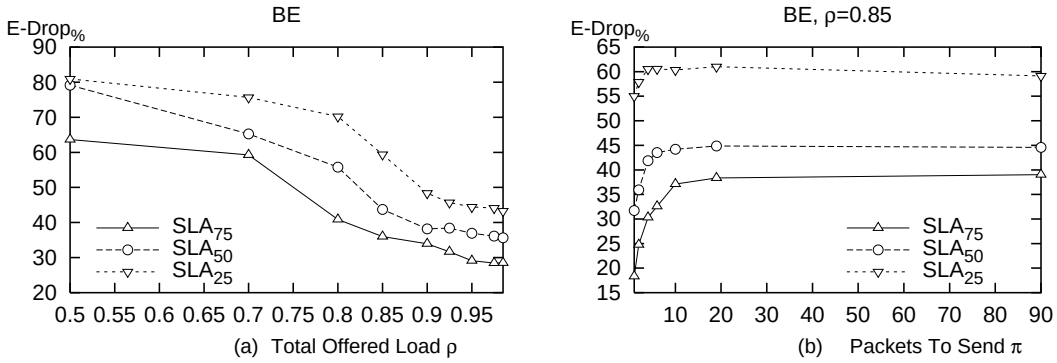


Figure 7.39: BE Early Drop Percentage (HTTP, $SLA_{25,50,75}$)

Another main difference affecting DS and BE services is the behavior of the maximum congestion window value achieved by flows during a transmission: these values are superimposed in Fig. 7.40.

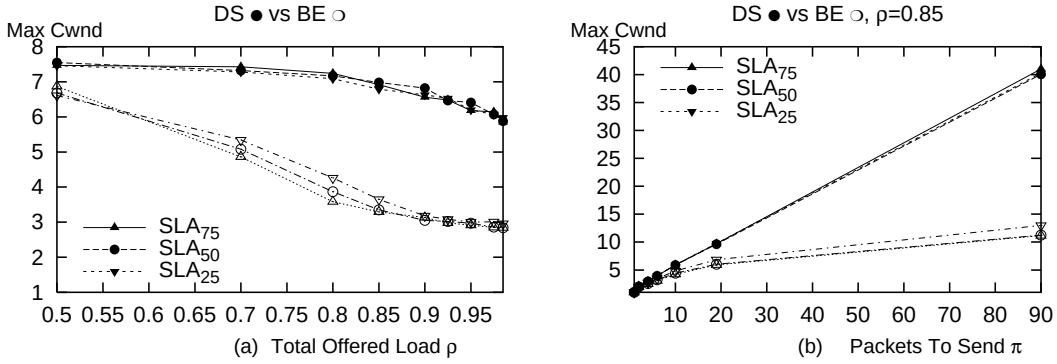


Figure 7.40: DS and BE Maximum Congestion Window (HTTP, $SLA_{25,50,75}$)

When the maximum cwnd is averaged over all flow sizes, as in Fig. 7.40(a), BE and DS cwnd have an opposite concavity, at least for SLA_{75}. DS cwnd performance degradation is gradual,

independent of SLA, less influenced by ρ, and its worst case values are doubles with respect to BE. BE cwnd linear decrease begins at low network loads and furthermore, as indicated from other parameter analysis, the higher SLA imposes serious service limitations: this results in a lower achievable cwnd value over all load ranges, with the exception of heavy congestion load range, where degradation is equal for any SLA in place. When the cwnd is analyzed with respect to flow lengths, BE cwnd seems to be proportional to the square root of π, whereas DS cwnd is directly proportional to π.

7.5 SLA_{50} Simulations: Varying DiffServ Load

In all the scenarios used during previous simulations, the HTTP traffic volume generated by each of the DS and BE clouds was modeled to equal, on average, the bandwidth granted for that specific service: thus each cloud's HTTP load (i.e ρ_{DS} and ρ_{BE}) was constant and set according to the in place SLA, while the overall offered HTTP load ρ was made variable; then, by means of FTP introduction, the traffic produced by each cloud was forced to exceed the limits imposed by the established SLA, offering the opportunity to examine the performances of partly out-of-profile HTTP flows at varying overall HTTP loads.

In the scenario presented in this section, the peering DS domains still offer two services, Best Effort and DS Assured Rate, and they have contracted a SLA_{50}: that is, half of the bottleneck's bandwidth is reserved to a DS while the other half is available to the share between DS excess traffic and BE flows. However, in order to cover a larger scale traffic sensitivity analysis, the varying simulation parameters are symmetrical to the previous ones; overall HTTP load is no longer variable, being fixed to $\rho = 0.85$, while each cloud's fractioning of that overall load is made variable: specifically, being $\rho_{DS} + \rho_{BE} = 1$, both DS and BE results will be analyzed as a function of ρ_{DS}. Furthermore, since $\rho = 0.85$ and SLA_{50} is in place, the DS load threshold $\rho_{DS,th}$ used to discriminate between *in* and *out* packets can be expressed as a function of both the total HTTP load ρ and the $x\%$ assured rate percentage as $\rho \cdot \rho_{DS,th} = x\%/100$, giving $\rho_{DS,th} \simeq 0.588$; simulations were conduced for overall HTTP loads ρ other than $\rho = 0.85$: Appendix D reports complete simulation results obtained when $\rho = 0.975$ and $\rho_{DS,th} \simeq 0.513$, some of which are used here with comparison purposes.

This scenario was designed to analyze and compare each service's performances reaction to SLA exceeding, enhancing the possibility of a finer load parameter tuning with respect to approach of over-subscribed traffic generation via FTP introduction. Similarly to previous analysis, the behavior of DS and BE aggregate characteristics will be plotted with respect to ρ_{DS} and π parameters:

- as a function of the DS cloud fractioning ρ_{DS} of the overall HTTP traffic load $\rho = 0.85$, for a set of flows with different amount of packets to send $\pi \in \{1, 10, 90\}$
- as a function of π for a set of $\rho_{DS} \in \{0.4, 0.5, 0.6\}$;

The DS/BE ratio plots would no longer be meaningful if both parameter values were sampled at $\rho_{cloud} = \rho_{DS}$, since traffic comparison would be effectuated between an oversubscribed case and the complementary undersubscribed one. Recalling the previous obtained results, it is not difficult to hypothesize that, being DS flows anyway overperformant with respect to BE ones, their performance improvement would be sensible in any case; this is due to green packets protection, assuring better performances to DS flows, coupled with the fact that even though $\rho_{DS} = 1$ the amount of green packets would be anyway greater than the red amount: this can be directly inferred from $\rho_{DS,th} > 0.5$. Furthermore, when $\rho_{DS} > \rho_{DS,th}$, DS oversubscribed traffic will be red marked from the TSWTCM: this will enhance performances of BE traffic, competing for a narrow bandwidth share with DS too aggressive and therefore penalized flows; it is not unwise to predict that DS performances will be negatively influenced by the growing amount of red packets as long as ρ_{DS} grows, implying a relative performance degradation with respect to lower DS cloud's loads. As a result, the DS performance gain obtained would decrease as long as ρ_{DS} increases, thus giving no indication useful in analysis of different services' performances.

7.5.1 Packet Drop and Early Drop

DS and BE services reaction to traffic over and under subscription are really different: in order to provide a clearer completion time performance description, it has been decided to firstly examine the factors which it does depend on.

Both per-flow and aggregated packet drop performance analysis, especially in the DS service case, cannot be untied from per-color statistics reported in Tab. 7.10, where Δ is used to indicate packet drops, Π^{sx} counts the packets *transmitted at the source* and the statistic are reported as percentages; when $\rho = 0.85$ the number of packets totally created in the simulation will be roughly 700k: different DS HTTP loads will thus unbalance per-cloud fractioning of the total packet number in a measure proportional to each cloud's load. To be more precise, the average number of packets to send $\overline{\Pi}$ determined by the model (rather than the Π^{sx} packet count) directly depends on the load ρ and on the line rate B; however, though Π^{sx} depends on network specific reaction to congestion, the maximum achieved value of the Π^{sx} difference within the ρ_{DS} range is $\Pi^{sx}(\rho_{DS} = 0.1) - \Pi^{sx}(\rho_{DS} = 0.9)$ =29k (less of 4% of Pi^{sx}).

ρ_{DS}	Δ/Π^{sx}	$\Delta_{BE}/\Pi^{sx}_{BE}$	$\Delta_{DS}/\Pi^{sx}_{DS}$	$\Pi^{sx}_{Red}/\Pi^{sx}_{DS}$	Δ_{Red}/Δ_{DS}
0.1	12.37	13.49	0.00	0	-
0.3	15.71	20.89	0.04	0.04	97.22
0.4	14.53	21.72	0.38	0.61	90.58
0.45	12.06	19.79	0.60	1.14	94.50
0.5	12.36	21.20	1.31	3.14	98.70
0.55	10.62	19.35	1.88	5.31	99.01
0.6	8.74	16.75	2.57	8.86	99.85
0.7	8.29	15.92	4.52	19.87	99.95
0.9	8.70	15.89	7.80	39.25	100.00

Note: all the ratio values are expressed as percentages

Table 7.10: BE and DS Drop Related Percentages: Packet Colors Effects

From first column of Tab. 7.10 it can be gathered that overall drop percentage Δ/Π^{sx} is lower when DS traffic, rather than BE, exceeds it SLA; this is partly due to specific MRED settings: when traffic is mostly red marked, the number of contemporary active active flows grows, and thus the red drop probability, being the red buffer size actually narrow. DS and BE aggregated drop percentages are reported in second and third table columns, while their per-flow size aggregate's drop probability are plotted in Fig. 7.41. Evidence is that DS drops show simple monotonic dependence on cloud load, whereas this is not true for BE traffic; above $\rho_{DS} = 0.475$, thus when AR service uses $\sim 80\%$ of its reserved resources, DS packet drop percentage slope abruptly increases: this happens because DS flows begin to be constituted from a growing sensible amount of red marked packets (as the last but one column of Tab. 7.10 reports) constituting almost the totality of DS drops (from last table's column); however, even when more than 40% of DS traffic opportunisticly participate to the excess bandwidth share, the DS drop percentage is roughly the half of BE one. BE traffic performance is negatively influenced by DS traffic presence especially when $\rho_{DS} \in [0.2, 0.7]$: for loads outside that range, this means that a minimal amount of green packets does not massively increase BE drops, and further that BE traffic better profits of the free line rate when it is created at a rate lower than the former. Experiments run at $\rho = 0.975$ do not show a local minimum of BE drop percentage (as for $\rho = 0.975$ and $\rho_{DS} = 0.475$ allowing to hypothesize that this is due to untrustable data); furthermore, when the overall offered load is pushed near to the line rate, BE drops are less unfairly distributed among flows with different π, but, especially for short flows, there is a reduced performances improvement when $\rho_{DS} \notin [0.2, 0.7]$ with respect to $\rho_{DS} \in [0.2, 0.7]$.

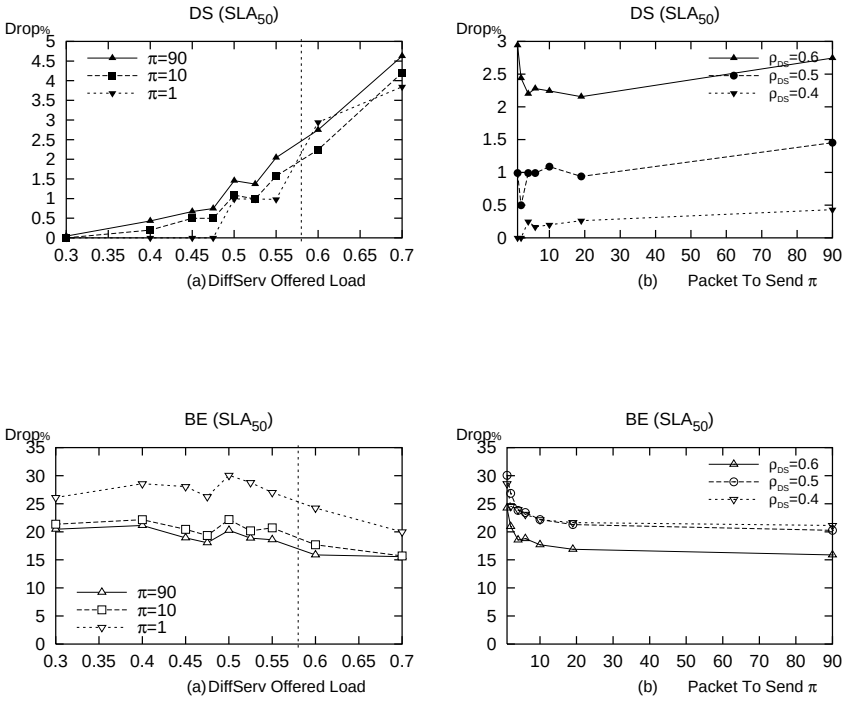


Figure 7.41: DS and BE Drop Percentage: Varying ρ_{DS} (HTTP, $\rho = 0.85$)

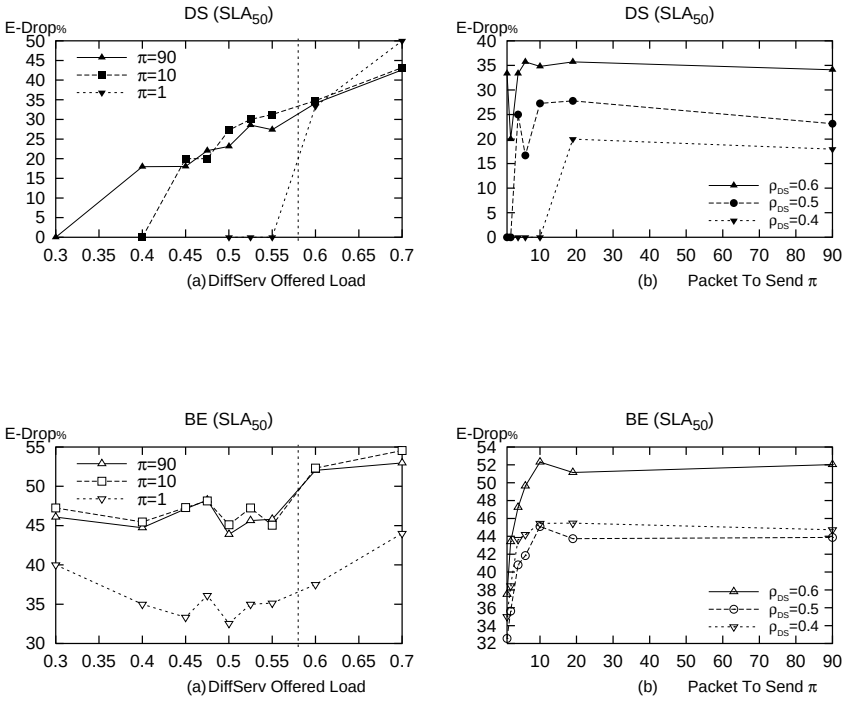


Figure 7.42: DS and BE Early Drop Percentage: Varying ρ_{DS} (HTTP, $\rho = 0.85$)

Early drop percentages, shown in Fig. 7.42 for both DS and BE services, increases as long as ρ_{DS} increases whether packets belong to AR flows: RED protects DS aggressive flows from the opportunistic transmission beyond the assured bandwidth by signalling via early drop action. Best Effort traffic overloads network when $\rho_{DS} < 0.3$: as a consequence, a sustained early drop actions are performed by RED gateway; as long as $\rho_{BE} < 0.3$ diminishes, green packet begin to profit of early congestion detection mechanism, and influences the averaged red queue size computation: with the deployed MRED parameter set, red packets are more likely to be dropped due to buffer overflow, and red queue is working mainly as a drop tail. Finally, when DS traffic exceeds its profile, both BE and DS packets are more likely incur of early drop: the reduced BE traffic creation rate, coupled with a certain degree of early drop effectiveness in reducing DS flow's aggressiveness, diminish BE packet probability of finding full buffers: this further participates to lower BE drops amount at high ρ_{DS}.

7.5.2 Congestion Window and Fast Recovery

The maximum congestion window gain of BE flows when DS cloud load exceeds $\rho_{DS,th}$ is remarkable though not impressive, as depicted in Fig. 7.43: with respect to BE's lower value the average gain is kept below 8%; conversely, for DS long flows, the maximum cwnd achieved at $\rho_{DS} = 0.6$ is ~ 1.6 times lower than what it was at $\rho_{DS} = 0.1$. This reduced aggressiveness of DS flows at high ρ_{DS} let BE flows improve its performance: with respect to a similar number of BE packets sent within a window, long DS flows will be less likely to monopolize the queue's occupancy, preventing BE flows starvation and further reducing both their late drop and total drop percentages.

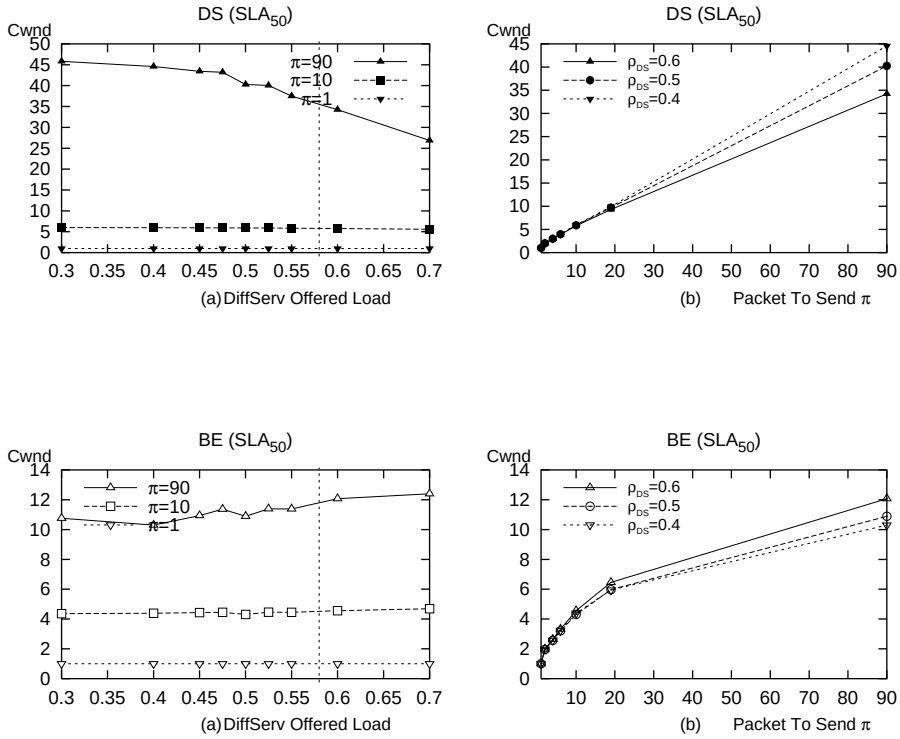


Figure 7.43: DS and BE Congestion Window: Varying ρ_{DS} (HTTP, $\rho = 0.85$)

Fast recovery plots are reported in Fig. 7.44; DS performance degradation is less noticeable than BE performance improvement: in the latter case, both intermediate and long flows doubles their performances at $\rho_{DS} = 0.9$ with respect to those achieved at $\rho_{DS} = 0.1$. In the intermediate DS flow case ($\pi = 10$), the small amount of dropped packets does not provide a confident estimate of Fast Retransmit occurrency. However, it must be said that these FR fractioning results of the drop percentages differs from those obtained when the overall HTTP load reaches $\rho = 0.975$: in the latter case, there is no BE performance improvement due to DS oversubscription, and both BE and DS performances are monotonicly degraded for $\rho_{DS} \geq \rho_{DS,th} \simeq 0.513$, eventually reaching worse values.

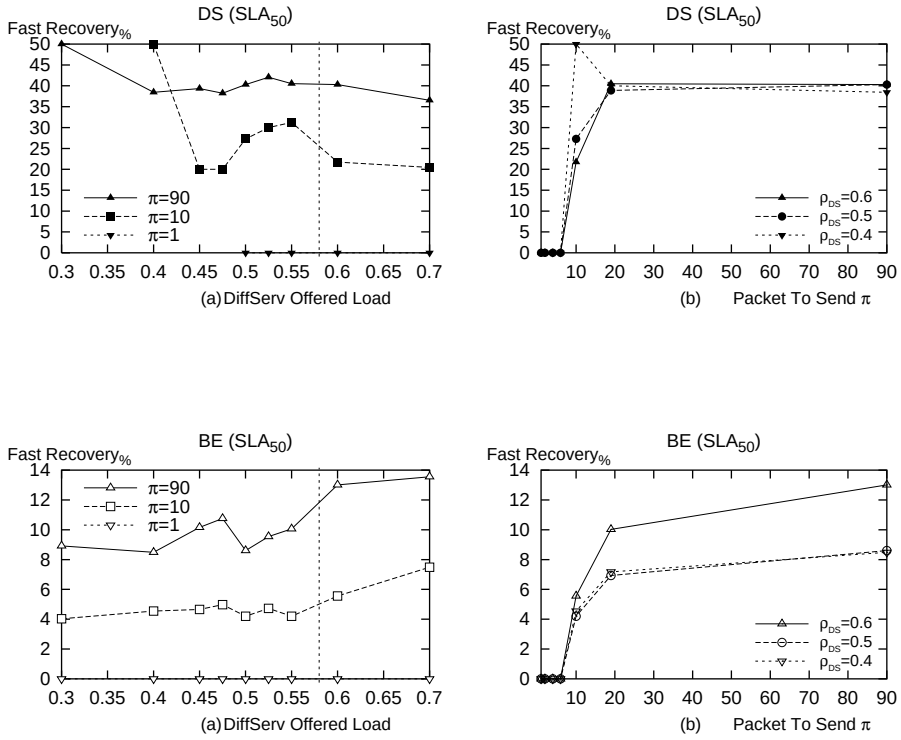


Figure 7.44: DS and BE Fast Recovery Percentage: Varying ρ_{DS} (HTTP, $\rho = 0.85$)

7.5.3 Completion time

HTTP flow's response time is an important observable user-centric metric, whose quantitative dependence on the factors earlier described is not straightforward; anyway, a qualitative cause-effect explanation of completion time behavior with respect to ρ_{DS} can be quite easily given, especially in the DS case. Fig. 7.45 depicts the flow's response time when $\rho = 0.85$ whereas the $\rho = 0.975$ case is shown in Fig. 7.45: in both cases, DS time performances monotonicly increase as long as ρ_{DS} increases and their main difference is that in the latter case performance degradation is much stronger; reasons to this behavior can be found in different forwarding assurances of in-profile and out-of-profile traffic, topics which have already been deeply developed in earlier simulation analysis.

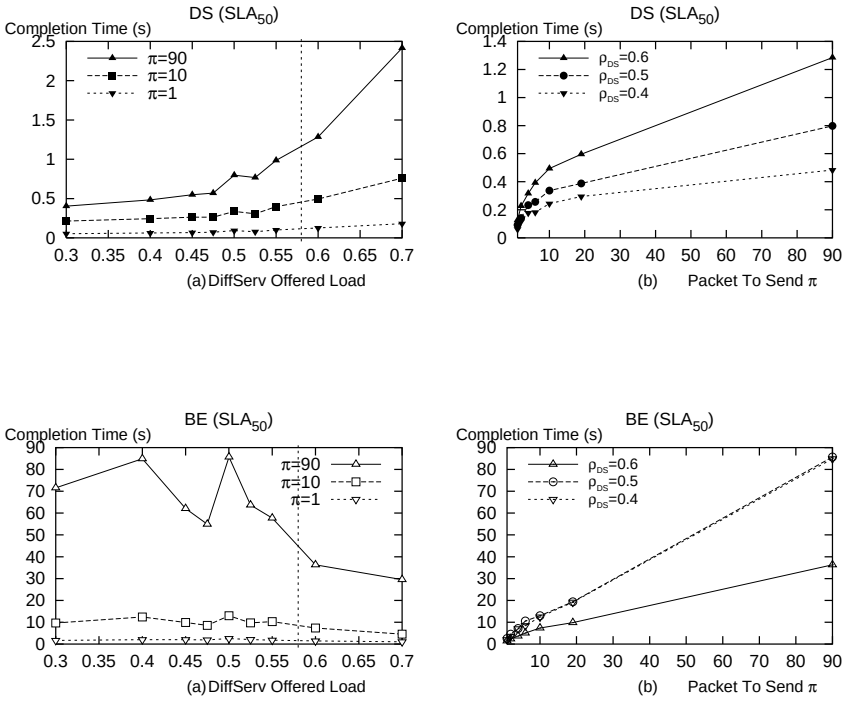


Figure 7.45: DS and BE Completion Time: Varying ρ_{DS} (HTTP, $\rho = 0.85$)

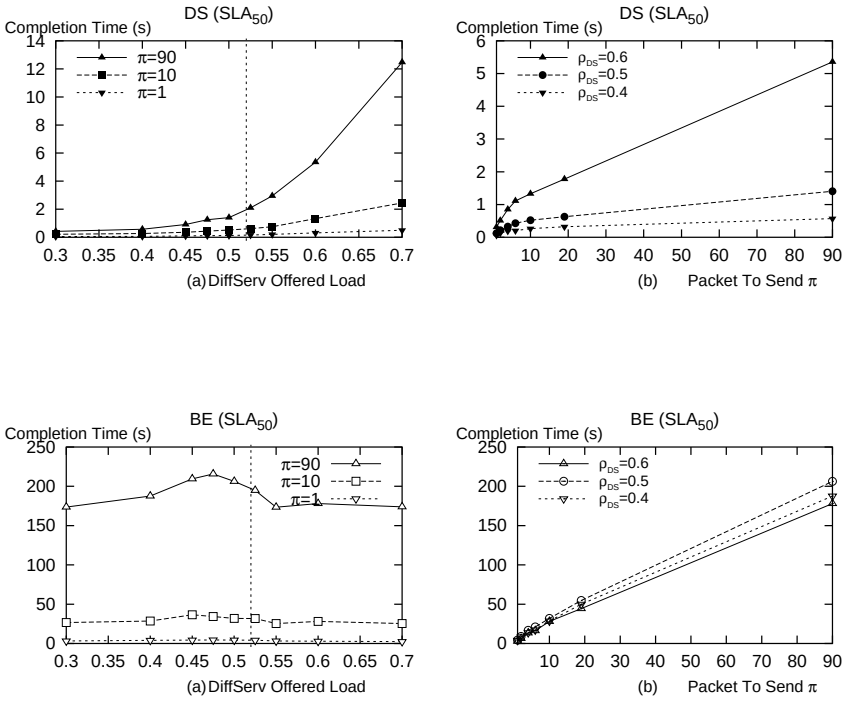


Figure 7.46: DS and BE Completion Time: Varying ρ_{DS} (HTTP, $\rho = 0.975$)

BE traffic performances behavior when $\rho = 0.85$ is not monotonic on ρ_{DS}; rather, in order to gives reasons to the observed results, we may define two ρ_{DS} ranges: a *low* load range $\rho_{DS} < \rho_{DS,th}$ and an *high* one $\rho_{DS} > \rho_{DS,th}$.

At very low DS loads network congestion is mainly due to BE traffic and then, as ρ_{DS} grows, DS contribution to congestion grows as well. It is thus reasonable to hypothesize that BE performances at very low loads are determined, as approximation of the $\rho_{DS} = 0$ case, mainly from the specific MRED parameter set: this has been already proved, in previous simulation analysis, to yield consistently poorer performance with respect to DS traffic one. Then, the introduction of in-profile DS traffic will surely penalize further BE marked packets, in order to ensure DS packets a better level of protection: the augmented BE drop percentage and the corresponding early drop decrease cause completion time to grow in that load region. However, as DS traffic volume approaches its reserved bandwidth, DS packets will begin to be red marked by the TSWTCM: this will happen during short congestion periods caused by traffic burstiness; BE traffic completion time gradually starts to decrease, since BE packets will start to compete with a growing amount of equal drop precedence DS red packets: this is reflected in DS performances by increasing packet drop and early drop other than DS flow's congestion window decrease.

At high DS loads, a smaller BE traffic volume takes better advantages of the assigned SLA_{50}: i.e., longest flow's response time at $\rho_{DS} = 0.7$ is ~ 30, whereas it was more than two (~ 2.33) times greater at $\rho_{DS} = 0.5$ (at $\rho_{DS} = 0.3$). The percentage of DS red marked traffic grows further, and a small amount of BE traffic benefit of competition with more aggressive out-of-profile flows; among the factors contributing to the consistent time performance improvement, we can identify the BE intermediate and long flows congestion window growth (whereas DS one sensibly decreases), entailing a better profit of fast recovery/fast retransmit mechanism whether possible, without neglecting the importance of a reduced packet drop amount (opposite the corresponding DS drop and early drop growth).

If overall network load grows to $\rho = 0.975$, DS and BE completion time performances are similar to those described, and similar are the cause-effect reasons behind the different time behaviors; however, a major difference must be evidenced with respect to BE service: the increased congestion level appears to offset any of the improvements earlier noticed when $\rho_{DS} > \rho_{DS,th}$, bringing BE traffic to starvation; as a consequence, when observing Fig. 7.46, we may say that BE response time get equally bad performances when $\rho_{DS} \in (0.1, 0.4) \cup (0.55, 0.9)$, and even worse performance when $\rho_{DS} \in (0.4, 0.55)$

7.6 Conclusions

The short lived TCP flows simulation performed are based on a really simple network scenario, but the aspects that have been investigated (i.e., the impact of DiffServ service on a bursty traffic model and the mutual effects of bursty and infinite sources) are of key importance in the current Internet.

In this section, we examine the short lived flows simulations from the ISP's point of view, giving some HTTP flows monitoring guidelines (Sec. 7.6.1.1). Moreover, in order to be able to test the AR service efficiency in the case of HTTP traffic, we propose a service performance evaluation metric, trying to assess a quantitative relationship between user and system metrics (Sec. 7.6.1.2). Finally, possible directions of the needed DiffServ research, as well as needed enhancements to the implemented HTTP model are outlined in Sec. 7.6.2.

7.6.1 SLA Considerations

When dealing with short-lived TCP flows, the completion time is the primary metric, from the user's point of view, to test the effectiveness of a given DS service. Nevertheless completion time is not likely to be included, neither directly nor indirectly, among the SLS parameters of the correspondent SLA; rather, the SLA will involve quantitative bounds to system metrics, such as throughput, goodput and loss rate. Specifically, in the Assured Rate context, it is likely that

the service will be entirely specified in terms of the committed information rate (CIR), without any further constraint, e.g., on packet drop probability. Although it could be possible, for a SLA contract, to enforce a peak rate (PIR) separately from the committed one, however we will disregard this case in the following, assuming thus PIR=CIR.

The evaluation of system and user performances in the HTTP case may be done in rather different ways; moreover, monitoring of HTTP traffic performances is much more complicated than in long-lived TCP flows case, especially in case of run-time measurements performed by an ISP. It is therefore important to pinpoint a quantitative relationship between the user-centric metric (i.e. completion time) and the service-level metric (i.e. target rate). The performance model should permit to judge the performance perceived by the user from a system point of view, and also to compare the fairness of the service delivered under different flows parameters.

7.6.1.1 Performance Monitoring

Since the AR service level is entirely specified in terms of a rate measure, it would be interesting to associate the HTTP traffic aggregate performance with a bandwidth measure: in this section, we discuss the design difficulties of such monitoring approach.

The completion time (CT) of an HTTP flow can be associated with a bandwidth measure by evaluating the ratio of the flow length over the CT time interval; similarly, throughput can be sampled at any time instant t by computing the ratio of the data sent by the flow up to t. If an ISP is willing to sample the average throughput of its AR flows, then a per-flow state tracking is needed; moreover, evaluating the *aggregate* throughput (which could be directly compared to the aggregate CIR) via per-flow measurements is not straightforward.

Indeed, for continuous sources such as infinite FTP transfers, user throughput and system link utilization have a simpler dependence than in the short lived flow case, since performance sampling can be done in a common time interval (T_i, T_f) with $\Delta T = T_f - T_i$; indicating by D_i the amount of bytes sent by the i-th source during ΔT, we may calculate the aggregate throughput Thr_Σ as $Thr_\Sigma = \Sigma_i Thr_i = \Sigma_i (D_i/\Delta T_i) = (\Sigma_i D_i)/\Delta T$. In the HTTP traffic case, the former approach cannot be used since each flow transfer has a different time extension, and moreover the number of flows active during (T_i, T_f) may vary widely.

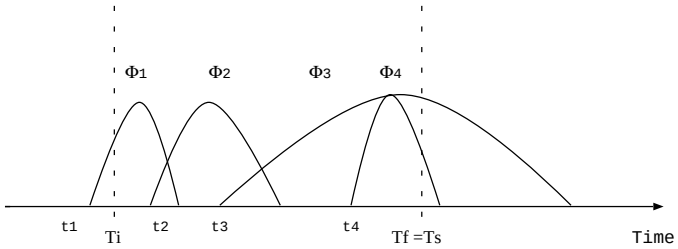


Figure 7.47: Aggregate Throughput Evaluation for Short-Lived Flows

In order to explain a possible approach to the aggregate throughput computation, it would be helpful to observe the scheme presented in Fig. 7.47, where each flow is depicted following the "bell" representation earlier introduced in Sec. 6.4: the i-th bell "base" measures the completion time of flow Φ_i but its "height" does not represent any meaningful parameter. When the averaging time interval is (T_i, T_f), each Φ_i differently contributes to the aggregate throughput Thr_Σ (e.g., Φ_2 contributes entirely to the sum, whereas others flows' contribution is only partial); moreover, monitoring cannot be focused only on connection starting and ending within (T_i, T_f), otherwise a significant amount of traffic would be disregarded (i.e., only Φ_2 performance would be considered). Another faulty approach would take into account only the flows starting after T_i (i.e., disregarding Φ_1), but this would lead to a possibly significant under-estimation of the real network situation. In order to avoid this and to be able to collect all the data relative to (T_i, T_f), the flows' state

variables sampling must necessarily begin *before* T_i; however, the warm-up sampling time $T_w < T_i$ depends on several factor, such as network congestion and active flow length, and can be hardly quantified. Nevertheless, it is important for the state tables to be up to date at the beginning of the sampling interval; every sent packet of each active flows must be tracked and associated to the corresponding system-time value: this ensures the ability to evaluate the throughput of flows whose transmission did not start and/or complete during (T_i, T_f).

Furthermore, due to HTTP traffic dynamics, performance sampling could not be spotty in order to obtain significant monitoring results: that is, once pinpointed a macroscopic interval $I_s = (T_i, T_f)$, then the sampling procedure should be repeated a number of times over different $(T_{i,1}, T_{f,1}) \subset I_s$ intervals, delaying each run, e.g., in a Montecarlo fashion. Several different approaches are possible to fraction I_s (e.g., intervals may have fixed or different time window sizes, may or may not overlap, ...) and different approaches can be used to provide a larger confidence with the results; also, the choice of the I_s extension with respect to network congestion level is a key of importance.

7.6.1.2 Service Evaluation Metrics

The completion time is the most meaningful and intuitive measure of HTTP flows performance; nevertheless, this parameter can be hard to relate to the subscribed service level. Furthermore, the analysis of the obtained HTTP results gave us several indications on the fact that shorter flows performances are more penalized than longer ones, even in DiffServ networks; however, a mere direct comparison of flow completion times achieved under different SLAs lacks any explicit reference to the established SLA itself.

The model should be able to to quantitatively express the completion time as a normalized measure of service efficiency, as a function of both the flow-length and the DS aggregate bandwidth. In other words, our goal is to find a possibly simple model allowing an ISP –as well as a network simulator– to be able to provide quantitative responses to a few key questions. Specifically, when a company subscribes to an Assured Rate service with an ISP whose configuration is SLA_x, knowing that a flow of length λ bytes is serviced in CT seconds, how well did that flow profit from the service? Moreover, what timely performance should expect a customer of SLA_y, under the same network condition, for a λ long flow? Finally, what is the impact of a specific network factor (e.g., among those studied for long lived flow) on SLA_x HTTP performances?

Let us indicate with $B_{used,i}$ the bandwidth used by the i-th flow to transmit λ_i data during the CT_i interval, thus $B_{used,i} = \lambda_i / CT_i$; then, we should define a theoretical $B_{th,i}$ bandwidth parameter to be used as a term of comparison with $B_{used,i}$. We may assume that the assured rate is identically distributed among each active flow of the aggregate: this represents an unrealistic hypothesis, but should be valid on long time scales; $B_{th,i}$ can then be computed as the ratio of the reserved bandwidth B_{SLA}, disregarding the possible excess, over the average number $\overline{n}_{\Phi,i}$ of flows active during CT_i. The efficiency η_i of the service delivered to the i-th flow can then be expressed as the ratio of $B_{used,i}$ over $B_{th,i}$: the parameter η_i therefore represents the bandwidth gain obtained by the i-th flow with respect to the rate that the AR service could ideally guarantee. The analytical expressions of the former defined parameters are:

$$\begin{cases} B_{used,i} &= \lambda_i / CT_i \\ B_{th,i} &= B_{SLA} / \overline{n}_{\Phi,i} \\ \eta_i &= B_{used,i} / B_{th,i} \end{cases}$$

If an ISP would implemented such a scheme to test its delivering efficiency as well as the achieved fairness among HTTP flows, then some simplifications would be needed. Actually, the average number of active flows $\overline{n}_{\Phi,i}$ might be calculated without sensible overhead: in fact, the number of active HTTP connections may widely fluctuate during CT_i, and intensive state tracking is needed to its estimation; rather, considering the ratio of the reserved bandwidth over the

maximum number $max(n_\Phi, i)$ of active connections during CT_i yield to the worst-case bandwidth measure B_{wc}. Since B_{wc} is a lower bound for B_{th}, the evaluation of the service efficiency via B_{wc} leads to an upper bound for η_i, which is less representive of the real situation (and less suited than a η lower bound). However, we may offset this over-estimation by considering each flow's goodput rather than its throughput, expressed here as the packets to send number π scaled by the constant packet size c; such a performance metric, to which we will refer as the *approximated* metric (with the purpose to discriminate it from the *ideal* metric from which it derives), is defined as:

$$\begin{cases} \tilde{B}_{used,i} &= c \cdot \pi_i / CT_i & \leq B_{used,i} \\ B_{wc} &= B_{SLA} / max(n_\Phi) & \leq B_{th} \\ \tilde{\eta_i} &= \tilde{B}_{used,i} / B_{wc} & \end{cases}$$

Although the relationship between $\overline{n}_{\Phi,i}$ and $max(n_{\Phi,i})$ is not simple to express, nevertheless it is unquestionably related to the congestion state the network experiences during CT_i. Then it should be considered that, during congestion periods, the actually sent data can be much greater than the data to send: this stems from possibly consistent loss rates, as we noticed earlier via simulation; conversely, $\overline{n}_{\Phi,i} \simeq max(n_{\Phi,i})$ may represent a valid approximation when the network congestion is low and constant, and may hold for a short time interval.

The defined models allow to test the efficiency of the service delivered to a specific flow, but can be easily extended to be representive of the overall service behavior; moreover, tracking the completion time of each flow as well as its length within the (T_i, T_f) interval of Fig. 7.47 may be simpler than the evaluation of the aggregate throughput in the same time interval. A possible service evaluation scheme may involve two distinct measures, with different sampling assumptions: the overall and the worst case service efficiency; in fact, due to relative ease of completion time sampling, this parameter can be monitored also during heavily congested network periods, reflecting thus the worst-case AR efficiency. It should be noted however that, in such case, the router's monitoring process should only perform the collection of the completion time and flow-length (CT, λ) pairs: any service efficiency computation should be postponed to avoid the introduction of unnecessary overhead. The averaged service evaluation may be more difficult to compute via spotty sampling and, due to HTTP flows dynamics, requires that monitoring is performed over larger time scales; moreover, the first metric would be better suited in this case.

Another extremely important point is that the η and $\tilde{\eta}$ metrics can be used as a measure of fairness among different flow sets: e.g., for a specific range of λ values, or for any RTT range, etc. Unfairness evaluation of a specific service SLA_x can be computed either per-flow $\mathcal{U}_{x,i}$ or aggregate $\mathcal{U}_x$; their analitic definitions are reported only for the ideal η metric but can be easily extended for the approximated $\tilde{\eta}$ metric as well:

$$\begin{cases} \mathcal{U}_{x,i} &= \eta_{x,i} / \eta_{x,\mu} \\ \mathcal{U}_x &= \eta_{x,\sigma} / \eta_{x,\mu} \end{cases}$$

To highlight the difference of the two models, we propose the evaluation, extendend over the whole simulation duration) of the SLA_{50} service for different AR traffic loads ρ_{DS}, when the overall network load is $\rho = 0.85$ (see Sec. 7.5). Fig. 7.48 shows the great differences among the ideal and the approximated model constitutive parameters: in (a) the maximum $max(n_\Phi)$ and average $\overline{n}_\Phi$ number of active flows are plotted as a function of ρ_{DS}, and in (b) the ratio of goodput over throughput is plotted for shortest 1-π, longest 90-π flows as well as its average over all flow lengths. From the latter case, we gather furthermore another evident confirmation to the TCP per-π unfairness: longest flow's goodput over throughput ratio reaches its minimum when the HTTP traffic is created at a rate equal to the reserved bandwidth but then stays substantially constant, whereas shorter flows' performance are monotonically degraded as long as ρ_{DS} grows.

It is important to notice that both ideal η and approximated $\tilde{\eta}$ models –while resulting in really different efficiency estimates– yield the same measure of both per-flow $\mathcal{U}_{x,i}$ and aggregate

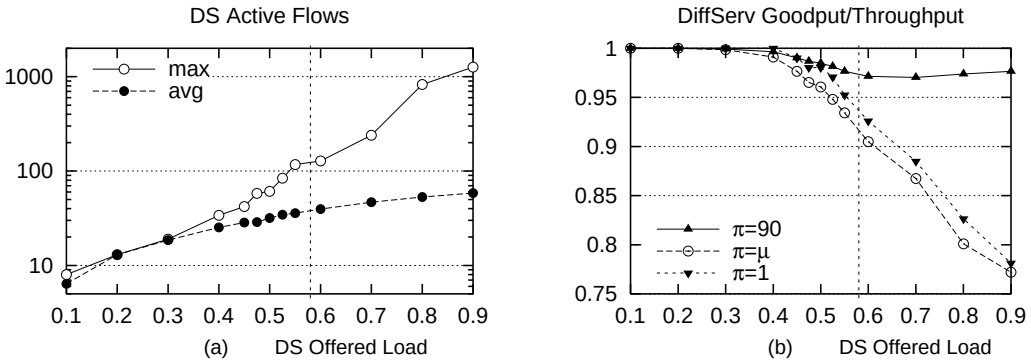


Figure 7.48: Parameters Differences Among Service Evaluation Models

$\mathcal{U}_x$ service unfairness. These observations are clearly shown in Fig. 7.49, which depicts on the y-axis both the service efficiency estimation (a) and service unfairness (b) as a function of the DS traffic load. As expected, the service efficiency decrease as long as the traffic volume grows – for both the ideal and approximated model; however, since the latter model depends on the maximum number of flows active at once, its slope is less smooth with respect to the ideal model, which uses an *averaged* value. Ideal service efficiency is compromised, thus lower than 1, above $\rho_{DS} = 0.5$: it should be considered (as earlier analyzed in Sec. 7.5), that the drop amount seldom increases at $\rho_{DS} = 0.5$ for any flow-length; moreover, the average completion time at $\rho_{DS} = 0.5$ for both BE and DS flows has a remarkable relative increase (DS flow completion time values are kept below 1 s even for longest 90-π flows for any ρ_{DS}.

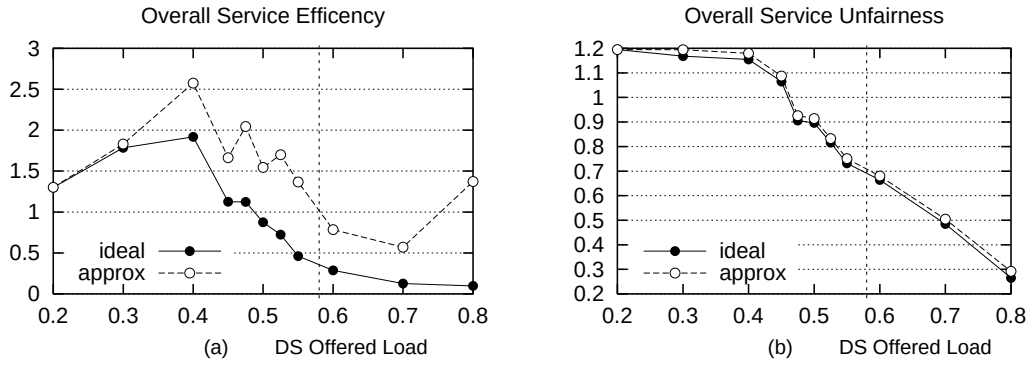


Figure 7.49: Comparison of Service Efficiency and Unfairness Models

However, it should be noticed that, in order to point out the different outcomes due to each specific metric's assumptions, metric comparison is performed in an extreme case: actually, the service's overall efficiency is calculated over a very large time interval, i.e. at least 600 seconds. Therefore, due to ρ_{DS} and ρ being constant within each simulation run, the average number of active flows represents a significant measure of the average congestion level, which is mainly determined by the model's parameter ρ_{DS} and ρ. Conversely, the maximum number of flows active at once is only representative of a shorter simulation period corresponding to the highest congestion level achieved by the network during that run: this parameter is thus mostly determined by TCP

dynamics reaction to simulation parameters. Furthermore, the maximum number of active flows can actually vary in a quite wide range, especially at high ρ_{DS}, even for a given (ρ_{DS}, ρ) parameter set. Eventually, it should be noticed that $\tilde{\eta}$ model would best represent the negative efficiency peak at high network congestion; however, in this case the completion time statistics should be sampled only over the correspondent narrower time period.

Tab. 7.11 reports the aggregate ideal service efficiency η_μ and its approximation $\tilde{\eta}_\mu$, as well as the service unfairness parameters. Per-flow length unfairness values are reported for flows with different π under different service loads; results are aggregate in the sense that they refer to the service experienced, on average, by flows willing to send π packets – where the averaging interval covers the whole simulation duration: justification of this approach comes from the fact that network load, as well as DS traffic load, is constant for each simulation run.

ρ_{DS}	0.1	0.3	0.5	0.6	0.7
η_μ	0.84	1.78	1.12	0.43	0.17
$\tilde{\eta}_\mu$	1.83	2.57	1.55	0.79	0.57
π	$\tilde{\eta}(\pi, \rho_{DS})/\tilde{\eta}_\mu(\pi, \rho_{DS})$				
1	0.29	0.33	0.39	0.52	0.76
2	0.30	0.32	0.45	0.6	0.67
4	0.31	0.33	0.43	0.59	0.70
6	0.60	0.62	0.66	0.92	1.07
10	0.77	0.80	1.02	1.06	1.12
19	1.16	1.25	1.34	1.43	1.28
90	3.58	3.34	2.71	1.88	1.40
$\tilde{\eta}_\sigma/\tilde{\eta}_\mu$	1.09	1.00	0.77	0.46	0.27

Table 7.11: Aggregate Service Utilization and Per-π Unfairness

Results for $\tilde{\eta}_\mu$ indicate that service is efficient only if the aggregate traffic does not exceed the reserved bandwidth: although DS flows completion time is far below the BE flows one, nevertheless the consistent increased amount of DS red marked packets, and therefore of DS packet drops, negatively influences DS flows ability of profiting of the reserved bandwidth.

Moreover, observing $\tilde{\eta}_\sigma/\tilde{\eta}_\mu$ behavior as ρ_{DS}, it can be gathered a trade-off between service fairness and efficency: fairness among flows increases as long as the overall service performance is degraded. From $\tilde{\eta}(\pi, \rho_{DS})/\tilde{\eta}_\mu(\rho_{DS})$ values inspection we can gather that all flows –longest flows being the exception– increase their performance, whereas longest flows over-performing is only lessened as ρ_{DS} cross the threshold. This is better shown in Fig. 7.50, which plots on the z-axis the per-flow length efficiency $\tilde{\eta}(\pi, \rho_{DS})$, normalized over the average service efficiency $\tilde{\eta}_\mu(\rho_{DS})$, as a function of both the DiffServ HTTP load ρ_{DS} and the flow length π expressed in packets. Evidence is that flows sending up to 4 packets obtains substantially identical performances: in order to give explanation to this phenomenon, we should consider specific TCP dynamics.

It is known that TCP's congestion control is only marginally driven by the rate at which data leave the source; rather, it depends on the segment sending rate and the correspondent ACKs receiving rate, both measured at the source. Moreover, TCP infers that a segment is lost either when the retransmission timer expires (RTO) or by the arrival of three duplicated ACKs triggering the FR algorithm: the former event is the least desirable, being usually much larger than the connection's RTT. However, FR mechanism cannot take over RTO unless cwnd is at least four time larger than the sender maximum segment size (SMSS) and the flow is long enough to allow the transmission of at least four segments, regardless of the actual data size in bytes. TCP's cwnd constraint is the main cause of the unfairness toward such short-lived flows, i.e. Φ_1,Φ_2 and Φ_4: the per-flow length fairness increase is a consequence of the increased probability for red-marked

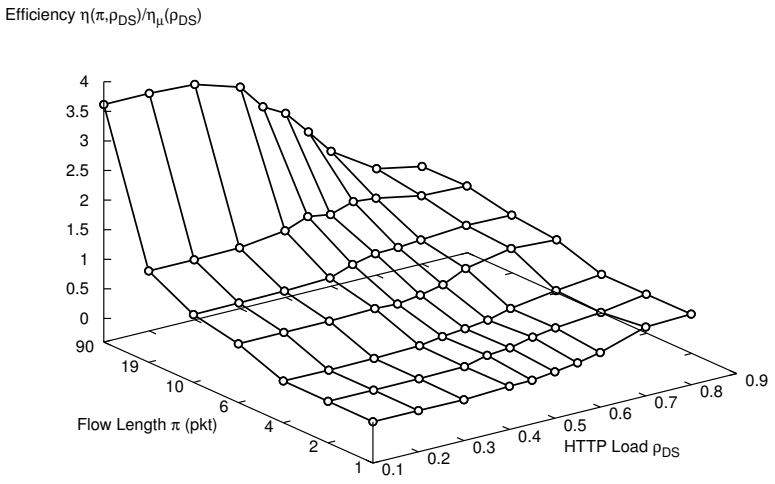


Figure 7.50: Per Flow Length Normalized Service Efficiency

packets of longest flows to incur in RTO.

Finally it should be noted that, rather than resulting in network decongestion, repeated forcing a short lived flow in RTO results mainly in excessive flow penalty, whereas transmission would otherwise end after sending a few more segments. Since today's Internet traffic is heavily represented by short-lived connections, the excessive TCP unfairness among flows with different lengths cannot be disregarded; however, solutions to this problem are not straightforward, since they may lead to consider specific TCP congestion control design: a number of works address the RTO penalty [RFC2415][TCP-SF], but intesive simulation and experimental testbed results are needed to test whether these approaches are no with unwanted draw-backs.

7.6.2 Future Directions

This study of DiffServ QoS architecture investigated the performance of TCP short and long lived flows crossing a domain supporting the Assured Rate service and the common Best Effort service. However, the analysis developed is far from being complete: therefore, we pinpoint further topics and directions for future research.

Unfairness Factors

From the long lived TCP simulation analysis, we found that DiffServ flows were not insensible to a set of network factors affecting TCP behavior: we identified these factors in connection's round trip time, packet size used by the application, number of micro flows within an aggregate, size of the target rate, number of different DS aggregates crossing the domain and overall DS reservation rate.

Moreover, interactions of different protocols at the transmission layer, the specific settings of deployed MRED active queue management gateway, the marker object and its settings yielded rather different performance. Finally, although we did not investigate it by simulation, different TCP stacks (i.e. Reno, NewReno, SACK, Vegas, ...) surely lead to different simulation results.

Even though the network scenario used only considered FTP and HTTP applications running over the Reno version of the TCP protocol, it can be thought that HTTP traffic will be

affected from the former factors *at least* as the FTP infinite sources. Moreover, HTTP traffic simulations raised other fairness concerns: all the simulation data indicated that DiffServ architecture advantages relatively long HTTP flows performance against shorter flows. It would be therefore important to couple the study of the former common network factors with the "intrinsic" HTTP unfairness: e.g., observe per-flow length performance of different aggregates having different connection RTT.

HTTP *ns* Engine

There is a main lack of the *ns* HTTP engine developed: the absence of fine FTP traffic tuning parameters. First of all, a main FTP load threshold $\rho_{FTP,}$ is clearly needed; whether FTP sources are either infinitely sending or absent, their only effect is to push the network load toward the line rate: then, depending on the HTTP load ρ, a substantially constant traffic volume is differently fractioned between HTTP and FTP traffic types. Moreover, it was noticed in some simulations that the FTP presence may have negative effects on the simulation run between (t_{stop}, t_{end}): the FTP tended to take over short HTTP flows, especially in the BE clouds; this effect, although reasonable –since FTP flows have the possibility to grow their cwnd while this is denied to short lived flows– is nevertheless not desirable.

Another interesting feature involves the differentiation of controlled FTP sources number $N_{FTP,i}$ among each cloud i, differently distributing $\rho_{FTP,i}$ among these sources; this would allow a more realistic traffic case, where both the number of long lived sources flows within each cloud varies and where furthermore source are differently sized within each traffic aggregate.

Finally, when the i-th cloud's HTTP traffic ended all the activated connections while other clouds are still sending data, FTP sources of cloud i may be resized to produce the same traffic volume as though HTTP were still active: this implies that, at t_{end}, $\rho_{FTP,i}$ is incremented by $\rho_{HTTP,i}$; however, it should be tested whether this approach does not lead to the already noticed drawback due to long lived flows FTP aggressiveness.

Apart from specific FTP traffic control, the engine should be enhanced to permit straightforward differentiation of common network factors, allowing to set different values among clouds. Actually, the size of the aggregate target is driven by the SLA setting, whereas the number of HTTP flows within the aggregate is determined by both the overall load ρ and the per-cloud load $\rho_{HTTP,i}$. Therefore, the remaining factor are the packet size and the connection propagation delay.

The former should not be statically set as for the long lived flows simulations: rather, the possibility of sending the true amount of data should be determined by the model instead of its discretization due to the packet size being constant; this could be done allowing only the last (or the first for really short flows) packet of a flow to be smaller than the otherwise constant packet size. However, this poses some increased coding difficulty and additional per-flow state tracking –but only a few overhead– in order to collect the desired statistics. Furthermore, it should be investigated whether the *ns* objects' and RED implementation's handling of variable packet size is configurable in a straightforward manner (e.g., when the flow size is heuristically determined the mean packet size directly reflects the average flow length).

The RTT can be either statically assigned to each flow of the i-th cloud (RTT_i), or randomly chosen at each flow activation among a coarse set $(RTT_{i1}, RTT_{i2}, \ldots, RTT_{ij})$ different for each cloud, or finally both the approaches can be implemented together.

Finally, although the engine self-implements several focused statistic collection methods and a specific per-cloud, per-flow trace mechanism (alternative to the exhaustive standard *ns* per-packet event tracing), nevertheless the large number of scenarios and flows tended to

produce an huge amount of simulation data. As a *caveat programmer*, it should be said that the desired statistics computation from the simulation data collected is not likely to be a light-hearted task: the possibility of using an SQL-based database rather than intensive Tcl/Perl/Shell scripting must not be disregarded. Furthermore, a number of libraries exists that provide methods for the ease of the database creation, query and manipulation.

Performance Evaluation of Alternate Solutions

The proposed performance models should be used, tested and further developed in order to find a flexible service evaluation scheme, useful to measure both the service unfairness and efficiency. However, the need is felt of a first grade analytic tool allowing to test system reactions to different marker objects and settings, policy schemes and the such, allowing further to choose which approach, among different possibilities, is the most worthy to be further investigated.

For example, unfairness due to short flows penalty might be corrected addressing short flows' requirements in the design of TCP congestion control algorithm. Rather, fairness may be increased by deliberately dropping ACK packets in routers: a probabilistic drop action would be taken basing the ACK selection on a stateless decision upon the occupancy of the queue by multiple ACKs of the same HTTP flow. Longer flows will be more likely to be penalized in the sense that they will possibly be pushed in fast recovery, therefore reducing their aggressiveness and allowing short flows to complete their transfers. Such an approach is similar to RED in the sense that it takes "early" probabilistic ACK drop decisions; indeed its goal concerns fairness among differently size flows rather than queue control. From a performance point of view, this mechanism is likely to yield efficiency gain to short flows, increasing further the overall service fairness; on the other hand, it must be considered the possibility that it leads to an overall efficiency loss: a well understood service performance metric is clearly needed in order to be able to properly configure the stateless selective ACK dropping mechanism in routers.

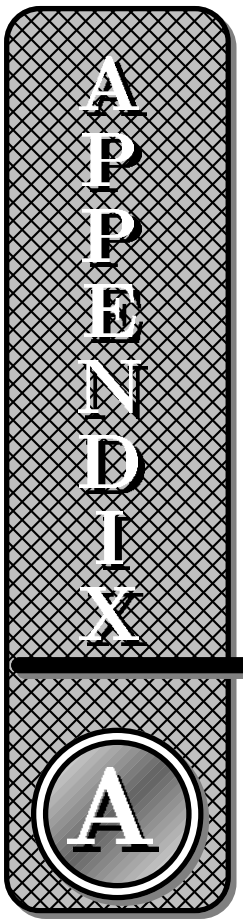
Interesting Scenarios

The short lived flow simulations tested the performances of different AR service levels when the rest of the traffic is forwarded as Best Effort.

A first step may involve, without the need of varying any other scenario parameters, the investigation of a DS domain offering a few different SLAs at the same time to different customers using HTTP and FTP applications: e.g., SLA_{25} and SLA_{50} AR services may be both implemented in the domain and the rest of the traffic should be forwarded as best effort. It should then be observed how the domain can guarantee different contemporary levels of assurance at one time: per-cloud performances should be relatively examined, service pre-emption investigated, services efficiency and fairness compared. Additionally, the DS domain might be enhanced to support an EF PHB-based service beyond the AR PDB, e.g. the Virtual Wire PDB, whose configuration and admission rule are simpler than in the AF case, to investigate the mutual influence of several Differentiated Services.

Then, with the earlier introduced extensions to the engine, a quantitative study of the RTT influence on HTTP flows would be suitable in order to have full parameter control over specific scenarios; moreover, traffic flows may be bidirectional, thus creating congestion on the ACKs backward path, and furthermore ACKs could be allowed to be lost, unlike in our scenario. On the other hand, it should be considered that an experimental testbed implementation allow a more realistic evaluation, featuring actual traffic patterns, so that contemporary deployment of these two investigation approaches should not be disregarded.

Finally, an important scenario evolution step would involve modifications of the network topology: this should be done, firstly, to test whether the used model scales; however, before the setup of a more complex topology case, it would be interesting to invest in a deep analysis of the admission rule for, e.g., a AR one-to-any PDB, in order to test the ability of a properly configured service to fulfill the SLA contract.



ns Code

Long Lived FTP Flows

A.1 Introduction

This report details the implementation of a set of OTcl classes used to build DiffServ-oriented simulations scripts for the *ns* Network Simulator. These classes are based on the DiffServ architecture implementation of Nortel Network Open IP Group.

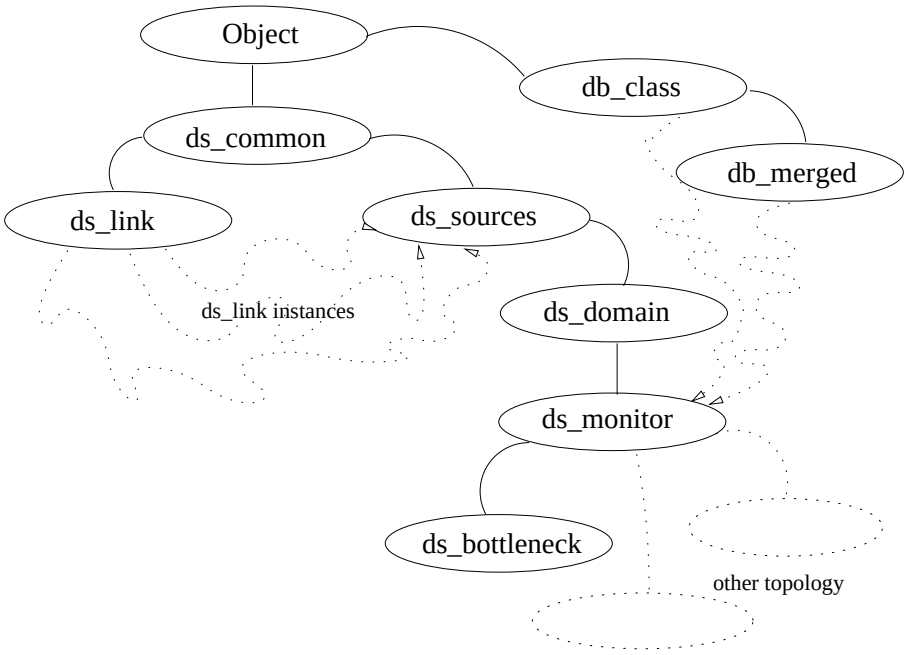


Figure A.1: OTcl Classes Set

The synopsis introduces the elements of the set, that is every *ds_XXX* classes, and illustrates their hierarchy.

Inspecting the scheme with a **top-down** approach, `ds_common` and `db_class` are at the same level in the object tree hierarchy:

- `db_class` is an object that handles bidimensional data tables with a special regard to columnar data, featuring facilities to obtain basic statistic analisys, using some of the Tcl procs defined in `list&array.tcl` script

- `ds_common` is just a container class defining some basic values and implementing a debug method that will be used by any of the DiffServ oriented class set.

One level down,

- `db_merged` enhances the databases capability, while

- `ds_link` handles a unidirectional flow aggregate, originating form a source node, entering the DS domain at a particular edge and finally reaching a destination node

- `ds_sources` handles a vector of flow aggregates, supposing that they have the same ingress edge and destination nodes.

Then `ds_domain` defines a formal structure in which domain path, nodes and queues should be ranked in order to take advantage of `ds_monitor`'s flow monitoring features; the latter class implements an alternative to ns trace files, calculating the data of interest during the simulation –rather than parsing ns standard trace files– and using the database objects introduced here.

A class derived from `ds_monitor` should take care of setting up DiffServ domain internals (that is, nodes, path, policy mechanism, etc.) coherently with the underlying structure: `ds_bottleneck` is such an example.

From a **bottom-up** point of view, `ds_bottleneck` defines a rather simple network topology, consisting of an ingress DS edge (to which source nodes will be attached) connected by a DS core router to a DS egress edge (considering unidirectional flows) and finally to a destination node. The edge and destination node have been created by a `ds_domain` class object in order to give to `ds_monitor` a conventional structure to base the monitoring tasks on; `ds_monitor` uses `db_merged` and `db_class` instances to handle simulation data, retrieved from `ds_domain`'s queues and `ds_link`'s objects. A set of `ds_link` instances is handled by a `ds_sources` object, with the assumption that every link produce unidirectional flows having the same (source,dest) nodes pair. A `ds_link` object supports various micro-flows within an aggregate, produced by heterogeneous (FTP over TCP, CBR over UDP) applications, with the ability to report per-flow or per-aggregate informations.

The rest of the document is organized as follows:

- Sec. A.2 illustrates the template structure used to describe each ds_XXX class
- from Sec. A.4 to Sec. A.8 each class is described in detail; this has the purpose to be a valid reference guide, ordered in a logical order
- Sec. A.9 illustrates a full OTcl example, and can be seen as a short tutorial complementary to the preceding section
- Sec. A.10 and Sec. A.11 explain in detail the database structure implemented by `db_class` and its derived class `db_merged`
- Sec. A.12 briefly describes some list processing Tcl routine commonly used in the code
- finally, Sec. A.13 gives a summary of all the classes implemented, their `heritage`, `instvars` and `instprocs` and acts as a lookup table in place of a reference index.

A.2 Class ds_XXX

heritage: for multiple inheritance, OTcl determines a linear inheritance ordering that respects all the local superclass ordering; note however that for a given class, `info superclass` and `info heritage methods` will not generally produce the same result.

instprocs: these are the class methods, common to every class instance; anyway, each instance of that class can define its own `proc` s, which are not accessible from others class instances.

instvars: these variables are accessible from any class method, either via `$self set varname` or by declaring `instvar varname` in method scope. A class inherits all the instvar of a class from which it derives.

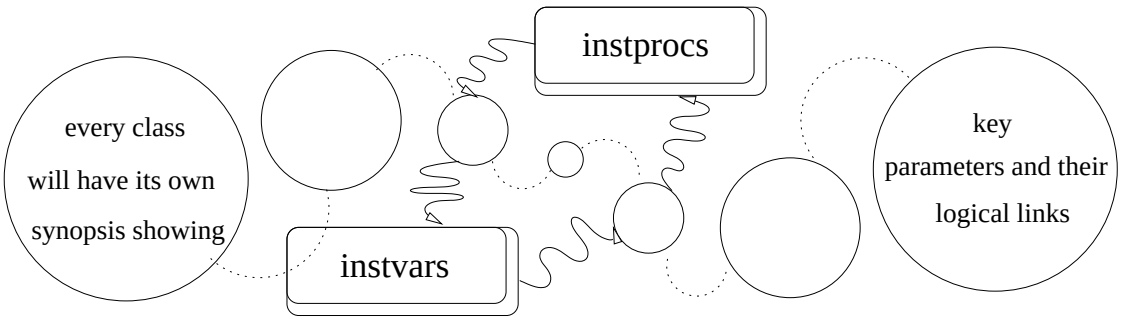


Figure A.2: ds_XXX Synopsis

Class instvars

`Class ds_XXX -superclass ds_YYY ds_ZZZ`
`ds_YYY` and `ds_ZZZ` are the classes from which `ds_XXX` inherits methods and data. Knowing class dependances and `instvars` is fundamental to understand the purposes of an object, since all of that class' methods (i.e., OTcl's `instprocs`) will handle and manipulate them.

`every`
instvar will then be introduced in this section...

Class instprocs

`...and {every}`

instproc will be described in detail in this one. Relationship between object's methods and data iwll be exposed and classes interdependance underlined as well to give a clear and complete idea of the internal class implementation; some relevant fragments

```
#of ds_XXX OTcl code will be shown
```

will also be shown and commented.

A.3 Class ds_common

heritage: Object

instprocs: init dbg destroy infoclass arraynames

instvars: nodes_ avg_ time_ debug_

Global vars

`ns`
Simulator instance that will be used for any proc scheduling, nodes generation, etc ..., by all the methods of any *ds_XXX*class.

`rng`
Random variable of NS class RNG, its seed defaults to 0.

`dim_`
Array that contains the measure unit names of ds_link's variables. This because a – no longer maintained – `GNUplot` class used ds_common's debug mechanism and vars, coupled with ds_monitor's output files, to automatize the GNUplot command files production.

Class instvars

```
Class ds_common -superclass Object
```

`nodes_`
Array with three elements:

`$nodes_(ds)` whose value corresponds to the number of DiffServ-enhanced flows used in the current simulation

`$nodes_(be)` keeping the number of Best Effort ones and

`$nodes_(tot)` whose value is the sum of the former and the latter.

`avg_`
Array with two elements: `$avg_(siz)` and `$avg_(rtt)`, basically used by ds_link's `appval` method which does measure some flow's parameters.

`time_`
Array defining `$time_(start)` the `$time_(stop)` and simulation's parameters

`debug_`
Array that contains the file channels for every *ds_XXX instance* that uses debug mechanism. An important note is that the current debug implementation uses different filenames and filechannels for *appending* debug information; these are produced exclusively for the *base* class of every instance, ignoring the classes from which it derives; an alternative approach would use a single filechannel (and consequently a single filename) to exhaustively collect every (class,method) pair's debug messages.

Class instprocs

`init {}`
This method gives default values to the simulation parameters described in instvar section. Notice that each object whose base class derives from ds_commonwill get a copy of those instvars: if you want to change them, either modify the default values or remember to set their values for any *ds_XXX* instance.

`arraynames {args}`
This method returns the array names of an instvar outside an object.

`dbg {args}`
This method puts debug information contained in `$args` strings into the filehandle contained in `$debug_([$self info class])`, pointing to a file named `"debug.[$self info class]"`. Before flushing the output, the name of the calling instance of that class is prepended to `$args`.

`infoclass {}`
This method is especially useful to become acquainted with OTcl classes, reporting info about `heritage, instprocs, instvars` of any derived class calling instance. The section Sec. A.13 has been built filtering the following ns code's output:

```
puts [[ds_common     instance] infoclass]
puts [[ds_link       instance] infoclass]
puts [[ds_sources    instance] infoclass]
puts [[ds_domain     instance] infoclass]
puts [[ds_monitor    instance] infoclass]
puts [[ds_bottleneck instance] infoclass]
```

The fist line of code output is, i.e:

```
#
# Class : ds_common : [ com : infoclass ]
# Heritage : Object
# $class instances : com
# $class instprocs : init dbg destroy infoclass arraynames
#
# $self : com
# $self procs :
# $self info vars : nodes_ avg_ time_ debug_
```

`destroy {}`
This OTcl destructor properly unsets all variables of an object, then closes debug file channels, to avoid data losses. No explicit call is needed for a destructor method, since the OTcl class mechanism automatically provides it.

A.4 Class ds_link

heritage: ds_common Object

instprocs: app infolink type size pir linkudm init appval edge cir almost_all dest linkval bw rtt appvar linkvar

instvars: nodes_ nflows_ avg_ time_ debug_ nd id rtt_ siz_ type_ DSCP_ cir_ pir_ bw_ edge_ e_id dest_ d_id tcp_ ftp_ udp_ cbr_ sink_ app_ par_

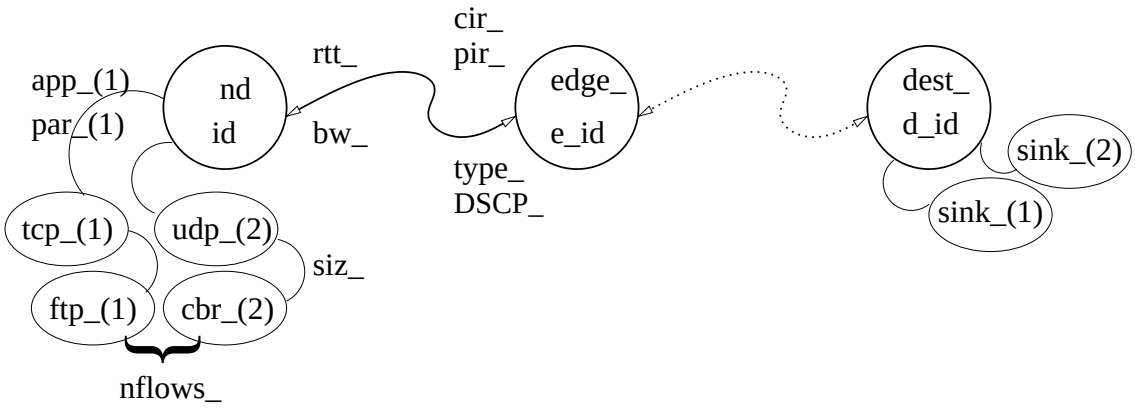


Figure A.3: ds_link Synopsis

Class instvars

`Class ds_link -superclass ds_common`

`nd, id`
A link, first of all, needs a starting point: `$nd` is the ns-node of id `$id`, to which every source agent and application will be attached. `$nd` is a ns node instance created by ds_link.

`edge_, e_id`
Then, comes the DiffServ domain ingress node `$edge_` of `$e_id` id; `$nd` and `$edge_` are connected by a DropTail link of Round Trip Time `$rtt_` ms and bandwidth `$bw_`. `$edge_` is just a reference to a previously created (precisely by a ds_domain class instance) ns node instance.

`dest_, d_id`
Finally, the destination `$dest_` node of id `$d_id`; here one can attach all the sink agenst of the applications having `$nd` as source node. `$dest_` is just a reference to a previously created (precisely by a ds_domain class instance) ns node instance.

`rtt_`
This is the `$nd` $\Longleftrightarrow$ `$edge_` link's Round Trip Time, expressed in milliseconds.

`bw_`
This is the `$nd` $\Longleftrightarrow$ `$edge_` link's bandwidth, expressed in Mbps.

`siz_`
This is the packet size produced by applications having `$nd` as source node.

`type_, DSCP_`
: Variable `$type_` is a string (which can assume the value `"ds"` or `"be"`) discriminating whether a micro-flow is DiffServ-compliant or just a Best Effort flow; in the latter case `$cir_` and `$pir_` are set to 0.0 Mpbs and `$DSCP_` to 00, while in the former `$DSCP_` is set to 10 (Green codepoint) and `$cir_`, `$pir_` are left unset.

`cir_, pir_`
: These are, respectively, the Committed Information Rate and the Peak Information Rate, expressed both in Mbps, that configure the TSW3CM marker.

`nflows_`
: Many source applications may share the same source node: `$nflows_` variable keeps the number of individual micro-flows that, starting from source node `$nd` and entering DiffServ domain in `$edge_` node, will reach a common `$dest_` node.

`app_, par_`
: These are arrays of strings, having both `$nflows_` elements. Current implementation of ds_link allows the use of `"tcp"` and `"udp"` agents which are the possible values of each `$app_()` element; the array `$par_()` contains the necessary application parameters.

`tcp_, ftp_, sink_`
: In the case that `$app_($i)=="tcp"` (with $0 \leq$ `$i` $<$ `$nflows_`), then a TCP agent `$tcp_($i)` is attached to source node `$nd`, an FTP application `$ftp_($i)` is attached to `$tcp_($i)` and a TCP sink `$sink_($i)` is attached to the node. The parameter's array `$par_($i)` should contain nothing or a dummy value.

`udp_, cbr_, sink_`
: In the case that `$app_($i)=="udp"` (with $0 \leq$ `$i` $<$ `$nflows_`), then a UDP agent `$udp_($i)` is attached to source node `$nd`, a CBR application `$cbr_($i)` is attached to `$udp_($i)` and a Loss Monitor sink `$sink_($i)` is attached to the node. The parameter's array `$par_($i)` contains the CBR rate parameter with fully specified measure unit.

Class instprocs

`init {}`
: Creates a source-node `$nd` and store its numeric id in `$id` using the global instance of the simulator, set to 0 the number `$nflows_` of applications, then calls ds_common's `init` instproc

`rtt, size, bw, cir, pir {value}`
: Each of the these instprocs just set the respective (thus `$rtt_`, `$siz_`, `$bw_`, `$cir_`, `$pir_`) variable value, then calls the debug function. For questions about variable's measure units, refers to instvar section.

`edge, dest {noderef}`
: Both the instprocs set the respective (thus `$edge_`, `$dest_`) variable value to a previously created ns-node, then stores that node id in the correspondant var (thus `$e_id`, `$d_id`).

`type {typestr}`
: Choosing a link type implies the choise of an initial DiffServ codepoint and a fixed Service Level Specification common to all the micro-flows of the aggregate. By setting `$type_` to `"be"`, the `$DSCP_` is set to 00 and `$cir_` and `$pir_` to 0.0 Mpbs; if `$type_` is `"ds"` then `$DSCP_` is set to Green codepoint 10.

`app {args}`
: This method requires that `$rtt_`, `$siz_`, `$bw_`, `$dest_` and `$edge_` are already properly set. The reason is that when the first flow is created, it sets up a duplex link between the source node and the ingress edge:

```
if {$nflows_==0} {
  $ns duplex-link $nd $edge_ [expr $bw_]Mb [expr $rtt_]ms DropTail
}
```

and the packet size will be used in agent's configuration. The argument `$args` passed to the function will be interpreted as a list containing two strings: the application name and (possibly) its parameter, in that order. Only two protocols are currently supported, `"tcp"` and `"udp"`, with their respective applications agents FTP and CBR; if none of these parameter is passed to the instproc, a fatal error will occur. Several micro-flows are supported within an aggregate, and vectors `$par_()`, `$app_()`, `$sink_()` will have the same dimension. At each call, this method preliminary increments `$nflows_`, which will be used as index of any of the mentioned vectors; moreover, it will be used for the others coupled arrays: (`$tcp_()`,`$ftp_()`) and (`$udp_()`, `$cbr_()`), meaning that there may be a gap between two indexes of any of these vectors (and if one vector has such an index discontinuity, then all the others will). Now supposing that currently `$nflows_-1` are already defined, then setting $i to `$nflows_`, the i-th flow will be built in the following way: in the case that the first element of `$args` matches the string `"tcp*"`

- a TCP agent (`set $tcp_($i)[new Agent/TCP]`) is created
- then attached to the source (`$ns attach-agent $nd $tcp_($i)`)
- then a FTP agent (`set $ftp_($i) [new Application/FTP]`) is created
- and attached (`set $ftp_($i) [$tcp_($i) attach-app FTP]`) to TCP agent
- finally, a TCP sink (`set $sink_($i) [new Agent/TCPsink]`) is created
- then attached (`$ns attach-agent $dest_ $sink_($i)`) to the destination node
- and connected to TCP source (`$nsconnect $tcp_($i) $sink_($i)`)
- FTP application is started at `$time_(start)` and stopped at `$time_(stop)`. If you experience syncronisation problems, either try to change the random variable `$rng`'s seed in ds_common, or shift each application starting time of a random time of the order of μs.

Else, if the first element of `$args` matches `"udp*"`:

- `$app_($i)` is setted to `"udp"`
- while `$par_($i)` will assume the 2nd list value, if any; otherwise it is defaulted to 64Kb
- an UDP agent (`set $udp_($i) [new Agent/UDP]`) is created
- attached (`$ns attach-agent $nd $udp_($i)`) to the source node
- a CBR source (`$cbr_($i) [new Application/Traffic/CBR]`) is created
- attached (`$cbr_($i) attach-agent $udp_($i)`) to the UDP agent
- and its constant sending rate is setted to the value of `$par_($i)`
- a sink (`$sink_($i) [new Agent/LossMonitor]`) is creted
- then attached (`$ns attach-agent $dest_ $sink_($i)`) to the destination node
- and connected to UDP source (`$nsconnect $udp_($i) $sink_($i)`)
- CBR application is started at `$time_(start)` and stopped at `$time_(stop)`.

`appvar {}`

This method is used, coupled with `appval`, to produce the measurements of two performance key parameters for long-lived flows: Throughput and Goodput. Its purpose is to return the names of the parameters calculated and returned by `appval`, in the same order. The current implementation of this method returns the string `"throughput goodput"` if the link has any active flow, otherwise, it outputs an error message and exits.

`appval {args}`

This methods returns the values of the variables that are monitored. It is called by both time and flow monitors of ds_monitorclass to collect per-flow information. It firstly calculates both Throughput and Goodput on a per-flow basis (using `$tcp_()` or `$sink_()` objects depending on the flow's transport protocol layer and application) and then produces the correspondent aggregated-values, summing all the results for that source. The optional parameter `$args` changes the routine output in the following way:

> `"notdetailed"`: Only the global values (that is, the sum of each microflow's variable) for that source node are returned.
>
> `"justdetailed"`: Every microflow's variable value for that source node is appended to the result string, prepended by a comment # character, then the string is returned. The comment # character is useful for reasons that will be clearer later: all the simulation flow's data collected by a monitor are handled by a `db_class` object, which is described later in section Sec. A.10.

The default behaviour of the routine, when no arguments are passed on the command line, is to return both results: the string produced by `"notdetailed"` as a header, then the `"justdetailed"` per micro-flow ones.

`linkvar, linkudm, linkval {list}`

These three methods are the similar to the (`appval`,`appvar`) couple described earlier; all of them are focused on simulation parameters, i.e., ds_linkinstvars that are constant during a simulation, rather than calculated values. Each of them processes the ds_link's vars passed as the list `$list` and returns:

- `linkvar` returns variable names (that is, the same list that has been passed; but with a major difference: see below the `$list` special value)
- `linkudm` linkudm returns each measure unit name if defined (see `$dim_()` global var described in Sec. A.3) or `".."` otherwise
- `linkval` returns the values of these variables; it handles vectors, other than a simple scalar case, by returning vector's size (preceded by the string `"arr"`) or, in the case of a one-sized vector, by returning its only value.

If the parameter `$list` assumes the special value {`all`}, then each of these functions will call the `almost_all` function to determine the variables of interest. Information on links are collected by ds_sources's methods (Sec. A.5) using these instprocs.

`almost_all {}`

This method is called by one of the `linkvar, linkudm, linkval` to interpret the `"all"` keyword. It returns the list of the link parameters of interest, excluding

- `[[ds_common instance] info vars]`
- all the variables that are redundant or not in a *human readable* form: `$udp_()`, `$tcp_()`, `$ftp_()`, `$sink_()`, `$cbr_()`, `$edge_`, `$dest_`, `$nd`, `$d_id`, `$e_id`, `$DSCP_`

`infolink {}`

Returns ds_link's instance parameters information. Here's a –truncated– output example

```
#
#DSCP_ app_   bw_  cir_  d_id  dest_  e_id  edge_  ftp_  id   nd    nflows_  par_
#..    agent  Mbps Mbps  dest  node   edge  node   ..    $elf $elf  ..       ..
10      tcp   1    0.5   0     _o12   1     _o14   _o32  2    _o16  1        /
```

A.5 Class ds_sources

heritage: ds_common Object

instprocs: infosources setdest init fwrite setedge fread addlink

instvars: nodes_ avg_ time_ debug_ s_ edge_ dest_

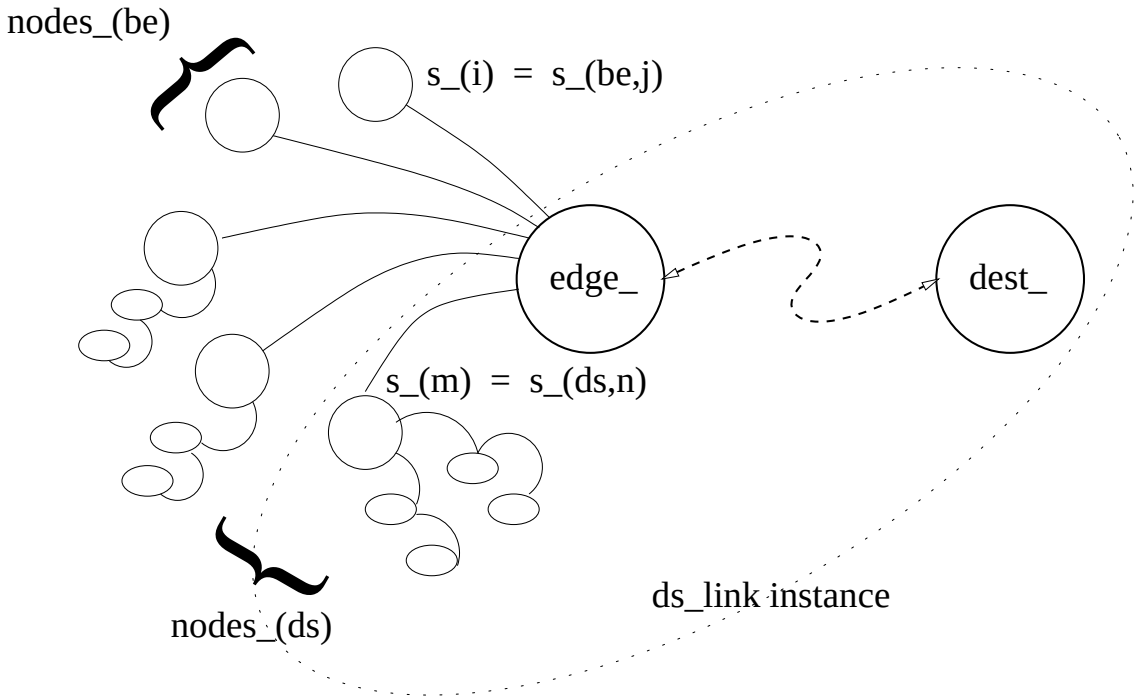


Figure A.4: ds_sources Synopsis

Class instvars

```
Class ds_sources -superclass ds_common
```

`nodes_`

This array has three elements:

`$nodes_(be)`: number of Best Effort source nodes

`$nodes_(ds)`: number of DiffServ source nodes

`$nodes_(tot)`: total number of the source nodes

`s_`

This variable is an array; each of its elements contains an handle to a single ds_link instance (see Sec. A.4). The indexes of this arrays are redundant, in the sense that two indexes points to the same element

`$s_(be,$x)`: where `$x` can assume all values between 1 and `$nodes_(be)`

`$s_(ds,$y)`: where `$y` can assume all values between 1 and `$nodes_(ds)`

`$s_($z)`: where `$z` can assume all values between 1 and `$nodes_(tot)`; this element has a *clone*, which index prefix depends in `[$s_($z) set type_]` (see Sec. A.4) and which ordinal number depends on how many others nodes of that type where previously initialized. Each vector's order reflects the order used in adding ds_link references to a ds_sources object.

`edge_`
This is the reference to the ingress node, common to all the sources. Note that the node is not created in this object (ds_domain cares of its construction, see Sec. A.6); however, ds_sources class takes care of properly setting every `$edge_` variable of each `$s_()` object.

`dest_`
This is the reference to the ingress node, common to all the sources. As for `$edge_`, this node is not created but only referenced here; however, ds_sources class takes care of properly setting every `$dest_` variable of each `$s_()` object. In the case that more than one destination is necessary, the implementation of ds_sources should be enhanced to support an array of destinations.

Class instprocs

`init {args}`
This method is totally virtual.

`setedge, setdest {noderef}`
Both these methods take exactly one argument on the comment line, which is the reference, respectively, to ingress node and destination node; ds_domain initialization method (see Sec. A.6) create this pair of nodes and takes care of properly call the respective method.

`addlink {linkref}`
The argument `$linkref` is the name of an existent ds_link object instance that will be added. The method takes care of updating nodes number

```
set ty [$linkref set type_]
incr nodes_(tot)
incr nodes_($ty)
```

then creates two references to the link object, appending it to two (global and per-type) *tails* of the `$s_()` array:

```
set s_($nodes_(tot)) $linkref
set s_($ty,$nodes_($ty)) $linkref
```

That way, you can access every source

- either by an ordinal index with `"ds"` and `"be"` flows appearing in the order you've putted them in:

```
for {set i 0} {$i<$nodes_(tot)} {incr i} {
   puts "source s_($i) type: [$s_($i) set type_]"
}
```

- or by a *multidimensional* array (which is in fact a hash) where the index is given by the combination of the flow-type and a numerical index:

```
foreach type {ds be} {
   foreach i [lsort [array names s_ $type,*]] {
      puts "source s_($i) id: [$s_($i) set id]"
   }
}
```

`fread, fwrite {fname}`
This two methods allow reading and writing of the simulation parameter's set. Anyway I discourage its use, being these routines not robust (who can write robust routines in Tcl without a try/catch mechanism?) and quite simplified (they consider that every micro-flow of an aggregate use the same transport and application layers).

`infosources {}`
This method produces an output like the following:

```
# sOrCi!  ds_sources src infos
#
#       Source  Nodes:      2     (1ds,  1be)
#
#app_   bw_   cir_   id    nflows_  par_  pir_  rtt_  siz_   type_
#agent  Mbps  Mbps   $elf  ..       ..    Mbps  ms    bytes  ..
tcp     1     0.5    0     1        /     0.75  10    1000   ds
udp     1     0      1     1        64Kb  0     20    1000   be
```

A.6 Class ds_domain

heritage: ds_sources ds_common Object

instprocs: infodomain init queue

instvars: edge_ nodes_ avg_ dn_ prefix_ time_ debug_ dest_ q_

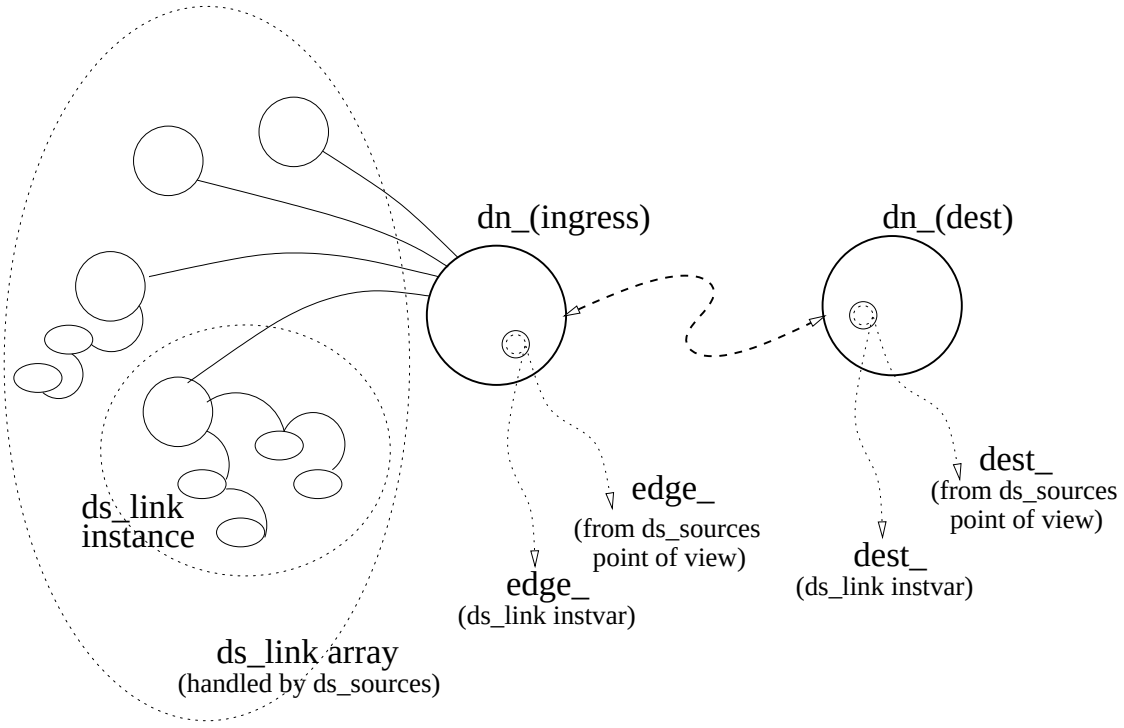


Figure A.5: ds_domain Synopsis

Class instvars

```
Class dsDom -superclass ds_sources
```

`prefix_`
: These are the prefixes used internally to distinguish between core nodes, edge nodes and destination nodes. It is just a common-sense notation,

```
$self set prefix_(core) c
$self set prefix_(edge) e
$self set prefix_(dest) d
```

`dn_`
: This array, whose name is the acronym of *domain node*, has two special indexes, whose elements refer to two nodes created by the constructor `init` of this class: `$dn_(ingress)` and `$dn_(dest)`. It is recommended that each array index be named according to the prefix notation described above, i.e: `$dn_(e1)`, `$dn_(c1)`, ..., `$dn_(d1)`, ..., being always possible to give an *alias* to `$dn_(ingress)` and `$dn_(dest)` (see ds_bottleneck internals in Sec. A.8 for an example).

`q_`
This array variable keeps a pointer to each domain queue; here, no queue will be set up, so `$q_` array won't be initialized. This was defined here just because a DiffServ queue is a logical object of a DiffServ domain.

Class instprocs

`init {}`
The `init` method creates two particular ns nodes, `$dn_(ingress)` and `$dn_(dest)`, without defining the in-between path yet. These two nodes are well-known to the reader, despite their names: they are handled by both ds_sources class methods and every ds_link instance has got a reference to them. These references have been called in the same way, respectively `$edge_`and `$dest_`, both for ds_sources (see Sec. A.5) and ds_link (see Sec. A.4); this methods set them up by calling the appropriate ds_sources methods, telling them which is the ingress and which is the destination of each source:

```
$self instvar dn_
$self setedge $dn_(ingress)
$self setdest $dn_(dest)
```

An example of class that defines the topology inside these two points of the domain is given by ds_bottleneckdescribed in Sec. A.8.

`queue {args}`
This method is listed just to keep in mind that a queue is an element of a vector `$q_` which is an ds_domain instvar. If `$args` is a valid `$q_` index, then `$q_($args)` is returned; if no parameter is passed on the command line or if `$args` isn't a valid queue index, then the list of the queue elements is returned (that is, `[array names q_]`).

`infodomain {}`
Rather than overloading a single info method and chaining the calls `eval $self next $args` , each *ds_XXX* class has its own. This simply means that calling

```
puts [dom infodomain]
```

will yield

```
# dOOm! ds_domain dom infos
#
# Domain nodes :
# All   dn_(...): [dest ingress]
# Edges dn_(e,...): []
# Cores dn_(c,...): []
# Dests dn_(d,...): [dest]
#
# Domain queues :
# Queues q_(...) : []
#
```

while we are still able to collect information about sources via

```
puts [dom infosources {all}]
```

obtaining

```
#       sOrCi!  ds_sources  dom   infos
#
#       Source  Nodes:      2     (1ds,    1be)
#
#app_   bw_     cir_        id    nflows_  par_  pir_  rtt_  siz_
#agent  Mbps    Mbps        $elf  ..       ..    Mbps  ms    bytes
tcp     1       0.5         0     1        /     0.75  10    1000
udp     1       0           1     1        64Kb  0     20    1000
```

or either information about a single source:

```
puts [[dom set s_(ds,1)] infolink]
```

whose outputs would be similar to those shown in ds_link's `infolink` instproc description (see Sec. A.4)

A.7 Class ds_monitor

heritage: ds_domain ds_sources ds_common Object

instprocs: flush_time timemon suite flowmon init samples progress done infomon flush_queue flush_flow flowstat run queuemon

instvars: edge_ nodes_ infostr_ avg_ dn_ prefix_ time_ debug_ nsamples_ dest_ s_ timemon_ flowmon_ suite_ CWND_ smpcoeff_ quiet_

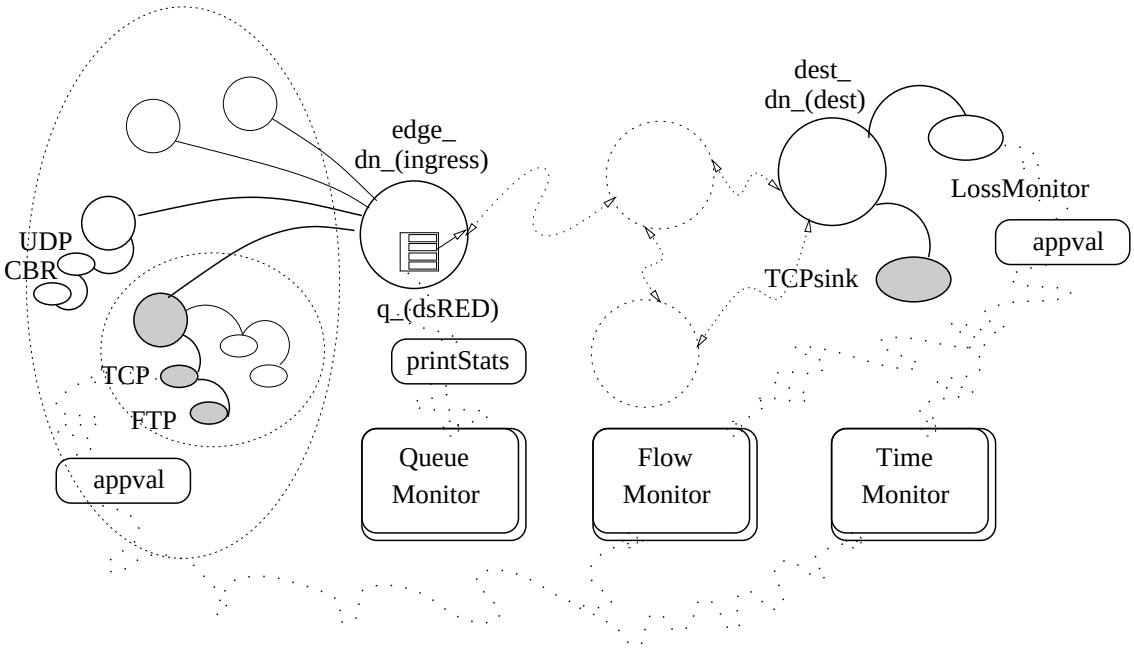


Figure A.6: ds_monitor Synopsis

Class instvars

```
Class ds_monitor -superclass ds_domain
```

`suite_`
Array variable holding string parameters for simulation file handling. See `suite` method in Sec. A.7 for further details.

`nsamples_`
Array which holds the requested number of samples in `$nsamples_(max)` and the current (thus at `$ns now`) number of samples in `$nsamples_(timemon)`.

`smpcoeff_`
This coefficient is calculated by the sampling instproc as

```
$self set smpcoeff_ [expr $time_(stop) / ($nsamples_(max) * 1.35)]
```

and it is used by the time monitor to determine the random delay to idle before next sampling instant:

```
set later [expr $now_ + [$rng exponential] * $smpcoeff_]
```

`timemon_`
: This array holds time-monitor file channels, whose names are determined following the suite notation.

`flowmon_`
: This array holds flow-monitor file channels, whose names are determined following the suite notation.

`quiet_`
: Used uniquely by `run` (which sets it to 1) and `done` (which tests if it's 1) methods to avoid the dumping of ns's scheduler queue at the end of simulation.

`CWND_`
: Array containing cwnd statistics; in current implementation they are collected by time-monitor (precisely by `flush_time` method) and they are printed on the `stdout` at the end of simulation (by `flush_flow` method). The array separates DiffServ from Best Effort data, keeping the maximum value of all the flows of that type and the calculated running average value; after some manipulation, a similar report is printed

	DS	BE	All
cwnd_(max)	22.6661	13.3404	22.6661
cwnd_(avg)	5.5294	4.0420	4.7883

Refer to the code of the mentioned methods for further details.

`infostr_`
: String returned by the `infomonitor` methods; every ds_monitor's method called should append its information to this instvar.

Class instprocs

`init {args}`
: The ds_monitor's constructor performs a few variables initialization, i.e. sets to 0 the number of samples of every monitor, before scheduling the `done` method a safe time after simulation's end `$time_(stop)`

`suite {args}`
: When dealing with a wide range of simulation scenarios over a same topology,

> **Murphy's Axiom of Filedynamics**
> The number of files produced by each simulation run tend to overcome
> *any* effort to keep them ranked in *any* order.

The argument passed on the command line should be a list composed of several elements. The first of them will be interpreted as the *subdirectory* of the current working directory where any produced file should be stored; from second to last, elements will be concatenated by a dot "." character, building an unique *file extension*. Each list element will be stored in the (zero-based) array `$suite_()` for direct access; this array has also other key values:

`$suite_(num)` holds the index of the last array element

`$suite_(fext)` gives the extension of the file evaluated as dot concatenation of each element of the list from `$suite_(1)` to `$suite_($suite_(num))` , prepended by a "." character

`$suite_(bpath)` holds a copy of subdirectory named `$suite_(0)`, which is created if inexistent

`samples {nsmp}`
This method allows you to choose the number of samples to be taken following a random-spaced Monte Carlo approach. The final number of samples, stored in `$nsamples_(timemon)`, will be surely a bit different from the requested `$nsmp`, which is stored in `$nsamples_(max)`. Here's how `$smpcoeff` is calculated:

```
$self set smpcoeff_ [expr $time_(stop) / ($nsamples_(max) * 1.35)]
```

`infomon {}`
This method simply returns the string variable `$infostr_`, which contains information about monitors and flows possibly added by any ds_monitor method when called. I admit that I've never used it.

`flowstat {}`
This method is left totally virtual, and an overload consistent example is given by ds_bottleneck class in Sec. A.8. If necessary, its return values (necessarily a string) could be added by the flow-monitor to the head of the produced text file.

`flowmon, flush_flow {args}`
A Flow Monitor is implemented by the couple of methods `flowmon, flush_flow`. It does generate three tables containing per-flow data (which involves DS only, BE only and All flows). All that `flowmon` does is to schedule the other method of the pair,

```
$ns at [expr $time_(stop)+1]  "$self flush_flow $args"
```

`flush_flow`, which does the all the dirty work. The data values are collected at `$time_(stop)` (when the variables –should– have reached a stable state) from

- ds_link's instvars (such as Round Trip Time, `$siz_`, `$bw_`, Commited Information Rate, Peak Information Rate, `$type_`, etc ...), which are constant of the simulation scenario. The list of vars can passed on the command line, otherwise it defaults to the special value {`all`} (see Sec. A.4)
- `appval` ds_link's method
- `flowval` method which must be defined by top hierarchy class (see Sec. A.8 for an example)

Two of the tables created report DS and BE flows separately, and are implemented as `db_class` The following lines of code collect the data, iterate over all possible source ds_link indexes, and store in two databases

```
db_class ds
db_class be
foreach type {ds be} {
   foreach i [array names s_ $type*] {
     set    dataS "[$s_($i) linkval $args] "
     append dataS "[$s_($i) appval notdetailed] "
     append dataS "[$self flowval $i]"
     $type addR $dataS
   }
 }
```

The db_class implement some facilities to add data by rows (and by column) to calculate –painlessly– per-column Sum, Average, Variance and Standard deviation, other than storing the database on a file (if you find it not so evident, see Sec. A.10):

```
foreach type {ds be} {
   if {$nodes_($type)>0} {
      $type ReAnalize
      $type fwrite "$suite_(0)/flows.$type$suite_(fext)"
   }
}
```

The third table is produced merging the previously created databases and is implemented as `db_merged` object (see Sec. A.11). This classes extends the first and is able to distinguish between the origin of the data in calculating per-column Sum, Average, etc...

```
if {$nodes_(ds)>0 && $nodes_(be)>0} {
   db_merged all
   all mByR "ds be"
   all mAnalize "ds be"
   all fwrite "$suite_(0)/flows.all$suite_(fext)"
}
```

an example of the final (and, as usual, truncated) output is

```
#
#
#          Merged!    flowmonitor  {all}
#
#app_   bw_    cir_   id     nflows_  par_  pir_   rtt_   type_ throughput  goodput   trg+eq
#agent  Mbps   Mbps   $self  ..       ..    Mbps   ms     ..
tcp     1      0.25   18     1        /     0.25   20.0    ds   0.374765    0.349168  0.375000
tcp     1      0.25   0      1        /     0.25   20.0    ds   0.260492    0.256160  0.375000
tcp     1      0.25   2      1        /     0.25   20.0    ds   0.258654    0.254392  0.375000
tcp     1      0.25   4      1        /     0.25   20.0    ds   0.258801    0.254348  0.375000
tcp     1      0.25   6      1        /     0.25   20.0    ds   0.306371    0.295333  0.375000
tcp     1      0.25   8      1        /     0.25   20.0    ds   0.309675    0.298663  0.375000
tcp     1      0.25   10     1        /     0.25   20.0    ds   0.332829    0.314856  0.375000
tcp     1      0.25   12     1        /     0.25   20.0    ds   0.340064    0.322263  0.375000
tcp     1      0.25   14     1        /     0.25   20.0    ds   0.338437    0.320158  0.375000
tcp     1      0.25   16     1        /     0.25   20.0    ds   0.374892    0.349985  0.375000
tcp     1      0.0    19     1        /     0.0    20.0    be   0.128265    0.102234  0.125000
tcp     1      0.0    1      1        /     0.0    20.0    be   0.049312    0.040800  0.125000
tcp     1      0.0    3      1        /     0.0    20.0    be   0.050351    0.041400  0.125000
tcp     1      0.0    5      1        /     0.0    20.0    be   0.090827    0.074392  0.125000
tcp     1      0.0    7      1        /     0.0    20.0    be   0.085710    0.070028  0.125000
tcp     1      0.0    9      1        /     0.0    20.0    be   0.098227    0.080453  0.125000
tcp     1      0.0    11     1        /     0.0    20.0    be   0.117329    0.094706  0.125000
tcp     1      0.0    13     1        /     0.0    20.0    be   0.121598    0.096830  0.125000
tcp     1      0.0    15     1        /     0.0    20.0    be   0.123033    0.096849  0.125000
tcp     1      0.0    17     1        /     0.0    20.0    be   0.120838    0.095905  0.125000
#
#Analisi: ds, be, all
#SUM
#-      10.000 2.5000 90.000 10.00000 -     2.5000 200.00  -    3.154979    3.015326  3.750000
#-      10.000 0.0000 100.00 10.00000 -     0.0000 200.00  -    0.985490    0.793597  1.250000
#-      20.000 2.5000 190.00 20.00000 -     2.5000 400.00  -    4.140469    3.808923  5.000000
#AVG
#-      1.0000 0.2500 9.0000 1.000000 -     0.2500 20.000  -    0.315498    0.301533  0.375000
#-      1.0000 0.0000 10.000 1.000000 -     0.0000 20.000  -    0.098549    0.079360  0.125000
#-      1.0000 0.1250 9.5000 1.000000 -     0.1250 20.000  -    0.207023    0.190446  0.250000
#VAR
#-      0.0000 0.0000 33.000 0.000000 -     0.0000 0.0000  -    0.001806    0.001212  0.000000
#-      0.0000 0.0000 33.000 0.000000 -     0.0000 0.0000  -    0.000784    0.000468  0.000000
#-      0.0000 0.0156 33.250 0.000000 -     0.0156 0.0000  -    0.013061    0.013181  0.015625
#SDEV
#-      0.0000 0.0000 5.7445 0.000000 -     0.0000 0.0000  -    0.042496    0.034821  0.000000
```

```
#-      0.0000 0.0000 5.7445 0.000000 -     0.0000 0.0000  -    0.027991    0.021643  0.000000
#-      0.0000 0.1250 5.7662 0.000000 -     0.1250 0.0000  -    0.114287    0.114807  0.125000
```

This method is also used to manipulate and print cwnd data; refer to the code for implementation details.

`queuemon, flush_queue {args}`
The Queue Monitor method's couple have been simplified during classes evolution and now its behaviour is no longer time oriented: it just gives a report on overall packet dropping, calling, for each of the queue's vector `$q_` index passed on the command line, the dsRED's `printStat` method, whose output is:

```
Packets Statistics
=======================================
 CP  TotPkts   TxPkts   ldrops   edrops
 --  -------   ------   ------   ------
All   182873   182873        0        0
  0    20545    20545        0        0
 10   135416   135416        0        0
 11    26912    26912        0        0
```

All that `queuemon` method does is

```
$ns at $time_(stop) "$self flush_queue $args"
```

And `flush_queue`

```
foreach k $args {
     puts "queue $k"
     $q_($k) printStats
}
```

Beware, since there's no output syncronization between Tcl `puts` and C `printf` (used by `printStats`): this means that the foreach output may present all queue statistics *after all* the `"queue $k"` output lines.

`timemon, flush_time {}`
The Time Monitor is implemented by the couple of methods `timemon, flush_time`. The `timemon` instproc opens a file channel for each variable returned by the `appvar` (and consequently `appvar`) methods defined in the ds_link class; at each time-sample (by row) the routine writes the value of the sampling instant on the first column, then the data values of each flow (by column); the sorting order is alphabetical, meaning that, i.e, `"ds,10"` and `"ds,17"` will be printed *before* `"ds,2"`, as schematized below for a ten BE – ten DS flows example:

```
#
# Throughput TimeMonitor {  }
#time    ds,1    ds,10   ds,2          ds,9    be,1    be,10 be,2          be,9
1.00     0.17613 0.24576 0.16384  ...  0.13517 0.22733 0.233 0.18022  ...  0.07373
1.15     0.17683 0.23577 0.15361       0.11789 0.22506 0.251 0.15718       0.06115
 ...       ...                                ...                          ...
```

`timemon` method opens the filechannels, setting up their names using the `$suite_(bpath)` directory and the `$suite_(fext)` extension, writes a file header, sets up every `$CWND_` entry to 0 and then schedules the first `flush_time` execution after one second of simulation. As a side note, it will be said that the current implementation of `timemon` could be enhanced to support automatized file-name generation, provided that the appvar method returned value does not contain any *funny character*, that is, any character having a special meaning to the shell:

```
foreach ch [splitblank [$s_(1) appvar]] {
    set timemon_($ch)\
        [open "$suite_(bpath)/time.$ch$suite_(fext) w]
}
```

That way, the `arraynames` of `$timemon_` will then be the names of the application variables returned by appvar (case-sensitevely). As an example, if one of the `appvar`'s name contains a final `"."` dot character to indicate an abbreviation, concatenating `$ch` to `$suite_(fext)` would produce a `".."`; otherwise, if the `appvar`'s name contains slash `"/"` characters, an error would be surely produced unless you care to build the resulting directory tree in advance. An alternative approach (which uses some of the list's `proc` described in Section A.12) would imply a zero-based indexing

```
set ch [splitblank [$s_(1) appvar]]
foreach k [lessof [llength $ch]] {
    set timemon_($k)\
        [open "$suite_(0)/time.[lindex $ch $k]$suite_(fext) w]
}
```

but will not change the filenames (this numerical indexing is the one used by `flush_time` in the current implementation). The `flush_time` method is firstly scheduled by `timemon`, then it reschedules itself `$later`, provided that `$later<$time_(stop)`, where

```
set later [expr $now_ + [$rng exponential] * $smpcoeff_]
```

The main loop of this routine:

```
foreach type {ds be} {
   foreach i [lsort [array names s_ $type,*]] {
      set k 0
        ...
      foreach val [splitblank [$s_($i) appval notdetailed]] {
         puts -nonewline $timemon_($k) [format " %.5f" $val]
         incr k
      }
       ...
   }
}
```

This method is also used to collect cwnd average and peak values; refer to the code for implementation details.

`progress {args}`

Useful to give an idea of the real-time duration of a simulation, this method indicator will print out a message every $\frac{1}{\texttt{\$howmany}} \cdot \texttt{\$time\_(stop)}$ seconds of simulation, where `$howmany` is the real value passed on the command line. If no param is given, then `progress` will reschedule itself five times before the end of simulation.

`run {}`

This is a really pleonastic method; however call it if you want to avoid dumping the scheduler queue (`$self set quiet_ 1`) at the end of simulation (for example if you're piping your output with `tee` or redirecting it into some file).

`done {}`

It firstly closes all filechannels, to avoid output-data losses; then makes ns -globally- rest in peace, in a silent way if the simulation was started with ds_bottleneck's `run` method:

```
$ns halt ; $ns gen-map
if !{$quiet_} {$ns dumpq}
return 0
```

A.8 Class ds_bottleneck

heritage: ds_monitor ds_domain ds_sources ds_common Object

instprocs: flowstat setqueues init flowvar setnodes flowval infobottleneck

instvars: edge_ nodes_ infostr_ avg_ dn_ prefix_ time_ debug_ nsamples_ dest_ s_ timemon_ flowmon_ suite_ CWND_ smpcoeff_ quiet_ q_ bwneck

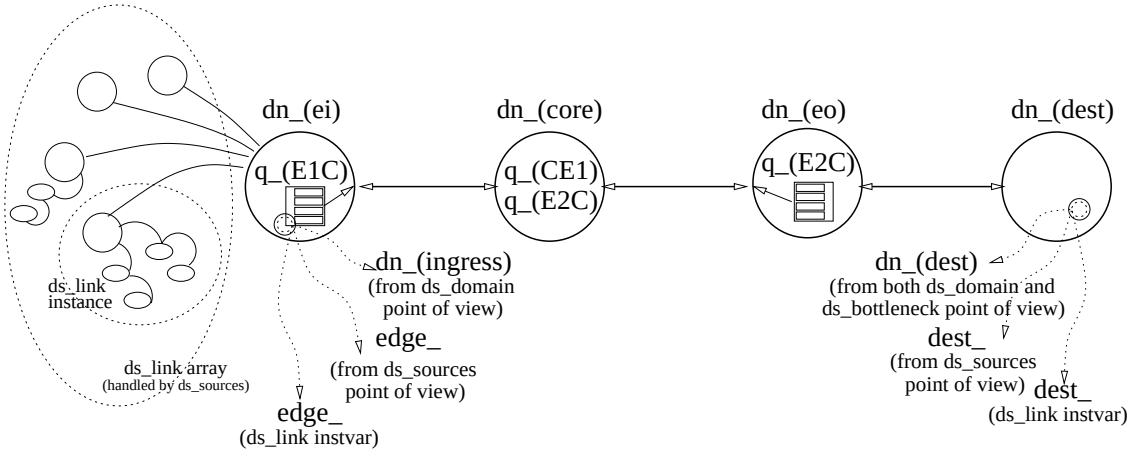


Figure A.7: ds_bottleneck Synopsis

Class instvars

```
Class ds_bottleneck -superclass ds_monitor
```

`bwneck`
: Surprisingly, the size of the bottleneck, expressed in Mbps. There are two duplex links which bandwidth is set to `$bwneck`: `$dn_(ei)` $\Longleftrightarrow$ `$dn_(core)` $\Longleftrightarrow$ `$dn_(eo)`.

Class instprocs

`init {args}`
: All that this method does is to set the size of the bottleneck to 5Mbps before handing over the task to the next class (ds_monitor) in the chain of initialization, via `eval $self next $args` call.

`setnodes {args}`
: This is the method which sets up the topology shown in Fig. A.7. It just creates two nodes (`$dn_(eo)` and `$dn_(core)`) while copies in `$dn_(ei)` a reference to `$dn_(ingress)` created by ds_domain initialization routine. Then the links between the DiffServ edge routers are created; it is to be noticed from the code that, due to the DiffServ philosophy (that is, push complexity to the edge), every duplex link will be created as two different single links: those links exiting core nodes are not required to perform any policing mechanism, while those entering are:

```
$ns simplex-link  $dn_(ei)   $dn_(core)  [set bwneck]Mb $RTT dsRED/edge
$ns simplex-link  $dn_(core) $dn_(ei)    [set bwneck]Mb $RTT dsRED/core
```

```
$ns simplex-link  $dn_(core) $dn_(eo)    [set bwneck]Mb $RTT dsRED/core
$ns simplex-link  $dn_(eo)   $dn_(core)  [set bwneck]Mb $RTT dsRED/edge
```

The last duplex link from egress edge to destination node will have a common DropTail queue and twice the bottleneck bandwidth

```
$ns duplex-link    $dn_(eo)   $dn_(dest)   10Mb $RTT DropTail
```

Notice that `$RTT` in not an instvar but just a local-scoped var.

`setqueues {args}`

This method is extremely important, as its task is to configure all DiffServ mechanisms and to set up the queue's vec `$q_` too. The two dsRED queues pictorially indicated in the synoptic, `$q_(E1C)` and `$q_(E2C)`, will accomplish policer tasks, and will monitored by ds_monitor's queuemon) routine, while `$q_(CE1)` and `$q_(CE2)` have only a routing task. Here a ds_common value (`$avg_(siz_)`) is used in setting up queues, then DiffServ codepoints are mapped to each virtual dsRED queues iterating over all queue indexes `$idx:`

```
$q_($idx) meanPktSize $avg_(siz_)
$q_($idx) set numQueues_ 1
$q_($idx) set NumPrec 3
$q_($idx) addPHBEntry 10 0 0
$q_($idx) addPHBEntry 11 0 1
$q_($idx) addPHBEntry 12 0 2
$q_($idx) addPHBEntry 00 0 2
```

After that, each virtual queue is configured with min_{th}, max_{th},max_p, either to work with a marker configured in three colour mode (TSW3CM)

```
#====================================
#   TSW3CM:       Grn Yel Red
#------------------------------------
$q_($idx) configQ 0 0 20 40 0.02
$q_($idx) configQ 0 1 10 20 0.10
$q_($idx) configQ 0 2  5 10 0.20
```

or in two colour mode (TSW2CM) setting `$cir_` equal to `$pir_` (in ds_link object) and extending Red's max_{th} to Green's min_{th} keeping the RED queue staggered

```
#====================================
#  TSW3CM used as TSW2CM:    Grn Red
#------------------------------------
$q_($idx) configQ 0 0 20 40 0.02
$q_($idx) configQ 0 2  5 20 0.20
```

Then, a policy entry for each of the sources is added to the policy table of each of the `$q_(E1C)` and `$q_(E2C)` queues:

```
$q_(E1C) addPolicyEntry $sourceid $destid   TSW3CM $dscp $cir $pir
$q_(E2C) addPolicyEntry $destid   $sourceid TSW3CM $dscp $cir $pir
```

and finally policer entry information (with `$qq` means either `E1C` or `E2C`) are added: Green will be pointed to Yellow and pointed to Red, Best Effort are never changed:

```
$q_($qq) addPolicerEntry TSW3CM 10 11 12
$q_($qq) addPolicerEntry TSW3CM 00 00 00
```

A final note about the `$args`: if the argument passed is the string `"quiet"`, then the following lines will not be executed:

```
$q_($qq) printPolicyTable
$q_($qq) printPolicerTable
```

which would be, in a 1-DS 1-BE case:

```
Policy Table(2):
Flow (0 to 3): TSW3CM policer, initial code point 10,
               CIR 500000.0 bps, PIR 750000.0 bytes.
Flow (1 to 3): TSW3CM policer, initial code point 0,
               CIR 0.0 bps, PIR 0.0 bytes.

Policer Table:
TSW3CM policer code point 10 is policed
     to yellow code point 11 and red code point 12.
TSW3CM policer code point 0 is policed
     to yellow code point 0 and red code point 0.

Policy Table(2):
Flow (3 to 0): TSW3CM policer, initial code point 10,
               CIR 500000.0 bps, PIR 750000.0 bytes.
Flow (3 to 1): TSW3CM policer, initial code point 0,
               CIR 0.0 bps, PIR 0.0 bytes.

Policer Table:
TSW3CM policer code point 10 is policed
     to yellow code point 11 and red code point 12.
TSW3CM policer code point 0 is policed
     to yellow code point 0 and red code point 0.
```

`flowvar {}`

This routine and `flowval` (see below) are coupled in the same way as ds_link's `appval` and `appvar` methods do. For completeness, a `flowudm` instproc should be implemented (which is not difficult to do) and ds_monitor's monitors should be enhanced accordingly. The return value is merely the string `"trg+eq"`, which is an actual shortkey for Target Rate Plus Equal Share of The Bottleneck's Excess Bandwidth.

`flowval {node}`

This method returns the value of the Target Equal Share, which is the sum of the Committed Target Rate (alias for Committed Information Rate) and the Equal Share, thus the share of the excess bandwidth equally distributed among all DiffServ and Best Effort flows; this literature parameter allows DS vs BE fairness comparison and DS, BE performances evaluation on a DiffServ-enhanced domain. The parameter `$node` is necessary and could be

- either `"$typestr,$x"` where `$typestr` is either `"ds"` or `"be"`, `$x` is an integer between 1 and `$nodes_($typestr)`.
- or `$y`, where `$s_($y)` is a valid array element, thus `$y` is an integer between 1 and `$nodes_(tot)`.

If you miss something at this point, go back to Sec. A.5 !

`flowstat {}`

This method reports some analysis of the current simulation characteristics, and it's an explicit overload of dummy virtual ds_monitor's method; the output example shown here is not intended to be meaningful, as we have only two sources:

```
# Flows    : (1 DS, 1 BE) = (50.000%, 50.000%)
# Allocated: 0.5 Mbps to DS, (10.0%)
# Total    : 5 Mbps
# Free     : 4.5 Mbps (90.0%)
# Eq.Share : 2.25 Mbps to each flow (90.0%)
```

`infobottleneck {}`

If the bottleneck have been properly setted up, `infobottleneck` and `infodomain` output would be something like:

```
# bOnK!  ds_bottleneck dom infos
#  _
# |_|  1 DiffServ
#    \
#     \                              +------+
#      +--[edge]--(core)--[edge]--| dest |
#     /  ingress          egress  +------+
#  _ /
# |_| 1 BestEffort
#
# Bottleneck Queues q_(...)  : CE1 E1C CE2 E2C
#              q_(CE1) : _o30
#              q_(E1C) : _o24
#              q_(CE2) : _o36
#              q_(E2C) : _o42
#
# Bottleneck Nodes  dn_(...) : eo dest core ei ingress
#              dn_(eo) :    _o22 : id 5
#            dn_(dest) :    _o18 : id 3
#            dn_(core) :    _o20 : id 4
#              dn_(ei) :    _o16 : id 2
#         dn_(ingress) :    _o16 : id 2
#

# dOOm! ds_domain dom infos
#
# Domain nodes :
# All   dn_(...): [eo dest core ei ingress]
# Edges dn_(e,...): [eo ei]
# Cores dn_(c,...): [core]
# Dests dn_(d,...): [dest]
#
# Domain queues :
# Queues q_(...) : [CE1 E1C CE2 E2C]
#
```

A.9 Script ds_example

Preliminary initialisation

```
#  ------------------------------------------------------
# /                                                      \
#/               ds_example.tcl             ___________/
#\_____________________________________________/.:nonsns:.
#
source "includes.tcl"
global ns rng
```

Just a word here, about the first line: Tcl's `source` mechanism is somewhat annoying unless you adopt the following stategy: you include every file you need only in your ns-script; as `source` rereads the file each time it is called, this may kill any object previously created if the object class is redefined. The `includes.tcl` file may contain the following lines, supposing that all the files are in `includes.tcl`'s directory:

```
source "list&array.tcl"
source "db_class.tcl"
source "db_merged.tcl"
source "ds_common.tcl"
source "ds_link.tcl"
source "ds_sources.tcl"
source "ds_domain.tcl"
source "ds_monitor.tcl"
source "ds_bottle.tcl"
```

Then, after defining a simple proc definition, let's go back to ds_example's initialization:

```
proc mod {n m} {
   while {$n>=$m} {
     set div [expr $n/$m]
     set n [expr $n-$m*$div]
   }
   return $n
}

#
# Scenario's const
#
set max_flow    20
set ds_num      15
set be_num      [expr $max_flow  - $ds_num]
set allocated   75   ;#%
set bottleneck  5.0  ;#Mbps
set PktSize     1500 ;#Bytes
set LinkBw      1    ;#Mbps
set Mflows      [lones $max_flow]
set Cir\
    expr ($allocated+0.0)*$bottleneck/(100.0*($ds_num+0.0))]
   #Pir = Cir
set RTT\
    [lscale {1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10} 10]
   #[RTT]=ms
```

ds_link Initialisation

First of all, it should be known that part of ds_link's objects initialization can be partially done at creation time, since Tcl interprets each "-str str1" pair (everything is a string in Tcl!) on the command line of a ds_link's instance declaration as a *–method args* pair (this is true also for -superclass!). It should be pointed out that lessof is a sort of < command (one of the miscellaneous functions described in Sec A.12)

```
foreach i [lessof $max_flow] {
   #
   # creating links
   #
   ds_link lnk($i)\
       -size $PktSize\
       -rtt  [lindex $RTT  $i]\
       -bw   $LinkBw
```

Inside the loop –where we are– the mod proc is used to produce –as promised– five Best Effort sources, whose initialization requires less methods calls than DiffServ ones:

```
   #
   #  5 BE links out of 20
   #
   if [mod $i 4] {
       lnk($i) type ds
       lnk($i) cir  $Cir
       lnk($i) pir  $Cir   ;# TSW2CM
   } else {
       lnk($i) type be
       #
       # DSCP_ , cir_ and pir_ will be set by type method
       #
   }
}
```

ds_bottleneck Initialisation

*

Step 1: setnodes
First of all, we absolutely need a bottleneck which properly sets (ds_domain) nodes, but not the queues yet; this also prepares ds_sources to fill up its source array by giving consistence to its $dest_ and $edge_ nodes:

```
#
# creating a bottleneck instance
#
ds_bottleneck bonk
bonk setnodes
```

Creating the bottleneck after each ds_link implies that source ids will start from 0 (rather than from the number of ns nodes otherwise created, which is not unconvenient).

*

Step 2: addlink
Each lnk but the first is then inserted in the bottleneck (*added* to the array from a ds_sources point of view). range (which is smartly *and* elegantly used here) is otherwise very useful, believe me.

```
foreach i [range 1 [expr $max_flow-1]] {
   #
   # adding all but last source to the bottleneck
   #
   bonk addlink lnk($i)

   foreach k [lessof [lindex $Mflows $i]] {
        #
        # adding app(s) to the source
        #
        lnk($i) app [list tcp_agent with_dummy_par]
   }
}
```

Here the innermost loop is not useful at all (since all the elements of $Mflows list are equal to unity –for a total of twenty), but its purpose is just to give as many ideas as possible in automatizing the building of a scenario.

*

Step 3: app
It is important to point out the fact that ds_link's application set up is possible only *after* having added that link to the bottleneck, because this step sets up ds_link's $edge_and $dest_, being ds_linkclass unaware up to now. The next line of code fills the gap of the first link's application. It shows how a CBR source can be initializated and can be seen as a reminder of the last point: it may be better to say things twice if they're important, isn't?

```
#
# note the order:  addlink lnk; lnk app
#
bonk addlink lnk(0) ; lnk(0) app [list udp_cbr_source 64Kb]
```

Anyway, app tries to match "udp*" or "tcp*" and no param is needed by tcp; we add a micro-flow to the aggregate of *ns* id 2 (which corresponds to ds_example's lnk(2)):

```
lnk(2) app [list tcp_agent_with_an_ftp_application\
                 just_to_test_microflows_feature]
```

*

Step 4: setqueues
Now it is time to set up all DiffServ parameters. This crucial step is entirely done by the following method:

```
#
#  now telling the bottleneck which are the sources
#  and setting up its queues.
#
#bonk setqueues quiet
bonk setqueues
```

Especially if you're not so confident with Tcl and OTcl classes, use and extend the following debug tools to your convenience; an example of their output can be found in each class `instproc`'s section. Remember to erase before each simulation the debug file produced by the `dbg` method (since these files are open to append data); its report can be useful in testing and implementation phases.

```
puts [bonk infoclass]
puts [bonk infobottleneck]
puts [bonk infomon]
puts [bonk infodomain]
puts [bonk infosources]
puts [bonk flowstat]
```

*

Step 5: `run`
The monitors can easily be set; using a shell script and command line parameter interpretation, one can appreciate the usefulness of a `suite`-like notation.

```
set suitedir    "example"
set suite       [list $suitedir\
                  "[set ds_num]DS_[set be_num]BE"\
                  "$allocated%_AS"\
                  "pir==cir"]

bonk  suite    $suite
bonk  samples  300
bonk  flowmon  {all}
bonk  queuemon E1C CE2
bonk  timemon
bonk  progress 5
```

Ready to run...

```
#
# Hi, Ho... Let's Go!
#

#bonk run
$ns run
```

*

Step 6: Output
If the we had used ds_bottleneck's run method, we wouldn't have seen the lines below:

```
Contents of scheduler queue (events) [cur time: 330.000000]---
t:330.043840 uid: 2188291 handler: 0x855e6e4
t:330.057840 uid: 2188295 handler: 0x8386be4
t:330.061440 uid: 2188288 handler: 0x851a56c
t:330.063040 uid: 2188289 handler: 0x854651c
t:330.089440 uid: 2188296 handler: 0x83db4a4
t:330.133440 uid: 2188290 handler: 0x8514ab4
t:330.148640 uid: 2188293 handler: 0x852adac
```

```
t:330.149040 uid: 2188294 handler: 0x853b62c
t:330.202240 uid: 2188297 handler: 0x8540dac
t:330.245040 uid: 2188292 handler: 0x851fe24
t:330.357440 uid: 2188298 handler: 0x855c5cc
t:330.412640 uid: 2188299 handler: 0x8556d5c
t:330.696240 uid: 2188300 handler: 0x85514bc
t:332.799120 uid: 2188190 handler: 0x84430cc
t:338.961040 uid: 2057812 handler: 0x8509b24
```

Listing `./example` directory, we find:

```
flows.ds.15DS_5BE.75%_AS.pir==cir
flows.all.15DS_5BE.75%_AS.pir==cir
flows.be.15DS_5BE.75%_AS.pir==cir
time.G.15DS_5BE.75%_AS.pir==cir
time.T.15DS_5BE.75%_AS.pir==cir
output.piped.with.tee
```

And file `flows.all.15DS_5BE.75%_AS.pir==cir` looks like...

```
#
#          Merged!    flowmonitor  {all}
#
trg+eq
#app_ bw_    cir_    id     nflows_ par_  pir_   rtt_    siz_      type_  Tput    Gput     Teq
#agentMbps   Mbps    $elf   ..      ..    Mbps   ms      bytes     ..
tcp   1      0.25    13     1       /     0.25   70.0    1500      ds     0.2920  0.28224  0.3125
tcp   1      0.25    1      1       /     0.25   10.0    1500      ds     0.3746  0.35259  0.3125
tcp   1      0.25    14     1       /     0.25   80.0    1500      ds     0.2758  0.26657  0.3125
arr2  1      0.25    2      2       arr2  0.25   20.0    1500      ds     0.4773  0.43914  0.3125
tcp   1      0.25    15     1       /     0.25   80.0    1500      ds     0.2739  0.26553  0.3125
tcp   1      0.25    3      1       /     0.25   20.0    1500      ds     0.3577  0.33861  0.3125
tcp   1      0.25    17     1       /     0.25   90.0    1500      ds     0.2709  0.26332  0.3125
tcp   1      0.25    5      1       /     0.25   30.0    1500      ds     0.3388  0.32142  0.3125
tcp   1      0.25    18     1       /     0.25   100.0   1500      ds     0.2628  0.25607  0.3125
tcp   1      0.25    6      1       /     0.25   40.0    1500      ds     0.3222  0.30807  0.3125
tcp   1      0.25    19     1       /     0.25   100.0   1500      ds     0.2627  0.25581  0.3125
tcp   1      0.25    7      1       /     0.25   40.0    1500      ds     0.3215  0.30785  0.3125
tcp   1      0.25    9      1       /     0.25   50.0    1500      ds     0.3030  0.29028  0.3125
tcp   1      0.25    10     1       /     0.25   60.0    1500      ds     0.2909  0.28022  0.3125
tcp   1      0.25    11     1       /     0.25   60.0    1500      ds     0.2948  0.28396  0.3125
tcp   1      0       4      1       /     0      30.0    1500      be     0.0552  0.04497  0.0625
tcp   1      0       8      1       /     0      50.0    1500      be     0.0854  0.07069  0.0625
tcp   1      0       12     1       /     0      70.0    1500      be     0.0815  0.06818  0.0625
tcp   1      0       16     1       /     0      90.0    1500      be     0.0617  0.05192  0.0625
udp   1      0       0      1       64Kb  0      10.0    1500      be     0.0726  0.06276  0.0625
#Analisi: ds   be    all
#SUM
#-    15.000 3.7500  150.0  16.000  -     3.750  850.00  22500.00  -      4.7195  4.51173  4.6875
#-    5.0000 0.0000  40.00  5.0000  -     0.000  250.00  7500.000  -      0.3365  0.31854  0.3125
#-    20.000 3.7500  190.0  21.000  -     3.750  1100.0  30000.00  -      5.0561  4.83028  5.0000
#AVG
#-    1.0000 0.2500  10.00  1.0666  -     0.250  56.666  1500.000  -      0.3146  0.30078  0.3125
#-    1.0000 0.0000  8.000  1.0000  -     0.000  50.000  1500.000  -      0.0673  0.06370  0.0625
#-    1.0000 0.1875  9.500  1.0500  -     0.187  55.000  1500.000  -      0.2528  0.24151  0.2500
#VAR
#-    0.0000 0.0000  32.66  0.0622  -     0.000  822.22  0.000000  -      0.0029  0.00218  0.0000
#-    0.0000 0.0000  32.00  0.0000  -     0.000  800.00  0.000000  -      0.0001  0.00018  0.0000
#-    0.0000 0.0117  33.25  0.0475  -     0.011  825.00  0.000000  -      0.0137  0.01222  0.0117
#SDEV
#-    0.0000 0.0000  5.715  0.2494  -     0.000  28.674  0.000000  -      0.0545  0.04675  0.0000
#-    0.0000 0.0000  5.656  0.0000  -     0.000  28.284  0.000000  -      0.0135  0.01357  0.0000
#-    0.0000 0.1082  5.766  0.2179  -     0.108  28.722  0.000000  -      0.1172  0.11056  0.1082
```

...surely lighter that ns trace-files. `time.?.15DS_5BE.75%_AS.pir==cir` files are ready to GNU-plot, and we can parse (maybe with `db_class`,`db_class`, maybe with the Pathologically Eclettic Rubbish Lister *perl*) the listed output.

The Tcl Script

```
#  ------------------------------------------------------
# /                                                      \
#/               ds_example.tcl             ___________/
#\_________________________________________/.:nonsns:.
#
source "includes.tcl"
global ns rng

proc mod {n m} {
   while {$n>=$m} {
     set div [expr $n/$m]
     set n [expr $n-$m*$div]
   }
   return $n
}

#
# Scenario's const
#
set max_flow    20
set ds_num      15
set be_num      [expr $max_flow  - $ds_num]
set allocated   75   ;#%
set bottleneck  5.0  ;#Mbps
set PktSize     1500 ;#Bytes
set LinkBw      1    ;#Mbps
set Mflows      [lones $max_flow]
set Cir\
    expr ($allocated+0.0)*$bottleneck/(100.0*($ds_num+0.0))]
   #Pir = Cir
set RTT\
    [lscale {1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10} 10]
   #[RTT]=ms
foreach i [lessof $max_flow] {
   #
   # creating links
   #
   ds_link lnk($i)\
       -size $PktSize\
       -rtt  [lindex $RTT  $i]\
       -bw   $LinkBw

   #
   #  5 BE links out of 20
   #
   if [mod $i 4] {
       lnk($i) type ds
       lnk($i) cir  $Cir
       lnk($i) pir  $Cir
   } else {
       lnk($i) type be
       #
       # DSCP_ , cir_ and pir_ will be set by type
       #
   }
}
#
```

```
# creating a bottleneck instance
#
ds_bottleneck bonk
bonk setnodes

foreach i [range 1 [expr $max_flow-1]] {
   #
   # adding all but last source to the bottleneck
   #
   bonk addlink lnk($i)

   foreach k [lessof [lindex $Mflows $i]] {
        #
        # adding app(s) to the source
        #
        lnk($i) app [list tcp_agent with_dummy_par]
   }
}
#
# note the order:  addlink lnk; lnk app
#
bonk addlink lnk(0) ; lnk(0) app [list udp_cbr_source 64Kb]
#
#  now telling the bottleneck which are the sources
#  and setting up its queues.
#
#bonk setqueues quiet
bonk setqueues
puts [bonk infoclass]

puts [bonk infobottleneck]
puts [bonk infomon]
puts [bonk infodomain]
puts [bonk infosources]
puts [bonk flowstat]

set suitedir    "example"
set suite       [list $suitedir\
                  "[set ds_num]DS_[set be_num]BE"\
                  "$allocated%_AS"\
                  "pir==cir"]

bonk  suite    $suite
bonk  samples  300
bonk  flowmon  {all}
bonk  queuemon E1C CE2
bonk  timemon
bonk  progress 5

#
# Hi, Ho... Let's Go!
#

#bonk run
$ns run
```

A.10 Class db_class

heritage: Object

instvars: nC nR RC tail_ head_ prec_

instprocs: avgR parsequery avgC fwrite tail getRStr addR sortC Reanalyze sdevR addComment addC init dimR analyze precision sdevC allNumbers dimC zaptail struct LinearRegressionR sortbyC sumR setC sumC parsecol getR addStr head fread varR calC getC varC puts query

This simple class can handle bidimensional tables, with a special regard to columns. Moreover, it has specialized functions that speed-up the manipulation of the table, such as parsing of literal expressions and the ability to automacally compute sum, average, variance and standard deviation; querying the database is implemented with literal parsing. The insertion of comments inside the database is handled by the introduction of the special character #, and comments can span over multiple lines. The scheme presented in Tab. A.1 introduces some of the keywords of this class in parallel to the logical structure of the data, described in Sec. A.10; the instprocs of the class, grouped in five different categories, are described in Sec A.10; finally, examples of db_class use are presented in Sec A.10.

```
              0          1           2            3     ...  $nC

$head_    "#comment string\n#spanning over several lines"
 0        $RC(0,0)    $RC(0,1)    $RC(0,2)     $RC(0,3)
0,str     "#another\n#boring\n#comment string"
 1        $RC(1,0)    $RC(1,1)    $RC(1,2)     $RC(1,3)
 2        $RC(2,0)    $RC(2,1)    $RC(2,2)     $RC(2,3) ...
2,str     "#YEEKS! Yet e-nother e-nnoying Komment String"
 3        $RC(3,0)    $RC(3,1)    $RC(3,2)     $RC(3,3)
 .
 .
$nR                          ...
$tail_
```

Table A.1: An introductory scheme for the db_class

Class instvars

`Class db_class -superclass`
This class derives directly from OTcl `Object` class.

`nC_`
Maximum value that a valid column index can assume, that is the number of columns decremented by one.

`nR_`
Maximum value that a valid row index can assume, that is the number of rows minus one.

`RC`
It is a hash table which looks like a particular zero-based, bidimensional array. if `$x` in `[0..$nR]` and `$y` in `[0..$nC]` then `$RC($x,$y)` is the *cell* value corresponding to row `$x` and column `$y`; every cell can contain either numbers or strings (avoid special characters). If some lines of text comment have been inserted between rows `$x` and `$x+1`, then `$RC($x,str)`

contains that text. Conventionally, every text line in the database file will be preceded by the # character; the reasons for this choice is that GNUplot skips such lines.

`prec_`
Set to 6 by default, this is the amount of decimal digits with which (almost) all numbers of the database are formatted on output. Anyway, it only affects the output: all calculations on data will be done with the available precision. The external routine (which isn't a class instproc) `apply_prec` can discriminate between strings, integers and reals. In the latter case, the format is applied only if the number of decimals of that real number exceeds `prec_` thus applying the precision limit only if it makes sense.

`head_`
The database data not only support comments anchored to particular rows, but treats especially the very first (as well as the least, see below) comment line, using the `head_` variable to address it.

`tail_`
Last comment line of the database. It is used by methods such as `analyze` and `Reanalyze` to practically extract and report some per-column information of the database (sum, average, variance and standard deviation).

Class instprocs

General Purpose

`init {}`
Defaults `head_` and `tail_` to # a blank comment line, six digits of decimal precision, `$nR` and `$nC` to -1 (remember that `RC` array is zero-based).

`INT = dimR {}`
Returns the vertical dimension, that is the number of rows, thus `$nR+1`.

`INT = dimC {}`
Returns the horizontal dimension, that is the number of columns, thus `$nC+1`.

`precision {val}`
Sets the number of decimal precision digits to `$val`.

`head {str}`
Appends `"\n#$str"` to `head_`

`tail {str}`
Appends `"\n#$str"` to `tail_`

`zaptail {}`
To reset the tail line means to zap all comment lines; this may be useful to re-analyze the database after some modifications.

`STR=struct {}`
Useful for debug purposes, it does returns the sorted list of every cell of the database, including possible comment line indexes.

`puts {}`
This is really self-explainatory

`analyze {}`
This method adds some information about the database columns to the tail: adding a row for each of the following functions: sum, average, variance and standard deviation; each element of that row is evaluated on each column of the database. See Sec. A.10 for examples of its use.

`Reanalyze {}`
It works as the previous one, but pre-emptying the tail string.

Input/Output

`fwrite {fname}`
A database can be flushed on a textfile by the means of this command, which requires only the filename to be passed. The structure of the database, following the scheme given in Tab. A.1, is then reproduced on the file. This is the only method which calls the `apply_prec` routine (briefly described in `instvar prec_` paragraph) to format its output according to the user's needs. Its output is piped through the shell command `column -t` to quickly order the output.

`fread {fname}`
The input side is a little bit more complicated than the output, since the method has to recognize the right item (head, cells, comments or tail) to be able to put it in the right place. It will certainly work on files produced by `fwrite`; it will work on simple hand-made files too, but its robustness is not guaranteed for any larger and exotic hand-made file.

Column Oriented Commands

`LIST = getC {col}`
Returns a list containing all elements of column `$col`. It does not perform any error checking.

`addC {dataL}`
Adds a column of data to the database, incrementing `$nC`. The parameter `dataL` must be a list of congruent dimensions, since no error-checking is performed: in fact

- if the data list length is shorter than what it should be, a run-time error is returned
- if otherwise the length is greater, then all the elements after the (`$nR+1`)-th will be ignored.
- whether the database is empty (thus `$nR` $\equiv$ `$nC` $\equiv -1$) the data is entirely added, and both dimensional variables are updated: `$nC++` (unfortunately in Tcl $\neg\exists$`++` operator) and `set nR [llength $dataL]`

`LIST = calC {fmla}`
This method performs simple parsing of the expression string `$fmla` and returns a list of the results. It is intended to work in column mode, and its parsing is quite simple: it just substitutes the *keywords* `#1,#2,...` (imperatively sourrounded by extra spaces), with the proper RC reference, then passing to Tcl's `[expr ...]` command. In further detail, a loop over all row indexes is implemented in `calC` with the *conventional* variable `row_idx`. Then the instproc `parsecol` is called: here the string is split on blanks and all the expressions `"#%d"` are sequentially examined; the integer after the # will be copied to `$col` and `"#$col"` expression is substituted with `"$RC{$row_idx,$col}`. Once all the substitutions are done, the control is passed back to the `calC` routine, which evaluates the `[expr ...]` and appends the result to the return list. Extra spaces around columns identifiers are necessary, which leads to writing expressions like `"( #1 *5+ ( #13 - #2 )/ #12 )"` or `"sqrt(abs( #1 - tanh #5 ))"`: remember that mathematical limitations are due to Tcl, but you can always provide your own routine.

`setC {col fmla}`
This method calls the previous one to evaluate the expression given as `$fmla`, column by column, putting the result list in column `$col` of the database; the method only allows the substitution of existing columns: all attempts to use `setC` with an index `$col > $nC` will be ignored.

`(0|1) = allNumbers {col}`
Test whether all the database elements of column `$col` are valid numbers (returning 1) or not (returning 0)

`STR = sumC, avgC, varC, sdevC {col}`
Each of these four functions (xxxC methods), respectively representing the sum, the average, the variance and its square root, preliminary calls `allNumbers` to ensure meaningfulness to the operation. On success, a string representing the result (formatted with precision `prec_`) of that operation evaluated on column `$col` is returned; if any of the cells of column `$col` is not a valid number, then the minus character `"-"` is returned

Row Oriented Commands

`addComment {row string}`
This sets `RC($row,str)` to `$string`, meaning that the comment string is not appended (thus replacing the old one if existent)

`addR {dataL}`
Adds a row to the database, incrementing `$nR`. No error check is performed, meaning that the list `dataL` must have proper dimensions in order to avoid unpredictable results.

`LIST = getR {row}`
Returns the list of elements belonging to any columns and to row `$row`.

`STR = getRStr {row}`
Returns, as a string, the list of elements belonging to any columns and to row `$row`, possibly appending to it the comment string `$RC($row,str)` if existent.

`STR = sumR,avgR,varR,sdevR {}`
Each of this xxxR functions iterates over all columns the correspondent xxxC method concatenating each result, and finally returns the string.

`STR = LinearRegressionR {x y}`
A simple linear regression is performed using data of column `$x` as independent set and column `$x` elements as the dependant values. A string containing a report is then returned.

Sorting and Querying

`sortbyC {col}`
This methods modifies the row order of the database by filtering it through the shell command `sort -k` around column `$col`. It produces a temporary hidden file `".sorted"`.

`STR = query {fmla}`
This method allows the user to query the database, employing the same columnar convention as those used for calculations: a number preceded by # character is to be intended as a reference to an element of that column number. The substitutions (done by `parsequery`) works as before, but here the expression is tested in a boolean context to decide whether or not each row matches the request. Since it isn't possible for a routine to return an object (nor a reference to an object), `query` returns the name of the hidden file which contains the results of the query. `"( #1 < #3 ) && ( #4 * #5 < 0)"` is an example of a valid `fmla`.

db_class examples

Both fragments of code below yield the same result: a 3x2 database filled by zeros, which we may later use coupled with onesXY to explain db_merged class. Let us suppose to be in an OTcl shell (where % is the prompt symbol); the `instproc puts` will discard any comment, head and tail, unless otherwise specified.

```
% db_class zero32
% zero32 addC {0 0 0}
% zero32 addC [list 0 0 0]
% zero32 puts
0 0
0 0
0 0
```

```
% db_class zero3_2
% zero3_2 addR {0 0}
% zero3_2 addR {0 0}
% zero3_2 addR {0 0}
% zero3_2 puts
0 0
0 0
0 0
```

If from any output we gathered the set of data reproduced below, we can, first of all, define a db_class instance. Notice that Tcl's `expr 3/2` yields 1 but neither `expr 3.0/2` nor `expr 3/2.0` yield 1. So the two portions of code below don't yield two equivalent db_class instances

```
% db_class a
% a addR {1.0 2.0 3.0 6.0}
% a addR {1.0 0.0 0.0 1.0}
% a addR {1.0 2.0 2.0 2.0}
```

```
% db_class ai
% ai addR {1 2 3 6}
% ai addR {1 0 0 1}
% ai addR {1 2 2 2}
```

if we want to add a column which reports interesting calculations on our database, we may use something like

```
% a addC [a calC "( #0 + #1 )/ #3"]
% a puts
1.0   2.0    3.0    6.0    0.500000
1.0   0.0    0.0    1.0    1.000000
1.0   2.0    2.0    2.0    1.500000
```

If our aim is to substitute an existing column performing any calculation on its (and others columns') data, we should use `setC` instead; in the example below we added a dummy sum in the expression in order to make Tcl work with real numbers (anyway this is not a robust way to build expressions: use real numbers instead).

```
% ai setC 3 "( #0 + #1 )/( #3 +0.0)"]
```

Let us degrade the precision and sort the database around the third column, that is column number 2. The precision is applied only to calculated columns unless the database is saved.

```
% a precision 3
% a sortbyC 2
% a puts
1.0   0.0    0.0    1.0    1.000
1.0   2.0    2.0    2.0    1.500
1.0   2.0    3.0    6.0    0.500
```

Now if you need to know the sum, average, variance and standard deviation of each column, just use `analyze` or `Reanalyze`. The output of `a puts` is here (and only here!) complete

```
% a analyze
% a head "The head will rest"
% a tail "but this will disappear"
% a Reanalyze
% a puts
#
#The head will rest
#
```

```
1.0 0.0 0.0 1.0 1.000
1.0 2.0 2.0 2.0 1.500
1.0 2.0 3.0 6.0 0.500
#
#SUM:
#3.000 4.000 5.000 9.000 3.000
#AVG:
#1.000 1.333 1.667 3.000 1.000
#VAR:
#0.000 0.889 1.556 4.667 0.167
#SDEV:
#0.000 0.943 1.247 2.160 0.408
```

Now for the mechanism of the query. A query cannot –as already said– return an object, and I do not like discarding data that may still be useful later (anyway, this is always possible: the second query mechanism is a way of consciously substituting data). That's the reason why a query returns a filename (which in fact, as is shown in the first query, will always be the same: `".query"`). So, going back to the short version of the `puts` instproc we have

```
% db_class b
% a query " #1 > 0 "
% b fread ".query"; b puts
1.0 2.0 2.0 2.0 1.500
1.0 2.0 3.0 6.0 0.500

% b fread [b query " #1 > 0 && #3 > #1"]
% b puts
1.0 2.0 3.0 6.0 0.500
```

Finally, we can store everuthing in a file; its output is slightly different, since it is formatted applying the specified precision and then aligning the columns. An example of it will be shown later at the end of Sec. A.11

```
% a fwrite "to_my_file"
```

A.11 Class db_merged

heritage: db_class Object

instvars: nC nR RC tail_ head_ prec_ mrglist_

instprocs: dimAgreeR init mAnalize dimAgreeC mAsSum mByR dimAgree mAsAvg mByC

This extension of the `db_class` provides the ability to merge two or more database objects. Suppose that a farmer would like to keep two separate databases of its products: `fresh_fruits` and `rotten_vegetables`. These databases are similar in the fact that each row represent a different item (from apples to zinnias): he may put the item's name in the first column, the produced quantity in the second, the price in the 3rd and so on. He then may want to know how much he earned from the fruits sale, from the vegetables sale (maybe not so much, in view of their quality), and his overall earnings. I would suggest that he uses a `db_merged` class for this purpose.

Class heritage and instvars

`Class db_merged -superclass db_class`
Refers to `db_class` to know its basic instvars and instprocs

`mrglist_`
This keeps the list of merged databases, indicated by their names, that is `fresh_fruits` and `rotten_vegetables`. Note that they must be existent `db_class` or even `db_merged` instances.

Class instprocs

`init {args}`
This sets the merging list to an empty list;

`(0|1) = dimAgreeC {lst}`
If all database objects in the list have equal numbers of columns the method returns 1, 0 otherwise. The `lst` parameter has access to a list of valid `db_class` or `db_merged` object instances names; this will also hold for every other method of the class.

`(0|1) = dimAgreeR {lst}`
If all database objects in the list have equal number of rows the method returns 1, 0 otherwise.

`(0|1) = dimAgree {lst}`
Verify if all database object in the list have identical rows and columnar dimensions. In that case, the set is said *homodimensional* and the function returns 1.

`mByR {lst}`
Produces a new database by sequentially appending each row of each database following the list order; `lst` elements must satisfy `dimAgreeC`. The `lst` parameter is stored in `mrglist_` instvar (like the next three routines `mByC`, `mAsSum`, `mAsAvg`)

`mByC {lst}`
Produces a new database by sequentially appending each column of each database following the list order; `lst` elements must satisfy `dimAgreeR`.

`mAsSum {lst}`
This method produces a database in which each element is the sum of the corresponding elements of each database of the set, which has to satisfy `dimAgree`, so as to be a homodimensional set. For clarity, it can be considered as an implementation of the matricial sum with two (or more) addenda.

`mAsAvg {lst}`
: This method produces a database in which each element is the average of the corresponding elements of each database of the set, which has to satisfy `dimAgree`, so as to be a homodimensional set.

`mAnalize {}`
: This method calls `Analize` for each database involved in the creation list, appending to those results the analysis of the calling database: it does something more than what a simple call to `Analize` would.

db_merged Examples

Working with already known example objects, the result of

```
dimAgree  {zero23 ones33}
dimAgreeR {zero23 ones33}
dimAgreeC {zero23 ones33}
```

are respectively (and obviously) 0, 0 and 1, allowing in the latter case to create an object `ex1` merging those two db_class instances by rows, then another object `ex2` merging those others db_class instances by column, and finally merge the newly created objects. The following lines of code

```
db_merged ex1; ex1 mByR {zero23 ones33}
db_merged ex2; ex2 mByR {zero51 ones51 zero51}
db_merged ex3; ex3 mByC {ex1 ex2}
```

produce the results

```
0  0  0        0  1  0        0  0  0  0  1  0
0  0  0        0  1  0        0  0  0  0  1  0
1  1  1        0  1  0        1  1  1  0  1  0
1  1  1        0  1  0        1  1  1  0  1  0
1  1  1        0  1  0        1  1  1  0  1  0

  ex1             ex2               ex3
```

When dealing with a set of homodimensional databases, it is possible to sum or average each object element over all databases of the set, thus

```
db_merged ex4; ex4 mAsSum {ex1 ex2 ex2}
db_merged ex5; ex5 mAsAvg {ex1 ex2}
```

will yield

```
0  2  0        0    0.5  0
0  2  0        0    0.5  0
1  3  1        0.5  1    0.5
1  3  1        0.5  1    0.5
1  3  1        0.5  1    0.5

  ex4               ex5
```

Finally, adding a head to the database, appending analysis and saving can be done with the commands

```
ex5 head "This is db_merged ex5"
ex5 mAnalize
ex5 fwrite "ex5.dbm"
```

Notice that every space, even those in head and tail are expanded by the shell pipe command `column -t`.

```
#This      is        db_merged  ex5
0.0        0.5       0.0
0.0        0.5       0.0
0.5        1.0       0.5
0.5        1.0       0.5
0.5        1.0       0.5
#Analisi:  ex1       ex2        ex5
#SUM
#3.000000  3.000000  3.000000
#0.000000  5.000000  0.000000
#1.500000  4.000000  1.500000
#AVG
#0.600000  0.600000  0.600000
#0.000000  1.000000  0.000000
#0.300000  0.800000  0.300000
#VAR
#0.240000  0.240000  0.240000
#0.000000  0.000000  0.000000
#0.060000  0.060000  0.060000
#SDEV
#0.489898  0.489898  0.489898
#0.000000  0.000000  0.000000
#0.244949  0.244949  0.244949
```

A.12 List procs

The file `list&array.tcl` contains some Tcl functions useful to manipulate lists. They will be merely listed here, since their purpose is quite often self-explainatory; for clarity purpose and reference they have been classified in several subsections depending on the parameter type.

`REAL = function( LIST )`

`lsum {lst}`
: Returns the sum of the elements of the input list `$lst`

`lavg {lst}`
: Returns the average of the elements of the input list `$lst`

`lvar {lst}`
: Returns the variance of the elements of the input list `$lst`

`lsdev {lst}`
: Returns the standard deviation of the elements of the input list `$lst`

`REAL = function( LIST )`

`lones {dim}`
: Creates a list containing `$dim` ones

`lzeros {dim}`
: Creates a list containing `$dim` zeros

`LIST = function( LIST )`

`lneg {lst}`
: Returns `-$lst`

`linv {lst}`
: Returns $1/$`$lst`

`lsqr {lst}`
: Returns `$lst` ^{2}

`lsqrt {lst}`
: Returns `$lst` $^{1/2}$

`LIST = function( LIST, REAL )`

`lminus {lst sub}`
: Subtracts `$sub` from each element of `$lst` and returns that list

`lplus {lst const}`
: Adds `$coeff` to each element of `$lst` and returns that list

`lscale {lst coeff}`
: Multiplies each element of `$lst` by `$coeff` and returns that list

`LIST = function( LIST, LIST )`

`lmult {lst1 lst2}`
Muliplies each element of `$lst1` by the correspondent of `$lst2`

`ldiv {lst1 lst2}`
Divides each element of `$lst1` by the correspondent of `$lst2`

`ladd {lst1 lst2}`
Adds each element of `$lst1` to the correspondent of `$lst2`

`lsub {lst1 lst2}`
Subtracts from each element of `$lst1` the correspondent of `$lst2`

`ldiff {list1 list2}`
Returns a list whose elements belong to `$lst1` but not to `$lst2`

Miscellaneous

`makearray {Lin Aout}`
The parameter `Aout` is the name of the zero-based array that will be created from the values of the list `$Lin`: i.e., `makearray {1 2 3} a` implies that `$a(1)==2`

`LIST = splitblank {string}`
Analogous to Tcl's `split`, this routine's peculiarity is to treat several consecutive blanks (spaces and tabs) as a single one and returns the list of tokens

`LIST = lnotmatch {lst exp}`
Returns elements of list `$lst` that do not match string `$exp`

`LIST = lmatch {lst exp}`
Returns elements of list `$lst` that match string `$exp`

`STR = fmtCol {str}`
Returns a string (possibly containing newlines) formatted by the pipe `echo $str | column -t`

`LIST = range {min max}`

Returns a list containing all the integers between (and including) `$min` and `$max`. This is the first of the three *lazy typist* procs. The following two lines of code are equivalent:

```
foreach k [range $min $max] { ... }
for {set k $min} {$k<=$max} {incr k} { ... }
```

`LIST = lessof {max}`
This is the second one: The Less You Type, The Better You Feel. The following two lines...

```
foreach k [lessof $max] { ... }
for {set k 0} {$k < $max} {incr k} { ... }
```

`LIST = upto {max}`
Third. *Frustra fit per plura quod fieri potest per pauciora.* (W.Hockham)

```
foreach k [upto $max] { ... }
for {set k 0} {$k <= $max} {incr k} { ... }
```

A.13 The Classed Classes

ds_common -superclass Object

instprocs	init	arraynames	destroy	infoclass	dbg	
instvars	nodes_	avg_	time_	debug_		

ds_link -superclass ds_common

instprocs	almost_all	init	type	size	cir	pir	edge
	dest	bw	rtt	app	appval	appvar	linkvar
	linkval	linkudm	infolink				
instvars	nodes_	nflows_	avg_	time_	debug_	nd	id
	rtt_	siz_	type_	DSCP_	cir_	pir_	bw_
	edge_	e_id	dest_	did_	tcp_	ftp_	udp_
	cbr_	sink_	app_	par_			

ds_sources -superclass ds_common

instprocs	infosources	setdest	init	fwrite	setedge	fread	addlink
instvars	nodes_	avg_	time_	debug_	s_		

ds_domain -superclass ds_sources

instprocs	infodomain	init	queue				
instvars	edge_	nodes_	avg_	dn_	prefix_	time_	debug_
	dest_	q_					

ds_monitor -superclass ds_domain

instprocs	flush_time	timemon	suite	flowmon	init	samples	progress
	done	infomon	flush_queue	flush_flow	flowstat	queuemon	run
instvars	edge_	nodes_	infostr_	avg_	dn_	prefix_	time_
	debug_	nsamples_	dest_	s_	timemon_	flowmon_	suite_
	CWND_	smpcoeff_	quiet_	q_			

ds_bottleneck -superclass ds_monitor

instprocs	flowstat	setqueues	init	flowvar	setnodes	flowval	infob [a]
instvars	edge_	nodes_	infostr_	avg_	dn_	prefix_	time_
	debug_	nsamples_	dest_	s_	timemon_	flowmon_	suite_
	CWND_	smpcoeff_	quiet_	q_	bwneck		

db_class -superclass Object

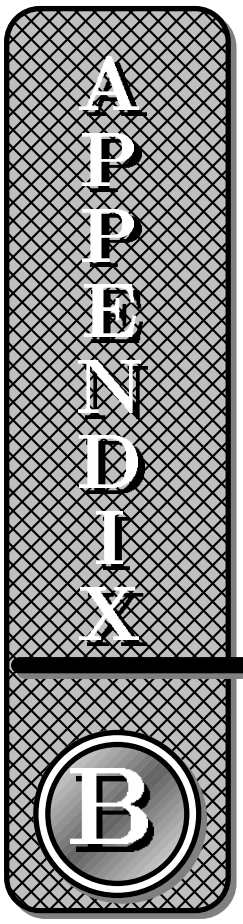
instvars	nC	nR	RC	tail_	head_	prec_	
instprocs	init	Analize	ReAnalize	sortbyC	query	fread	fwrite
	head	tail	zaptail	getRStr	addStr	addComment	LinRegR [b]
	dimR	addR	getR	avgR	sdevR	sumR	varR
	dimC	addC	getC	avgC	sdevC	sumC	varC
	setC	calC	allNumbers	parsecol	parsequery	puts	struct
	precision						

db_merged -superclass db_class

instvars	nC	nR	RC	tail_	head_	prec_	mrglist_
instprocs	dimAgreeR	init	mAnalize	dimAgreeC	mAsSum	mByR	dimAgree
	mAsAvg	mByC					

[a] Its complete name is `infobottleneck`, which was too long to fit in this table

[b] Here too `LinearRegressionR` was too long



Simulation Results

Long Lived FTP Flows

B.1 Target Rate Suite

TRG @ ALLOC		Target Rate			DS Overall		Link
		$\sim(\mu-\sigma)$	$\sim\mu$	$\sim(\mu+\sigma)$	μ	σ	Σ
5 @ 50 %	Trg	0.278	0.463	0.741	0.500	0.302	2.500
	Thr	0.383	0.528	0.753	0.555	0.250	4.314
	Thr/Trg	1.378	1.141	1.017	1.346	0.458	1.726
	Thr/Tfs	0.689	0.570	0.509	0.673	0.229	*0.863*
10 @ 50 %	Trg	0.136	0.273	0.364	0.250	0.131	2.500
	Thr	0.237	0.379	0.453	0.345	0.121	4.390
	Thr/Trg	1.735	1.389	1.246	1.623	0.529	1.756
	Thr/Tfs	0.868	0.695	0.623	0.812	0.265	*0.878*
15 @ 50 %	Trg	0.090	0.181	0.241	0.167	0.082	2.500
	Thr	0.203	0.291	0.344	0.267	0.082	4.480
	Thr/Trg	2.251	1.609	1.429	1.890	0.682	1.792
	Thr/Tfs	1.125	0.805	0.715	0.945	0.341	*0.896*

Table B.1: Target Rate: DS Performance, $\rho_{DS} = 50\%$

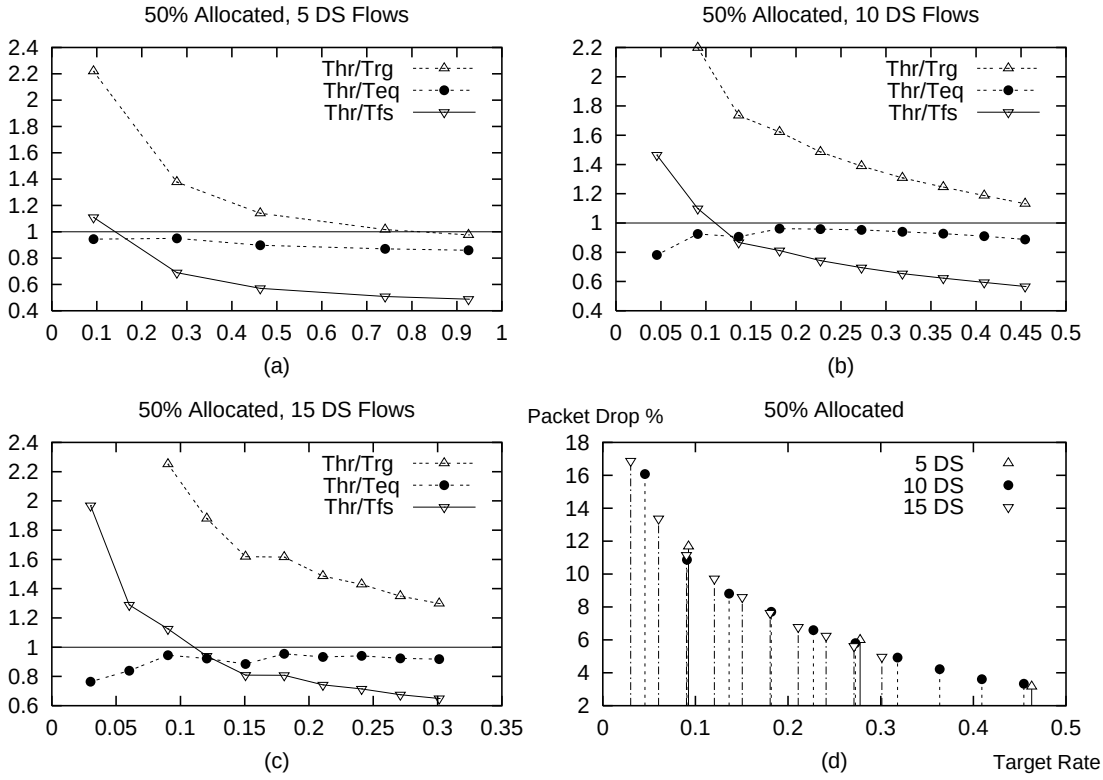


Figure B.1: TARGET: Same Allocation

TRG @ FLOW		Target Rate			DS Overall		Link
		$\sim(\mu-\sigma)$	$\sim\mu$	$\sim(\mu+\sigma)$	μ	σ	Σ
10 @ 25 %	Trg	0.068	0.114	0.182	0.125	0.065	1.250
	Thr	0.210	0.258	0.320	0.273	0.063	4.144
	Thr/Trg	3.073	2.267	1.762	2.943	1.819	3.315
	Thr/Tfs	0.768	0.567	0.441	0.736	0.455	*0.829*
10 @ 50 %	Trg	0.136	0.273	0.364	0.250	0.131	2.500
	Thr	0.237	0.379	0.453	0.345	0.121	4.390
	Thr/Trg	1.735	1.389	1.246	1.623	0.529	1.756
	Thr/Tfs	0.868	0.695	0.623	0.812	0.265	*0.878*
10 @ 75 %	Trg	0.205	0.409	0.545	0.375	0.196	3.750
	Thr	0.272	0.452	0.574	0.419	0.173	4.690
	Thr/Trg	1.331	1.105	1.053	1.252	0.306	1.251
	Thr/Tfs	0.998	0.829	0.789	0.939	0.230	*0.938*

Table B.2: Target Rate: DS Performance, $N_{DS} = 10$

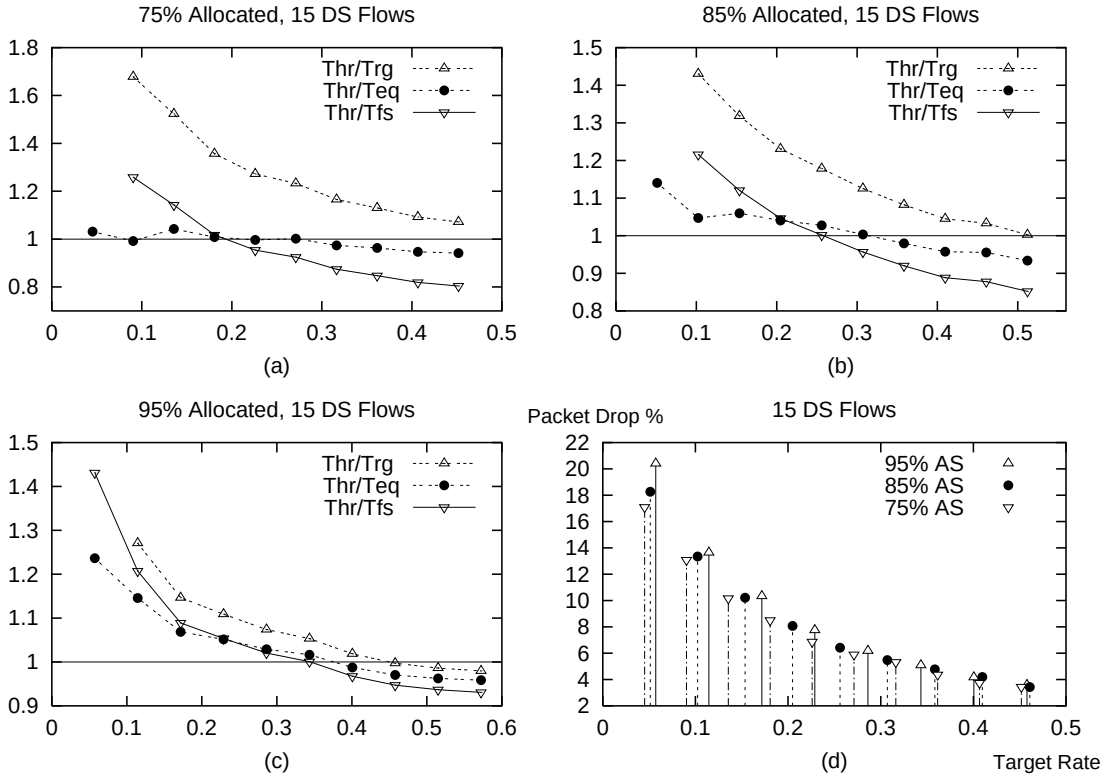


Figure B.2: TARGET: High Flow Number

TRG @ HIGH-f		Target Rate			DS Overall		Link
		$\sim(\mu-\sigma)$	$\sim\mu$	$\sim(\mu+\sigma)$	μ	σ	Σ
15 @ 75 %	Trg	0.136	0.271	0.361	0.250	0.123	3.750
	Thr	0.206	0.333	0.408	0.306	0.113	4.766
	Thr/Trg	1.523	1.228	1.130	1.367	0.349	1.271
	Thr/Tfs	1.142	0.921	0.847	1.025	0.262	*0.953*
15 @ 85 %	Trg	0.154	0.307	0.410	0.283	0.140	4.250
	Thr	0.202	0.344	0.428	0.318	0.126	4.856
	Thr/Trg	1.318	1.121	1.045	1.222	0.241	1.143
	Thr/Tfs	1.121	0.953	0.888	1.038	0.205	*0.971*
15 @ 95 %	Trg	0.172	0.343	0.458	0.317	0.156	4.750
	Thr	0.197	0.364	0.456	0.331	0.142	4.978
	Thr/Trg	1.146	1.059	0.997	1.105	0.141	1.048
	Thr/Tfs	1.089	1.006	0.947	1.050	0.134	*0.996*

Table B.3: Target Rate: DS Performance at High Reserved Bandwidth, $N_{DS} = 15$

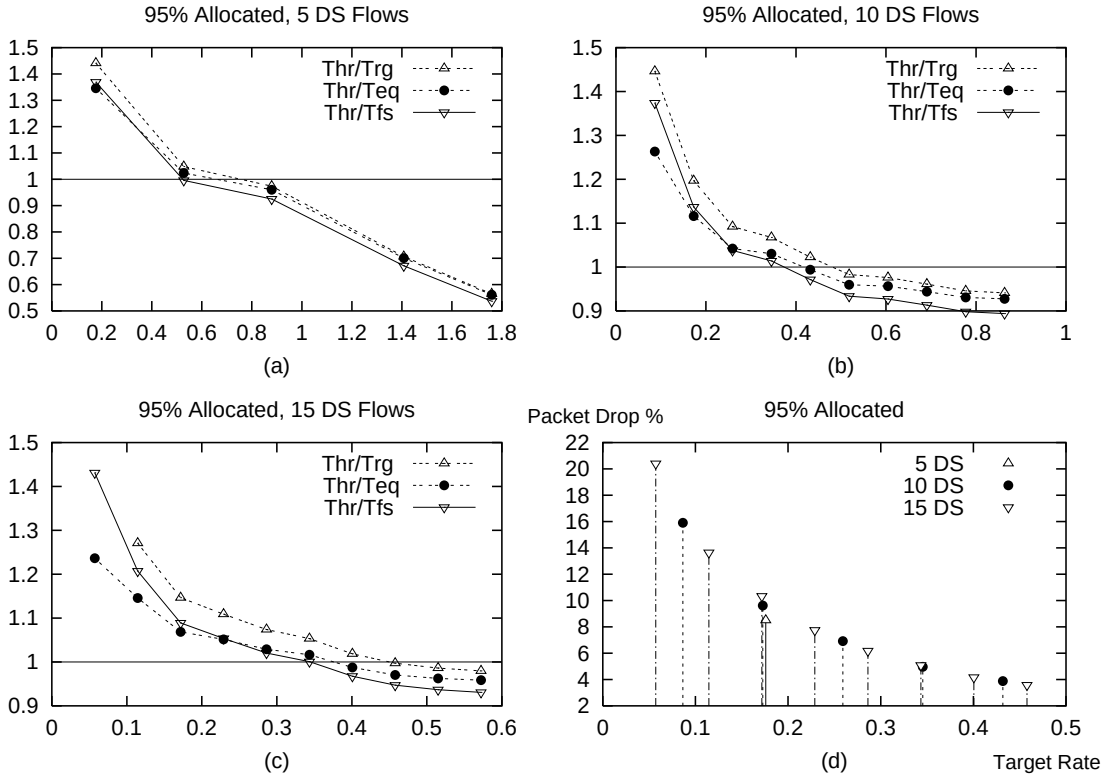


Figure B.3: TARGET: High Allocation

TRG @ CIR		5 @ 25 %		10 @ 50 %		15 @ 75 %	
		DS	BE	DS	BE	DS	BE
TRG$\sim(\mu-\sigma)$	Trg	0.139	-	0.136	-	0.136	-
	Thr	0.283	0.134	0.237	0.096	0.206	0.039
	Drops	7.66%	15.11%	8.81%	20.28%	10.17%	24.67%
	Thr/Trg	2.039	-	1.735	-	1.523	-
	Thr/Teq	0.868	0.716	0.905	0.766	1.042	0.620
TRG$\sim(\mu+\sigma)$	Trg	0.370	-	0.364	-	0.361	-
	Thr	0.484	0.134	0.453	0.096	0.408	0.039
	Drops	3.56%	15.11%	4.22%	20.28%	4.37%	24.67%
	Thr/Trg	1.306	-	1.246	-	1.130	-
	Thr/Teq	0.867	0.716	0.927	0.766	0.963	0.620

Table B.4: Target Rate: DS vs BE Per-Target Rate Performance

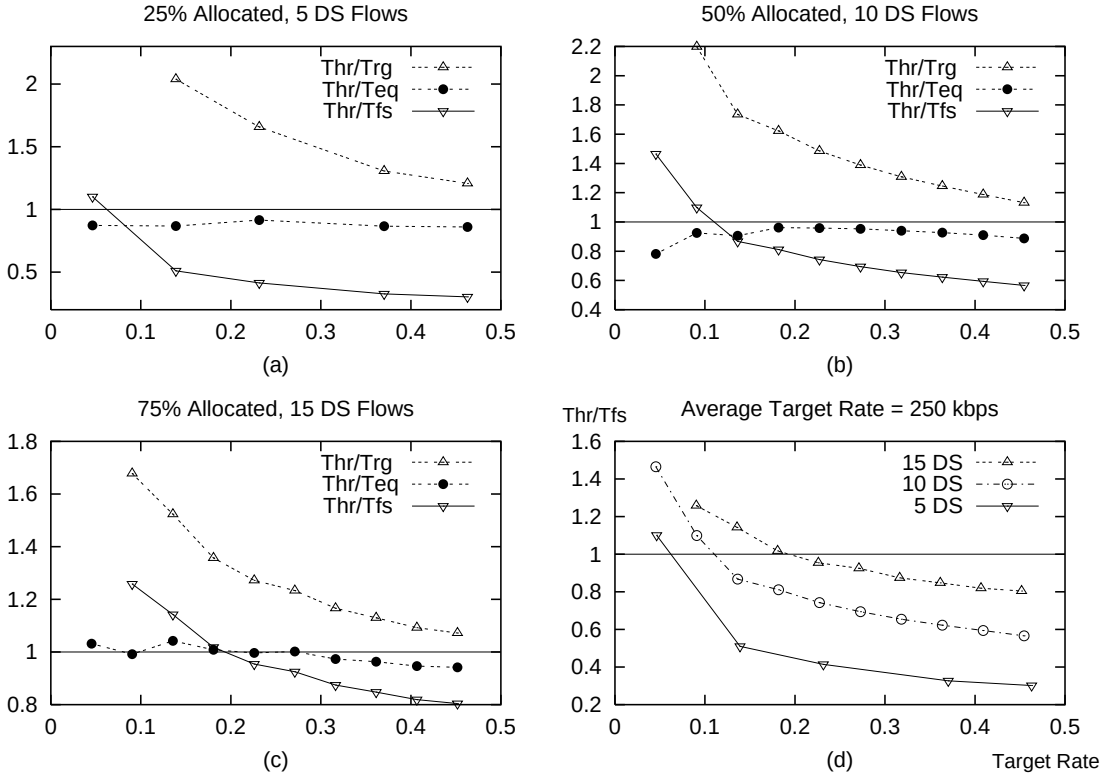


Figure B.4: TARGET: Same Target Rate

TRG @ CIR		5 @ 25 %		10 @ 50 %		15 @ 75 %	
		DS	BE	DS	BE	DS	BE
Per-Flow	Trg	0.250	-	0.250	-	0.250	-
μ	Thr	0.383	0.146	0.345	0.094	0.306	0.036
	Drops	6.18%	14.75%	7.19%	19.99%	7.67%	25.87%
	Thr/Trg	2.123	-	1.623	-	1.367	-
	Thr/Teq	0.876	0.780	0.915	0.752	0.988	0.578
Per-Flow	Trg	0.151	-	0.131	-	0.123	-
σ	Thr	0.129	0.016	0.121	0.025	0.113	0.007
	Drops	2.98%	0.59%	3.73%	1.40%	3.94%	0.66%
	Thr/Trg	1.178	-	0.529	-	0.349	-
	Thr/Teq	0.020	0.087	0.050	0.203	0.030	0.113
Overall	Link ρ	0.3826		0.6902		0.917	
	Drops	5.18 %		5.94 %		6.26 %	

Table B.5: Target Rate: DS vs BE Overall Performance, Trg = 250 kbps

TRG @ DRP		TRG			DS Overall		Link
		min Mbps	avg Mbps	max Mbps	*avg*	*sdev*	*tot*
CIR	5 @ 25 %	7.66%	5.61%	3.56%	6.18%	2.98%	10.28%
	10 @ 50 %	8.81%	5.80%	4.22%	7.19%	3.73%	8.88%
	15 @ 75 %	10.17%	5.67%	4.37%	7.67%	3.94%	7.01%
FLOW	10 @ 25 %	11.15%	8.57%	6.89%	8.74%	2.45%	10.64%
	10 @ 50 %	8.81%	5.80%	4.22%	7.19%	3.73%	8.88%
	10 @ 75 %	7.57%	4.11%	2.59%	5.82%	4.00%	6.33%
HIGH-f	15 @ 75 %	10.17%	5.67%	4.37%	7.67%	3.94%	7.01%
	15 @ 85 %	10.21%	5.38%	4.19%	7.48%	4.33%	6.32%
	15 @ 95 %	10.33%	5.17%	3.58%	7.34%	4.96%	5.50%
ALLOC	5 @ 50 %	5.99%	3.16%	1.64%	4.58%	4.01%	9.13%
	10 @ 50 %	8.81%	5.80%	4.22%	7.19%	3.73%	8.88%
	15 @ 50 %	11.15%	7.59%	6.25%	8.94%	3.33%	9.20%
HIGH-a	5 @ 95 %	2.99%	0.80%	0.00%	2.46%	3.21%	5.54%
	10 @ 95 %	6.92%	3.88%	1.86%	5.11%	4.40%	3.99%
	15 @ 95 %	10.33%	5.17%	3.58%	7.34%	4.96%	5.50%

Table B.6: Target Rate: Overall Packet Drop Results

B.2 Round Trip Time Suite

RTT @ CIR		Round Trip Time			DS Overall		Link
		10 ms	50 ms	100 ms	μ	σ	Σ
5 @ 25 %	Thr	0.531	0.332	0.269	0.362	0.095	4.126
	Thr/Trg	2.124	1.327	1.078	1.448	0.378	3.301
	Thr/Tfs	0.531	0.332	0.269	0.362	0.095	*0.825*
10 @ 50 %	Thr	0.454	0.321	0.256	0.323	0.060	4.339
	Thr/Trg	1.814	1.286	1.025	1.293	0.240	1.736
	Thr/Tfs	0.907	0.643	0.512	0.647	0.120	*0.868*
15 @ 75 %	Thr	0.367	0.285	0.241	0.286	0.036	4.573
	Thr/Trg	1.469	1.141	0.962	1.146	0.144	1.219
	Thr/Tfs	1.102	0.856	0.722	0.859	0.108	*0.915*

Table B.7: RTT: DS Performance, Trg = 250 kbps

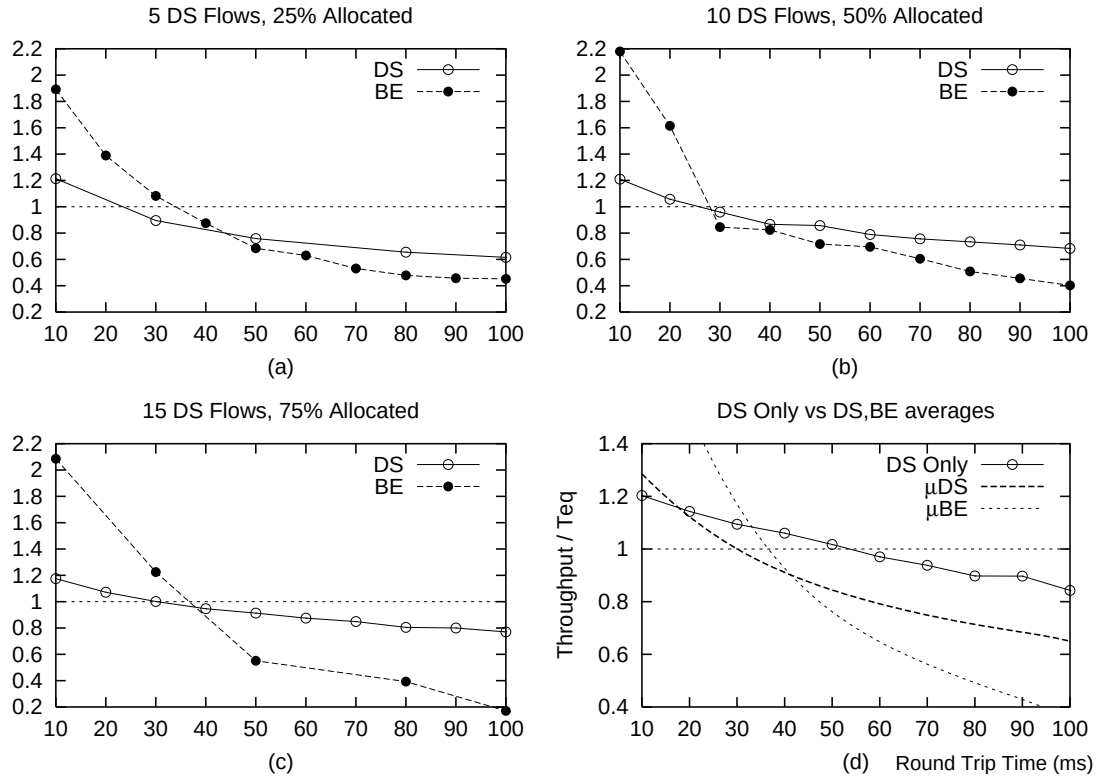


Figure B.5: RTT: Same Target

RTT @ FLOW		Round Trip Time			DS Overall		Link
		10 ms	50 ms	100 ms	μ	σ	Σ
10 @ 25 %	Thr	0.426	0.233	0.176	0.256	0.078	4.124
	Thr/Trg	3.406	1.866	1.407	2.047	0.623	3.299
	Thr/Tfs	0.851	0.466	0.352	0.512	0.156	*0.825*
10 @ 50 %	Thr	0.454	0.321	0.256	0.323	0.060	4.339
	Thr/Trg	1.814	1.286	1.025	1.293	0.240	1.736
	Thr/Tfs	0.907	0.643	0.512	0.647	0.120	*0.868*
10 @ 75 %	Thr	0.486	0.393	0.335	0.395	0.045	4.594
	Thr/Trg	1.296	1.048	0.894	1.054	0.121	1.225
	Thr/Tfs	0.972	0.786	0.671	0.791	0.090	*0.919*

Table B.8: RTT: DS Performance, $N_{DS} = 10$

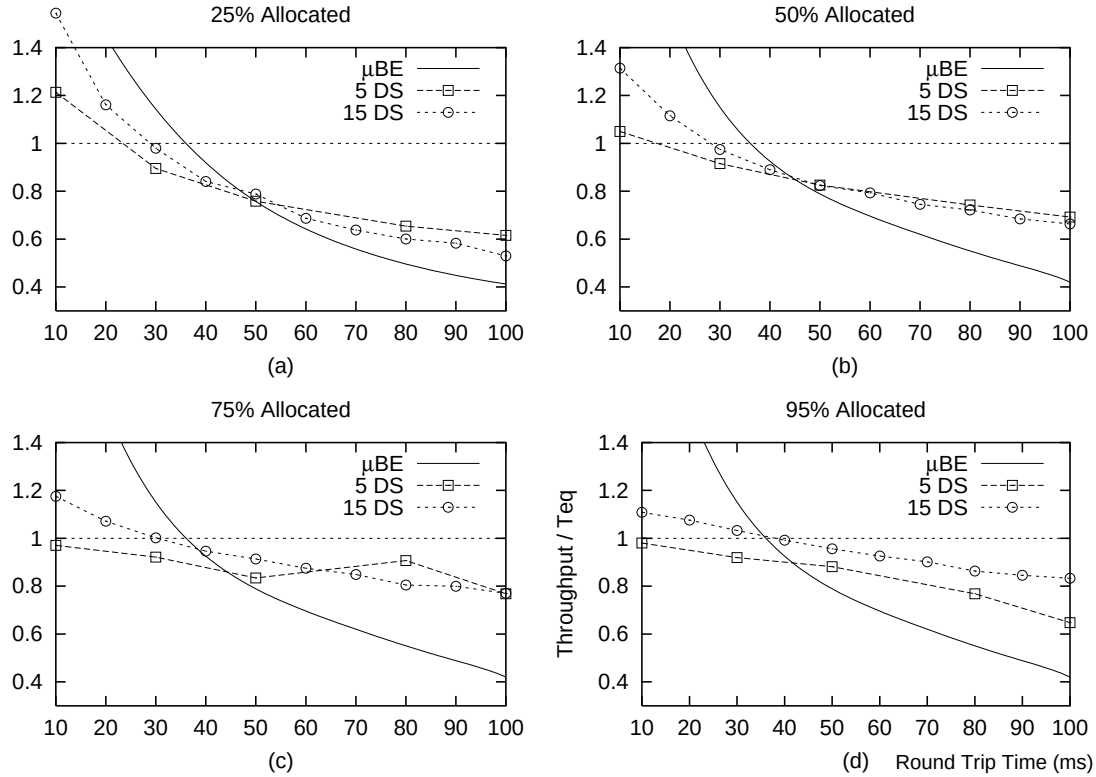


Figure B.6: RTT: Same Flow Number

RTT @ HIGH-f		Round Trip Time			DS Overall		Link
		10 ms	50 ms	100 ms	μ	σ	Σ
15 @ 75 %	Thr	0.367	0.285	0.241	0.286	0.036	4.573
	Thr/Trg	1.469	1.141	0.962	1.146	0.144	1.219
	Thr/Tfs	1.102	0.856	0.722	0.859	0.108	*0.915*
15 @ 85 %	Thr	0.372	0.296	0.257	0.301	0.034	4.652
	Thr/Trg	1.314	1.045	0.907	1.063	0.119	1.095
	Thr/Tfs	1.117	0.888	0.771	0.903	0.101	*0.930*
15 @ 95 %	Thr	0.365	0.315	0.274	0.313	0.029	4.771
	Thr/Trg	1.152	0.994	0.866	0.989	0.092	1.004
	Thr/Tfs	1.095	0.944	0.823	0.940	0.087	*0.954*

Table B.9: RTT: DS Performance at High Reserved Bandwidth, $N_{DS} = 15$

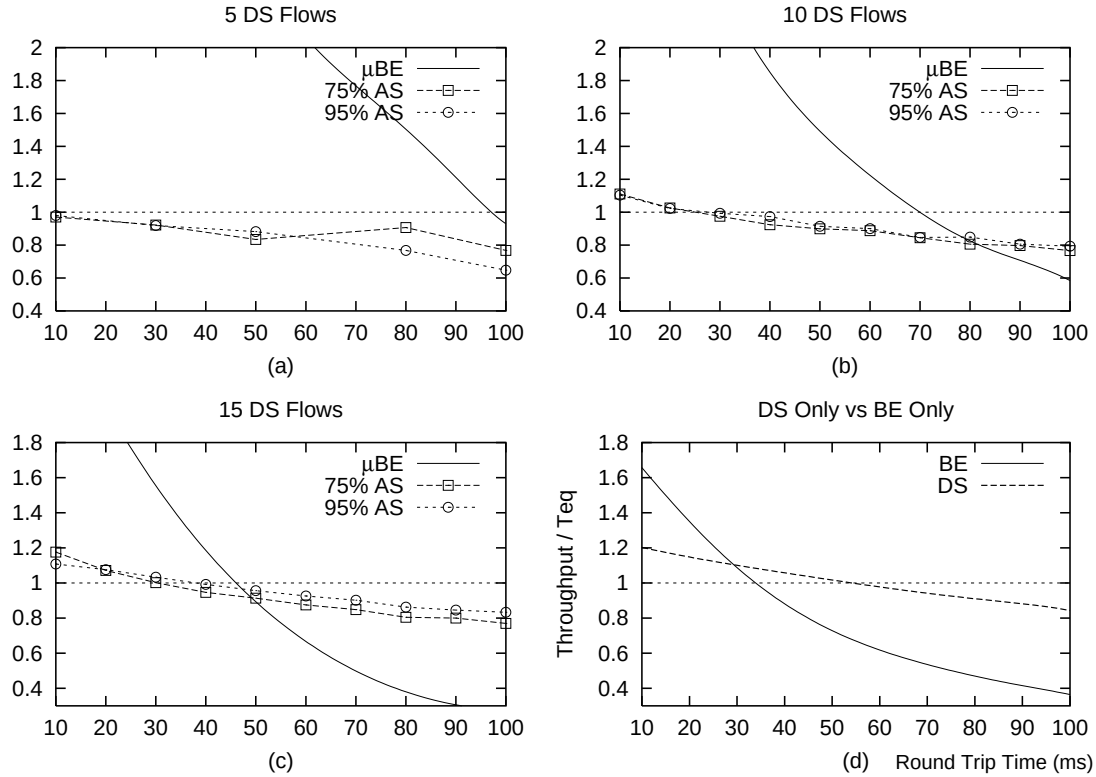


Figure B.7: RTT: High Allocation

RTT @ ALLOC		Round Trip Time			DS Overall		Link
		10 ms	50 ms	100 ms	μ	σ	Σ
5 @ 50 %	Thr	0.656	0.516	0.433	0.528	0.080	4.333
	Thr/Trg	1.312	1.032	0.865	1.056	0.159	1.733
	Thr/Tfs	0.656	0.516	0.432	0.528	0.080	*0.867*
10 @ 50 %	Thr	0.454	0.321	0.256	0.323	0.060	4.339
	Thr/Trg	1.814	1.286	1.025	1.293	0.240	1.736
	Thr/Tfs	0.907	0.643	0.512	0.647	0.120	*0.868*
15 @ 50 %	Thr	0.383	0.240	0.194	0.252	0.054	4.330
	Thr/Trg	2.300	1.441	1.161	1.511	0.322	1.732
	Thr/Tfs	1.150	0.721	0.581	0.755	0.161	*0.866*

Table B.10: RTT: DS Performance, $\rho_{DS} = 50\%$

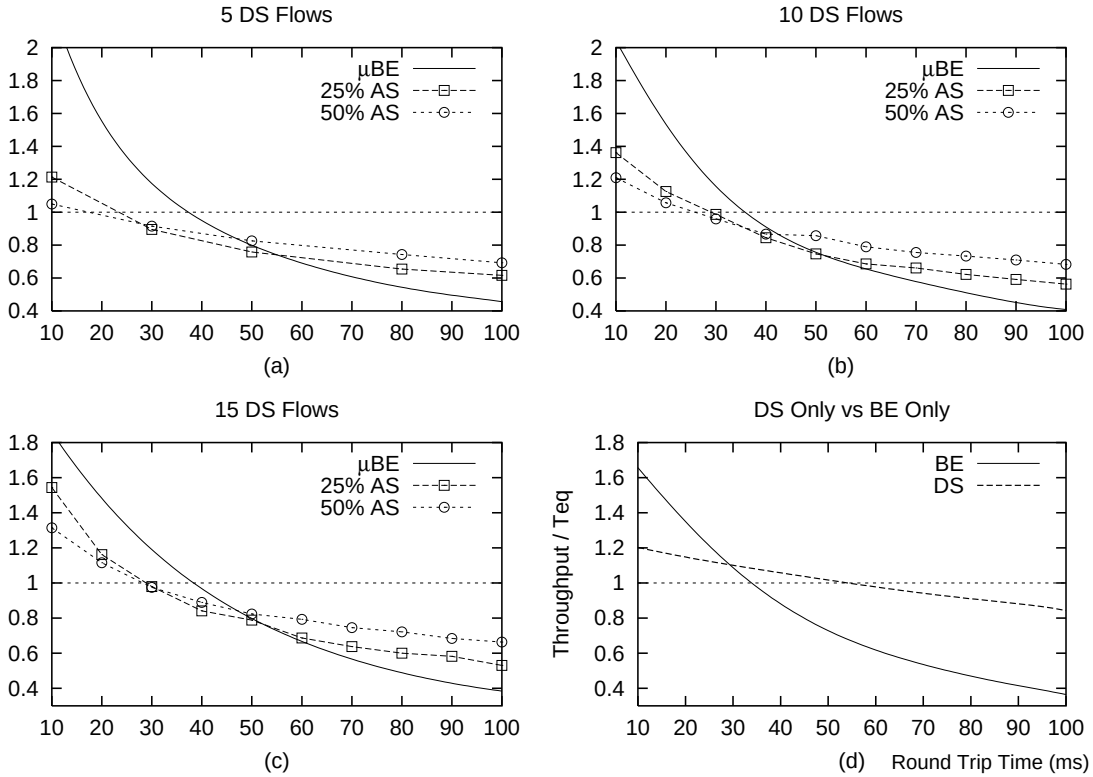


Figure B.8: RTT: Low Allocation

RTT @ HIGH-a		Round Trip Time			DS Overall		Link
		10 ms	50 ms	100 ms	μ	σ	Σ
5 @ 95 %	Thr	0.944	0.848	0.623	0.808	0.114	4.869
	Thr/Trg	0.993	0.893	0.656	0.850	0.120	1.025
	Thr/Tfs	0.944	0.849	0.623	0.808	0.114	*0.974*
10 @ 95 %	Thr	0.538	0.446	0.387	0.449	0.047	4.759
	Thr/Trg	1.133	0.938	0.814	0.945	0.099	1.002
	Thr/Tfs	1.076	0.891	0.773	0.897	0.094	*0.952*
15 @ 95 %	Thr	0.365	0.315	0.274	0.313	0.029	4.771
	Thr/Trg	1.152	0.994	0.866	0.989	0.092	1.004
	Thr/Tfs	1.095	0.944	0.823	0.940	0.087	*0.954*

Table B.11: RTT: DS Performance, $\rho_{DS} = 95\%$

RTT @ CIR		5 @ 25 %		10 @ 50 %		15 @ 75 %	
		DS	BE	DS	BE	DS	BE
RTT=10 ms	Thr	0.531	0.355	0.454	0.272	0.367	0.130
	Drops	4.92%	8.27%	5.81%	10.32%	7.13%	14.66%
	Thr/Trg	2.124	-	1.814	-	1.469	-
	Thr/Teq	1.214	1.891	1.210	2.180	1.175	2.086
RTT=50 ms	Thr	0.332	0.128	0.321	0.090	0.285	0.034
	Drops	2.95%	10.36%	3.82%	15.20%	4.41%	25.08%
	Thr/Trg	1.327	-	1.286	-	1.141	-
	Thr/Teq	0.758	0.684	0.857	0.716	0.913	0.551
RTT=100 ms	Thr	0.269	0.085	0.256	0.050	0.241	0.011
	Drops	1.70%	10.27%	2.11%	17.27%	2.65%	31.42%
	Thr/Trg	1.078	-	1.025	-	0.962	-
	Thr/Teq	0.616	0.453	0.683	0.403	0.770	0.171

Table B.12: RTT: DS vs BE Per-RTT Performance

RTT @ CIR		5 @ 25 %		10 @ 50 %		15 @ 75 %	
		DS	BE	DS	BE	DS	BE
Per-Flow	Thr	0.362	0.154	0.323	0.111	0.286	0.055
μ	Drops	3.13%	10.53%	3.62%	15.25%	4.36%	23.93%
	Thr/Trg	1.448	-	1.293	-	1.146	-
	Thr/Teq	0.827	0.824	0.862	0.885	0.916	0.885
Per-Flow	Thr	0.095	0.079	0.060	0.067	0.036	0.043
σ	Drops	1.20%	0.89%	1.28%	2.04%	1.34%	5.90%
	Thr/Trg	0.378	-	0.240	-	0.144	-
	Thr/Teq	0.216	0.421	0.160	0.540	0.115	0.696
Overall	Link ρ	0.8252		0.8678		0.9146	
	Drops	7.20 %		6.45 %		5.45 %	

Table B.13: RTT: DS vs BE Overall Performance, Trg = 250 kbps

RTT @ DS vs BE		RTT			Overall		
		10 ms	50 ms	100 ms	μ	σ	Σ
BE Only	Thr	0.423	0.168	0.096	0.197	0.102	3.930
	Drops	6.35%	8.14%	7.28%	7.76%	0.67%	7.46%
	Thr/Teq	1.691	0.671	0.383	0.786	0.409	0.786
DS Only	Thr	0.298	0.255	0.211	0.252	0.028	5.031
	Drops	7.18%	4.71%	3.05%	4.77%	1.40%	4.91%
	Thr/Teq†	1.191	1.018	0.846	1.006	0.112	1.006
† *Teq≡Trg being all the linerate allocated to DiffServ flows*							

Table B.14: RTT: DS-Only vs BE-Only Performance

RTT @ DRP		RTT			DS Overall		Link
		10 ms	50 ms	100 ms	*avg*	*sdev*	*tot*
CIR	5 @ 25 %	4.92%	2.95%	1.70%	3.13%	1.20%	7.20%
	10 @ 50 %	5.81%	3.82%	2.11%	3.62%	1.28%	6.45%
	15 @ 75 %	7.13%	4.41%	2.65%	4.36%	1.34%	5.45%
FLOW	10 @ 25 %	6.42%	4.89%	3.85%	5.08%	0.98%	7.25%
	10 @ 50 %	5.81%	3.82%	2.11%	3.62%	1.28%	6.45%
	10 @ 75 %	4.61%	2.70%	1.41%	2.70%	0.99%	5.05%
HIGH-f	15 @ 75 %	7.13%	4.41%	2.65%	4.36%	1.34%	5.45%
	15 @ 85 %	6.65%	4.14%	2.39%	4.09%	1.28%	4.86%
	15 @ 95 %	6.21%	3.39%	2.04%	3.64%	1.23%	4.15%
ALLOC	5 @ 50 %	3.13%	1.79%	0.64%	1.79%	0.85%	6.55%
	10 @ 50 %	5.81%	3.82%	2.11%	3.62%	1.28%	6.45%
	15 @ 50 %	7.42%	5.11%	3.30%	5.26%	1.47%	6.70%
HIGH-a	5 @ 95 %	0.42%	0.15%	0.00%	0.17%	0.16%	3.82%
	10 @ 95 %	3.89%	2.10%	1.09%	2.15%	0.84%	3.57%
	15 @ 95 %	6.21%	3.39%	2.04%	3.64%	1.23%	4.15%

Table B.15: RTT: Overall Packet Drop Results

B.3 Packet Size Suite

SIZE @ CIR		Packet Size			DS Overall		Link
		512 byte	768 byte	1024 byte	μ	σ	Σ
5 @ 25 %	Thr	0.329	0.381	0.410	0.344	0.047	3.841
	Thr/Trg	1.317	1.523	1.639	1.377	0.189	3.073
	Thr/Tfs	0.329	0.381	0.410	0.344	0.047	*0.768*
10 @ 50 %	Thr	0.305	0.331	0.375	0.315	0.043	4.145
	Thr/Trg	1.219	1.323	1.500	1.260	0.171	1.658
	Thr/Tfs	0.610	0.661	0.750	0.630	0.085	*0.829*
15 @ 75 %	Thr	0.278	0.296	0.321	0.284	0.027	4.483
	Thr/Trg	1.111	1.183	1.284	1.137	0.110	1.195
	Thr/Tfs	0.833	0.888	0.963	0.853	0.082	*0.897*

Table B.16: Packet Size: DS Performance, Trg = 250 kbps

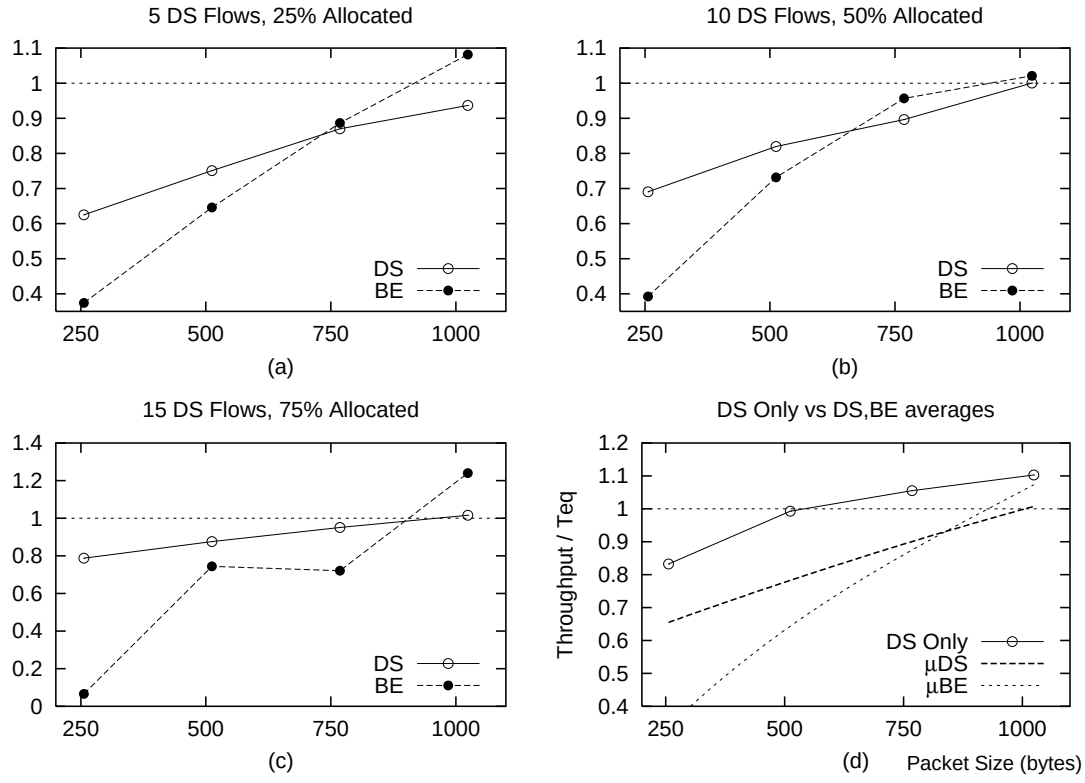


Figure B.9: SIZE: Same Target

SIZE @ FLOW		Packet Size			DS Overall		Link
		512 byte	768 byte	1024 byte	μ	σ	Σ
10 @ 25 %	Thr	0.230	0.283	0.314	0.243	0.058	3.859
	Thr/Trg	1.844	2.267	2.508	1.945	0.465	3.087
	Thr/Tfs	0.461	0.567	0.627	0.486	0.116	*0.772*
10 @ 50 %	Thr	0.305	0.331	0.375	0.315	0.043	4.145
	Thr/Trg	1.219	1.323	1.500	1.260	0.171	1.658
	Thr/Tfs	0.610	0.661	0.750	0.630	0.085	*0.829*
10 @ 75 %	Thr	0.390	0.423	0.437	0.397	0.033	4.460
	Thr/Trg	1.041	1.127	1.166	1.060	0.088	1.189
	Thr/Tfs	0.780	0.845	0.874	0.795	0.066	*0.892*

Table B.17: Packet Size: DS Performance, $N_{DS} = 10$

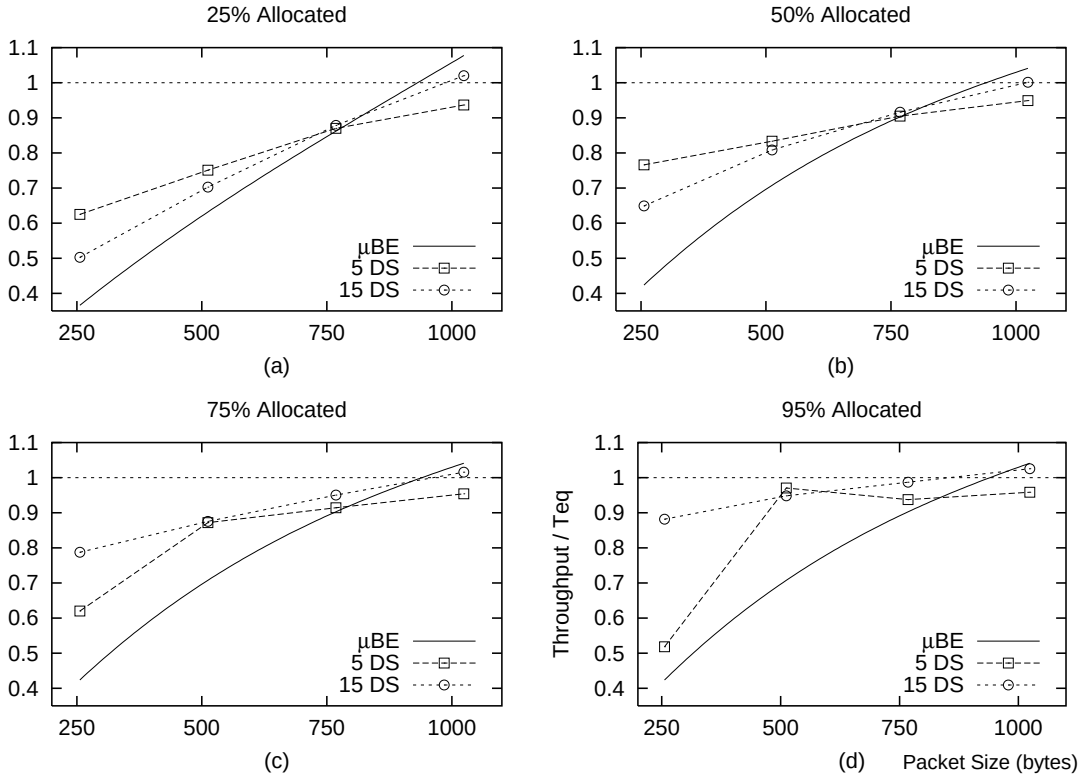


Figure B.10: SIZE: Same Flow Number

SIZE @ HIGH-f		Packet Size			DS Overall		Link
		512 byte	768 byte	1024 byte	μ	σ	Σ
15 @ 75 %	Thr	0.278	0.296	0.321	0.284	0.027	4.483
	Thr/Trg	1.111	1.183	1.284	1.137	0.110	1.195
	Thr/Tfs	0.833	0.888	0.963	0.853	0.082	*0.897*
15 @ 85 %	Thr	0.293	0.312	0.340	0.302	0.025	4.581
	Thr/Trg	1.035	1.101	1.200	1.065	0.087	1.078
	Thr/Tfs	0.880	0.936	1.020	0.905	0.074	*0.916*
15 @ 95 %	Thr	0.313	0.328	0.336	0.316	0.018	4.771
	Thr/Trg	0.987	1.036	1.062	0.999	0.057	1.004
	Thr/Tfs	0.938	0.984	1.008	0.949	0.054	*0.954*

Table B.18: Packet Size: DS Performance at High Reserved DS Bandwidth, $N_{DS} = 15$

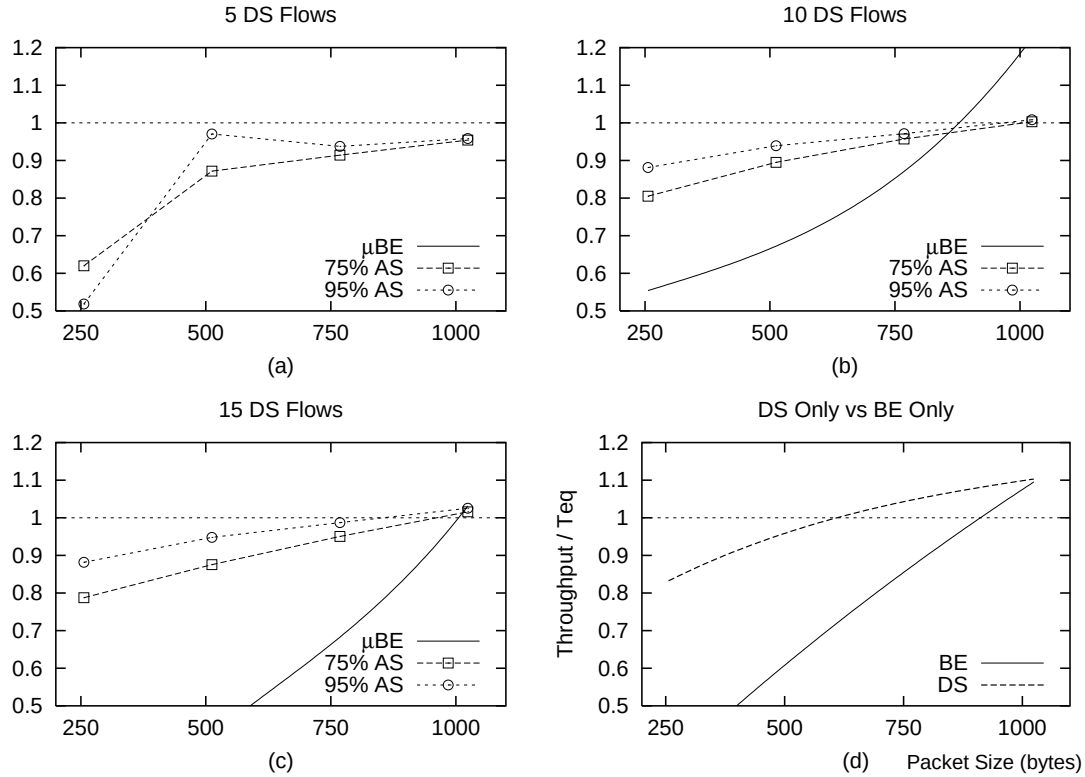


Figure B.11: SIZE: High Allocation

SIZE @ ALLOC		Packet Size			DS Overall		Link
		512 byte	768 byte	1024 byte	μ	σ	Σ
5 @ 50 %	Thr	0.521	0.566	0.593	0.536	0.040	4.237
	Thr/Trg	1.043	1.131	1.186	1.072	0.079	1.695
	Thr/Tfs	0.521	0.566	0.593	0.536	0.040	*0.847*
10 @ 50 %	Thr	0.305	0.331	0.375	0.315	0.043	4.145
	Thr/Trg	1.219	1.323	1.500	1.260	0.171	1.658
	Thr/Tfs	0.610	0.661	0.750	0.630	0.085	*0.829*
15 @ 50 %	Thr	0.237	0.264	0.286	0.247	0.040	4.129
	Thr/Trg	1.423	1.586	1.717	1.480	0.238	1.652
	Thr/Tfs	0.711	0.793	0.859	0.740	0.119	*0.826*

Table B.19: Packet Size: DS Performance, $\rho_{DS} = 50\%$

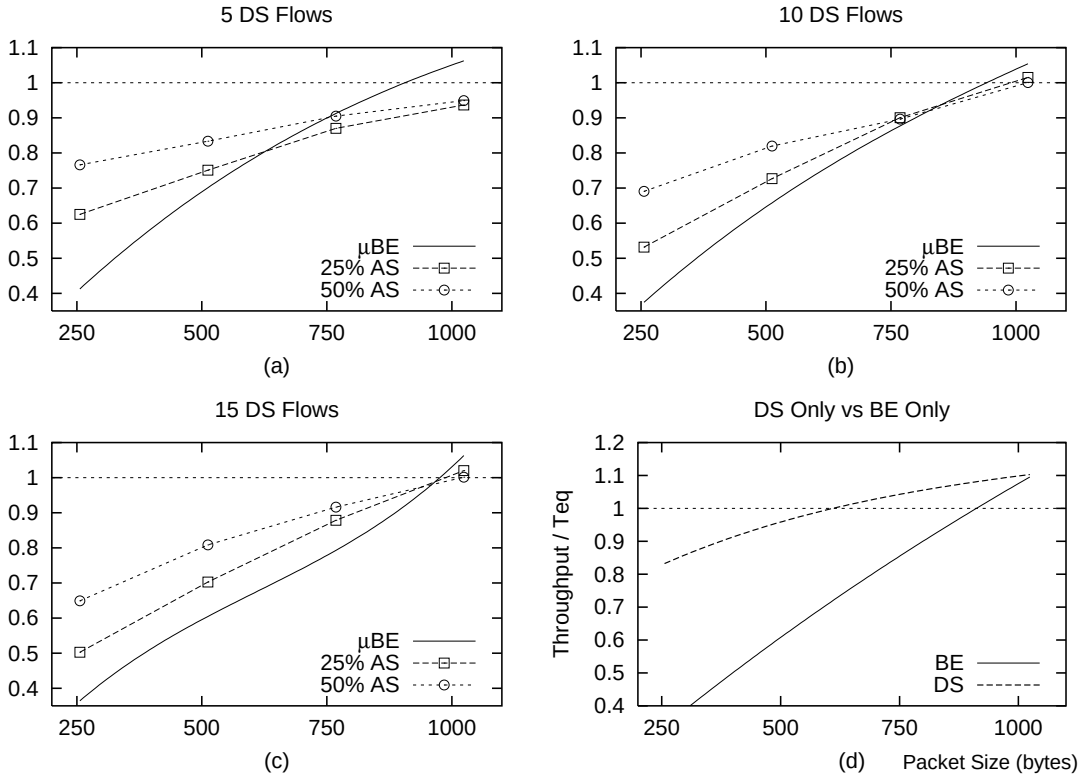


Figure B.12: SIZE: Low Allocation

SIZE @ HIGH-a		Packet Size			DS Overall		Link
		512 byte	768 byte	1024 byte	μ	σ	Σ
5 @ 95 %	Thr	0.936	0.902	0.922	0.838	0.170	4.946
	Thr/Trg	0.985	0.950	0.971	0.882	0.179	1.041
	Thr/Tfs	0.936	0.902	0.922	0.838	0.170	*0.989*
10 @ 95 %	Thr	0.461	0.475	0.489	0.461	0.023	4.718
	Thr/Trg	0.971	1.000	1.030	0.970	0.049	0.993
	Thr/Tfs	0.922	0.950	0.978	0.921	0.046	*0.944*
15 @ 95 %	Thr	0.313	0.328	0.336	0.316	0.018	4.771
	Thr/Trg	0.987	1.036	1.062	0.999	0.057	1.004
	Thr/Tfs	0.938	0.984	1.008	0.949	0.054	*0.954*

Table B.20: Packet Size: DS Performance, $\rho_{DS} = 95\%$

SIZE @ CIR		5 @ 25 %		10 @ 50 %		15 @ 75 %	
		DS	BE	DS	BE	DS	BE
SIZE=512 byte	Thr	0.329	0.118	0.305	0.088	0.278	0.033
	Drops	3.07%	12.28%	3.63%	18.35%	4.33%	26.41%
	Thr/Trg	1.317	-	1.219	-	1.111	-
	Thr/Teq	0.752	0.632	0.813	0.703	0.889	0.526
SIZE=768 byte	Thr	0.381	0.157	0.331	0.116	0.296	0.045
	Drops	4.71%	13.23%	5.38%	19.52%	6.01%	26.72%
	Thr/Trg	1.523	-	1.323	-	1.183	-
	Thr/Teq	0.870	0.838	0.882	0.928	0.947	0.721
SIZE=1024 byte	Thr	0.410	0.187	0.375	0.136	0.321	0.077
	Drops	5.45%	14.08%	6.80%	20.18%	7.52%	26.69%
	Thr/Trg	1.639	-	1.500	-	1.284	-
	Thr/Teq	0.937	0.999	1.000	1.084	1.028	1.240

Table B.21: Packet Size: DS vs BE Per-Packet Size Performance

SIZE @ CIR		5 @ 25 %		10 @ 50 %		15 @ 75 %	
		DS	BE	DS	BE	DS	BE
Per-Flow	Thr	0.344	0.141	0.315	0.099	0.284	0.044
μ	Drops	3.52%	12.71%	4.15%	19.09%	4.93%	26.69%
	Thr/Trg	1.377	-	1.260	-	1.137	-
	Thr/Teq	0.787	0.754	0.840	0.795	0.909	0.703
Per-Flow	Thr	0.047	0.052	0.043	0.029	0.027	0.025
σ	Drops	1.43%	0.77%	1.91%	1.24%	2.24%	1.59%
	Thr/Trg	0.189	-	0.171	-	0.110	-
	Thr/Teq	0.108	0.276	0.114	0.235	0.088	0.398
Overall	Link ρ	0.7682		0.829		0.8966	
	Drops	8.80 %		8.01 %		6.18 %	

Table B.22: Packet Size: DS vs BE Overall Performance, Trg = 250 kbps

SIZE @ DS vs BE		SIZE			Overall		
		512 byte	768 byte	1024 byte	μ	σ	Σ
BE Only	Thr	0.163	0.213	0.281	0.185	0.071	3.693
	Drops	8.83%	9.25%	9.37%	8.96%	0.47%	9.10%
	Thr/Teq	0.651	0.853	1.123	0.739	0.284	0.739
DS Only	Thr	0.249	0.263	0.275	0.249	0.026	4.980
	Drops	3.90%	6.00%	8.26%	4.99%	2.29%	5.22%
	Thr/Teq†	0.997	1.053	1.101	0.996	0.102	0.996

† *Teq≡Trg being all the linerate allocated to DiffServ flows*

Table B.23: Packet Size: DS-Only vs BE-Only Performance

SIZE @ DRP		SIZE			DS Overall		Link
		512 byte	768 byte	1024 byte	*avg*	*sdev*	*tot*
CIR	5 @ 25 %	3.07%	4.71%	5.45%	3.52%	1.43%	8.80%
	10 @ 50 %	3.63%	5.38%	6.80%	4.15%	1.91%	8.01%
	15 @ 75 %	4.33%	6.01%	7.52%	4.93%	2.24%	6.18%
FLOW	10 @ 25 %	5.85%	7.39%	8.43%	6.04%	2.02%	9.07%
	10 @ 50 %	3.63%	5.38%	6.80%	4.15%	1.91%	8.01%
	10 @ 75 %	2.64%	3.74%	5.08%	2.87%	1.63%	5.38%
HIGH-f	15 @ 75 %	4.33%	6.01%	7.52%	4.93%	2.24%	6.18%
	15 @ 85 %	3.34%	5.58%	7.35%	4.41%	2.19%	4.91%
	15 @ 95 %	2.81%	4.74%	6.37%	3.82%	1.94%	4.11%
ALLOC	5 @ 50 %	1.35%	2.35%	2.89%	1.61%	0.97%	7.15%
	10 @ 50 %	3.63%	5.38%	6.80%	4.15%	1.91%	8.01%
	15 @ 50 %	5.84%	7.83%	9.08%	6.39%	2.47%	8.06%
HIGH-a	5 @ 95 %	0.01%	0.26%	0.33%	0.12%	0.14%	4.06%
	10 @ 95 %	1.84%	2.74%	3.90%	2.05%	1.42%	2.86%
	15 @ 95 %	2.81%	4.74%	6.37%	3.82%	1.94%	4.11%

Table B.24: Packet Size: Overall Packet Drop Results

B.4 μFlow Number Suite

μFLOW @ CIR		Micro-Flow Number			DS Overall		Link
		1	4	8	μ	σ	Σ
5 @ 25 %	Thr	0.332	0.524	0.711	0.476	0.132	4.812
	Thr/Trg	1.326	2.095	2.843	1.905	0.529	3.850
	Thr/Tfs	0.332	0.524	0.711	0.476	0.132	*0.962*
10 @ 50 %	Thr	0.305	0.429	0.531	0.398	0.085	4.824
	Thr/Trg	1.219	1.715	2.125	1.591	0.338	1.930
	Thr/Tfs	0.609	0.857	1.063	0.796	0.169	*0.965*
15 @ 75 %	Thr	0.271	0.347	0.404	0.332	0.049	5.051
	Thr/Trg	1.086	1.386	1.615	1.326	0.195	1.347
	Thr/Tfs	0.814	1.040	1.211	0.995	0.146	*1.010*

Table B.25: μ-Flow: DS Performance, Trg = 250 kbps

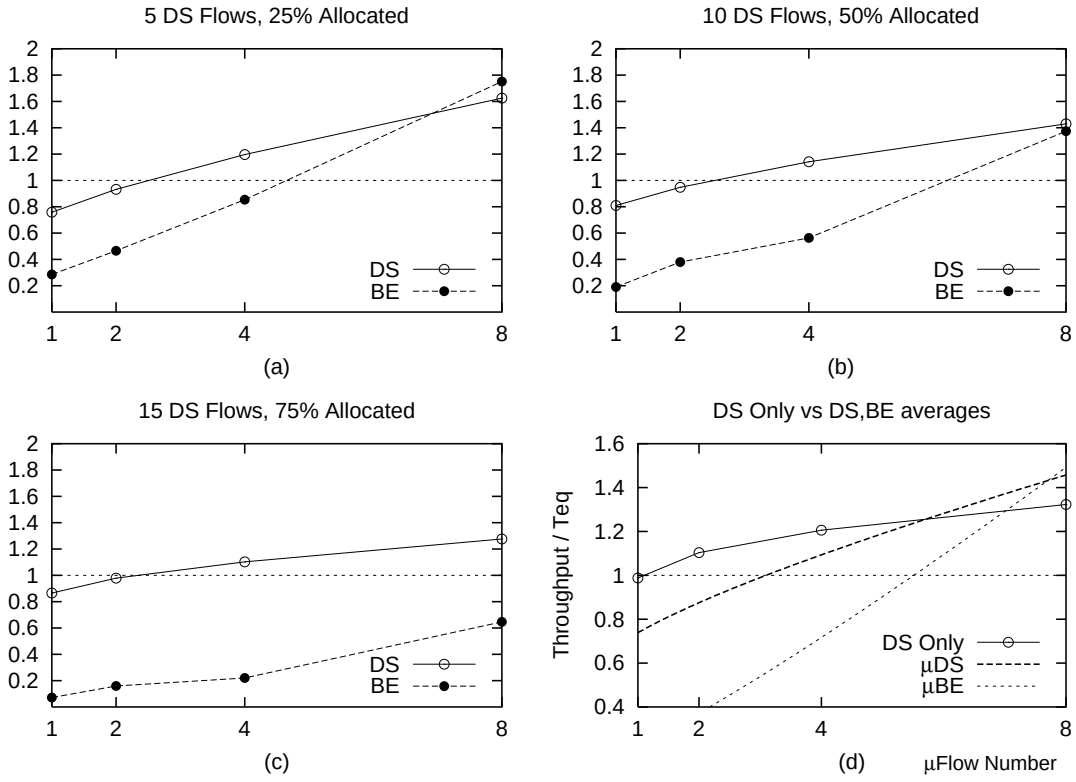


Figure B.13: μFLOW: Same Target

μFLOW @ FLOW		Micro-Flow Number			DS Overall		Link
		1	4	8	μ	σ	Σ
10 @ 25 %	Thr	0.205	0.344	0.505	0.319	0.105	4.709
	Thr/Trg	1.641	2.750	4.044	2.555	0.844	3.767
	Thr/Tfs	0.410	0.687	1.011	0.639	0.211	*0.942*
10 @ 50 %	Thr	0.305	0.429	0.531	0.398	0.085	4.824
	Thr/Trg	1.219	1.715	2.125	1.591	0.338	1.930
	Thr/Tfs	0.609	0.857	1.063	0.796	0.169	*0.965*
10 @ 75 %	Thr	0.387	0.494	0.561	0.465	0.063	4.892
	Thr/Trg	1.033	1.316	1.496	1.241	0.169	1.305
	Thr/Tfs	0.774	0.987	1.122	0.930	0.126	*0.978*

Table B.26: μ-Flow: DS Performance, $N_{DS} = 10$

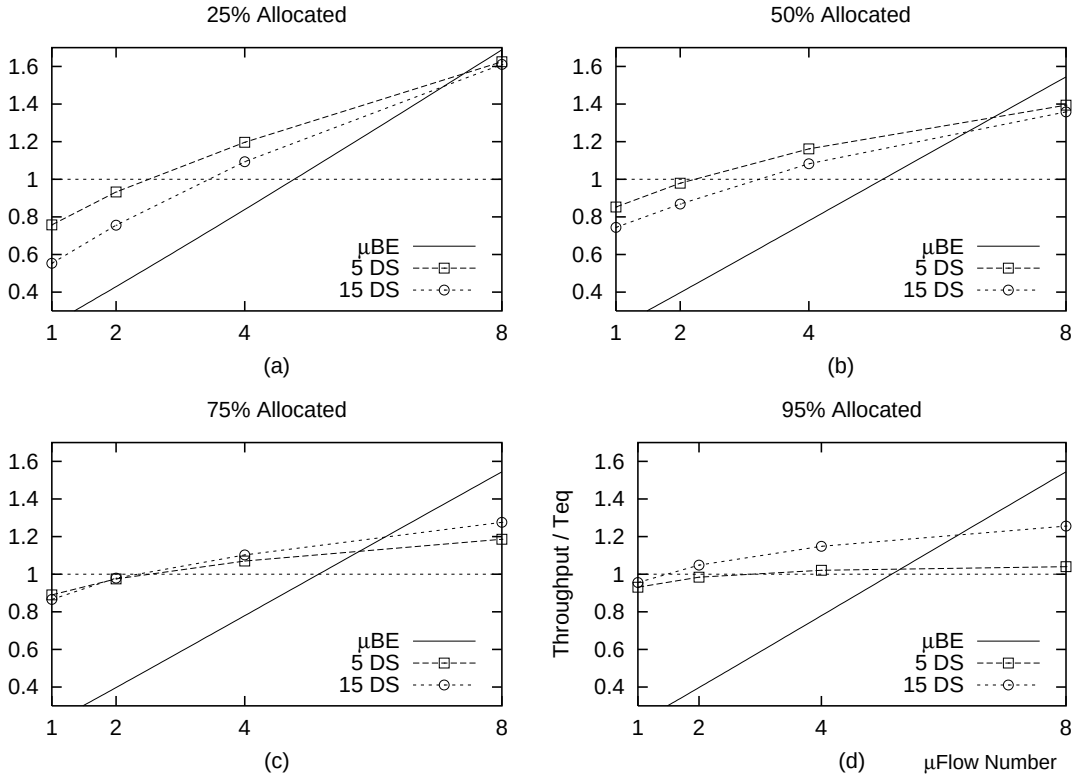


Figure B.14: μFLOW: Same Flow Number

μFLOW @ HIGH-f		Micro-Flow Number			DS Overall		Link
		1	4	8	μ	σ	Σ
15 @ 75 %	Thr	0.271	0.347	0.404	0.332	0.049	5.051
	Thr/Trg	1.086	1.386	1.615	1.326	0.195	1.347
	Thr/Tfs	0.814	1.040	1.211	0.995	0.146	*1.010*
15 @ 85 %	Thr	0.290	0.361	0.407	0.348	0.044	5.263
	Thr/Trg	1.025	1.276	1.438	1.229	0.156	1.238
	Thr/Tfs	0.871	1.084	1.222	1.045	0.133	*1.053*
15 @ 95 %	Thr	0.313	0.380	0.410	0.364	0.038	5.475
	Thr/Trg	0.990	1.201	1.294	1.149	0.119	1.153
	Thr/Tfs	0.940	1.141	1.230	1.092	0.114	*1.095*

Table B.27: μ-Flow: DS Performances at High Reserved Bandwidth, $N_{DS} = 15$

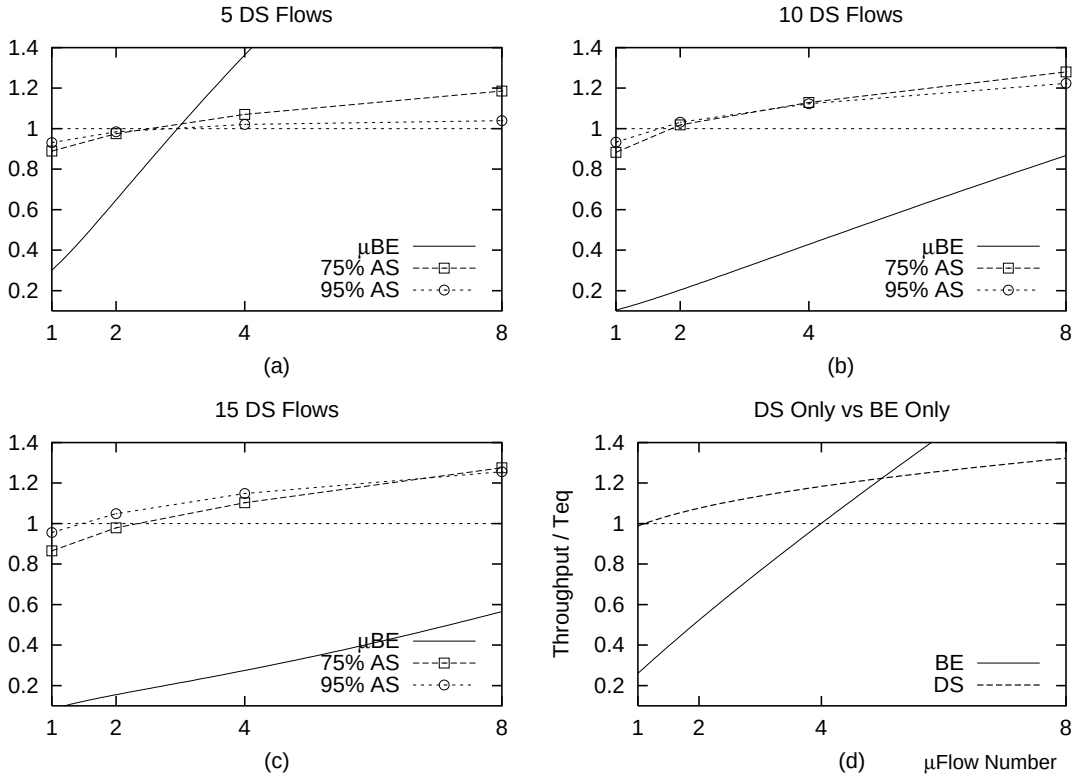


Figure B.15: μFLOW: High Allocation

μFLOW @ ALLOC		Micro-Flow Number			DS Overall		Link
		1	4	8	μ	σ	Σ
5 @ 50 %	Thr	0.533	0.726	0.871	0.671	0.118	4.995
	Thr/Trg	1.065	1.452	1.743	1.341	0.236	1.998
	Thr/Tfs	0.532	0.726	0.872	0.671	0.118	*0.999*
10 @ 50 %	Thr	0.305	0.429	0.531	0.398	0.085	4.824
	Thr/Trg	1.219	1.715	2.125	1.591	0.338	1.930
	Thr/Tfs	0.609	0.857	1.063	0.796	0.169	*0.965*
15 @ 50 %	Thr	0.218	0.314	0.385	0.298	0.070	4.794
	Thr/Trg	1.309	1.883	2.309	1.790	0.417	1.918
	Thr/Tfs	0.655	0.942	1.154	0.895	0.209	*0.959*

Table B.28: μ-Flow: DS Performance, $\rho_{DS} = 50\%$

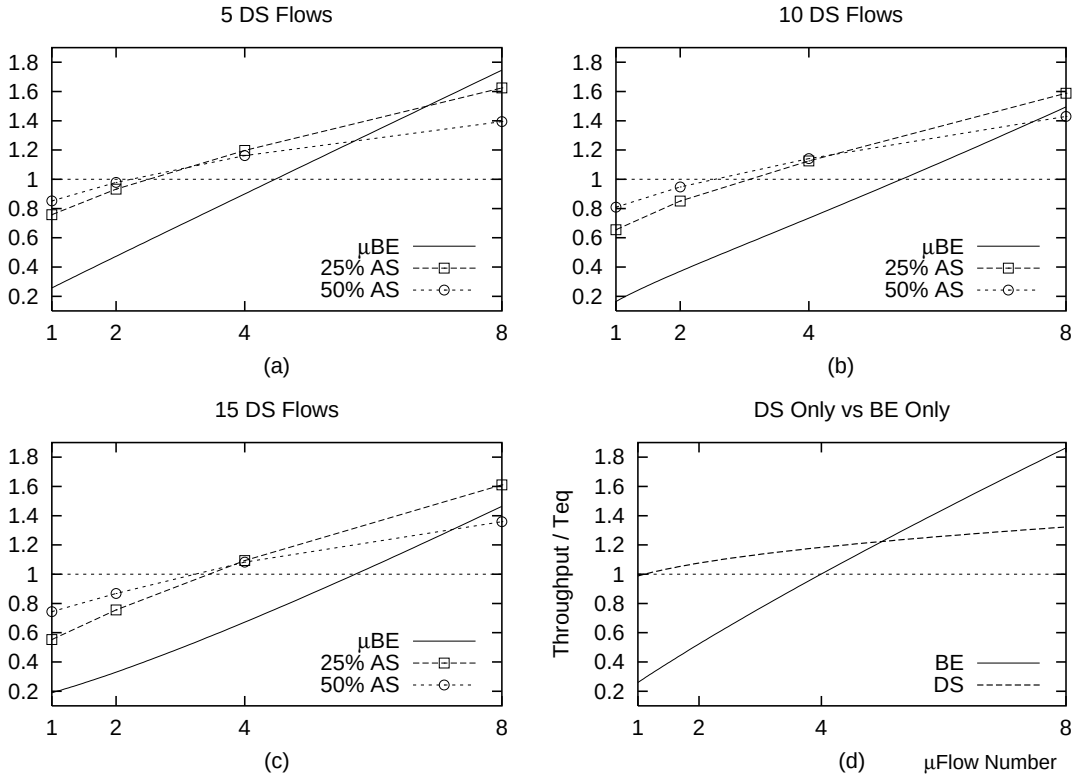


Figure B.16: μFLOW: Low Allocation

μFLOW @ HIGH-a		Micro-Flow Number			DS Overall		Link
		1	4	8	μ	σ	Σ
5 @ 95 %	Thr	0.896	0.983	1.001	0.955	0.036	5.064
	Thr/Trg	0.943	1.034	1.053	1.005	0.038	1.066
	Thr/Tfs	0.896	0.983	1.001	0.955	0.036	*1.013*
10 @ 95 %	Thr	0.459	0.552	0.598	0.521	0.052	5.264
	Thr/Trg	0.966	1.161	1.259	1.096	0.110	1.108
	Thr/Tfs	0.917	1.103	1.196	1.041	0.105	*1.053*
15 @ 95 %	Thr	0.313	0.380	0.410	0.364	0.038	5.475
	Thr/Trg	0.990	1.201	1.294	1.149	0.119	1.153
	Thr/Tfs	0.940	1.141	1.230	1.092	0.114	*1.095*

Table B.29: μ-Flow: DS Performance, $\rho_{DS} = 95\%$

μFLOW @ CIR		5 @ 25 %		10 @ 50 %		15 @ 75 %	
		DS	BE	DS	BE	DS	BE
μFLOW=1	Thr	0.332	0.062	0.305	0.027	0.271	0.004
	Drops	6.04%	24.16%	7.19%	32.57%	7.42%	45.57%
	Thr/Trg	1.326	-	1.219	-	1.086	-
	Thr/Teq	0.758	0.328	0.812	0.213	0.869	0.072
μFLOW=4	Thr	0.524	0.129	0.429	0.077	0.347	0.014
	Drops	12.43%	25.80%	15.13%	31.88%	16.39%	51.54%
	Thr/Trg	2.095	-	1.715	-	1.386	-
	Thr/Teq	1.197	0.691	1.143	0.613	1.109	0.220
μFLOW=8	Thr	0.711	0.320	0.531	0.178	0.404	0.040
	Drops	15.27%	24.92%	19.34%	32.11%	21.35%	46.08%
	Thr/Trg	2.843	-	2.125	-	1.615	-
	Thr/Teq	1.625	1.708	1.417	1.427	1.292	0.646

Table B.30: μ-Flow: DS vs BE Per-Micro-Flow Number Performance

μFLOW @ CIR		5 @ 25 %		10 @ 50 %		15 @ 75 %	
		DS	BE	DS	BE	DS	BE
Per-Flow	Thr	0.476	0.162	0.398	0.085	0.332	0.016
μ	Drops	10.47%	24.80%	12.64%	32.76%	14.29%	47.77%
	Thr/Trg	1.905	-	1.591	-	1.326	-
	Thr/Teq	1.089	0.864	1.061	0.677	1.061	0.251
Per-Flow	Thr	0.132	0.108	0.085	0.060	0.049	0.013
σ	Drops	3.14%	0.80%	4.63%	1.25%	5.24%	2.17%
	Thr/Trg	0.529	-	0.338	-	0.195	-
	Thr/Teq	0.302	0.578	0.225	0.479	0.156	0.203
Overall	Link ρ	0.9624		0.9648		1.0102	
	Drops	18.20 %		16.94 %		15.56 %	

Table B.31: μ-Flow: DS vs BE Overall Performance, Trg = 250 kbps

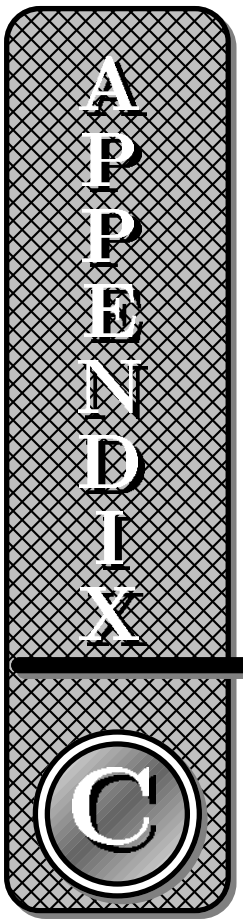
μ**FLOW @ DS vs BE**		μFLOW			Overall		
		1	4	8	μ	σ	Σ
BE Only	Thr	0.052	0.254	0.409	0.231	0.154	4.626
	Drops	21.54%	20.84%	20.42%	20.61%	0.66%	20.47%
	Thr/Teq	0.209	1.016	1.634	0.925	0.615	0.925
DS Only	Thr	0.247	0.304	0.330	0.289	0.031	5.775
	Drops	7.02%	16.21%	20.98%	13.72%	5.29%	14.29%
	Thr/Teq†	0.988	1.214	1.319	1.155	0.124	1.155

† *Teq≡Trg being all the linerate allocated to DiffServ flows*

Table B.32: μ-Flow: DS-Only vs BE-Only Performance

μ**FLOW @ DRP**		μFLOW			DS Overall		Link
		1	4	8	*avg*	*sdev*	*tot*
CIR	5 @ 25 %	6.04%	12.43%	15.27%	10.47%	3.14%	18.20%
	10 @ 50 %	7.19%	15.13%	19.34%	12.64%	4.63%	16.94%
	15 @ 75 %	7.42%	16.39%	21.35%	14.29%	5.24%	15.56%
FLOW	10 @ 25 %	10.33%	16.97%	19.99%	15.01%	3.69%	19.43%
	10 @ 50 %	7.19%	15.13%	19.34%	12.64%	4.63%	16.94%
	10 @ 75 %	4.74%	12.84%	17.52%	10.52%	4.80%	12.69%
HIGH-f	15 @ 75 %	7.42%	16.39%	21.35%	14.29%	5.24%	15.56%
	15 @ 85 %	6.55%	16.02%	20.44%	13.63%	5.41%	14.61%
	15 @ 95 %	5.86%	14.78%	19.57%	12.65%	5.34%	13.37%
ALLOC	5 @ 50 %	3.01%	9.15%	12.08%	7.13%	3.15%	15.14%
	10 @ 50 %	7.19%	15.13%	19.34%	12.64%	4.63%	16.94%
	15 @ 50 %	9.91%	18.29%	22.51%	16.37%	4.85%	18.56%
HIGH-a	5 @ 95 %	0.42%	2.37%	3.27%	1.66%	1.02%	4.05%
	10 @ 95 %	3.42%	11.39%	15.64%	8.93%	4.58%	9.99%
	15 @ 95 %	5.86%	14.78%	19.57%	12.65%	5.34%	13.37%

Table B.33: μ-Flow: Overall Packet Drop Results



ns Code

Short Lived HTTP Flows

C.1 HTTP Flows Model

In the need of simulate HTTP flows over a DiffServ network, various approach are feasible. Even if an OO approach would seem allurant, it does pose indeed some not neglectable software engineering difficulties; the solution presented here proposes a series of Tcl scripts which interact –with some minor modifications to *ns* C++ code– with existing *ns* agents; this choice, as will be clearer later (see Sec. C.2.2), improves simulator performances, offering beside that

- definition of HTTP flows aggregate (HTTP Clouds, described in Sec. C.1.3.2)
- multiple Cloud support, each with its own source/destination/SLA
- re-use of *ns* agents to speed up simulation setup and run
- light-weighted flow-oriented tracing mechanism
- wide flexibility in using different network scenarios
- possibility of variating the short TCP flow model

C.1.1 HTTP Scripts Overview

The table Tab. C.1 shortly lists the Tcl script files which functions list is reported in Sec. C.3. The main script, `HTTP.tcl`, just `sources` them all. Beyond that, it consists of only two procedure calls:

```
MAIN_init
HTTP_run
```

Although it may seems inconvenient, the "patch" mechanism (described later in Sec. C.1.6) permits to beneficiate of a high flexibility just following a convenient queues and nodes naming notation.

Tcl File		Task
HTTP.general.tcl	⇝	general purpose routines, mostly output oriented
HTTP.option.tcl	⇝	{OPTION/GLOBAL/MAIN}_init routines take care of command line option parsing as well as global variables setup
HTTP.mapping.tcl	⇝	HTTP Classifier code, used to enhance HTTP Cloud mechanism in Tcl/C++ interactions
HTTP.cloud.tcl	⇝	HTTP_cloud flows aggregate setup and creation
HTTP.monitor.tcl	⇝	Per-Flow, Per-Cloud Tcl Tracing Mechanism
HTTP.patch.tcl	⇝	Scenario Initialization, with "patching" features
HTTP.http_init.tcl	⇝	Responsible of HTTP Clouds creation accordingly to topology parameters
HTTP.background.tcl	⇝	Optional background FTP sources handling
HTTP.core.tcl	⇝	HTTP flows generation, handling, stats collection; resolves Tcl/C++ interactions
HTTP.queues.tcl	⇝	Queue's and other stats's dumping procedures
HTTP.finish.tcl	⇝	Simulator run/finish/dump procedures

Table C.1: Tcl Script Index

In the latter, these files will be referenced accordingly to the following notation:

`HTTP.xxx.tcl` ⇝ ◁xxx▷

Additionally, a Tcl proc `yyy` defined in ◁`xxx`▷ will be referred as ◁`xxx`▷`yyy`, while a reference to a C++ method will be expressed in the classic form `Object::method`. No details are given on proc's and method's parameters here: for further information, please refers to the code.

C.1.2 Main Script Syntax

```
Usage:
        ns HTTP.tcl  -par1 val1 .. -parn -valn

Arguments indicated with * are mandatory

------------------------------------------------------------------------
(def|*) [-param  value] meaning
========================================================================
 2      [-routno N]     N is number of routers of the HTTP cloud
 fctry  [-model  M]     M is the name of the HTTP model (fctry|pareto)
------------------------------------------------------------------------
 *      [-SLAn   S]     -SLA0 for cloud0, -SLA1 for cloud1 etc.
                        S is a quoted string following Nortel convention:
                        'TSW2CM $DSCP $cir'
                        'TSW3CM $DSCP $cir $pir'
                        'TokenBucket $DSCP $cir $cbs'
                        'srTCM $DSCP $cir $cbs $ebs'
                        'trTCM $DSCP $cir $cbs $pir $pbs'
 *      [-CLn    C]     Normalized per-cloud offered load (in a single
                        clouded scenario, CL0 is defaulted to 1)
 *      [-load   X]     Normalized overall offered load
------------------------------------------------------------------------
 *      [-start  sec]   Simulator's starting-time of HTTP connections
 *      [-stop   sec]   Simulator's stop time of HTTP connections
 *      [-halt   sec]   Simulator's halt time
------------------------------------------------------------------------
 0      [-ftpno  N]     Number of background long lived FTP sources
 *      [-bwneck bw]    Bottleneck's bandwidth (Mbps)
 666    [-seed   seed]  Seed of the randomizer
 1000   [-psize  size]  Mean Packet Size (Bytes)
------------------------------------------------------------------------
 0      [-monidx S]     String idx prepended to trace-file produced
 .      [-mondir S]     Base directory of trace-file produced
------------------------------------------------------------------------
        [-ppost  file]  Post Process Code; full access to globals
        [-topo   file]  Specifies a TOPOLOGY definition script
        [-ffser  file]  Specifies a DiffServ policy rules script
========================================================================
 off    [-quiet  Oo]    Do You Want Me Talkative Or Introverse ?
        [-nam]         If you want to namtrace-all the simulation...
```

Table C.2: Output of `ns HTTP.tcl` command

`-routno`

number of per-Cloud *ideal* routers which HTTP sources are attached. Such a router is ideal in the sens that it has *huge* bandwidth and *neglectable* RTT: it is used to model a traffic

multiplexer; for some considerations on its setting, see Sec. C.2.2.5.

`-model`

The factory model chooses –via an integer randomized variable x_i– the length λ of flow i from an empiric distribution Λ, while the interarrival time is determined by a Poisson process ϕ_i.

$$\begin{cases} \overline{\lambda} = 13.6\ KB & \text{Average flow length} \\ \overline{\tau} = \rho \cdot B/\overline{\lambda}\ sec & \text{Average interarrival time, where} \\ & \quad B \text{ is the linerate} \\ & \quad \rho \text{ the offered load} \\ \lambda_i = \Lambda[x_i] & \text{Flow } i \text{ length, with} \\ & \quad \Lambda \text{ vector storing the empirical flow length} \\ \tau_{i+1} = \overline{\tau} \cdot \phi_i & \text{Next HTTP flow arrival time, with} \\ & \quad \phi_i \text{ exponential distribution with average 1} \end{cases}$$

The Pareto model can be described as:

$$P(\lambda_i > x) = \begin{cases} (x/\delta)^{-\gamma} & x \geq \delta \\ 1 & x < \delta \end{cases}$$

with

$$\begin{cases} \gamma = 1.3 & \text{Thus Hurst parameter H=0.85} \\ \delta = 1\ KB & \text{Minimum flow length} \\ \overline{\lambda} = 4.33\ KB & \text{Average flow length} \end{cases}$$

`-SLA0,-SLA1,...`

each cloud's SLA can be changed directly on the command line, while the source/destination nodes pair is determined (either in ◁`patch`▷ or in an external file), by the following naming notation:

$$\begin{cases} \text{Cloud } \texttt{\$i} \text{ source:} & \texttt{\$dn\_(src,\$i)} \\ \text{Cloud } \texttt{\$i} \text{ destination:} & \texttt{\$dn\_(dst,\$i)} \end{cases}$$

It should be noticed that, that way, two different cloud may have different source and share the same destination node. Furthermore, the initial DiffServ Code Point of the HTTP flows (and eventually FTP background traffic) packets can be used to create Best-Effort clouds.

`-load,-CL0,-CL1,...`

Select the total normalized offered load ρ of the network via `-load` and distribute it per-cloud via `-CL`$_i$; these parameters are used to evaluate each cloud's average interarrival time $\overline{\tau_{f}}_i$:

$$\overline{\tau_{f}}_i = \frac{\overline{\lambda} \cdot 8}{\texttt{CL}_i \cdot \rho \cdot B}$$

being the average flow length $\overline{\lambda}$ expressed in bytes and the bottleneck link rate B expressed in Mbps, the formula expresses $\overline{\tau_{f}}_i$ in μs. Note that the sum of the `-CL`$_i$ should be the unity; wheter the simulation uses a single cloud `CL0` is defaulted to 1.

`-start,-stop,-halt`

The HTTP sources will start sending data at the specified `start` instant, and no new flow will be started after the `stop` time. However, simulation will run until the minimum of the last HTTP flow finished time and the forced `halt` time. The (always mandatory) starting time option is used to *warm up* the simulation: the (eventually existent) FTP traffic background sources are started at a random time uniformly distributed between 0 and `start/2`, so before any HTTP flow starts.

`-ftpno`

This is the number of per-cloud per-router background sources that will be created: this mean that if `-routno 10` and the scenario uses two HTTP clouds, setting `-ftpno 10` would produce a total of 200 FTP sources.

`-bwneck` bw] The bottleneck's bandwidth, expressed in Mbps, is used by the HTTP model to calculate the HTTP flow's interarrival average time; the default topology use this value –stored in global array `$op(bwneck)`– to setup links accordingly

`-seed`

Seed of the randomizer, diabolically defaulted to 666.

`-psize`

The mean packet size, expressed in bytes, is used by TCP agents and RED queues for internal calculations:

```
Agent/TCP set packetSize_   $op(psize)
Queue/RED set mean_pktsize_ $op(psize)
```

`-mondir,-monidx`

Respectively the base directory of and the string prepended to the trace-files produced by the simulation (see Sec. C.1.5 for further details).

`-ffser,-topo,-ppost`

Please refer to section Sec. C.1.6

`-quiet`

Setting it `on` will just make the standard output silent.

`-nam`

Allow to `namtrace-all` the simulation. But beware, since enabling *ns* trace will both slow down the simulation timing performances and require a lot of disk space (roughly 2 MB per second of HTTP traffic simulation on a 10 Mbps bottleneck without FTP background sources): this option is left essentially for debug purposes when a change to either *ns* code or HTTP scripts set is made.

C.1.3 Short TCP Flows Implementation

C.1.3.1 HTTP Flows

An HTTP flow is emulated sending a certain amount of packets via an FTP application, over a TCP source to a TCPSink. Flow size and flow scheduling times are choosen accordingly to a model, defined in ◁core▷: at present time, two different models, `-pareto` and `-fctry`, have been implemented in ◁core▷ `new_HTTP_start`; their initialization is done by ◁http_init▷ `HTTP_model_init`, while their analitycal definition can be found in Sec. C.1.2.

Each HTTP flow will have its own flow id (which equals TCP's fid) and uniquely use the FTP/TCP/TCPSink touple during its transmission. Some HTTP binded state-vars have been added to TCP to make it easier to collect per-HTTP-flow informations (from `ns-default.tcl`):

```
#
# DiffServ per-HTTP-flow statistics
#      (in TCPs agents for HTTP implementation)
#
Agent/TCP set ldrop_no_   0
Agent/TCP set edrop_no_   0
Agent/TCP set be_pkt_no_  0
Agent/TCP set grn_pkt_no_ 0
Agent/TCP set yel_pkt_no_ 0
Agent/TCP set red_pkt_no_ 0

#
# TCPs per-HTTP-flow state vars
#      (used in HTTP implementation)
#
Agent/TCP set http_pkt_no_     0
Agent/TCP set http_old_pkt_no_ 0
Agent/TCP set http_ack_no_     0
Agent/TCP set http_old_ack_no_ 0

Agent/TCP set timeout_no_ 0
Agent/TCP set fast_recovery_no_ 0
Agent/TCP set packet_to_send_no_  -1
              #transmitted by an HTTP flow
Agent/TCP set HTTP_locked_  0
              #by an HTTP flow
Agent/TCP set avg_cwnd_  0
Agent/TCP set max_cwnd_  0
```

The Tcl script is able to build FTP/TCP/TCPSink *ns* agents touples at run-time wheter needed by simulation.

C.1.3.2 HTTP Clouds

An HTTP Cloud is an aggregate of HTTP flows, all having common:

- Source node
- Destionation node
- DiffServ Service Level Specification.

The HTTP flows of the same aggregate are equally distributed among a certain number of *ideal* routers; the routers are said to be ideal in the sense that they have *huge* bandwidth and *neglectable* RTT; indeed, they model a traffic multiplexer.

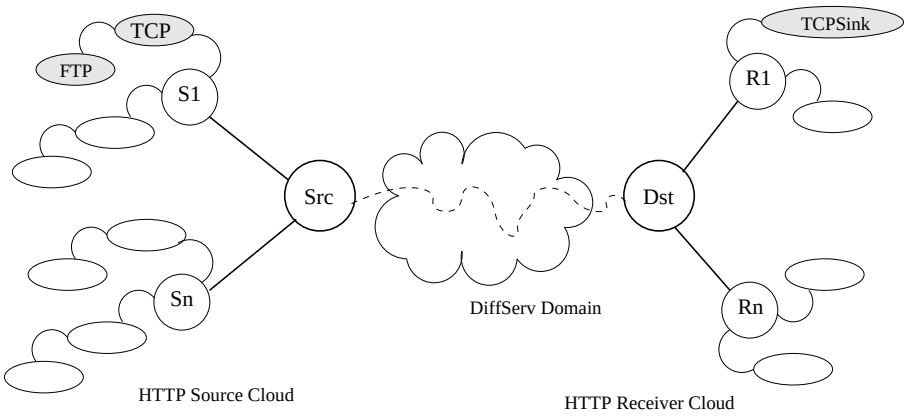


Figure C.1: Synoptic of an HTTP Cloud

Figure Fig. C.1 indicates them as S_i –the source routers– and R_i –the receiver ones, while the shadowed FTP, TCP, TCPSink agents individuate an unique HTTP flow of the HTTP Cloud. In addition to that parameters, every cloud has its own

- Offered Load: this means that it is possible to analize a network –let's say an University LAN– with overall normalized load 0.95, in which the AF traffic of Professors's Cloud has 0.7 guaranteed load and the Poor Student's one has just 0.3
- Initial DSCP: it is useful to easily define Best Effort clouds, and a sort of Fair Best Effort ones: due to Strict Priority Queuing implementation of DiffServ Nortel libraries routers, the Best Effort queue won't be serviced until all the other queues (EF, AF1 (g,y,r), ... AF1 (g,y,r)) are empty; an hack to solve this is to create a cloud that always sends red packets (that way BE packets are served with the same effectiveness that AF red ones), and to consider it a (fairer) Best Effort one, in the sense that flows coming from that Cloud are allowed to compete with other DS Clouds for the extra share of the bandwidth.

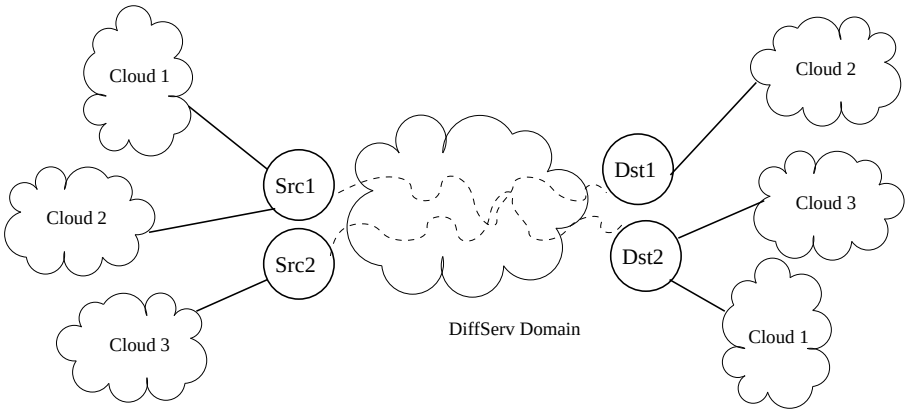


Figure C.2: Synoptic of multiple HTTP Clouds

An HTTP Cloud is thus the top level structure used to manage HTTP Flows. Its creation is basically done by ◁`cloud`▷ script functions:

```
src_HTTP_cloud $i $src
```

```
dst_HTTP_cloud $i $ds
SLA_HTTP_cloud $i $SLAstr
create_HTTP_cloud $i
```

where `$i` is the cloud id (integer or string, Tcl makes no difference!) `$src` and `$dst` are references to *ns* nodes and `$SLAstr` is exactly what we would write using Nortel's `Policy::addPolicyEntry`. Then, once the topology is defined and so are the queues, ◁`patch`▷ `DiffServ_init` proc calls ◁`cloud`▷ `addPolicyEntry_HTTP_cloud` simply that way

```
addPolicyEntry_HTTP_cloud $queue $cloudid
```

Finally, ◁`core`▷ `start_HTTP_flows` is called once, for each cloud, by ◁`finish`▷ `HTTP_run`: the former starts a cloud's flows arrival process and then reschedule itself, following the classical event driven simulation model:

```
start_HTTP_flows $cloudid
```

C.1.4 HTTP Handling Engine

This section focuses on describing how the HTTP clouds are handled both at Tcl and C++ level. This is mainly done by either a C++ call of Tcl functions defined in ◁`core`▷ or –inversely– by Tcl calling of C++ methods: to resolve every object instance addressing, an intesive use of ◁`mapping`▷ functions is made; the most relevant functions in understanding the HTTP cloud mechanism are presented in synoptic of Fig. C.3, which shows as well their interactions.

C.1.4.1 HTTP Flow Starts

As explained earlier, ◁`finish`▷ `HTTP_run` takes care to schedule the first HTTP flow starting call for each cloud, via ◁`core`▷ `new_HTTP_start`, which then reschedule itself after a time period determined by the HTTP model wheter the next activation time isn't greather that the maximum activation time `$op(stop)`.

```
set DELAY     [expr $web(avg_wait,CL$cloudid)*[$rng exponential]]
set web(next) [expr [$ns now] + $DELAY]

if {$web(next) <= $op(stop)} {
    $ns at-now "activate_FTP $cloudid [ packet_to_send ]"
    $ns at $web(next) "start_HTTP_flows $cloudid"
} else {
    if [expr $cloudid==($cloud(num)-1)] finish_HTTP
}
```

When a new flow has to be started, ◁`core`▷ `activate_FTP`, checks if a free source exists in the per-cloud stack managed in ◁`mapping`▷. If there is, then it is popped via ◁`mapping`▷ `pop_HTTP_source` by , and the FTP is instructed to produce some (according to the flow length distribution model) packets; if there is no free source available, then 10 new sources per-router are added to that cloud before instructing the flow to start sending data.

Thus HTTP agent tuples are dinamically created only when needed: this aspect of the scripts, combined with source re-use feature drastically reduce simulation system times needs.

C.1.4.2 HTTP Flow Ends

When a started TCP transfer reachs its end, `TcpAgent::recv_newack_helper` will make the following call to ◁`core`▷ `finish_HTTP_flow`, before resetting TCP variables:

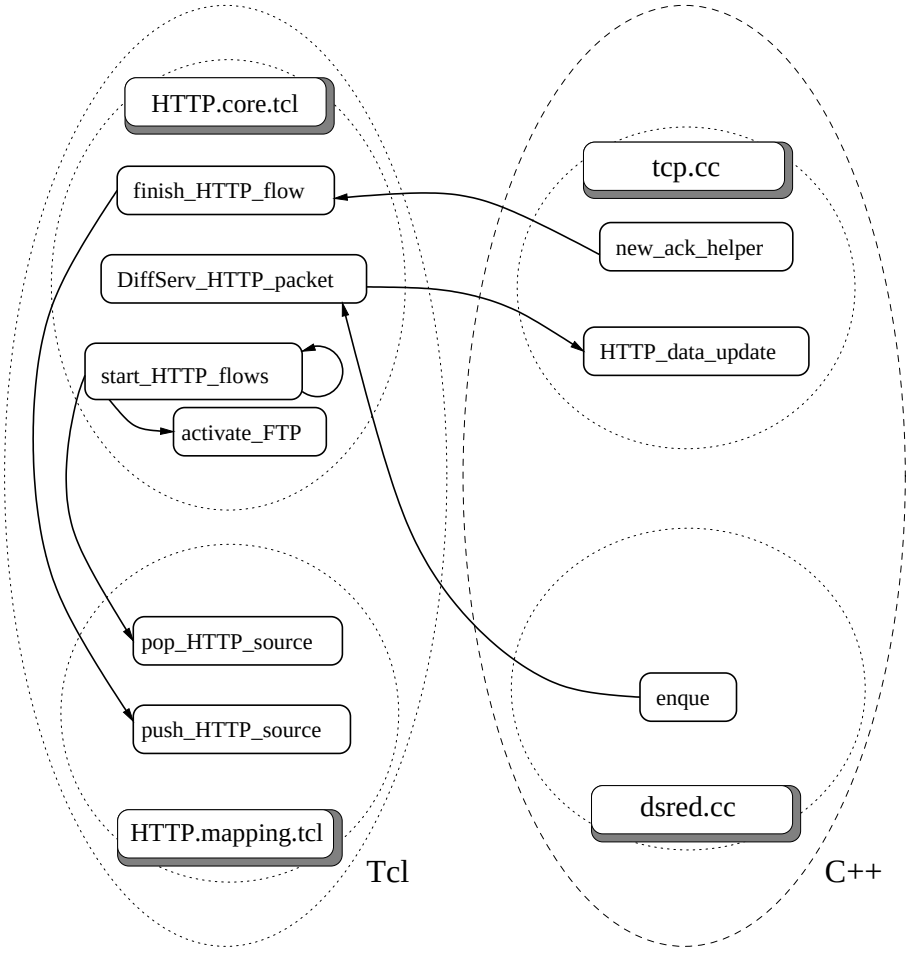


Figure C.3: Tcl/C++ Interactions Roadmap

```
Tcl::instance().evalf("catch \"finish_HTTP_flow %s %s %s\"",
          this->app_->name(),    //  FTP application name
          this->name(),          //  TCP source name
          this->target_->name()  //  TCPSink sink
          );
```

the latter's task is to dumps the stats, collected via ◁core▷ `retrieve_TCP_vars`, to calculate per-cloud ones, and to free then the HTTP source (using ◁mapping▷ `push_HTTP_source`): a *non* object oriented mapping critheria is implemented in ◁mapping▷ to resolve all ambiguity and to handle per-fid source classement. Thus HTTP agent tuples that ended transfering data are put in the free tuple `STACK`, and are available as free sources: all status variables are cleared no that no old status information can affect a new flow. The `finish_HTTP_flow` routine has also the task to test whether the simulation has reached its ending point:

```
if [expr (!$web(active)) && ($this(now) > $op(stop))] {
   HTTP_bye-bye
}
```

C.1.4.3 HTTP Packet Enqueued

In addition to that, when a packet is enqueued, a C++ $\rightsquigarrow$ Tcl $\rightsquigarrow$ C++ signaling mechanism is performed (between queue and TCP agent) to collect per-HTTP flow DiffServ stats about packet marking and e/l-dropping. `dsREDQueue::enque` method instruct Tcl about the DiffServ per-flow stats that he has collected; another kind of mapping must be used here (as we can access to the IP packet header), evaluated as a complete {source,dest}×{addr,port},fid reference:

```
int PKT_code = redq_[queue].enque(pkt, prec, ecn);
Tcl::instance().evalf("catch \"DiffServ_HTTP_packet %d:%d,%d:%d,%d %d %d\"",
                        iph->saddr(), iph->sport(),
                        iph->daddr(), iph->dport(), iph->fid_,
                        PKT_code, codePt
                      );
```

◁core▷ `DiffServ_HTTP_packet` find the TCP source agent correspondant to that reference via ◁mapping▷ `get_HTTP_DiffServ` (if there is not, this means that the packet is an ACK of the backward path). Then it check if the queue belong to the ingress queue's list (which must be appropriately stored in `$op(INGRESS)`, see rule Q.2 in Sec. C.1.6.2) or if a dropping event happened, in which case it updates the per-flow status variables calling `TcpAgent::HTTP_data_update`

```
set tcp [get_HTTP_DiffServ $full_id]
if {$tcp!=0} {
   set is_INGRESS [expr !([lsearch -exact $op(INGRESS) $QUEUE]<0)]
   set is_DROPPED [expr $EVENT==0 || $EVENT==2]
   if [expr $is_INGRESS || $is_DROPPED] {
      $tcp HTTP_data_update $EVENT $DSCP
   }
}
```

Additionally, it does update the global hash variable `DROP` which keeps complete statistics about packets count: it can be indexed as

```
$DROP($Cloud,$Color,$Event)
$DROP($Cloud,$Color,$Event,$App)
```

where

$$\begin{cases} \$Cloud & \in & \{TOT,0,1,\ldots,\$cloud(num)\} \\ \$Color & \in & \{All,BE,DS,Grn,Yel,Red\} \\ \$Event & \in & \{TOT,TX,DRP,ED,LD\} \\ \$App & \in & \{HTTP,FTP\} \end{cases}$$

C.1.5 HTTP Simulation Example

To illustrate HTTP script's output format, a 2-Cloud HTTP simulation using a basic Source Node ⇜⇝ DiffServ Edge Router ⇜⇝ Destination Node scenario, where both source and destination nodes are shared by the identically configured HTTP Clouds, will be run with the following call (implemented in shell script `httest`):

```
ns HTTP.tcl\
    -start 1   -stop 5    -halt 30 \
    -ffser PATCH.test/test.ffser   \
    -topo  PATCH.test/test2cl.topo \
    -SLA0 'TSW2CM 10 800000'       \
    -SLA1 'TSW2CM 10 800000'       \
    -CL0 0.5   -CL1 0.5            \
    -routno 2  -load 0.8           \
    -ftpno 10  -bwneck 10          \
    -mondir test -monidx "2ftp" -nam
```

Listing the contents of `./test` directory, we'll find the following trace files:

Trace File	File Size		Stored Data
2ftp.CLOUD.0 2ftp.CLOUD.1 2ftp.OUT.web	20k 21k 36k	⇝	WEB Traffic Trace
2ftp.OUT.ftp	9.9k	⇝	FTP Traffic Dump
2ftp.OUT.tot	5.5k	⇝	Total Traffic Dump
2ftp.OUT.queue	5.1k	⇝	Queues dump and further statistics
out.2ftp.nam	52M	⇝	All simulator's events trace
2ftp.OUT.scn 2ftp.OUT.DBG	11k 3.1k	⇝	Simulation Parameters, Performances and Debug Informations
.2ftp.POST.init 2ftp.POST	30k 0.5k	⇝	Post Process Tcl Files

C.1.5.1 Traffic Trace and Dump

Tracing involves only HTTP traffic: the monitoring function is called by ◁`core`▷ `finish_HTTP_flow`, thus when an HTTP flow finish sending its data: the traced variable reflects the order shown in the header below:

```
#=====================================================================
# fid tot now id P2S Ack PKT TO FR ED LD  BE Grn Yel Red Thr  Aw  Cw  CT
#---------------------------------------------------------------------
#   0   1   2  3   4   5   6  7  8  9 10  11  12  13  14  15  16  17  18
#=====================================================================
```

The HTTP source `fid`, belonging to cloud `id`, increments the number of `tot` flows finished. As sources are re-used, the `fid` doesn't uniquely identifies a flow –while `tot` does– but the HTTP touple used to generate it. The trace happens at `now` time, when the destination received the `P2S` packet to send determined by the model; `PKT` is the number of packets totally sent by the source for which the TCP Sink generated `Ack` acknowledgement (there's no Ack loss model in the simulation). The packets incurred in `TO` Time Outs, `FR` Fast Recovery, `ED` Early Drops and `LD`

Late Drops, and their colour distribution for that flow are reported under columns `BE`,`Grn`,`Yel`,`Red`. Finally the Throughput `Thr`, the average value of TCP Congestion Window`Aw`, the maximum TCP Congestion Window value reached `Cw` and flow's Completion Time `CT` are listed. The tracing is effectuated on per-cloud (`2ftp.CLOUD.0`, `2ftp.CLOUD.1`) and per-web (2ftp.OUT.web) base.

Dumping involves both HTTP and FTP background traffic, if present; the same variables will be monitored for FTP traffic: here the `tot` is the number of per-cloud FTP sources, `P2S` is meaningless while the `CT` indicates the living time of the background source. Dumping occurs twice during the simulation:

- at the HTTP new flow's schedulation end time $t_1 \leq$ `stop`
- at the last HTTP flow's –and simulation too– end time `stop` $< t_2 \leq$ `halt`

This is done mainly to ensure that HTTP simulation results are meaningful, wheter the extintion of the transitory doesn't massively affect the Web statistics issues. Firstly ◁`finish`▷ `finish_FTP`, wheter the background traffic is present, dumps:

- per-source variable's values
- variable's totals over all the sources
- variable's per-sources averages

Then ◁`finish`▷ `finish_HTTP` dumps:

- per-cloud variable's totals
- per-cloud per-flow variable's averages
- overall variable's totals
- overall per-flow variable's averages

Then the totals of the jointed Web and FTP background traffic is –eventually– dumped; dumping statistics affect mainly `2ftp.OUT.tot` and `2ftp.OUT.ftp`

C.1.5.2 Scenario and Debug Output

The whole initialization process output, which shows as well the simulation parameters, is stored in the scenario description file `2ftp.OUT.scn`. After the simulation run, the simulation timing specifications of the HTTP flow that started at dumping time t_1 as well as those who finished at dumping time t_2 are listed:

```
#============================================================================
# Timing Bla Bla Bla
#----------------------------------------------------------------------------
#
# ns time       start:1.0       stop:1.0       halt:30.0
#                 |               |              |
#               ---+-------+-------+------+------+---> t
#                          |              \_end
# last ended               |     s:[2.958], e:[15.474] => CT:[12.5164]
#                          \_start
# last started  s:[4.950], e:[5.230] => CT:[0.2808]
#
```

Some statistics on simulator performances are also reported:

```
#============================================================================
# Simulator Performances
#----------------------------------------------------------------------------
# Sources:
# Init + Run-Time : 40 + 26 = 66
# Flows:
# {Tot, Max, Avg} : {287, 58, 19.828}
# ns Speed:
# Sim Duration    : Setup + Session        = 8.926  + 120.074 = 129s
# Sim Length      : Last End - First Start = 14.474   ns_s
# Flow Run Rate   : Tot Flows / Sim Run    = 2.390    flw/sec
# Flow Tot Rate   : Tot Flows / Sim Tot    = 2.225    flw/sec
# Sim 'Rate'      : Sim Tot / Sim Length   = 8.296    s/ns_s
```

The debug file `2ftp.OUT.DBG` is created only wheter ◁`general`▷ global variable `debug` is setted to 1: debug informations can be written by a ◁`general`▷ `DEBUG $str` call.

C.1.5.3 Queues Output

At the end of the simulation, some queue statistic about queues and packets are collected by ◁`queues`▷ procs; this is the actual output of `2ftp.OUT.queue`:

```
#============================================================================
# Simulation HTTP Cloud 0 results[ Load: 0.5, SLA: TSW2CM 10 800000 ]
#----------------------------------------------------------------------------
# PKT:(43.73 %) 1778     DRP: (7.03  %) 125    All: 1778   AvgThr : 18944.521
# P2S:(43.84 %) 1617     ED : (35.20 %) 44     BE : 0      AvgAwnd: 0.30052
# TX :(43.88 %) 1653     LD : (64.80 %) 81     Grn: 743    AvgCwnd: 4.92701
# FLW:(47.74 %) 137      FR : (1.29  %) 23     Yel: 1035   AvgCT  : 0.75082
# flw:(43.10 %) 25       TO : (4.39  %) 78     Red: 0      AvgWait: 0.02732
#============================================================================
# Simulation HTTP Cloud 1 results[ Load: 0.5, SLA: TSW2CM 10 800000 ]
#----------------------------------------------------------------------------
# PKT:(56.27 %) 2288     DRP: (7.60  %) 174    All: 2288   AvgThr : 17246.569
# P2S:(56.16 %) 2071     ED : (25.86 %) 45     BE : 0      AvgAwnd: 0.31709
# TX :(56.12 %) 2114     LD : (74.14 %) 129    Grn: 757    AvgCwnd: 4.87333
# FLW:(52.26 %) 150      FR : (1.18  %) 27     Yel: 1531   AvgCT  : 1.06131
# flw:(68.97 %) 40       TO : (5.24  %) 120    Red: 0      AvgWait: 0.02732
#============================================================================
# Simulation WEB results
#----------------------------------------------------------------------------
# PKT: 4066       DRP: (7.35  %) 299        All: 4066
# P2S: 3688       ED:  (29.77 %) 89         BE : 0
# TX : 3767       LD:  (70.23 %) 210        Grn: 1500
# FLW: 287        FR:  (1.23  %) 50         Yel: 2566
# flw: 58         TO:  (4.87  %) 198        Red: 0
#============================================================================
# Simulation FTP results
#----------------------------------------------------------------------------
# PKT: 14133      DRP: (5.35  %) 756        All: 14133
# P2S: 0          ED:  (39.29 %) 297        BE : 0
# TX : 13377      LD:  (60.71 %) 459        Grn: 5054
# SRC: 20         FR:  (1.66  %) 235        Yel: 9079
#  - : 0          TO:  (1.44  %) 204        Red: 0
#============================================================================
# Simulation TOT results
#----------------------------------------------------------------------------
# PKT: 18199      DRP: (5.80  %) 1055       All: 18199
# P2S: 3688       ED:  (36.59 %) 386        BE : 0
# TX : 17144      LD:  (63.41 %) 669        Grn: 6554
# FLW: 287        FR:  (1.57  %) 285        Yel: 11645
# flw: 58         TO:  (2.21  %) 402        Red: 0
#----------------------------------------------------------------------------
```

```
#=============================================================================
# Q:out       All       BE       DS      Grn      Yel      Red
#-----------------------------------------------------------------------------
PKT         17126        0    17126     6544    10582        0
ED              0        0        0        0        0        0
LD              0        0        0        0        0        0
TX          17126        0    17126     6544    10582        0
#=============================================================================
# Q:in        All       BE       DS      Grn      Yel      Red
#-----------------------------------------------------------------------------
PKT         18199        0    18199     6554    11645        0
ED            386        0      386        0      386        0
LD            669        0      669       83      586        0
TX          17144        0    17144     6471    10673        0
#-----------------------------------------------------------------------------
```

All the queues referenced by the global vector `q_` are dumped, unless the global variable `$op(2DUMP)`, which sould be setted in the topology patch file, contains a list of valid `q_` indexes.

C.1.5.4 Post Process Output

To allow a wider range post process analisys on the simulation data, the following mechanism is implemented: an initialization Tcl script (`.2ftp.POST.init`) contains the code needed to rebuild the collected data vectors; this file is `sourced` in the main post process Tcl script (`2ftp.POST`), who declares those variables as `globals`:

```
#  ------------------------------------------------------
# /                                                      \
#/ PPROC test/2ftp.*                          ___________/
#\_________________________________________/.:nonsns:.

source "./.2ftp.POST.init" ;# init globals

#
# in the previous file you have all except per-flow data
# stored in the following global hash variables
#
global files_               ;# Monitor Params
global cloud_               ;# CLOUD
global web_                 ;# WEB
global ftp_                 ;# FTP
global tot_                 ;# TOT
global PKT_                 ;# Packets Statistic
global stat                 ;# Simulator Statisics
global Model                ;# HTTP Model data
```

This is mainly done to help in separate simulations data collection from its analisys: the next logical step would be to define a set of Tcl macros that extrapolate –in a more convenient manner then the standard shell-filtering approach– the wanted data from the stored vectors, then to source and use that macro set in the main post process Tcl script. The procedure wich convert an hash to Tcl code is ◁`general`▷ `VSAVE`, then the output is flushed by ◁`monitor`▷ `MONITOR`: the `PKT_` vector declared above, containing `DROP` hash data (see Sec. C.1.4.3), has been created with the following call:

```
MONITOR POST [VSAVE "Packets Statistic" "PKT\_" DROP]
```

C.1.6 The "Patch" Mechanism

At the price of coherency with a small set of naming rules, the "patch" mechanism allow wide flexibility in topology scenario definition and DiffServ queue setup. The basic idea is to `source` external fragments of code directly in the body of two ◁`patch`▷ macros: `TOPOLOGY_init` and `DiffServ_init`, which are called in that order during main script initialization. These two functions are just a *container*, which declares the needed global variables:

`ns` *ns* Simulator instance

`dn_` DiffServ Domain Nodes array

`q_` DiffServ Queues vector

`op` options vector

Belowe it is shown how the former macro is defined, being the latter very similar: it can be noticed that the factory topology too have been defined as an external patch file:

```
proc TOPOLOGY_init {} {
  global ns rng verbose
  global dn_ q_ op
    #
    # if $op(topo) is an empty string, use factory topology
    #
    if ![string compare $op(topo) ""] {
        set op(topo) "Tcl.PATCH/test.topo"
    }
    #
    # The patch file is sourced here
    #
    if ![file exists $op(topo)] {
       ERROR "The TOPOLOGY file you specified doesn't exists"
    }
    TVERBOSE  "Sourcing TOPOLOGY from $op(topo) PATCH"
    source $op(topo)
}
```

C.1.6.1 Node Naming Notation

The node naming notation gives just two simple rules:

N.1) Source node of cloud `$i` must be referenced in `dn_(src,$i)` global var

N.2) Destination node of cloud `$i` must be referenced in `dn_(dst,$i)` global var

These rules means simply that if we have one *ns* node at the ingress edge of a domain with a single core router, and two different egress routers:

```
# ingress
set door   [$ns node]

# core
set room   [$ns node]

# egress
set window [$ns node]
set bed    [$ns node]
```

then, supposing to have properly setted the links, if we want cloud of id `Belle` entering by the `door` and leaving from the `window`, we should write:

```
set dn_(src,Belle) $door
set dn_(dst,Belle) $window
```

Cloud `Sebastian` may enter from the `door` to and directly go to `bed` via:

```
set dn_(src,Sebastian) $door
set dn_(dst,Sebastian) $bed
```

This is beacuse ◁`http_init`▷ `HTTP_init` (called by ◁`option`▷ `MAIN_init`) sets up the clouds according to:

```
foreach i [lessof $cloud(num)] {
   src_HTTP_cloud $i $dn_(src,$i)
   dst_HTTP_cloud $i $dn_(dst,$i)
   SLA_HTTP_cloud $i $op(SLA$i)
   create_HTTP_cloud $i
}
```

C.1.6.2 Queue Naming Notation

The queue naming notation gives just three simple rules:

Q.1) All the DiffServ queues in DiffServ domain must be referenced in a global vector of name `q_`

Q.2) The global var `op(INGRESS)` must contain the DiffServ Ingress Edge queue's references list

Q.3) The global var `op(2DUMP)` may contain the list of DiffServ queues to dump, expressed as indexes of `q_` vector; otherwise all the queues referenced in `q_` will be dumped

The rule Q.1) is essential for the coherency of the patching mechanism. The queues definition is a topology matter, so they have to be defined by `TOPOLOGY_init` (which is called previously then `DiffServ_init`); wheter using unidirectional flows only the forward path queues need to be defined: to continue our example, we would write

```
set q_(Welcome) [[$ns link $door $core]   queue]
set q_(Pinelli) [[$ns link $core $window] queue]
set q_(Tired)   [[$ns link $core $bed]    queue]
```

indeed only TCP acknowledgment will flow on the backward path through the dual queues:

```
set q_(SeeYouSoon)   [[$ns link $core   $door] queue]
set q_(Resurrection) [[$ns link $window $core] queue]
set q_(WakeUp)       [[$ns link $bed    $core] queue]
```

In this example, we have only one ingress edge router, so, followin rule Q.2) we would set

```
set op(INGRESS)  $q_(Welcome)
```

while we can observe dumping of all the forward path edges, with repect of Q.3), writing

```
set op(2DROP)  [list Welcome Pinelli Tired]
```

otherwise all the queues will be dumped, according to

```
if ![info exists op(2DROP)] {
    set op(2DROP) [array names q_]
}
foreach qid $op(2DROP) {
        ...
}
```

C.1.6.3 Topology Patch Example

```
#
# 2-HTTP-Clouds Scenario, src <-> E <-> dst
#
#
#      Cloud0 S             Cloud0 R
#        \                   /
#         \   Bw  ---  Bw   /
#      src 0------|E|------0 dst
#         /   RTT ---  RTT  \
#        /                   \
#      Cloud1 S             Cloud1 R
#
#                              - -
#                     ----------\
#___________________/.:nonsns:.\-/
```

To define the simple 2-Cloud HTTP scenario ASCII-drawn above, with a Source Node ⇜⇝ DiffServ Edge Router ⇜⇝ Destination Node topology, the topology patch should just contains the lines:

```
#=====================================================================
#                 Domain Nodes dn_ (global)
#---------------------------------------------------------------------
  set dn_(src,0) [$ns node]     ;# src0 for HTTP-cloud 0
  set dn_(src,1) $dn_(src,0)    ;# and for HTTP-cloud 1 too

  set dn_(R)     [$ns node]     ;# DS router

  set dn_(dst,0) [$ns node]     ;# dst is the common destination
  set dn_(dst,1) $dn_(dst,0)


#=====================================================================
#                Domain Structural Parameters
#---------------------------------------------------------------------
  set BW    $op(bwneck)Mb
  set RTT   10ms

#=====================================================================
#                      Domain links
#---------------------------------------------------------------------
#     link type      from         to         rate  rtt  queue type
#---------------------------------------------------------------------
  $ns simplex-link  $dn_(src,0) $dn_(R)      $BW  $RTT  dsRED/edge
  $ns simplex-link  $dn_(R)     $dn_(dst,0)  $BW  $RTT  dsRED/edge

  $ns simplex-link  $dn_(dst,0) $dn_(R)      $BW  $RTT  dsRED/edge
  $ns simplex-link  $dn_(R)     $dn_(src,0)  $BW  $RTT  dsRED/edge


#=====================================================================
#          Domain queues q_ (global) used in DiffServ_init
#---------------------------------------------------------------------
  #
  # HTTP flows (last 2 queues are for the Backward path's Ack)
  #
  set q_(out)     [[$ns link $dn_(R)     $dn_(dst,0)] queue]
  set q_(in)      [[$ns link $dn_(src,0) $dn_(R)]     queue]
  set q_(Bin)     [[$ns link $dn_(dst,0) $dn_(R)]     queue]
```

```
set q_(Bout)    [[$ns link $dn_(R)     $dn_(src,0)] queue]

set op(INGRESS) [list $q_(in)]  ;# update stats
set op(2DUMP)   [list out in]   ;# dumping queues idx
```

C.1.6.4 DiffServ Patch Example

```
#
# 2-HTTP-Clouds Scenario, src <-> R <-> dst
#
#   Cp  MinTh MaxTh  E
# ----------------------
#   10    20   40   0.02
#   11     5   20   0.20
# 12-00    -   -     -
#                          ----------^\^
#_________________________/.:nonsns:.\o/
```

The natural complement to the topology patch example presented in previous section, to produce the above two color marker oriented RED settings the DiffServ patch file should be similar to:

```
foreach qid [array names q_] {

   #=====================================================================
   #                       DiffServ PHB Entries
   #---------------------------------------------------------------------
     $q_($qid) meanPktSize $op(psize) ;# for RED calculations
     $q_($qid) set numQueues_ 1       ;# one AF service
     $q_($qid) set NumPrec 2          ;# with Green or Yel precs.
     $q_($qid) addPHBEntry 10 0 0     ;# Grn
     $q_($qid) addPHBEntry 11 0 1     ;# Yel
     $q_($qid) addPHBEntry 00 0 1     ;# BE

   #=====================================================================
   #                     DiffServ RED Parameters
   #---------------------------------------------------------------------
     $q_($qid) configQ 0 0 20 40 0.02
     $q_($qid) configQ 0 1 10 20 0.10

   #=====================================================================
   #                    DiffServ HTTP Cloud Setup
   #---------------------------------------------------------------------
     foreach cid [lessof $cloud(num)] {
         addPolicyEntry_HTTP_cloud $q_($qid) $cid
     }
}
```

C.2 HTTP Simulation Performances

C.2.1 System Memory Consumption

The following table lists *approximately* the memory consumption of simulation sessions with various memory conservation solutions. The bottleneck has been determined to be the routing table, which is $O(n^2)$ for flat routing. In addition, n (number of routing entries) is always 2^k. That is why we observe roughly the same routing table size for 513-node and 1024-node simulations ($k = 10$).

Nodes	System Memory Consumption (MB)			
	Connectivity ∼1.8	Delay Bind	Hierarchical Connectivity	∼3–4
512-1023	16	10	—	
1024-2047	46	40	(1040 nodes)	16
2048-4095	180	160	(2080 nodes)	40
4096-8191	720	640	(5120 nodes)	169
8192-16384	2886	2560	(10075 nodes)	1049

Table C.3: System Memory Consumption

C.2.2 System Time Consumption

In the need of simulate HTTP connections on wide network scenarios, we'll have to define a reasonable analytical model that let us preliminary investigate simulation's parameters such as number of event and in-flight packets per unit of time. This is done with the intent to find out the largest viable topology with regard of today system's possibilities.

C.2.2.1 Analitical Model

Definitions

$$\begin{cases} B & \text{link speed [bit/s]} \\ \lambda & \text{average flow length [bit]} \\ \rho & \text{target link utilization} \\ \tau_f & \text{new flow interarrival time} \\ N & \text{average number of active flows} \\ MSS & \text{packet size} \\ RTT & \text{two-way propagation delay} \\ n_f & \text{number of in-flight packets in a link} \\ t_p & \text{packet transmission time} \end{cases}$$

Resulting average number of flows necessary to feed a link which outputs $\rho \cdot B$:

$$N = \frac{1}{\tau_f} = \frac{\rho \cdot B}{\lambda}$$

with $B = 2.4$ Gbps, $\lambda = 10$ kBytes and $\rho = 0.8$ we have:

$$\begin{cases} N \simeq 27000 \text{ flows} \\ \tau_f \simeq 40\mu s \\ 1/\tau_f = 25000 \text{ event}/s \end{cases}$$

being $1/\tau$ the scheduling rate of new flow events. The number of in-flight 1000-byte packets in a link with $RTT = 100$ *ms neglecting the queueing delay*, being $MSS = 1000$ Bytes, is

$$n_f = B \cdot \frac{RTT/2}{MSS} = 15000 \text{ packets}$$

actually, in the case of TCP traffic, the number of in-flight packets is (roughly) twice as much:

$$\begin{cases} n'_f = 30000 \text{ packets} \\ \tau_p = MSS/B = 3.3\mu c \\ 1/\tau_p \simeq 300000 \text{ event}/s \end{cases}$$

where $1/\tau_p$ represents the scheduling rate of new packet transmission.

It must be pointed out that the number of packets affects the memory consumption; furthermore, the larger the number of in-flight packets, the longer the scheduler will take to scan the scheduler queue while inserting a new event (i.e., reception of a packet at an intermediate node).

The aggregate $1/\tau_a$ scheduling rate, defined as the sum of new flows and new packets scheduling rate would then be

$$\frac{1}{\tau_a} = \frac{1}{\tau_f} + \frac{1}{\tau_p} = 325000 \text{ event}/s$$

C.2.2.2 General System Performances

We ran a few sample simulations on an stripped-down, simplified scenario: each node has only one FTP application attached and the network domain is *not* DiffServ compliant. The performance indices we obtained ought to be scaled up by X factor when a full-blown DiffServ scenario (including schedulers, markers, queue monitors, etc.) is simulated.

Scenario: N TCP connections on a bottleneck link; max cwnd for each connection set to a value that allows the aggregate traffic to fill the pipe at any time

Workstation specs: AMD-K6-II, 350Mhz, 128MB RAM

CPU Setup time: CPU time needed to create the objects in the scenario (mainly, the *ns* agents and their connections)

CPU Session time: CPU time needed to simulate 10 seconds of network operations once ended the initialisation process

Observing Tab. C.4[1], we gather that the CPU simulation time exhibits a logaritmic growth with respect to the number of connections: as it will be shown later, this is true only when dealing with FTP traffic in a restrained scenario. The setup time growth, on the other hand, appears to be linear, at least when the system is not running out of memory; this result will hold also for HTTP flows on a DiffServ domain.

C.2.2.3 Agent Creation Rate

As explained earlier in Sec. C.1.3.1, each HTTP flow use a tuple of `TCP`, `TCPSink` and `FTP` ns objects: so we can roughly measure the creation time of an HTTP flow source by simply timing the object creation, neglecting both *ns* connections startup and attaching requirements and Tcl layer mapping and referencing costs which are proper to

[1] Our workstation ran out of memory on the last experiment, soon after setting up all the required connections, so the simulation time is for the latter case only an estimate based on previous simulations results.

	System Time Consumption		
Nodes	Setup	Session	Total
1	∼0	∼0	<1
5	∼0	1	1
10	1	1	2
100	7	2	9
500	31	3	34
1000	63	4	67
5000	331	16	347
		...	
27000	3900	$\sim 60^{\dagger}$	$\sim 3960^{\dagger}$

† *Estimate based on previous simulations*

Table C.4: System Time Consumption

```
new Agent/TCP
new Agent/TCPSink
new Application/FTP
```

The creation of a thousand of object triplet needed by a correspondant thousand of contemporary HTTP flows requires ∼12MB of memory and nearly 60 seconds to be completed. Of course, no packets were sent during this step: we just measured system requirement via the following loop:

```
new Simulator
set TIC [exec date +%s]
foreach k [upto 1000] {
    if [expr !($k%50)] {
        set TOC  [expr [exec date +%s] - $TIC]
        puts "RATE= [expr $k/$TOC]"
    }
    new Agent/TCP; new Agent/TCPSink; new Application/FTP
}
```

Which tell us that –for a limited number of flows– the object creation rate remains constant, allowing the instatiation of 18 TPC/FTP/Sink triplets every second (system time consumption was of 0.24 seconds during the whole run).

However, to get more realistic results, we should consider also *ns* agents connection and DiffServ setup time as well: implementating loop similar to the previous one, we gather that we can create, in the need, sligthly less than ∼14 HTTP flows per second. The DiffServ/BestEffort scale factor is ~ 1.28, thus building a DiffServ scenario requires $\sim 28\%$ more CPU time that a plain Best Effort one: just to give a rough idea, 6 minutes are required minutes to set up a scenario of 5000 flows, and 65 minutes when 27000 flows are created; from Fig. C.4, where both the sources creation rate and creation time length are plotted against the number of HTTP flows created, we gather that as the number of HTTP flows grows, their creation rate will decrease.

C.2.2.4 CPU Session Time

Once the scenario has been created, we consider the CPU time needed to run ∼8.5 *ns* seconds of simulated network traffic, comparing FTP issues to HTTP ones. The simulated time has been choosen accordingly to last HTTP flow ending time for the HTTP triplet $t_{start} = 1$s, $t_{stop} = 5$s, $t_{halt} = 10$s; results are shown in Tab. C.5.

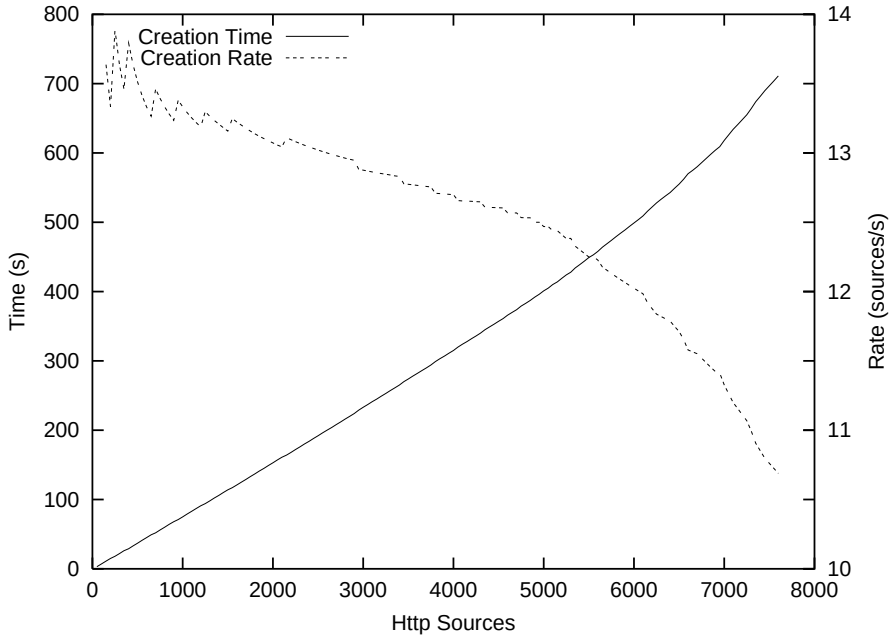


Figure C.4: DiffServ HTTP Source Creation

FTP		Timing (sec)		
Sources	Flows	Setup	Session	Overall
170	170	11	2	13
740	740	47	4	51
1470	1470	94	5	99
7460	7460	515	30	545

HTTP		Timing (sec)		
Sources	Flows	Setup	Session	Overall
170	240	12	12	24
740	1000	54	46	100
1470	2200	108	107	215
7460	11100	549	603	1125

Table C.5: DiffServ HTTP vs FTP Performances

From the previous table, we can remark two interesting points: First of all, the HTTP/FTP session time scale factor is ∼20, as Fig. C.5 depicts, while the HTTP/FTP setup time scale factor remains close to the unity. Moreover, both of the applications seem to have a time cost which is linear on the sources number; in Fig. C.6 the setup time and session time are plotted against the FTP/TCP/TCPSink tuples number for both FTP and HTTP application. In the case of HTTP application –in which the number of sources at a given time equals the greatest number of contemporary active flows up to that simulation istant– we can say that the CPU simulation time

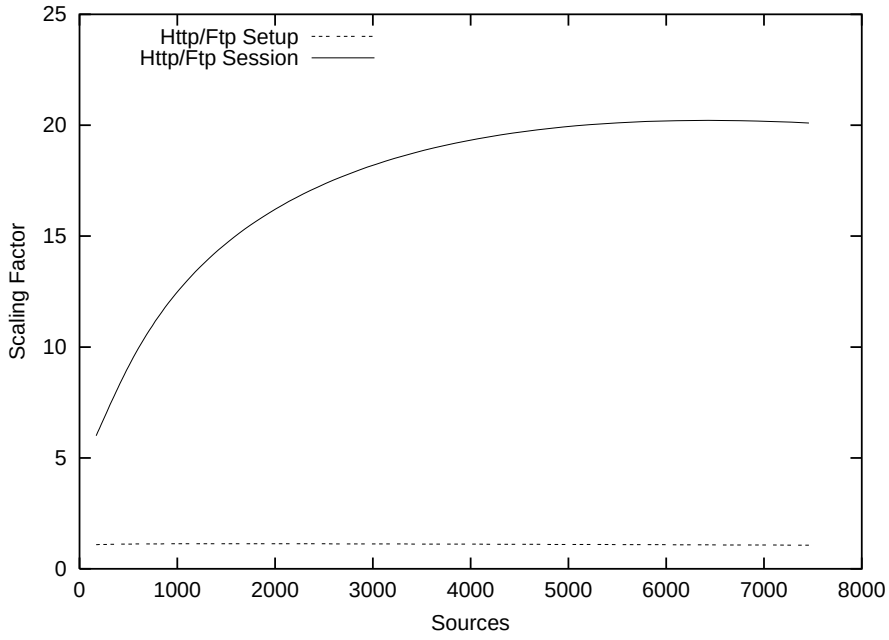


Figure C.5: HTTP over FTP Scale Factor

is roughly linear on HTTP flows number.

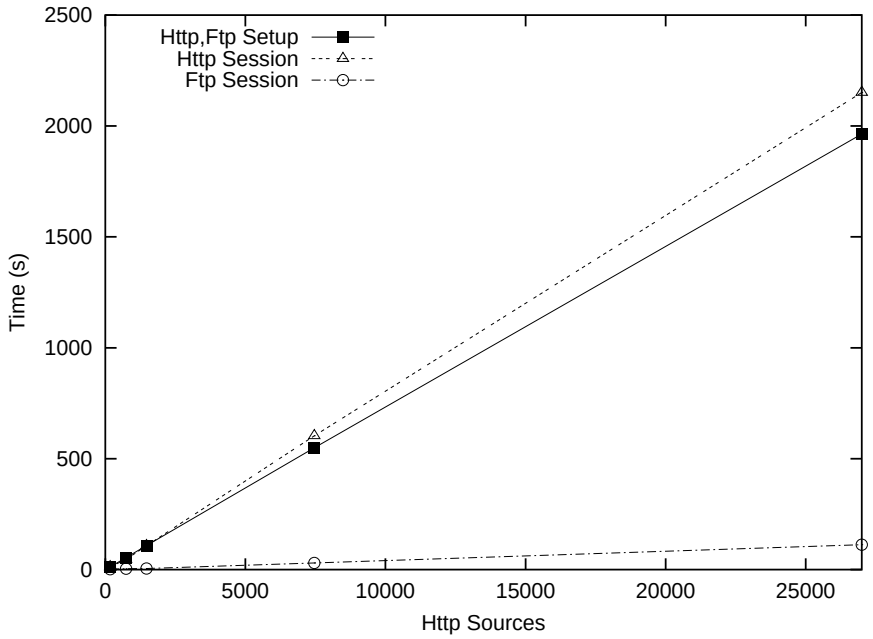


Figure C.6: HTTP vs FTP Session Time

Finally, observe that the number of sources totally created represents the maximum number of HTTP flows contemporary active. Obserbing table Tab. C.6, reporting simulation issues for different simulation timings and for different bottleneck sizes , we notice that the ratio between

Max Flows (simultaneous) variable and Tot Flows (during the whole simulation) is ∼1.5; moreover, the two per-flow time ratios variables listed are almost constant too: Session/Flow ≃ 0.65s and Overall/Flow ≃ 0.11s.

		$(t_{start},t_{stop},t_{halt})$			
		(1,10,20)		(1,15,30)	
Bottleneck Size		10 Mbps	50 Mbps	10 Mbps	50 Mbps
ns ended at		13.31	16.32	17.22	21.62
Flows	Max	346	1660	484	2482
	Tot	491	2517	719	3721
Time	Overall	59s	290s	80s	420s
	Sources	24s	118s	34s	177s
	Session	35s	172s	46s	243s
Max/Tot Flow		1.41	1.51	1.48	1.50
Session/Flow		0.71s	0.68s	0.69s	0.65s
Overall/Flow		0.12s	0.12s	0.11s	0.11s

Table C.6: Simulator Peformances: Viables Scenarios

Even if we raise –indiscriminately– the bottleneck size, we observe their values stability, as table Tab. C.7 shows for $(t_{start},t_{stop},t_{halt})$=(1,10,20) :

Bottleneck Size		10 Mbps	50 Mbps	100 Mbps	500 Mbps	1 GBps
ns ended at		13.31	16.32	15.16	6.61	3.47
Flows	Max	346	1660	3264	10914	9702
	Tot	491	2517	4985	14707	11630
Time	Overall	59s	290s	570s	9300s ‡	1500s
	Sources	24s	118s	251s	...	693s
	Session	35s	172s	319s	...	807s
Max/Tot Flow		1.41	1.51	1.53	1.34 †	1.20 †
Session/Flow		0.71	0.68	0.64	...	0.67
Overall/Flow		0.12	0.12	0.11	0.63	0.13

‡ This result is due to intensive hard disk swapping

† These results are due to *ns* simulation early interruption

Table C.7: Simulator Peformances: Raising Bottleneck Size

A final *remarque* can be made about prevision of simulation duration: for a 10 Mbps bottleneck, one must count –as upper bound– ∼5.5s for each *ns* second of simulation, that is, (slightly more than) half of an hour for every minute of simulated newtwork traffic. Extending the bottleneck size, with a 50 Mbps bottleneck *ns* needs ∼29s per unit of time, while for a 100 Mbps one ∼47s would be required, which is roughly linear on the bottleneck linerate: as an estimate, to simulate one minute of HTTP traffic flows over a 500Mbps bottleneck, nearly 4 hours of CPU are needed. This hypotesis holds only wheter system memory requirement can be provided: the 500 Mbps column of table Tab. C.7 states that a duration many times longer would be needed if swapping was required.

C.2.2.5 Source Provisioning Strategy

In this section we'll discuss some considerations about HTTP "Ideal Routers" and their setting as `-routno` option of the main `HTTP.tcl` script. These routers, acting as traffic multiplexer, have the same meaning that subdirs in a directory: they allows to attach more HTTP flows to a cloud, since each node has a maximum number of objects that it can handle, knowing that each flow needs at least 3 obj. They have been developed because in earlier in earlier versions of *ns* the addressing mechanism was a limitation but this isn't true anymore. Being

$$\begin{cases} ns \text{ HTTP Start Time} & 1\ s \\ ns \text{ HTTP Stop Time} & 5\ s \\ ns \text{ HTTP Halt Time} & 10\ s \\ \text{Bottleneck rate} & 10 \text{ Mbps} \end{cases}$$

Last HTTP started at	4.943	4.332	4.195	4.387
ns Simulation ended at	8.364	8.676	7.865	8.732
Sources Per-Router	10	10	10	10
Router Per-Cloud	1	2	5	10
Initially Created Sources	10	20	50	100
Run Time Created Sources	154	150	120	80
Totally Used Sources	164	170	170	180
Reused Sources	81	75	79	63
Number of flows	245	245	249	243
In flight Pkts	5043	4959	4969	5103
Setup Time	1s	2s	4s	8s
Session Time	20s	22s	20s	15s
Overall (HTTP only)	21s	24s	24s	23s
Overall (with FTP)	28s	28s	26s	29s

Table C.8: Under Provisioned Case Performances

Thus the global simulation time is not influenced by HTTP cloud initial under-provisioning, resulting only in a variable fraction of the simulation time between initialization and analisys.

Router Per-Cloud	10	10	10	10
Sources Per-Router	20	30	40	50
Sources per-Cloud	200	300	400	500
Setup Time	15s	22s	29s	36s
Session Time	19s	19s	12s	11s
Overall Time	34s	41s	41s	47s

Table C.9: Over Provisioned Case Performances

The results for the over-provisioned cases are shown in Tab. C.9, where we have roughly

$$\begin{cases} \text{Number of flows} & \sim 240 \\ \text{In flight Pkts} & \sim 5000 \\ \text{Last HTTP started at} & \sim 4.3 \\ \text{ns Simulation ended at} & \sim 8.7 \end{cases}$$

One can conclude that over-provisioning a scenario doesn't implies any efficency gain: altough it would be interesting to re-use instanced simulations objects, this is neverthless impossible after `$ns halt` command.

Finally, it must be pointed out that the use of hundreds of routers negatively influence `dsPolicy.cc`. Set `MAX_POLICY` to a quite upper bound, but remeber that the as the entry table is static, you won't be able to get as many entries as wanted, but rather you will be surely able to get bogus simulatio results and segmentation fault errors.

C.2.3 System Disk Space Consumption

In this section the light-weight trace mechanism alternative to standard *ns* one, will be analized in the details of its implementation as well as its performances, expressed in terms of the size of the trace file produced.

C.2.3.1 MONITOR Implementation

The *ns* command `nam-traceall` is as pre-powerful as post-painful: that's why in ◁`monitor`▷ script functions have been implemented. Basically two commands,

```
MONITOR $channel_keyword $str
TRACE   $trace_keyword   vector
```

accomplish both the tracing and dumping functions (see Sec. C.1.5.1): the former's task is just to flush the output string `$str` over multiple channels parsing the `$channel_keyword` string, while the latter care to format the data contained in `$vector` accordingly to the kind of trace specified via `$trace_keyword` string.

The output flusing over multiple channels is implemented via ◁`general`▷ `M-puts` which simply implement a `puts` loop over the list returned by `MONITOR_channels` parsing function:

```
proc MONITOR {what str} {
   M-puts [MONITOR_channels $what] $str
}
```

The `vector` passed to the `trace` function is build by two ◁`core`▷ routines:

```
retrieve_FTP_vars $TCP $fid vector
retrieve_WEB_vars $cid $fid vector
```

they both call the following one, who gets from *ns* `$TCP` agent the HTTP and DiffServ binded variables:

```
retrieve_TCP_vars $TCP vector
```

The `MONITOR` and `TRACE` call is usually combined, both for tracing and dumping purposes, as:

```
MONITOR where [TRACE what vector]
```

◁`core`▷ `finish_HTTP_flow` uses the monitor that way:

```
retrieve_WEB_vars $cid $fid this
MONITOR "web$cid" [TRACE "web" this]
```

the keyword `web$cid` will instruct the monitor to write to the per-cloud trace file and to the web-traffic one, while with the next call:

```
MONITOR WEB [TRACE WEB this]
```

the informations would be putted to the overall dump file too, and finally:

```
MONITOR ALL [TRACE HDR dummy]
```

will produce the traced variable header (see sec Sec. C.1.5.1 to be printed to all the trace files. For further details on the mapping between the `TRACE`'s keywords and the traced variables, and between `MONITOR`'s one and the trace files, please refers to ◁`monitor`▷ script code.

C.2.3.2 MONITOR Performances

The monitoring mechanism effectiveness can be appreciated by observing the terribly light-size files you'll have to parse reported in table Tab. C.10 for different bottleneck sizes. Just to make an example, if you're dealing with large-scale web scenarios, lets saying a 500Mbps bandwidth bottleneck to fill-up with thousands (10k flows for 5sec of simulations) of HTTP flows and some FTP background sources, you will find lovely to parse a 10k lines where all the data of interest are already calculated.

Bottleneck Size (Mpbs)	10	50	100	500
Totally Created Sources	170	740	1470	7460
Totally Activated Flows	240	1000	2200	11100
Per-Flow Trace File Size (KB)	25	100	240	914

Table C.10: Trace File Size

As a complement to the Tab. C.10, it will be said that disk consumption rates are roughly 500 bytes/FTP-source, 1000 bytes/HTTP-cloud, 130 bytes/HTTP-flow. To make a comparaison with `namtrace-all` feature, we'll shortly analize the disk space consumption of the simulation example scenario of Sec. C.1.5. From Tab. C.11, it can be gathered that `MONITOR` processes 385 times less events, resulting in 475 times lighter-weighted size than *ns* one (115KB vs 52MB); furthermore, the gain factor raises to 1033 when counting only trace and dump (thus neglecting the scenario descriptions, queue dumping and post process Tcl files).

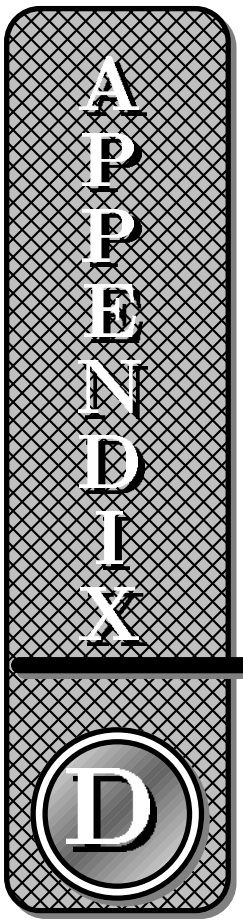
	`MONITOR`	*ns*
(Flows,Packets)	(287, 18199)	
Event	1449†	554054
Event/Flow	5	1930
Event/Pkts	0.08	30.5
Bytes/Event	80	100

† The number of lines of the whole `MONITOR` file set

Table C.11: `MONITOR` vs `namtrace-all` Comparaison

C.3 HTTP.xxx.tcl Procedure List

◁http_init▷	◁finish▷
HTTP_model_init {}	HTTP_run {}
HTTP_web_init {}	finish_FTP {}
HTTP_TCP_init {}	finish_HTTP {}
HTTP_init {}	HTTP_bye-bye {}
◁monitor▷	**◁core▷**
MONITOR_init {}	start_HTTP_flows {cloudid}
MONITOR {what str}	activate_FTP {cloudid pkt}
MONITOR_channels {what}	DiffServ_HTTP_packet {id DSCP EV}
MONITOR_close_all {}	finish_HTTP_flow {FTP TCP SINK}
MONITOR_flush_all {}	retrieve_TCP_vars {TCP VEC}
TRACE_init {kind VEC}	retrieve_FTP_vars {cid fid VEC}
TRACE {kind VEC}	retrieve_WEB_vars {cid fid VEC}
◁cloud▷	**◁mapping▷**
init_HTTP_cloud {}	pop_HTTP_freesource {cid}
SLA_HTTP_cloud {args}	push_HTTP_freesource {cid fid}
src_HTTP_cloud {args}	HTTP_classify {cid tcp ftp snk fid}
dst_HTTP_cloud {args}	FTP_classify {cid tcp ftp snk fid}
addPolicyEntry_HTTP_cloud	get_HTTP_fid {type agent}
{queue cloudid}	get_HTTP_cloud {fid}
check_HTTP_cloud {this}	get_HTTP_agent {fid agent}
create_HTTP_cloud {args}	get_HTTP_DiffServ {full_id}
attach_HTTP_cloud {args}	is_HTTP_fid {fid}
create_HTTP_source {id}	get_HTTP_app {fid}
◁patch▷	
DiffServ_init {}	TOPOLOGY_init {}
◁option▷	
MAIN_init {}	OPTION_init {}
GLOBAL_init {}	usage {}
◁queues▷	
QUEUE_dump {}	sim_CLOUD_results {cid}
get_QUEUE_stat {qid}	sim_OVERALL_results {str VEC}
◁general▷	
M-puts {chnList STR}	VERBOSE {str}
tic {args}	TVERBOSE {str}
toc {args}	BVERBOSE {str}
lessof {max}	VVEC {name vec}
range {min max}	ViVEC {name id vec}
ifexists {a}	VSAVE {title name vec}
isint {a}	VPUT {title name vec}
getopt {argc argv VEC}	VVERBOSE {name vec}
ERROR {str}	BOXED {str}
DEBUG {str}	TITLED {str}

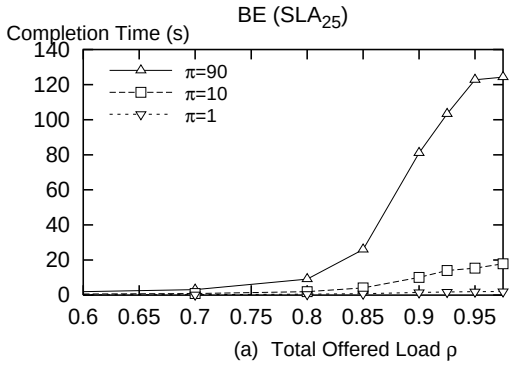


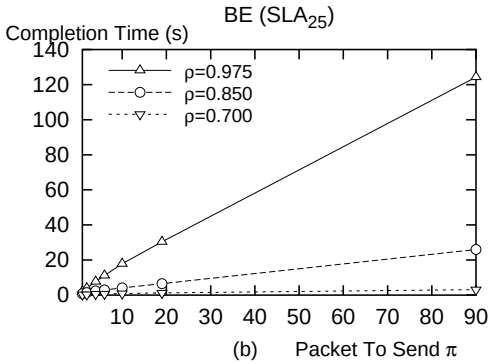
Simulation Results

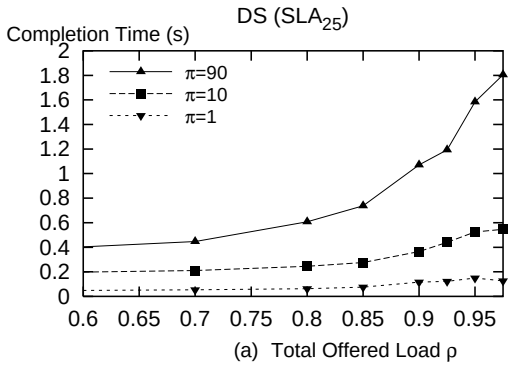
Short Lived HTTP Flows

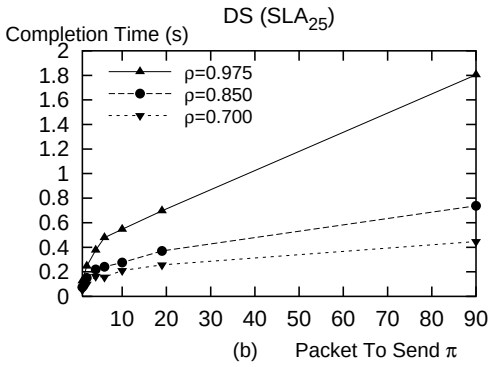
D.1 SLA_{25} Simulations

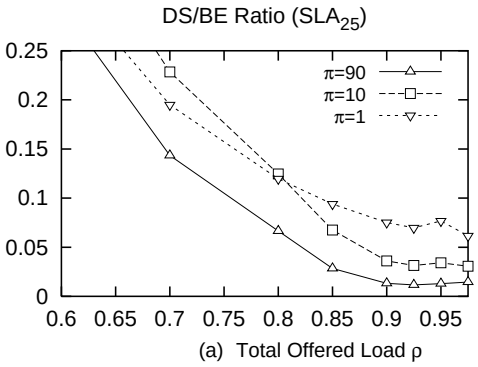
D.1.1 Completion Time (HTTP Traffic Only, SLA_{25})

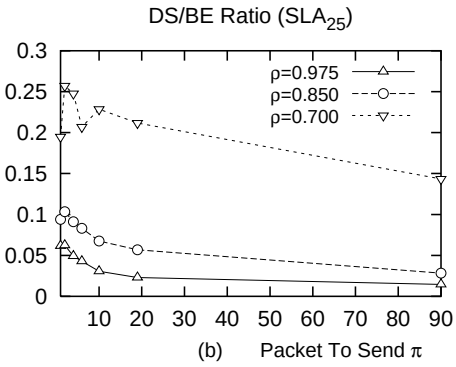




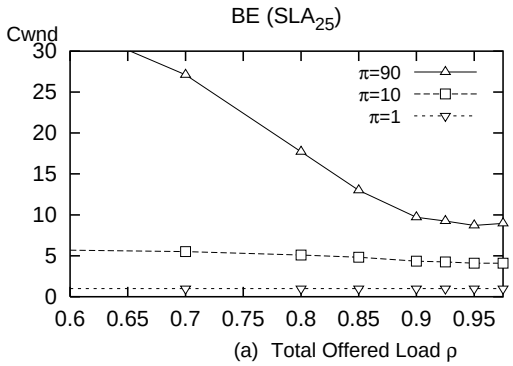


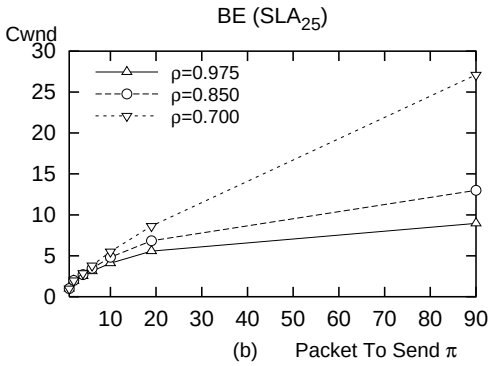


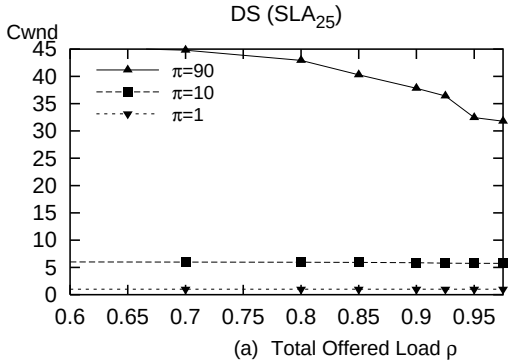


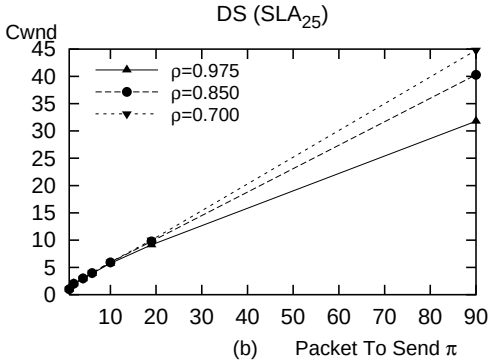


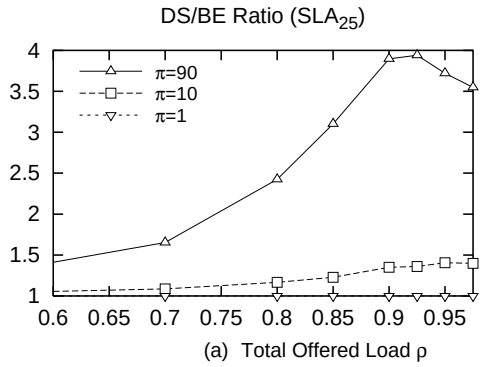
D.1.2 Congestion Window (HTTP Traffic Only, SLA_{25})

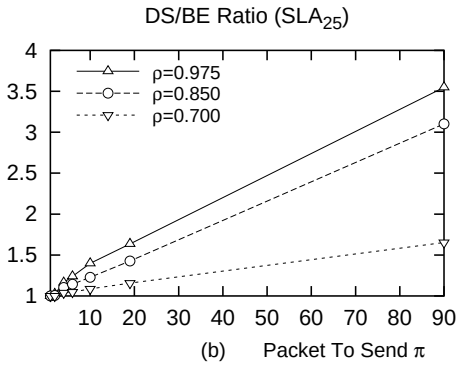




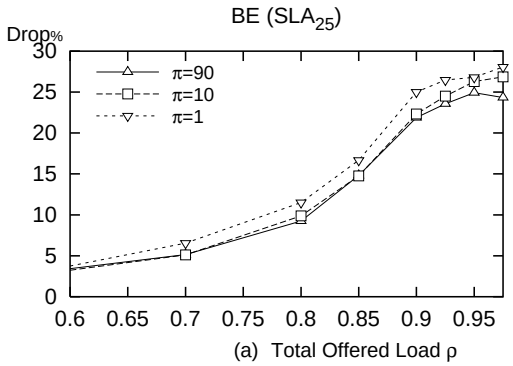


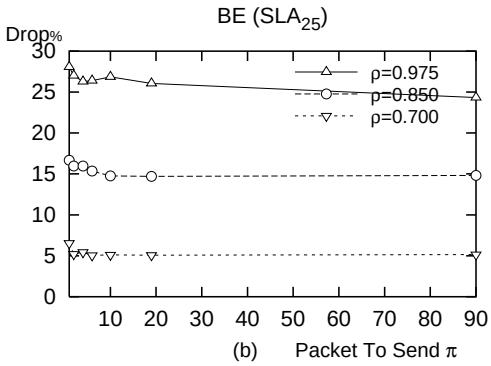


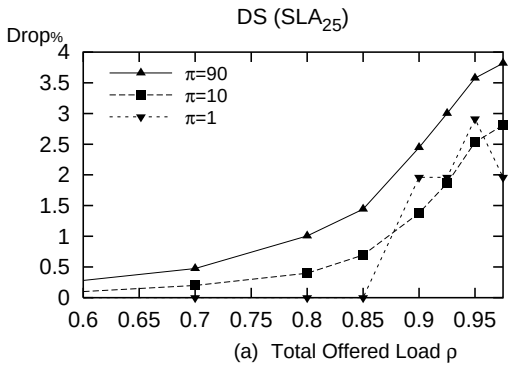


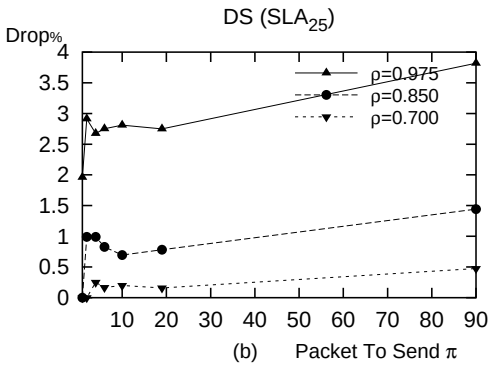


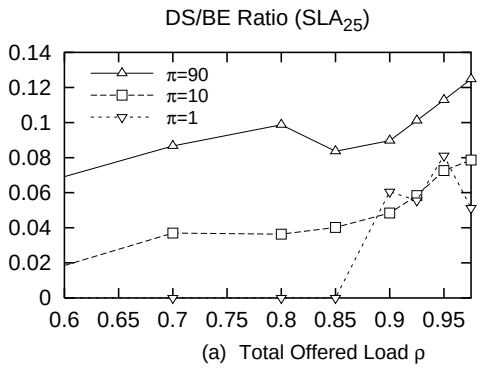
D.1.3 Packet Drops (HTTP Traffic Only, SLA_{25})

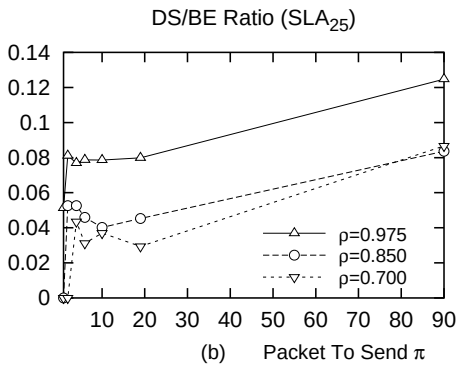




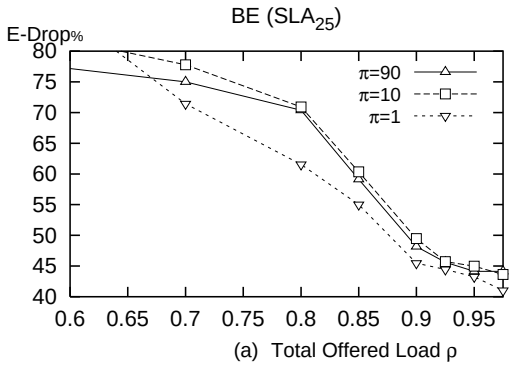


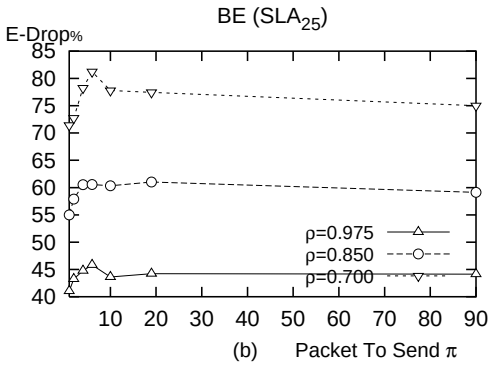


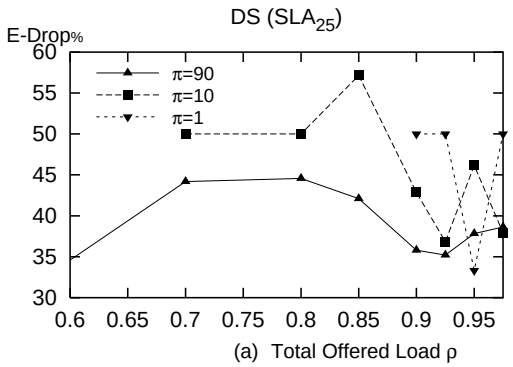


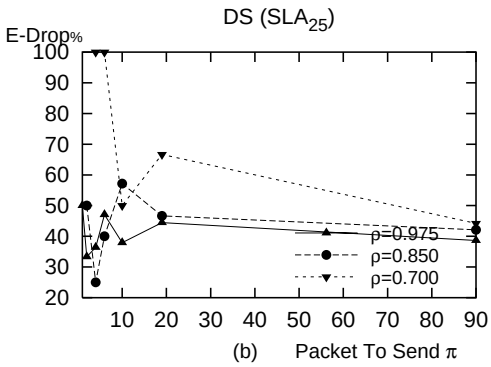


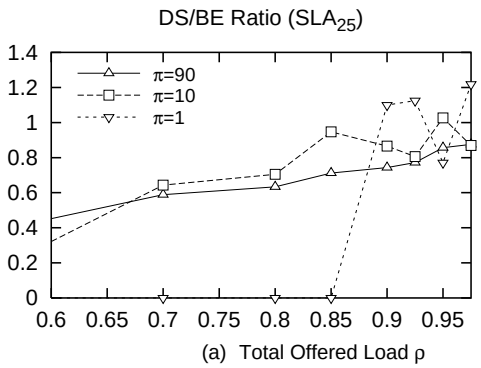
D.1.4 Early Drops / Packet Drops (HTTP Traffic Only, SLA_{25})

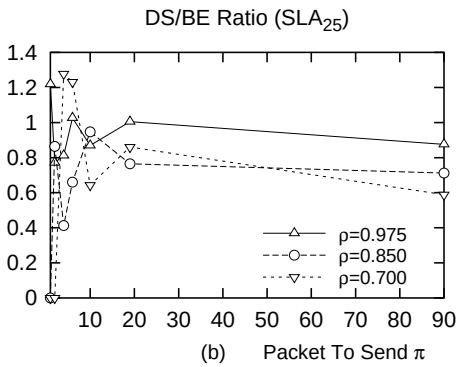




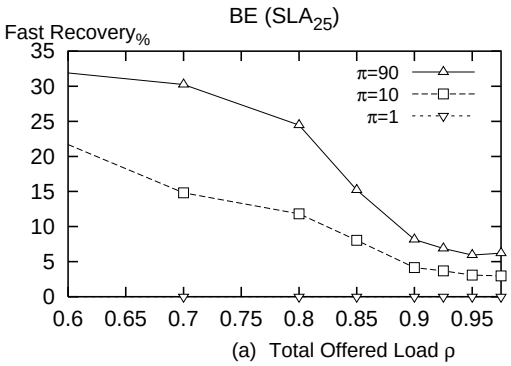


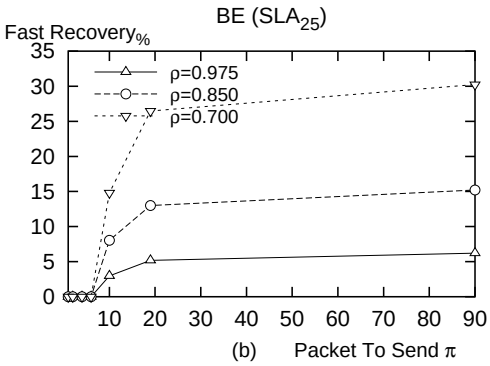


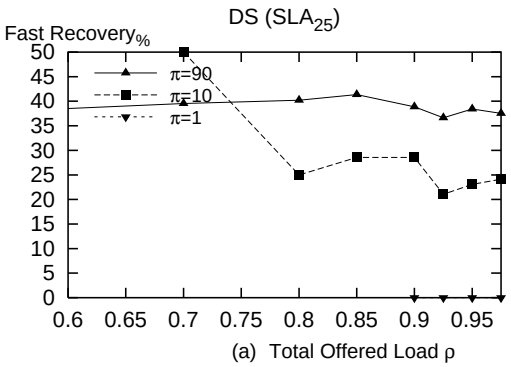


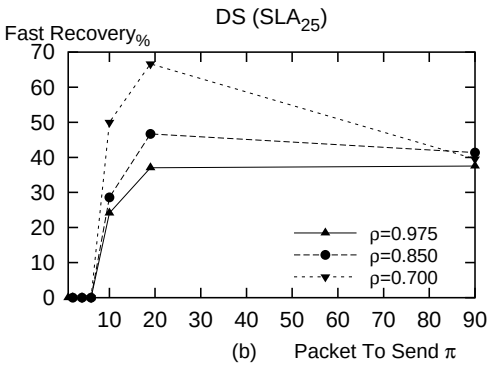


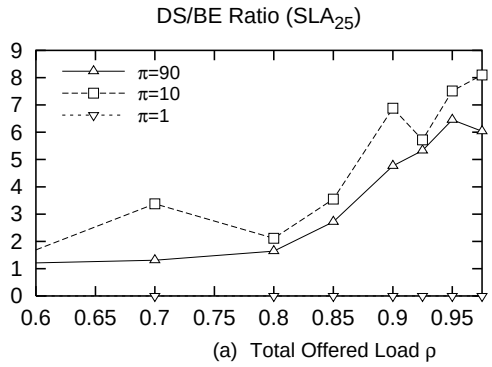
D.1.5 Fast Recovery (HTTP Traffic Only, SLA_{25})

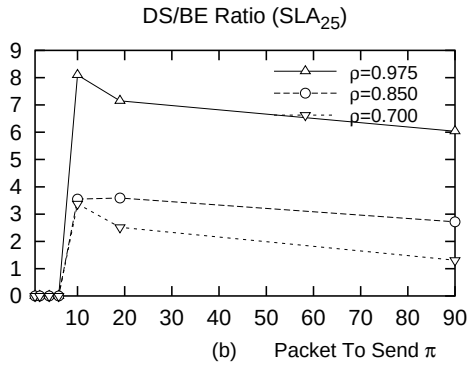




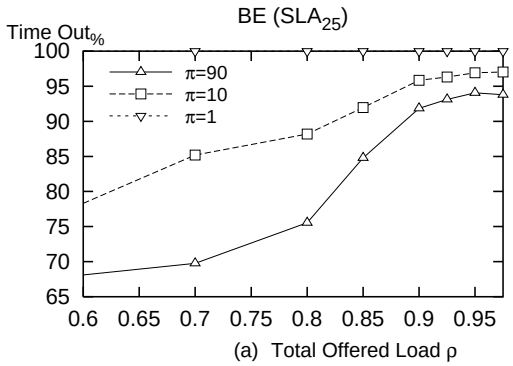


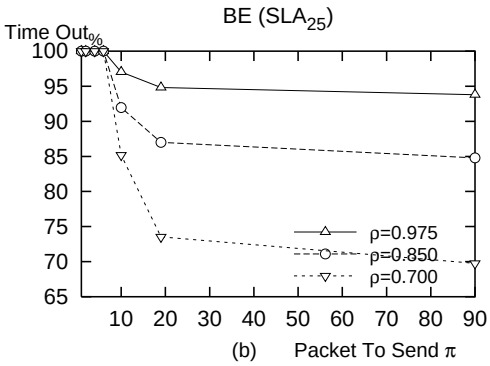


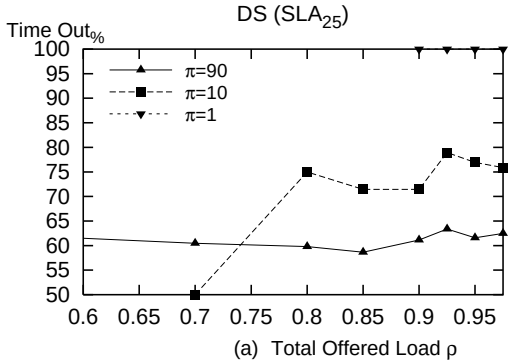


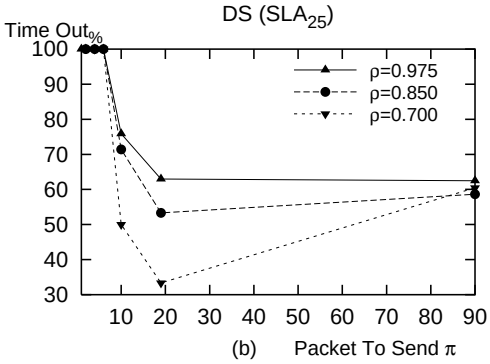


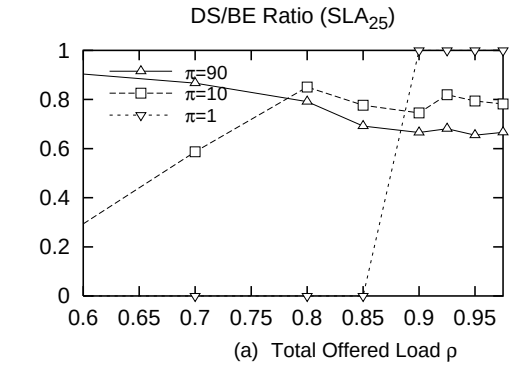
D.1.6 Time Out (HTTP Traffic Only, SLA_{25})

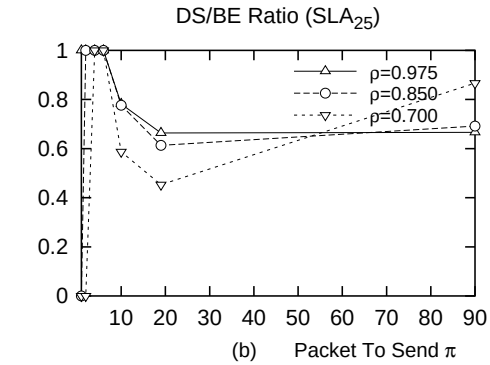






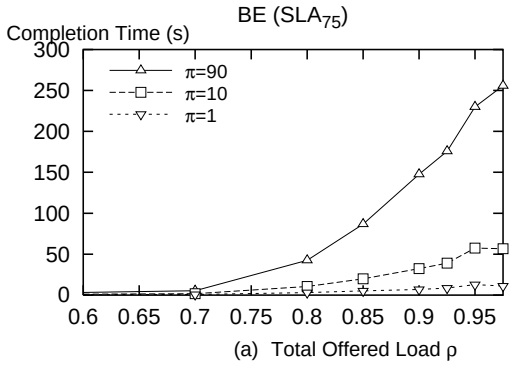


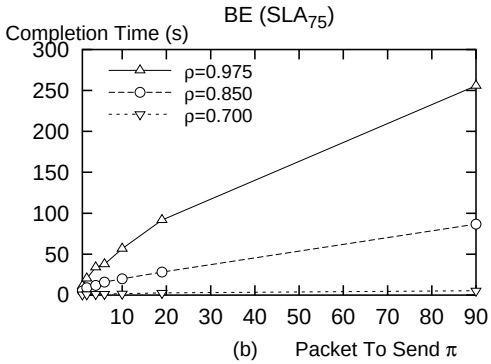


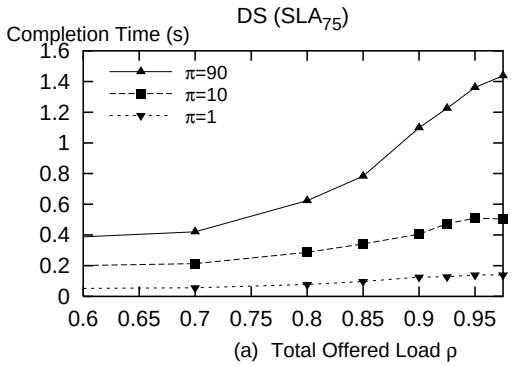


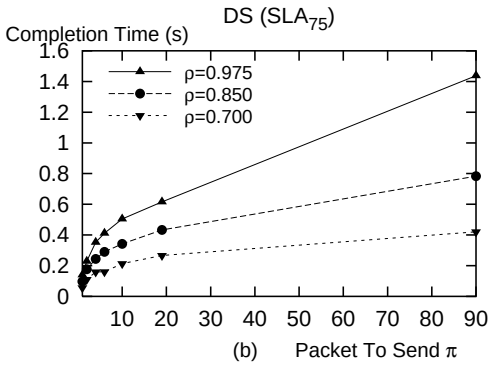
D.2 SLA$_{75}$ Simulations

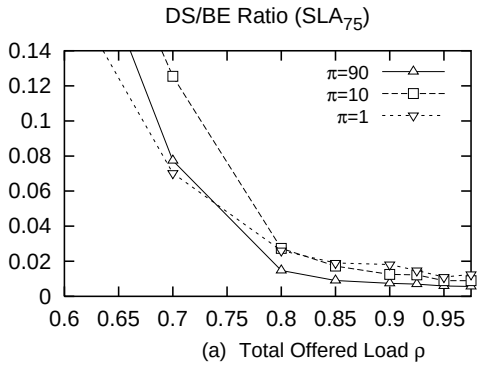
D.2.1 Completion Time (HTTP Traffic Only, SLA$_{75}$)

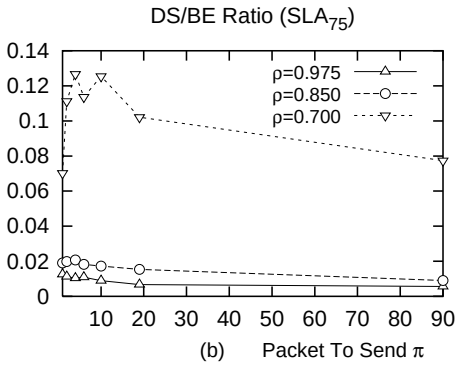




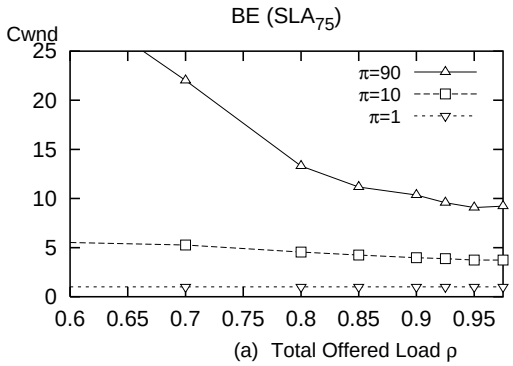


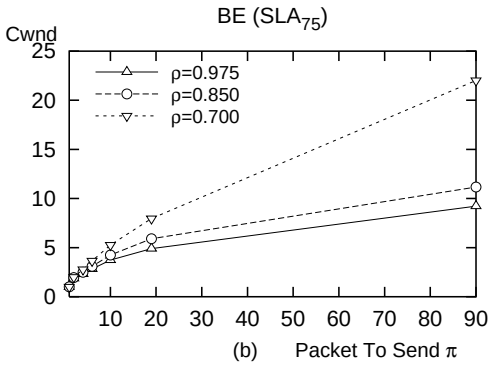


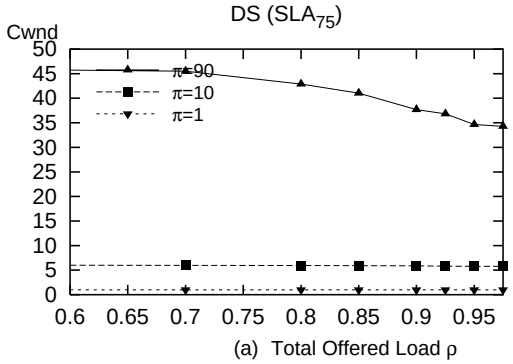


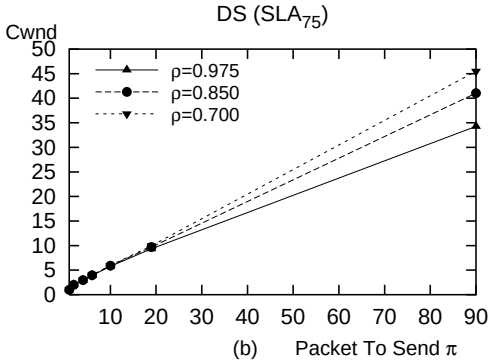


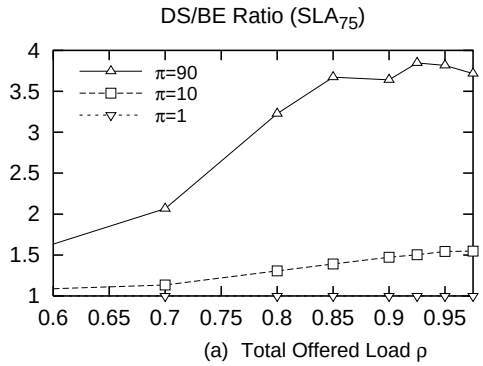
D.2.2 Congestion Window (HTTP Traffic Only, SLA_{75})

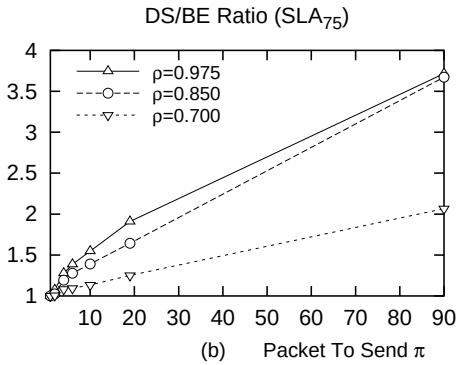




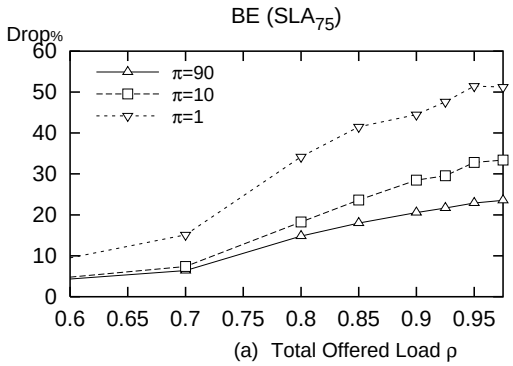


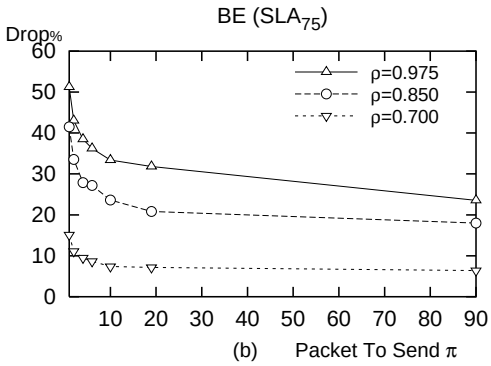


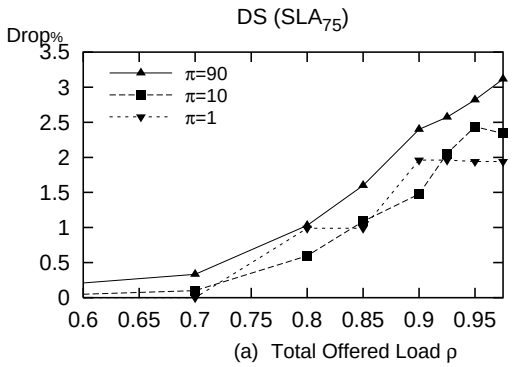


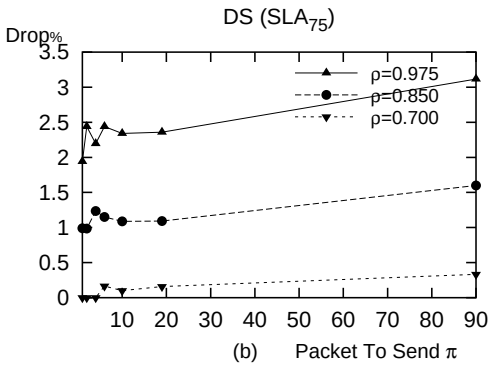


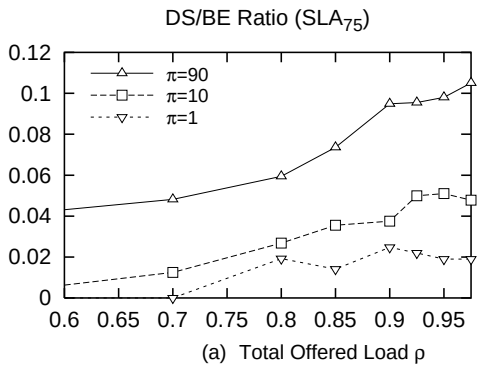
D.2.3 Packet Drops (HTTP Traffic Only, SLA_{75})

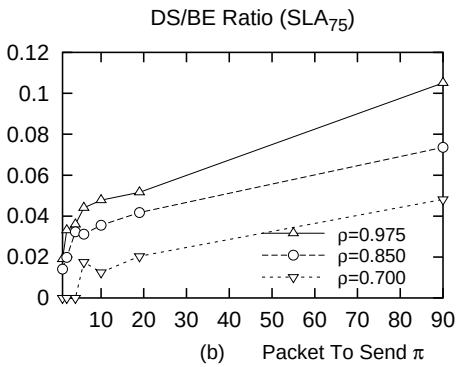




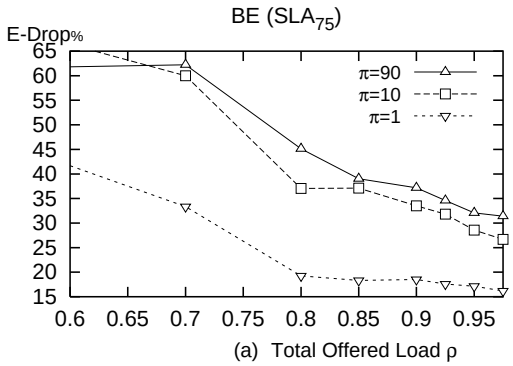


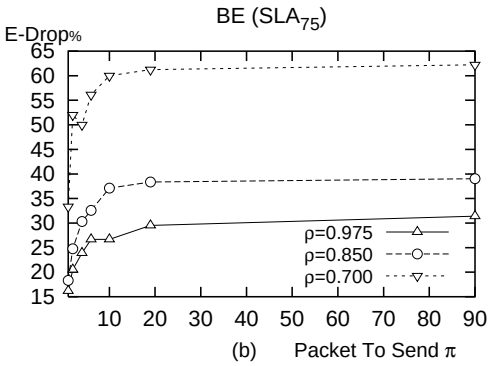


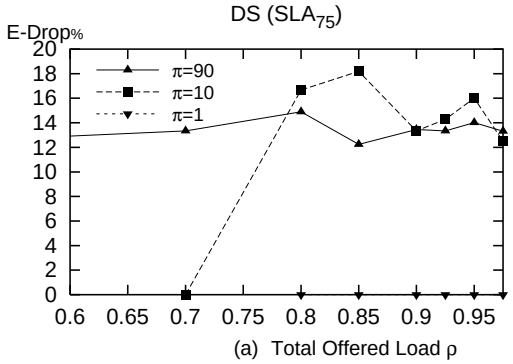


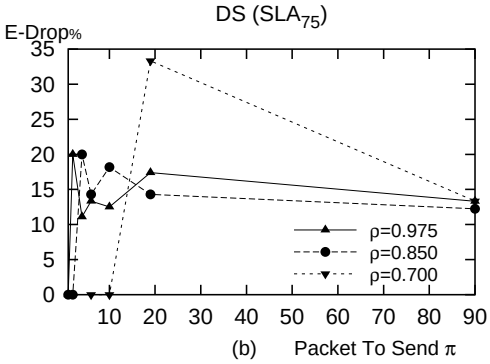


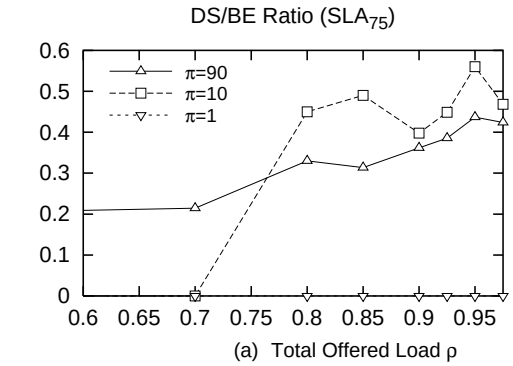
D.2.4 Early Drops / Packet Drops (HTTP Traffic Only, SLA_{75})

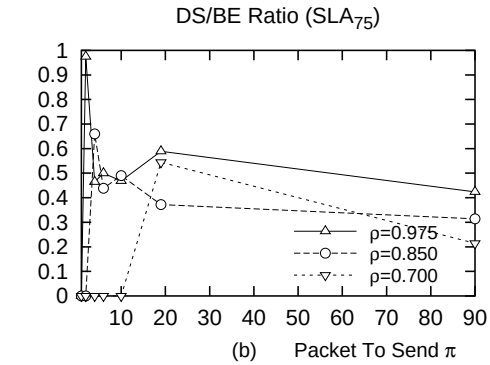




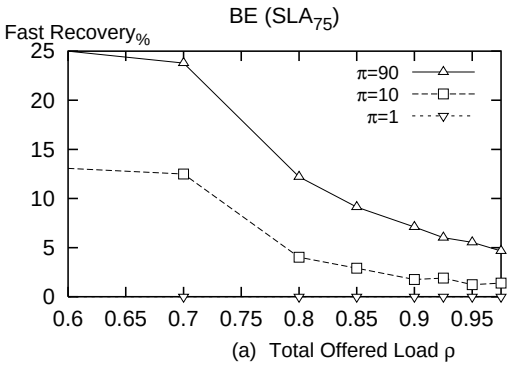


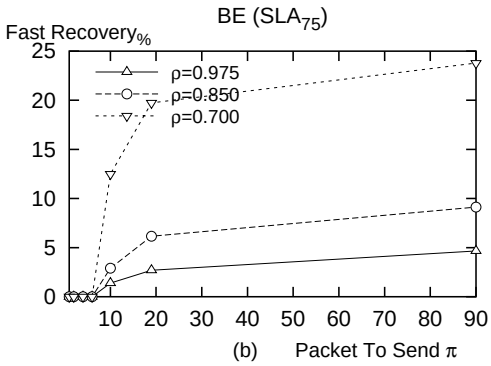


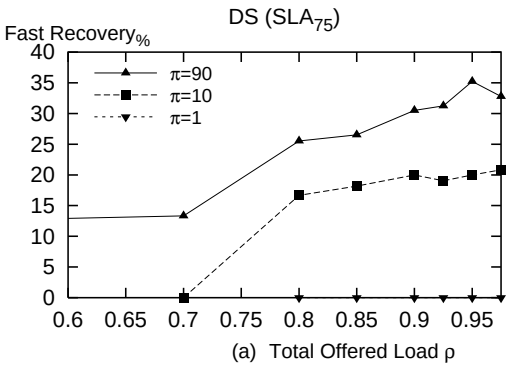


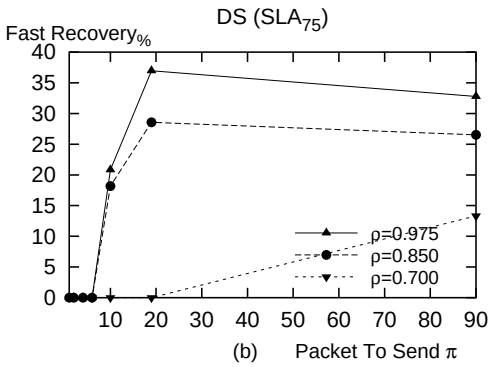


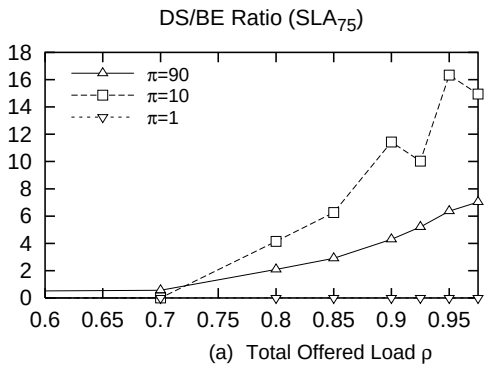
D.2.5 Fast Recovery (HTTP Traffic Only, SLA_{75})

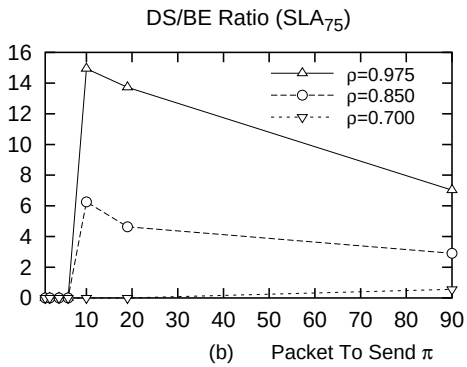




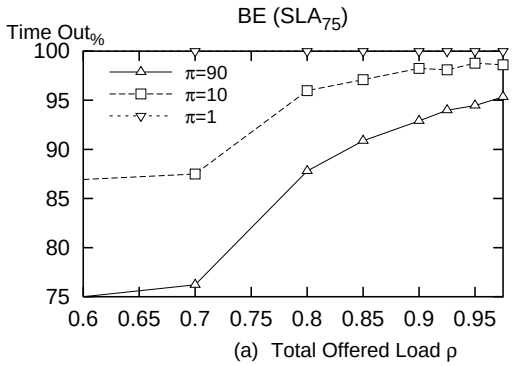


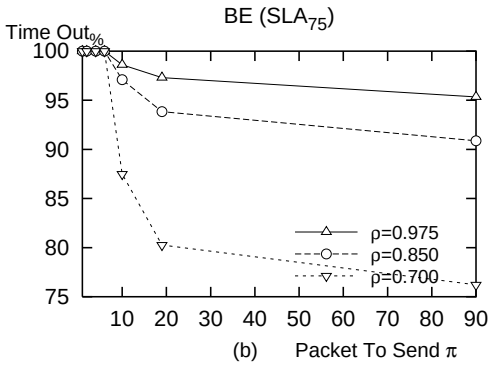


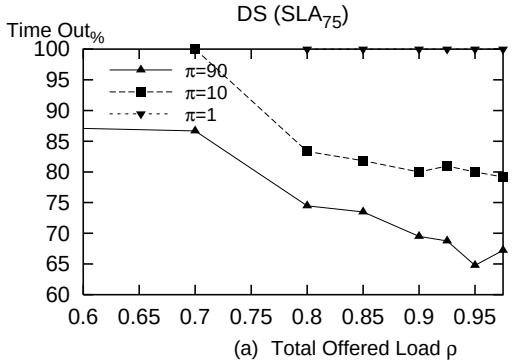


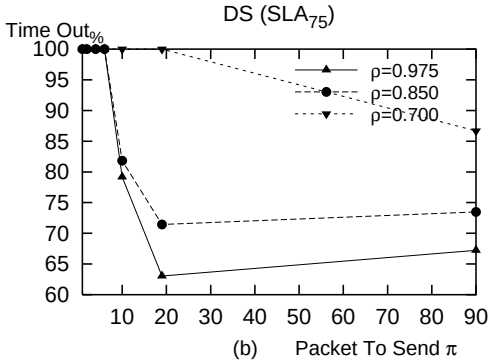


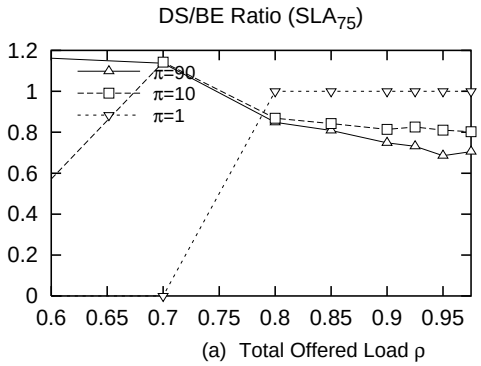
D.2.6 Time Out (HTTP Traffic Only, SLA$_{75}$)

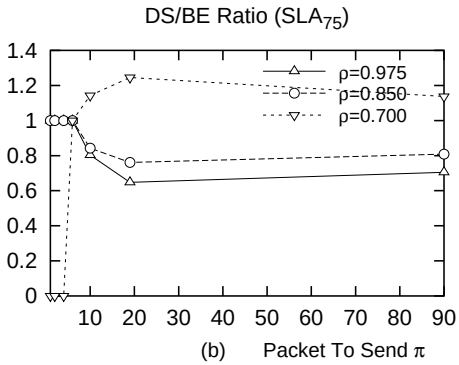






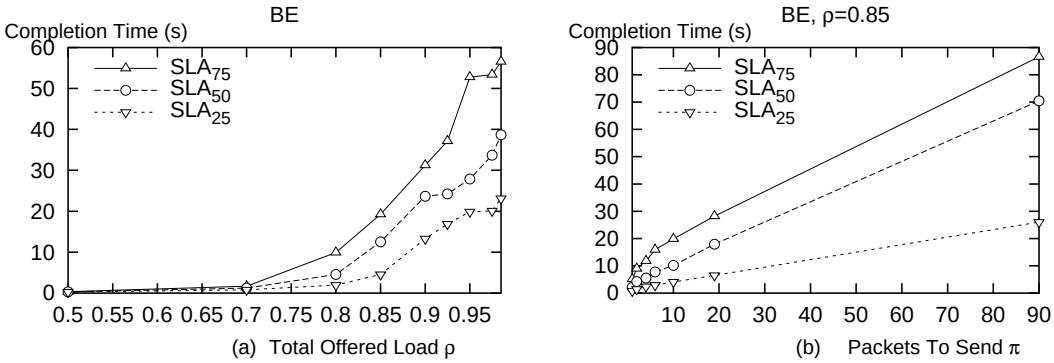


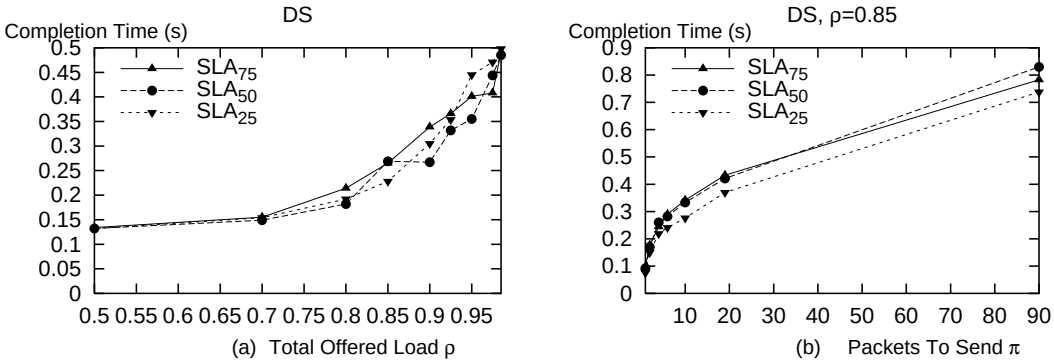


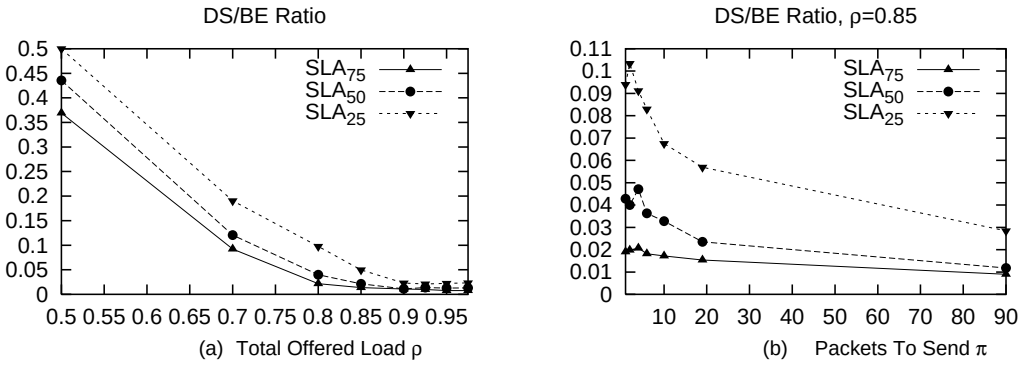


D.3 SLA_{25-75} Simulations Comparison

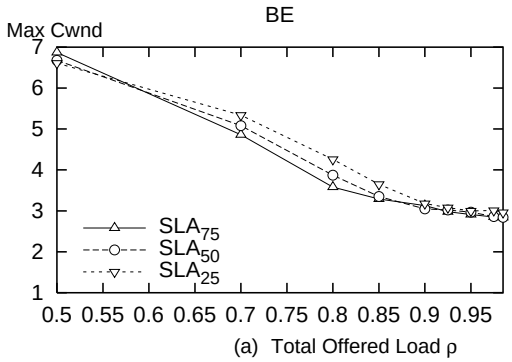
D.3.1 Completion Time (HTTP Traffic Only)

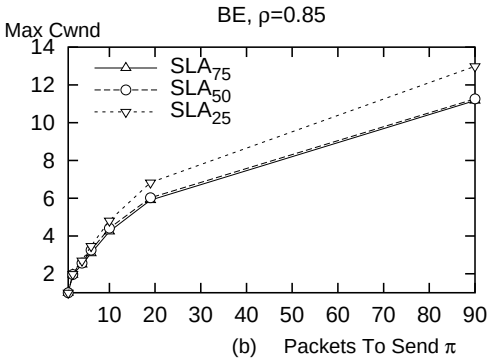


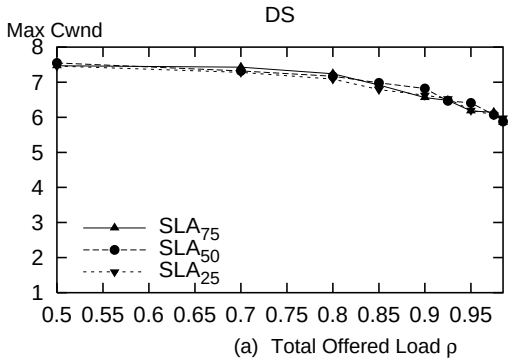


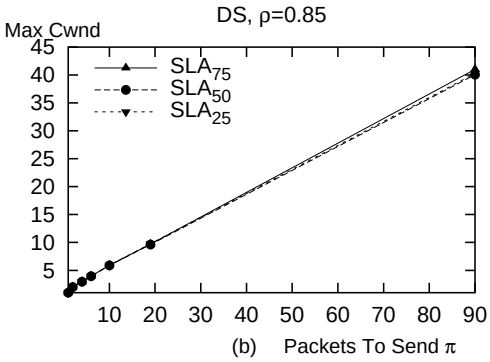


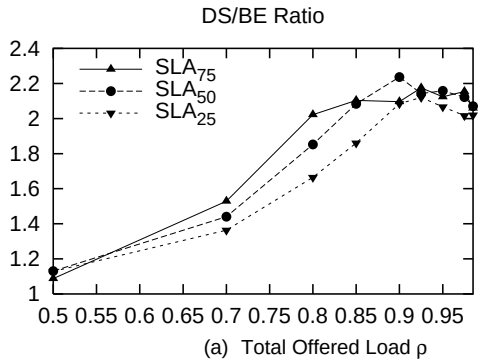
D.3.2 Congestion Window (HTTP Traffic Only)

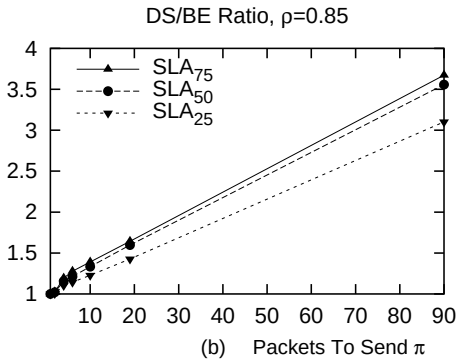




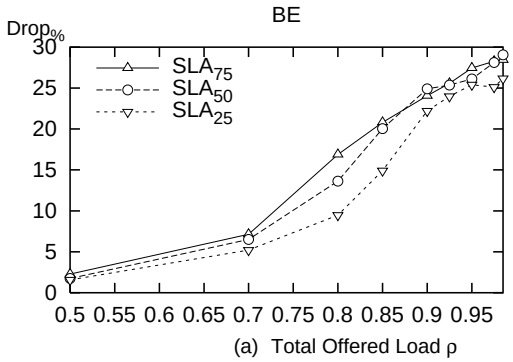


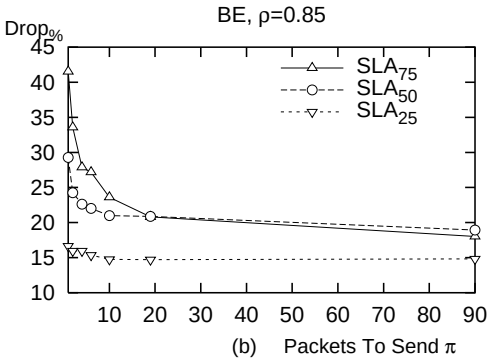


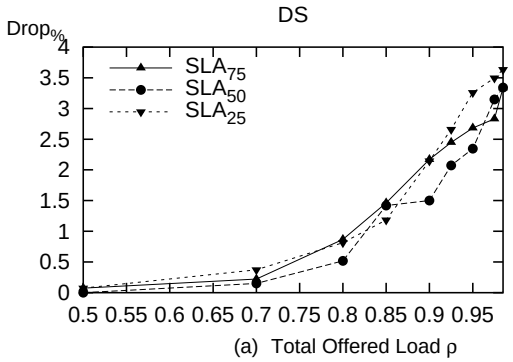


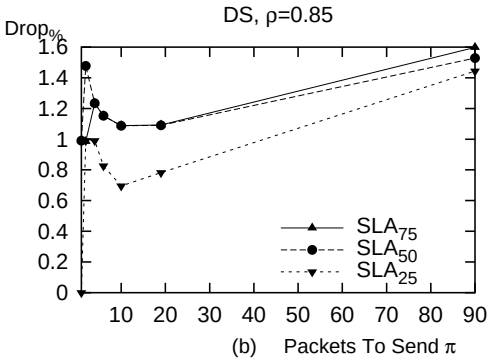


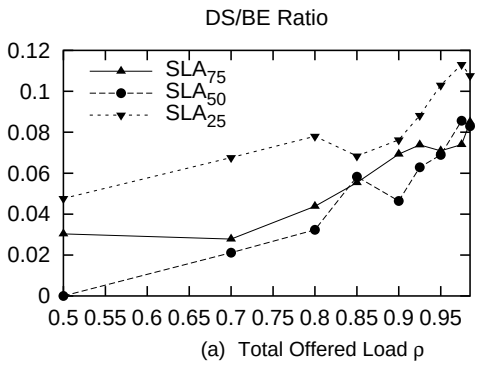
D.3.3 Packet Drops (HTTP Traffic Only)

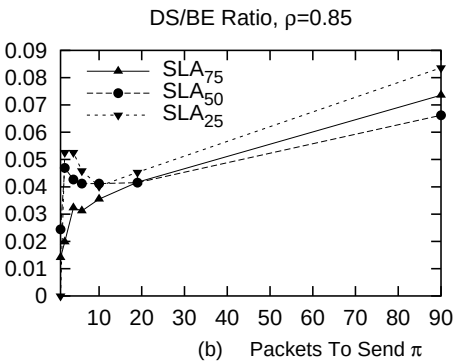




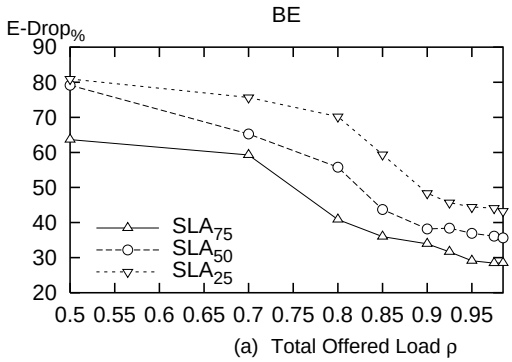


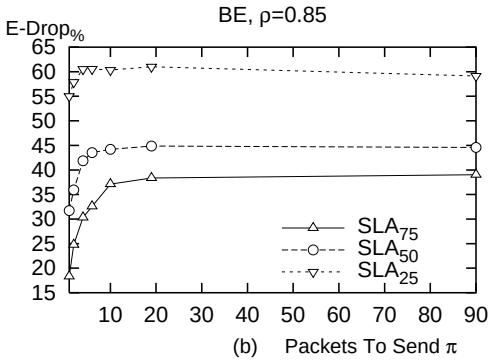


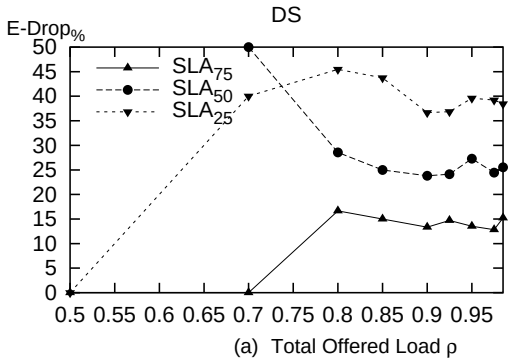


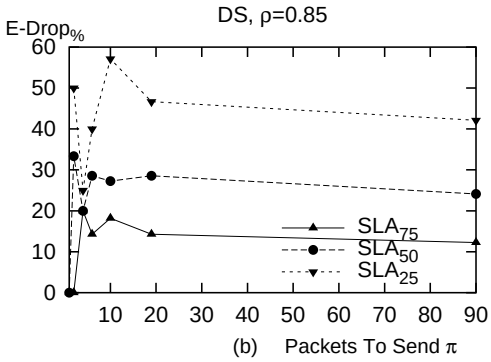


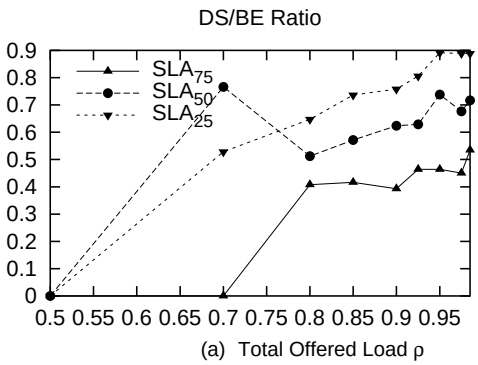
D.3.4 Early Drops / Packet Drops (HTTP Traffic Only)

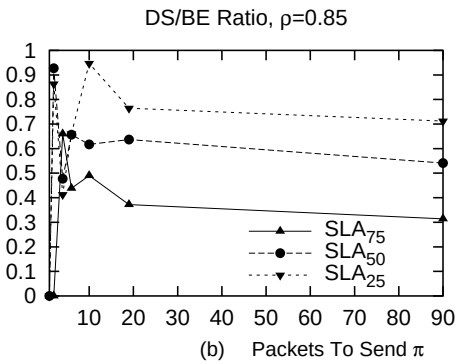




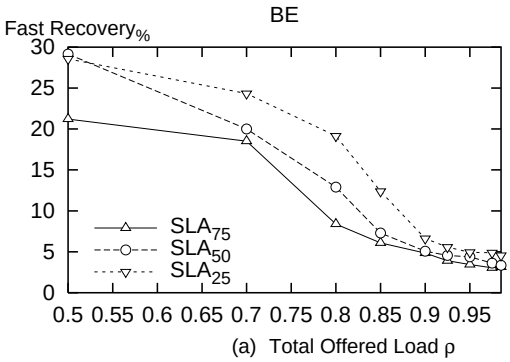


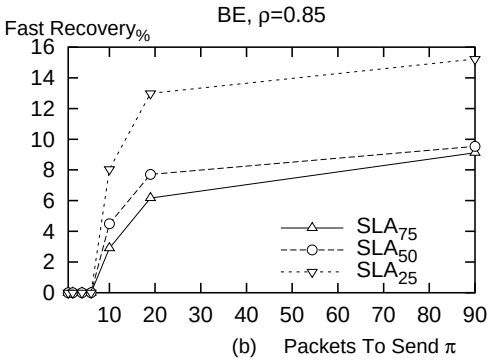


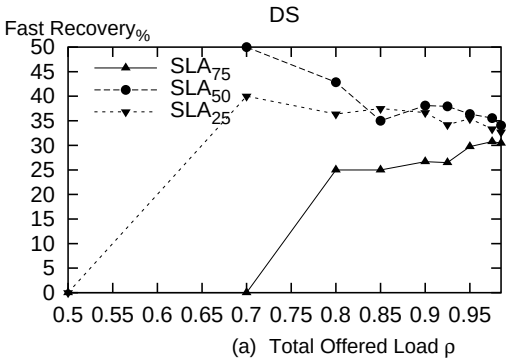


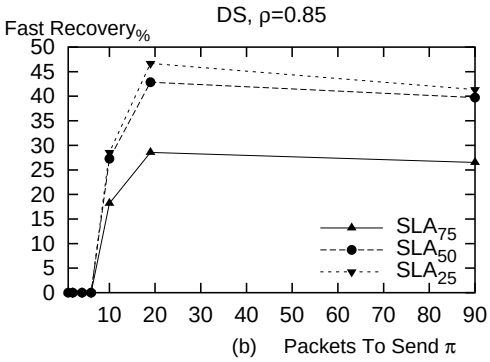


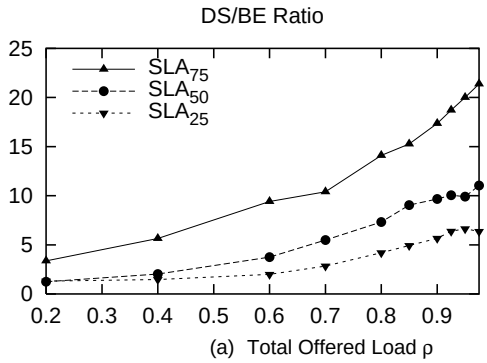
D.3.5 Fast Recovery (HTTP Traffic Only)

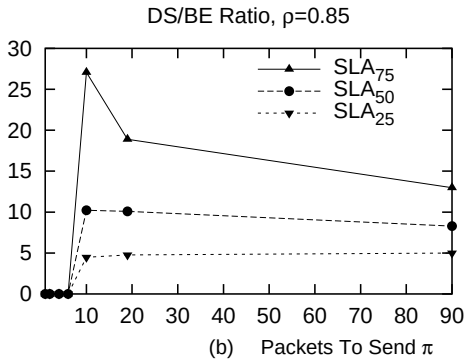




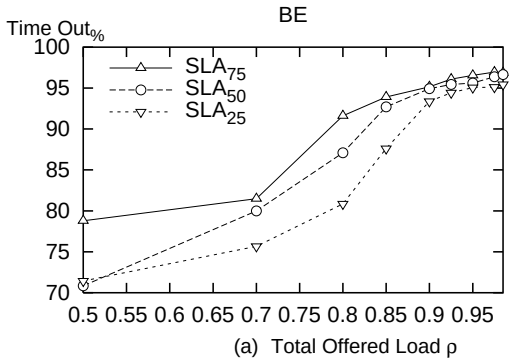


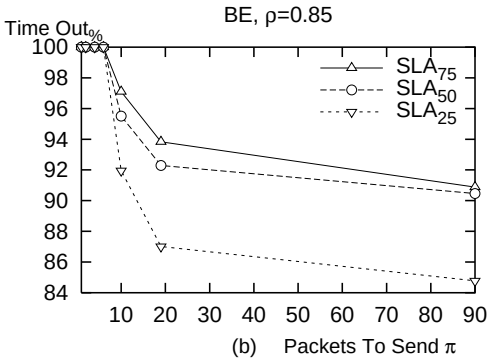


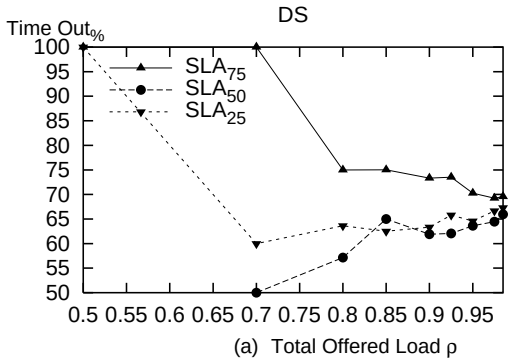


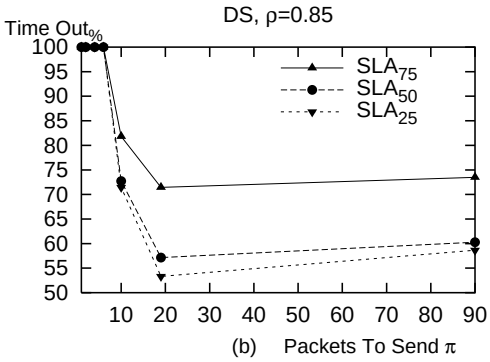


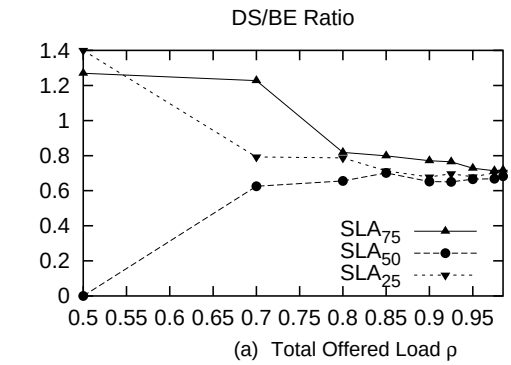
D.3.6 Time Out (HTTP Traffic Only)

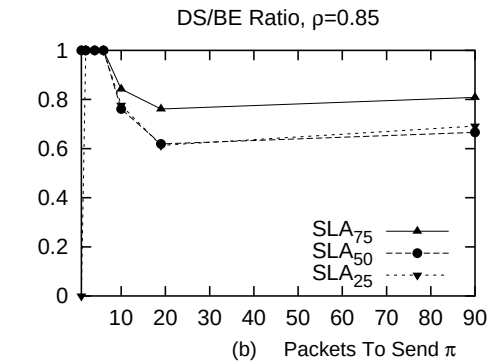






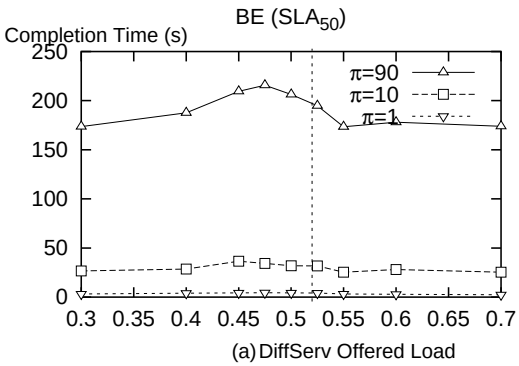


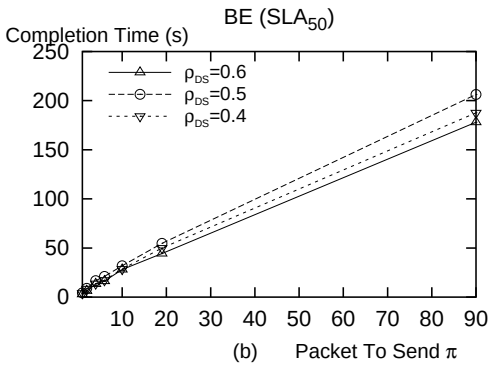


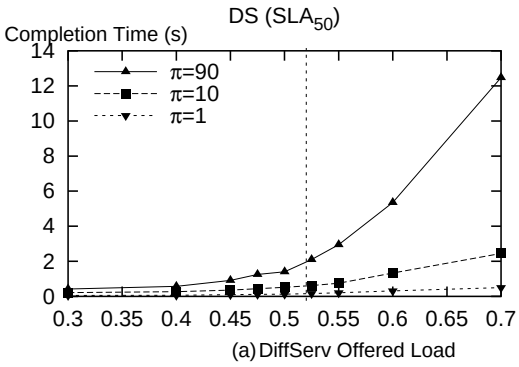


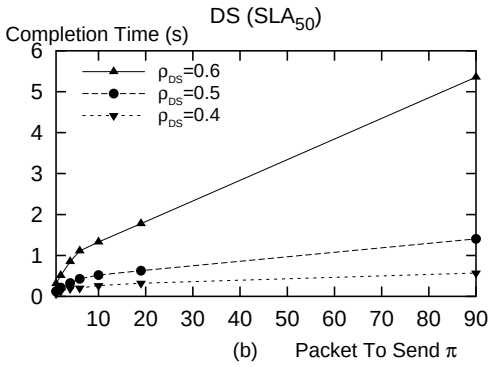
D.4 SLA$_{50}$, DiffServ Allocation Simulations

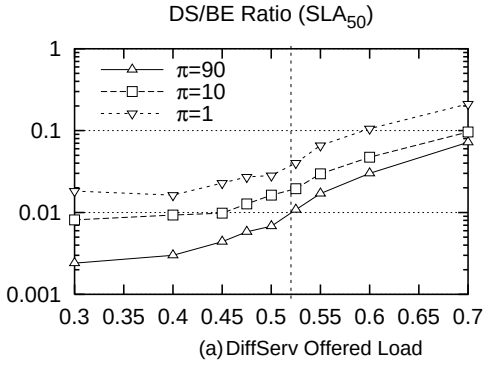
D.4.1 Completion time (ρ=0.975)

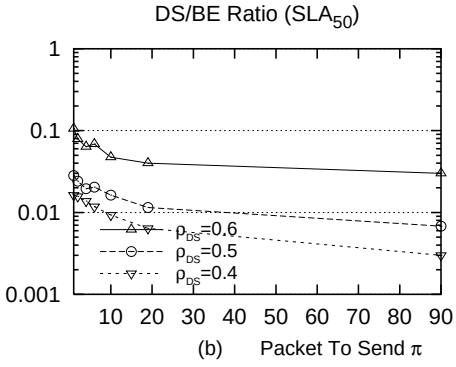




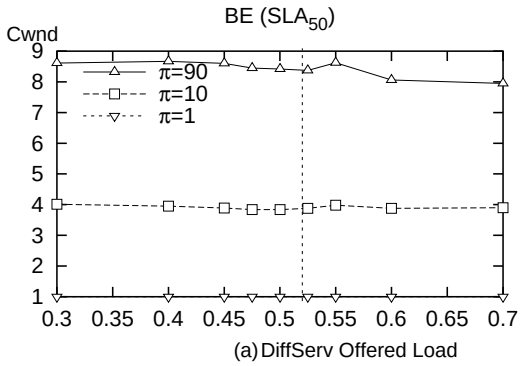


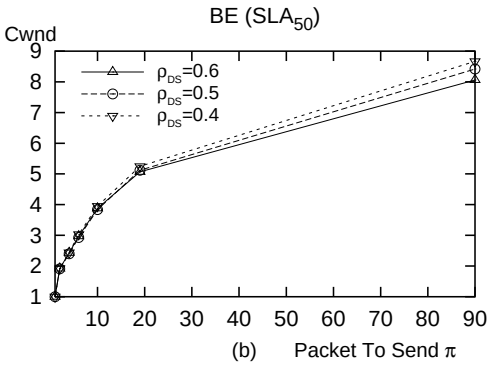


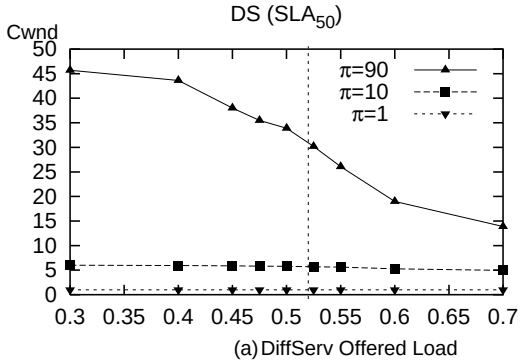


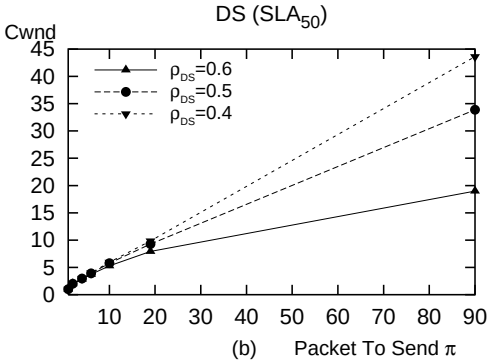


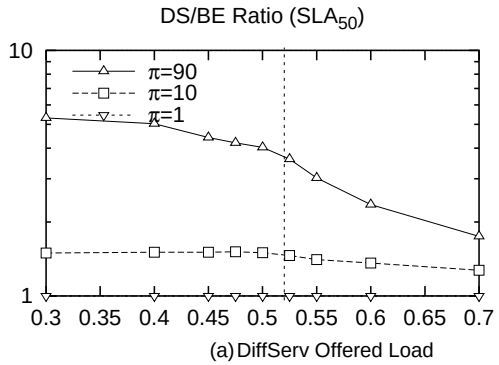
D.4.2 Congestion Window (ρ=0.975)

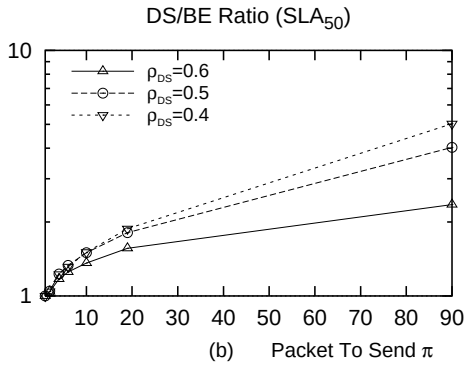




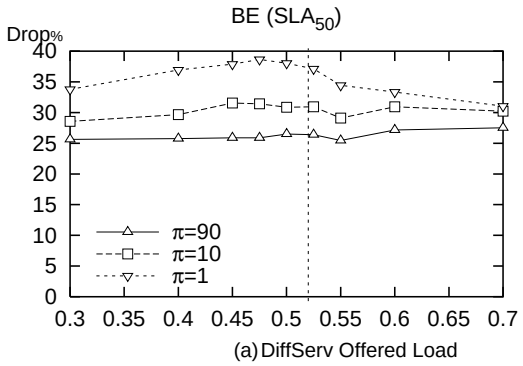


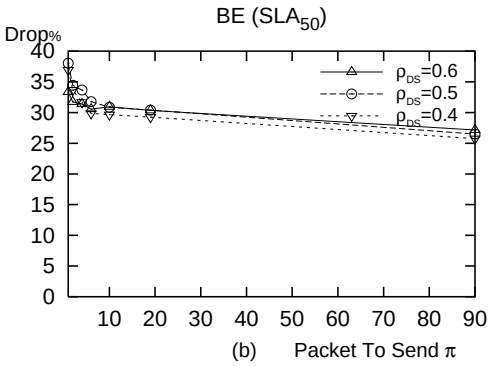


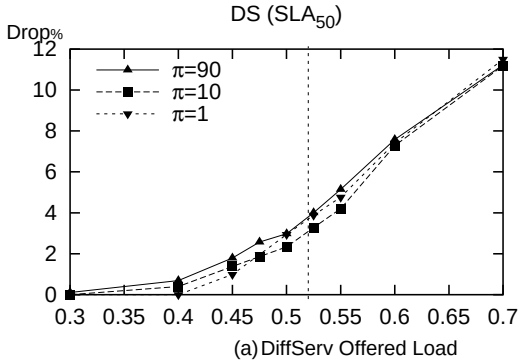


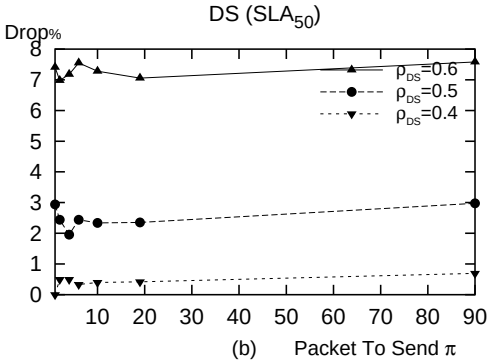


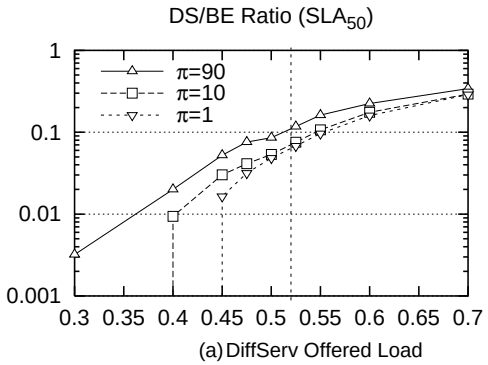
D.4.3 Packet Drops (ρ=0.975)

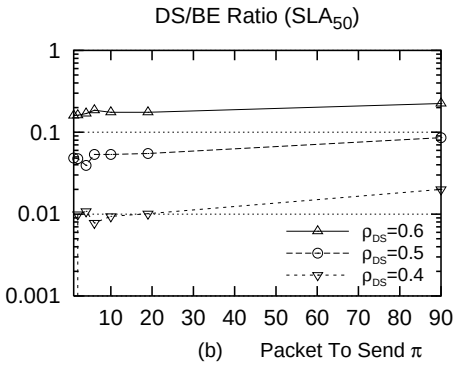




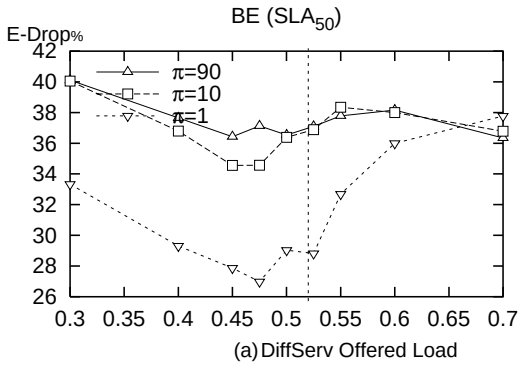


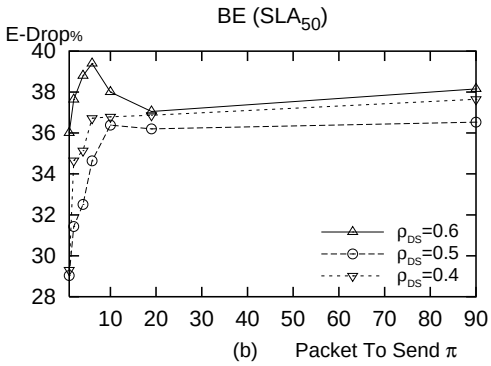


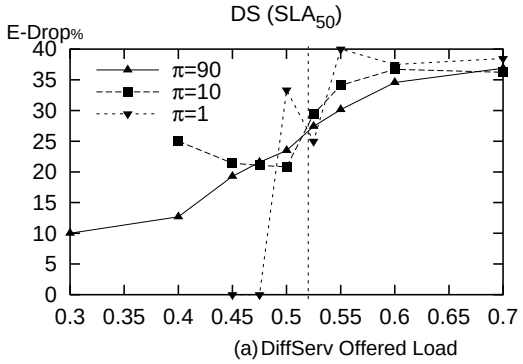


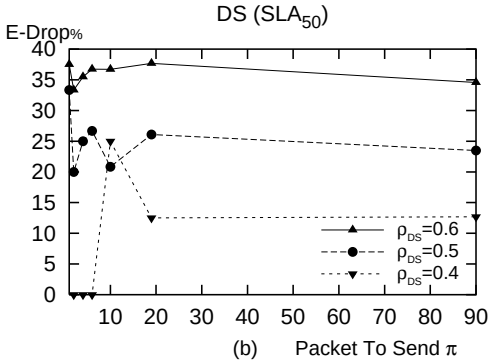


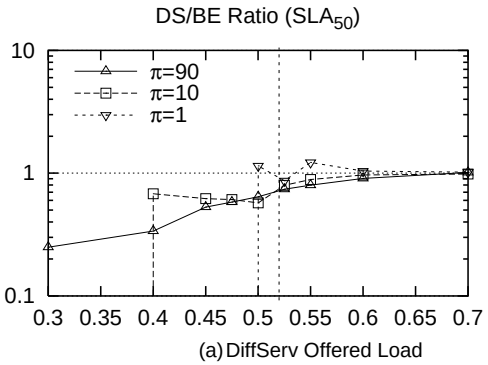
D.4.4 Early Drops / Packet Drops (ρ=0.975)

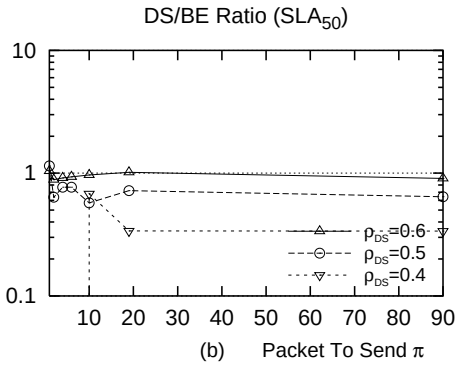




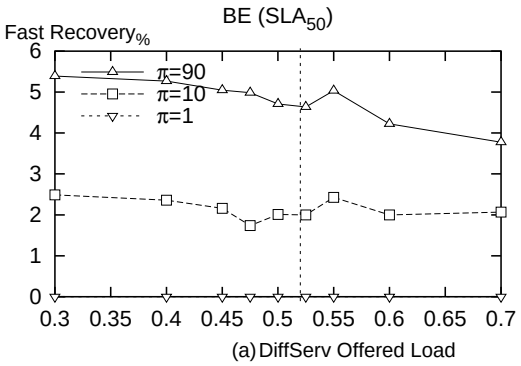


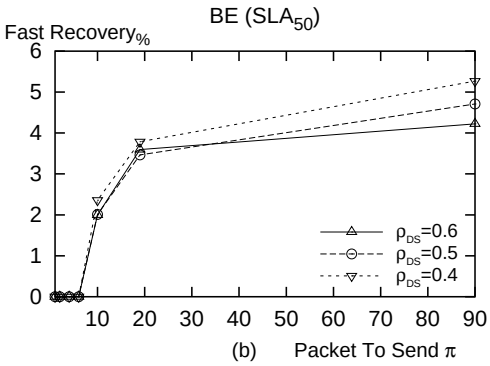


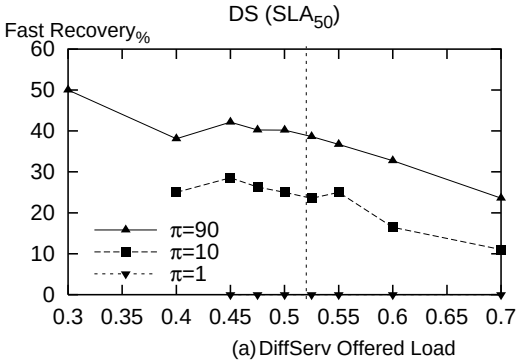


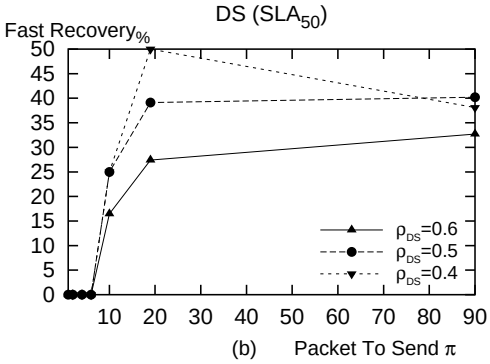


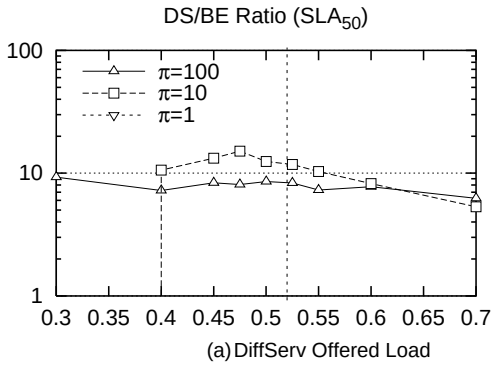
D.4.5 Fast Recovery (ρ=0.975)

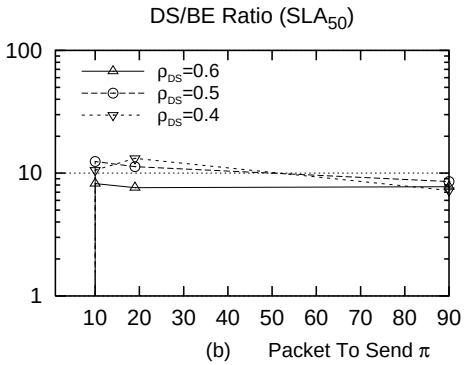




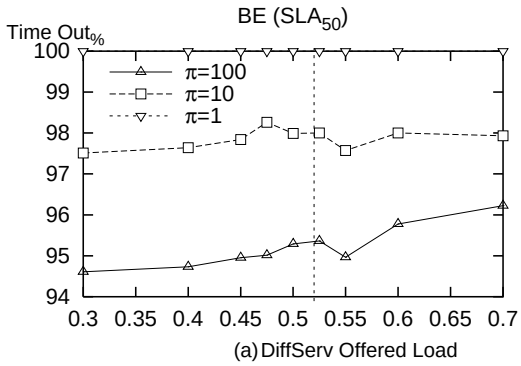


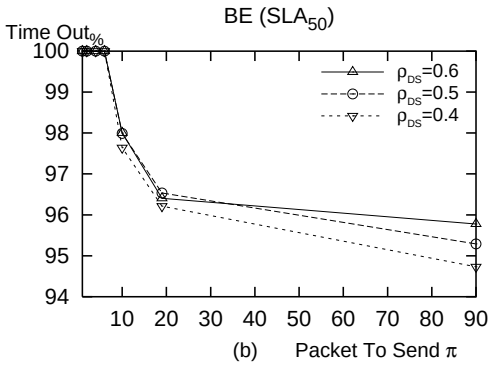


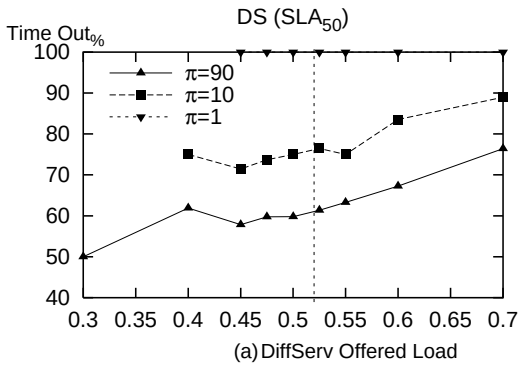


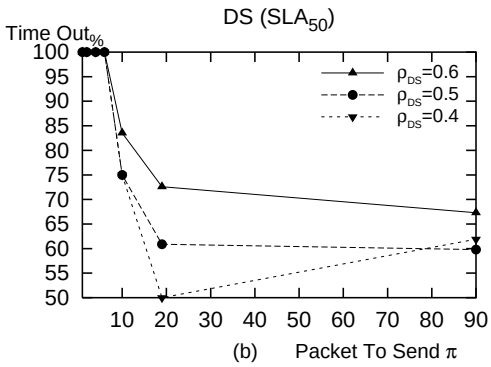


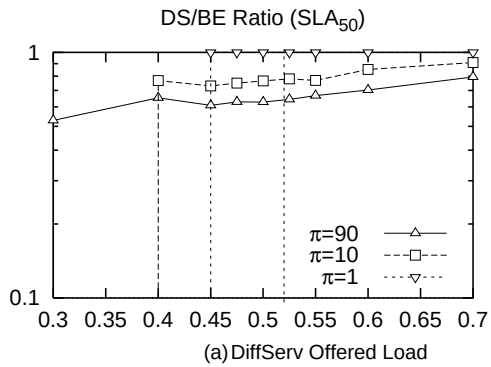
D.4.6 Time Out (ρ=0.975)

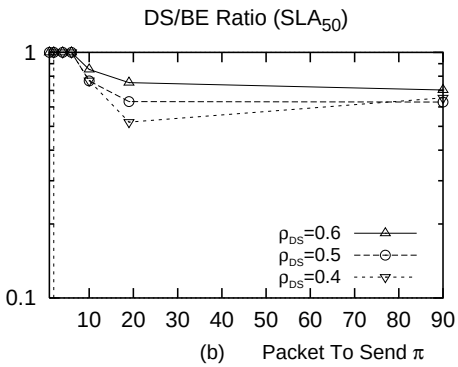












Bibliography

[RFC2474] K. Nichols, S. Blake, F. Baker and D. Black, *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*, RFC 2474, December 1998.

[RFC2475] D. Black, S. Blake, M. Carlson, E. Davies, Z. Wang and W. Weiss, *An Architecture for Differentiated Services*, RFC 2475, December 1998.

[RFC2597] J. Heinanen, F. Baker, W. Weiss and J. Wroclawski, *Assured Forwarding PHB Group*, RFC 2597, June 1999.

[RFC2598] V. Jacobson, K. Nichols and K. Poduri, *An Expedited Forwarding PHB*, RFC 2598, June 1999.

[RFC2638] K. Nichols, V. Jacobson and L. Zhang, *A Two-bit Differentiated Architecture for the Internet*, RFC 2638, July 1999.

[RFC2697] J. Heinan and R. Guerin, *A Single Rate Three Color Marker*, RFC 2697, September 1999.

[RFC2698] J. Heinan and R. Guerin, *A Two Rate Three Color Marker*, RFC 2698, September 1999.

[RFC2859] W. Fang, N. Seddigh and B. Nandy, *A Time Sliding Window Three Color Marker*, RFC 2859, June 2000.

[RFC2963] O. Bonaventure and S. De Cnodder, *A Rate Adaptative Shaper for Differentiated Services*, RFC 2963, October 2000.

[RFC2873] X. Xiao, A. Hannan, V. Paxson and E. Crabbe, *TCP Processing of the IPv4 Precedence Field*, RFC 2873, June 2000.

[RFC2990] G. Huston, *Next Steps for the IP QoS Architecture*, RFC 2990, November 2000.

[RFC3086] K. Nichols and B. Carpenter, *Definition of Differentiated Services Per Domain Behaviors and Rules for their Specification* , RFC 3086, April 2001.

[RFC3140] S. Brim, B. Carpenter and F. Le Faucheur, *Per Hop Behavior Identification Codes*, RFC 2836, June 2001.

[RFC791] J. Postel, *Internet Protocol*, STD 5, RFC 791, September 1981.

[RFC793] J. Postel, *Transmission Control Protocol*, STD 7, RFC 793, September 1981

[RFC1122] R. Braden, *Requirements for Internet hosts – communication layers*, STD 3, RFC 1122, October 1989.

[RFC1349] P. Almquist, *Type of Service in the Internet Protocol Suite*, RFC 1349, July 1992.

[RFC1812] F. Baker, *Requirements for IP Version 4 Routers*, RFC 1812, June 1995.

[RFC2309] R. Braden et al., *Recommendations on Queue Management and Congestion Avoidance in the Internet*, RFC 2309, April 1998.

[RFC2415] K. Poduri and K. Nichols, *Simulation Studies of Increased Initial TCP Window Size*, RFC 2415, September 1998.

[RFC2460] S. Deering and R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 2460, December 1998.

[RFC2581] W. Stevens et al., *TCP Congestion Control*, RFC 2581, April 1999.

[TCPCON] M. Mathis, J. Semske, J. Mahdavi and T. Ott, *The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm*, IEEE Journal, Vol. 15, June 1997.

[TCPMOD] J. Padhye, V. Firoiu, D. Towsley and J. Kurose, *Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation*, IEEE/ACM Transactions on Networking, Vol. 8, April 2000.

[TCP-SF] M. Mellia, C. Casetti and M. Meo, *TCP Smart Framing: Using Smart Segments to Enhance the Performance of TCP*, Globecom 2001, San Antonio, Texas, Novermber 2001

[RFC2998] Y. Bernet, R. Yavatkar, P. Ford, F. Baker et al., *A Framework for Integrated Services Operation Over DiffServ Networks*, RFC 2998, November 2000.

[RFC1633] R. Braden, D. Clark and S. Shenker, *Integrated Services in the Internet Architecture: An Overview*, RFC 1633, July 1994.

[RFC2205] B. Braden, L. Zhang, S. Berson, S. Herzog and S. Jamin, *Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification*, RFC 2205, September 1997.

[RFC2208] A. Mankin, F. Baker, R. Braden, M. O'Dell and A. Romanow, *Resource ReSerVation Protocol (RSVP) – Version 1 Applicability Statement*, RFC 2208, September 1997.

[DSRSVP] C. Radhakrishna, *Use of RSVP for Differentiated Services Signaling and Admission Control*, Work In Progress, `draft-rk-diffserv-rsvp-sig-00.txt`, December 2000.

[EXPALL] D.D. Clark and W. Fang, *Explicit Allocation of Best Effort Packet Delivery Service*, IEEE/ACM Transactions on Networking, Vol. 6 , August 1998.

[PROPDS] C. Dovrolis, D. Stiliadis and P. Ramanathan, *Proportional Differentiated Services: Delay Differentiation and Packet Scheduling*, In Proocedings ACM SIGCOMM, 1999.

[SIMA] K. Kilkki, *Simple Integrated Media Access (SIMA)*, Work In Progress, `draft-kalevi-simple-media-access-01.txt`, June 1997.

[SIMDS] P. Ferguson, *Simple Differential Services: IP TOS and Precedence, Delay Indication, and Drop Preference*, Work In Progress, November 1997.

[VWPDB] V. Jacobson, K. Nichols and K. Poduri, *The "Virtual Wire" Per-Domain Behavior*, Work in Progress, `draft-ietf-diffserv-pdb-vw-00.txt`, January 2001.

[ARPDB] N. Seddigh, B. Nandy and J. Heinanen, *An Assured Rate Per-Domain Behaviour for Differentiated Services*, Work In Progress, `draft-seddigh-pdb-ar-00.txt`, June 2001.

[RED] S. Floyd and V. Jacobson, *Random Early Detection Gateways fot Congestion Avoidance*, IEEE/ACM Transactions on Networking, Vol. 1, August 1993.

[REDNOTE] V. Jacobson, *Notes on Using RED for Queue Management and Congestion Avoidance*, In Proceedings of NANOG Workshop, 1998.

[REDFLU] V. Misra, W. Gong et al., *A Fluidbased Analysis of a Network of AQM Routers Supporting TCP Flows with an Application to RED*, In Proceedings of SIGCOMM, 2000.

[AQMSTDY] V.Firout and M.Borden, *A Study Of Active Queue Management for Congestion Control*, In Proceedings of INFOCOM, 1999.

[REDMOD] M. May, T. Bonald and J.C. Bolot, *Analytic Evaluation of RED Performance*, In Proceedings of INFOCOM, May 2000.

[REDNOT] M. May, J.C. Bolot, C. Diot and B. Lyles, *Reasons not to deploy RED*, In Proceedings of IWQoS, May 1999.

[REDWEB] M. Christiansen, K. Jeffay, D. Ott and F. Smith, *Tuning RED for Web Traffic*, In Proceedings of SIGCOMM, 2000

[MRED] R. Makkar, J. Salim, N. Seddigh, B. Nandy and J. Babiarz, *Empirical Study of Buffer Management Scheme for Diffserv Assured Forwarding PHB*, Nortel Tech. Report, May 2000.

[REDLIG] V. Jacobson, K. Nichols and K. Poduri, *RED in a Different Light*, Work In Progress, September 1999.

[AFSTDY] M. Goyal, A. Durresi, R. Jain, C. Liu, *Performance Analysis of Assured Forwarding*, Work In Progress, `draft-goyal-diffserv-afstdy-00.txt`, August 2000.

[AFEVAL] J. Ibanez and K. Nichols, *A Preliminary Simulation Evaluation of an Assured Service*, Work In Progress, `draft-ibanez-diffserv-assured-eval-00.txt`, August 1998.

[AFRES] N. Seddigh, B. Nandy and P. Pieda, *Assured Forwarding PHB: What Assurance does the Customer Have ?*, In Proceedings NOSSDAV, New Jersey, June 1999.

[BWRES] N. Seddigh, B. Nandy and P. Pieda, *Bandwidth Assurance Issues for TCP flows in a Differentiated Services Network*, In Proceedings of GLOBECOM, December 1999

[TCPUDP] N. Seddigh, B. Nandy and P. Pieda, *Study of TCP and UDP Interaction for the AF PHB*, Work In Progress, `draft-nsbnpp-diffserv-udptcpaf-01.txt`, February 2000.

[UDPDYN] P. Pieda, N. Seddigh and B. Nandy, *The Dynamics of TCP and UDP Interaction in IP-QoS Differentiated Service Networks*, In Proceedings of CCBR, Ottawa, November 1999.

[DSMARK] I. Yeom and N. Reddy, *Impact of Marking Strategy on Aggregated Flows in a Differentiated Services Network*, In Proceedings of IWQoS, May 1999.

[CSFQ] I. Stoica et al. *Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks*, In Proceedings of ACM SIGCOMM, September 1998.

[FMARK] A. Feroz, S. Kalyanaraman and A. Rao, *A TCP-Friendly Traffic Marker for IP Differentiated Services*, In Proceedings of IwQoS, June 2000.

[FTCON] I. Andrikopoulos, L. Wood and G. Pavlou, *A Fair Traffic Conditioner for the Assured Service in a Differentiated Services Internet*, In Proceedings of ICC, June 2000.

[SPFQ] N. Venkitaraman , J. Mysore, R. Srikant and R. Barnes, *Stateless Prioritized Fair Queueing*, Work In Progress, `draft-venkitaraman-diffserv-spfq-00.txt`, January 2001.

[GRAS] O. Bonaventure and S. De Cnodder, *A rate adaptive shaper for differentiated services*, Work In Progress, `draft-bonaventure-diffserv-rashaper-02.txt`, October 2000.

[QOSPIC] X. Xiao, *Internet QoS: the Big Picture*, IEEE Network, Vol. 13, April 1999.

[HTTPMOD] B. Mah, *An Empirical Model of HTTP Network Traffic*, In Proceedings of INFOCOM, Agust 1997.

[NS] *ns* Network Simulator, Available at *http://www.isi.edu/nsnam/ns/*.