# itmBench: Generalized API for Internet Traffic Managers [*]

Gali Diamant Leonid Veytser Ibrahim Matta Azer Bestavros
Mina Guirguis Liang Guo Yuting Zhang Sean Chen
*Computer Science Department*
*Boston University*
*Boston, MA, 02215, USA*
`{gali,veytser,matta,best,msg,guol,danazh,zyschen}@cs.bu.edu`
`http://www.cs.bu.edu/groups/itm/`

## Abstract

Internet Traffic Managers (ITMs) are special machines placed at strategic places in the Internet. *itmBench* is an interface that allows users (e.g. network managers, service providers, or experimental researchers) to register different traffic control functionalities to run on one ITM or an overlay of ITMs. Thus *itmBench* offers a tool that is extensible and powerful yet easy to maintain. ITM traffic control applications could be developed either using a kernel API so they run in kernel space, or using a user-space API so they run in user space. We demonstrate the flexibility of *itmBench* by showing the implementation of a kernel module that provides a differentiated network service. Due to space limitations, we refer the reader to [2] for a user-space module that provides an overlay routing service. Our *itmBench* Linux-based prototype is free software and can be obtained from `http://www.cs.bu.edu/groups/itm/`.
**Keywords:** Communications Software; TCP/IP Networks; Traffic Control Applications; Linux Prototype.

## 1 Introduction

**Motivation:** Internet Traffic Managers (ITMs) are special machines placed at strategic places in the Internet (e.g., in front of clients or servers, or between administrative domains) [9]. These ITMs should be capable of classifying packets as they go by into *classes*, and of intelligently controlling their transmission into the core of the infrastructure. The idea is to implement additional control functionalities at the ITMs while keeping the core of the Internet as simple as possible. For example, once ITMs classify packets at the edges or network boundaries, core routers may simply differentiate their services based on the class carried by the packet, e.g. using a simple class-based scheduling discipline. Thus each core router maintains a state for each one of the (few) classes, rather than maintaining a state for each flow of packets or connection. This architecture is consistent with scalable hierarchical designs, e.g. the Diff-Serv architectural concepts of the IETF (Internet Engineering Task Force) [1]. Figure 1 illustrates the hierarchy of routers. ITM functionalities can be placed at access or distribution points, whereas core routers are kept simple to keep up with the higher transmission rates.

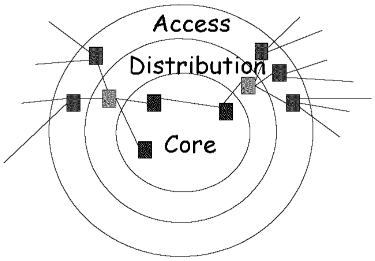

Figure 1: ITMs placed at access or distribution points

Figure 2 illustrates examples of placement and functionality of ITMs. An ITM placed in front of a farm of servers can perform *aggregation control*—the ITM would control the transmission of packets on several connections (flows) in a cooperative manner rather than letting them compete unproductively (and possibly unfairly) inside the network. An ITM placed at the edge of an administrative domain can perform *differentiated control*—the ITM would classify passing packets into classes so that core routers service them using a simple class-based scheduling discipline or route them using a class-based routing protocol. An ITM could also be placed in front of a set of clients to perform *proxy control*—the ITM acting as a wireless proxy would hide wireless losses from the source by acting as a server with reduced capacity.

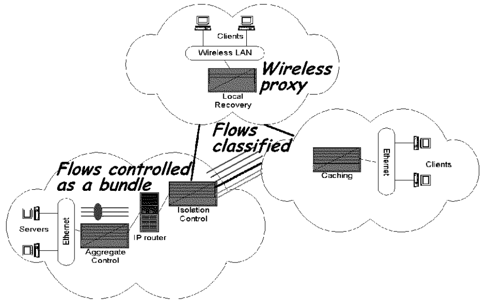

Figure 2: Placement and functionality of ITMs

**Our Contribution in this Paper:** The Internet has, in the last few years, witnessed similar architectures of special boxes placed in the "middle" of the Internet to improve its predictability and utilization. Our project, for which we describe its Linux-based prototype in this paper, unifies distinct functionalities (such as the aforementioned differentiated, aggregation, and proxy controls) in one common framework, that of ITMs. Such *uniform* infrastructure enables easier development through a user interface that is common to several traffic control functionalities.

[*] This work was supported in part by NSF grants ANI-0095988, ANI-9986397, EIA-0202067 and ITR ANI-0205294, and by grants from Sprint Labs and Motorola Labs.

The design of an ITM supporting those traffic management functionalities is based on a unified control-theoretic framework. Figure 3 shows the general architecture of an ITM. Typical of closed-loop feedback control systems, the ITM would consist of *control programs* implementing the considered functionalities. The parameters of these control programs would be dynamically adjusted based on *measurements*, e.g. the characteristics of the bottleneck resource such as its bandwidth and buffer space. In this project, we focus on the management of traffic from the Transmission Control Protocol (TCP) since the majority of bytes on the Internet (up to 95%) is attributed to TCP [10]. However, our generalized Application Programming Interface (API) we present in this paper can support TCP-based as well as non-TCP-based network services.

Specifically, through our *itmBench* API, ITM traffic control applications could be developed on one ITM or across an overlay of ITMs, either using a kernel API so they run in kernel space, or using a user-space API so they run in user space. Using an overlay of ITMs provides a scalable solution to managing traffic, especially across large-scale highly heterogeneous internets [3]. We demonstrate the flexibility of our *itmBench* API by showing the implementation of a kernel module that provides a differentiated network service. Due to space limitations, we refer the reader to [2] for a user-space module that provides an overlay routing service. Our *itmBench* Linux-based prototype is free software and can be obtained from `http://www.cs.bu.edu/groups/itm/`.

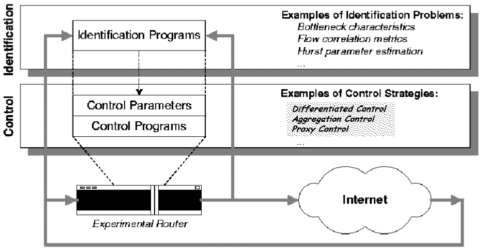

Figure 3: General architecture of an ITM

**Paper Outline:** The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 gives an overview of our ITM architecture. Section 4 describes in more detail the kernel-level *itmBench* API, i.e. the interface that allows a user to develop a traffic control application in kernel space. Section 5 gives an example kernel-level application, called *qos_mod*. In this application, the ITM classifies passing packets into short-flow and long-flow classes, so short-flow packets are preferentially treated inside the network. Section 6 presents an overview of the *itmBench* API for writing user-space applications, and describes *itmRoute*, an overlay routing application built using the user-space API. We refer the reader to [2] for more details. Finally, Section 7 concludes the paper.

## 2 Related work

A few other projects have implemented extensible router systems: Click [8] is a software for building modular routers. XORP [6] is a routing infrastructure that provides an extensible experimental protocol deployment utility. Netfilter [11], now part of the Linux kernel, was originally a Firewall software, and now provides packet filtering capabilites, using specific control tools.

After a thorough investigation of these systems, they turned out to be inadequate for our needs. Click and XORP deal with the core routing software itself, and focus on network protocols and their development through a rudimentary interface. Netfilter only provides packet filtering at different levels, but does not provide other functionalities. ITMs are different, as they don't handle core routing but rather provide a generalized infrastructure that allows different traffic control capabilities to reside in the same machine, and be controlled using a similar interface. This API supports the development of control programs as either kernel-level modules or user-space modules. A unique goal of our *itmBench* API design is to also support traffic management functionalities that span multiple ITM boxes, for example as in an overlay routing service or a virtual tunnel service.

## 3 Architecture

### 3.1 Overview

Applications (both kernel and user level) wishing to use the ITM need to register with a core kernel module. This module, called *itm_mod*, keeps track of registered applications, and forwards packets to them according to their specifications.

A control utility enables adding kernel modules and user-level modules, each can be turned on or off according to need. This design results in a system that is:

- *Extensible:* Adding and removing capabilities is easily done simply by loading only wanted modules;
- *Easy to deploy:* The architecture does not require recompiling the kernel, making it simple to use.

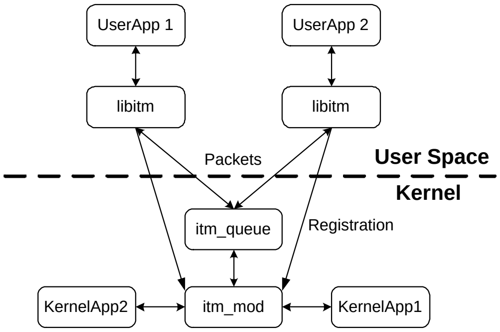

Figure 4: The ITM architecture

The ITM kernel module, *itm_mod*, communicates with the TCP/IP stack to retrieve packets. Both kernel-level and user-level modules register with the ITM module. In addition, user-level modules register with another kernel module, called *itm_queue*, which handles packets that need to move to user space; it demultiplexes the packets and forwards them to the appropriate user-level (traffic control) applications. Figure 4 shows the different components and connections between them. Applications (i.e. traffic control programs) can modify the packets, or leave them unchanged. They can claim the packets, causing them to drop, or reinject them to the TCP/IP stack, to be processed as normal.

## 3.2 An Event-Driven System

We chose to implement an event-driven Application Programming Interface (API), where for each event, a set of functions is defined. Users have to provide pointers to the functions, to be called when the event is triggered. The following five functions are currently supported:

- a *classification* function that specifies how to identify a class of packets or flows;
- a *logging* function that specifies how to log data and update class structures;
- a *processing* function that determines the action(s) associated with each event;
- a *class-update* function which defines how a class should be updated based on the logged data; and
- a *controller-update* function which allows for updating the controller parameters based on measured system state.

These functions are all called in this order, when the event an application registered for occurs. Events can be either synchronous (e.g. packet arrival at a specific layer) or asynchronous (e.g. periodic). Our API currently uses Linux kernel modules to enable different capabilities. Our current implementation was tested on both Linux 2.4.9 and Linux 2.4.20. Our current implementation requires netfilter to be installed as well, but since this is now part of the Linux kernel, it is very likely that netfilter is already installed on newer Linux machines.

## 4 Kernel Space *itmBench* API

The ITM module, *itm_mod*, is a Linux kernel module which serves as an intermediate layer between the Linux kernel and (traffic control) applications. It currently uses netfilter to capture packets that travel the TCP/IP stack, but this can be changed to work with any tool that provides similar capabilities. The ITM module needs to be loaded before any others can be used, as it provides them with necessary packet filtering capabilities, currently through netfilter. Thus, the ITM module actually hides the details of netfilter from the users. Also, if we ever decide to change the underlying layer and use a tool other than netfilter, only this *itm_mod* module will have to be re-designed, avoiding a painful process of updating many other modules.

Using netfilter hooks at the IP layer (where they are currently available), we can intercept each packet at each of five locations: pre-routing, IP-in, IP-forward, post-routing and IP-out. Figure 5 shows the relative location of these interception points during the routing process.

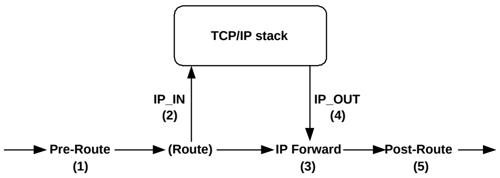

Figure 5: IP hook points

Applications that wish to use the *itm_mod* module need to register at the wanted point, giving the criteria for "interesting" packets (according to flow information, protocol, etc.) Registration for a stack-related event is made using the following function:

*itm_register_hook(where location, itm_info *data)*

The first parameter is the location in the TCP/IP stack at which to intercept the packets. The *itm_info* structure contains five pointers to functions (described in Section 3.2) and an identifying label for the registering module.

Figure 6 shows the members of the *itm_info* structure. Note that *class_func()* needs to return a value, which is later passed (as second argument) to *process_func()*. This value is the class into which the classification function classified this packet. If this value is 0, *process_func()* will not be called.

The ITM module also allows for the registration of a timer: an event to occur at specified intervals. This is done using the following function:

*itm_register_timer (void (*f)(unsigned long), int timeout)*

that will result in the function *f()* being called periodically, every *timeout* jiffies.

Also provided are the functions *itm_unregister_hook()* and *itm_unregister_timer()*, that need to be called when those services are no longer needed, or when the module is unloaded.

## 5 Application: Size-aware Differentiation

In this section, we use the *itmBench* API to implement the size-aware scheduling of TCP flows [5] as a kernel-level application.

**Motivation:** Scheduling policies that give preference to short (small) jobs, such as Shortest Job First (SJF) and Shortest Remaining Processing Time (SRPT) first scheduling, are long-known to be beneficial in reducing the mean response time of the system. Since the delivery of Internet documents can be viewed as an instance of the job scheduling problem, it has recently been shown that giving high priority to the transfer of small sized TCP flows is also beneficial. This differentiated service to different classes of TCP flows is provided simply by changing a field in the IP header of a passing packet. For a two-class service (i.e. short flows and long flows), an edge/boundary router (the ITM) starts marking packets of a flow as a "long-lived" TCP flow once the number of packets of that flow reaches a predetermined threshold. These marked packets are then treated with lower priority inside the network. This differentiated service is illustrated in Figure 7.

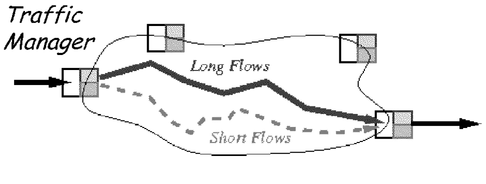

Figure 7: Classification of packets into short-flow and long-flow classes

**Implementation using the Kernel Space *itmBench* API:** The ITM maintains a counter for each flow, recording how many packets have been transmitted so far. By default, packets from every new flow obtain the highest priority. However, once this counter exceeds some pre-defined threshold, the priority of the remaining packets is reduced to the next lower level (as the flow to which these packets belong is now considered

```
typedef struct itm_info_t{
        char *label;
        itm_type type;           /*ITM_USER or ITM_KERNEL*/
        itm_interest interest;  /*has classification of "interesting" packets*/
        pid_t pid;               /*ITM_USER pid*/
        where hook;

#ifdef __KERNEL__
        /*ITM_KERNEL functions */

        int (*class_func)(where, struct sk_buff **);
        void (*log_func)(where, struct sk_buff **);
        int (*process_func)(where, int, struct sk_buff **);
        int (*update_func)(where, struct sk_buff **);
        int (*control_func)(where, struct sk_buff **);
#else
        /*ITM_USER functions */

        int (*class_func)(where, ipq_packet_msg_t *);
        void (*log_func)(where, ipq_packet_msg_t *);
        int (*process_func)(where, int, ipq_packet_msg_t *);
        int (*update_func)(where, ipq_packet_msg_t *);
        int (*control_func)(where, ipq_packet_msg_t *);
#endif
} itm_info;
```

Figure 6: The itm_info struct

long/large). Packet classification is accomplished by tagging a TOS (Type-of-Service) field, or DiffServ Code Point, in the packet header.

We implemented this differentiated functionality as a kernel module, *qos_mod*, using our *itmBench* API. Henceforth, we show code segments to demonstrate the use of the API.

*qos_mod* uses a table to keep track of TCP connections, identified by source+destination details (ipaddr+port). It counts the number of packets per flow and stores this value in the table entry which correponds to the connection. Using these counters, *qos_mod* classifies flows into `LONG` and `SHORT` according to the number of packets seen so far. A threshold value is given as a parameter when the module is loaded (using *insmod*), with a default value defined.

The first step is to register with *itm_mod*, when the module is loaded. Figure 8 shows how this is done in the module's __init section. The code shows how to define the classification, logging and processing functions, and how to register with *itm_mod* asking for packets in the IP_POST_ROUTE stage of the TCP/IP stack. The module does not use the other two functions, class-update and controller-update, so NULL pointers are passed.

Figure 9 shows the code for the classification function. Note that all the function does is to look at the current data logged and decide whether the flow is LONG, SHORT or still unknown, according to the number of packets seen so far.

The log function shown in Figure 10 either creates a new entry in the flows table or increments the number of packets seen for an existing flow.

Figure 11 shows code for the processing function. It is called with a packet and its class (as was determined by *classify()*). The TOS field in the IP header holds values that are the logical AND of the following:

| | |
|---|---|
| Normal-Service | 0x00 |
| Minimize-Cost | 0x02 |
| Maximize-Reliability | 0x04 |
| Maximize-Throughput | 0x08 |
| Minimize-Delay | 0x10 |

If the class of a packet is known (`LONG` or `SHORT`), the TOS field in its IP header is set so that a short flow will have the Minimize-Delay bit on, while for a long flow this bit is turned off. The default TOS value is defined in the *qos_mod* module as 0x10, so the first few packets of a new flow are given a low-delay service under the presumption that the flow is short.

Note that if the packet is modified, its IP checksum needs to be recalculated. We also need to notify netfilter of the change. The return value of `NF_ACCEPT` results in injecting the packet back to the place it was taken from (IP_POST_ROUTE in this case) in the TCP/IP stack.

## 6 User Space *itmBench* API

The *itmBench* API supports the dynamic loading of kernel-level modules (such as *qos_mod* described in Section 5) as well as user-space modules. Even though kernel (traffic control) applications excel in performance, they lack the flexibility and ease of implementation of user-space programs. The *itmBench* API infrastructure provides user-space access to captured packets to allow any application in user space to register with the ITM module and use all of the same services provided to kernel modules.

The user-space API is designed to be as close as possible to the kernel API—All the API functions and the registration functions are preserved with the same names and similar parameters. Referring to Figure 4, the implementation involves a kernel module, called *itm_queue*, and a user-space library, called *libitm*. The *libitm* library takes care of registration and packet transport, so all what the user application needs to do is to provide the implementation of the API functions (cf. Section 3.2).

```
itm_info data;

data.class_func = classify;
data.log_func = log;
data.process_func = process;
data.update_func = NULL;
data.control_func = NULL;
data.label = "qos_ctrl";
itm_register_hook(IP_POST_ROUTE, &data);
```

Figure 8: Using itm_register_hook()

```
int classify(where location, struct sk_buff **pskb)
{
        int class = 0; // unknown

        // look for connection in table, according to src+dst info

        if (found)
        {
                // mark packet as short if haven't seen too many
                // packets of this flow or if it's ending
                // otherwise, it's a long flow.

                if ((con->pkt_cnt <= THRESHOLD) || (tcph->fin))
                        class = SHORT;
                else class = LONG;
        }
       return class;
}
```

Figure 9: Sample classify_func() code

```
void log(where location, struct sk_buff **pskb)
{
        // look for connection in table, according to src+dst info
        if (found)
        {

                // if still haven't reached the threshold,
                // need to keep counting
                if (con->pkt_cnt <= THRESHOLD)
                {
                         (con->pkt_cnt) ++;
                }
        }
        else
        {

                // a new connection - add to table and start counter
                con = add_new_connection()
                con->pkt_cnt = 0;
        }
}
```

Figure 10: Sample log_func() code

We refer the reader to [2] for a detailed description of the implementation of the user-space API and how ITM applications can use this API to perform various traffic control functionalities in user space. We briefly mention here such application.

**Application: Overlay Routing -** A user-space overlay routing application, we call *itmRoute*, has been written using the user-space *itmBench* API and the *libitm* library. It works by listening to packets that are leaving the machine and tunneling them through the "closest" neighboring machine. The closeness of a neighbor is determined by sending small probe packets to the destination tunneled through each neighbor. The destination machine would then reply with an acknowledgment. The fastest received acknowledgment determines the "closest" neighbor. The status of a "closest" neighbor is refreshed periodically.

```
int process(where location, int class, struct sk_buff **pskb)
{
         if (class == 0)
                return NF_ACCEPT;

        // if a long flow, need to unset the Minimize-Delay bit
        if ((class == LONG) && (iph->tos & TOS))
                iph->tos = (iph->tos & IPTOS_PREC_MASK) & ~(TOS);

        // otherwise - short flow, so need to set Minimize-Delay bit
        else if ((class == SHORT) && (!(iph->tos & TOS)))
                iph->tos = (iph->tos & IPTOS_PREC_MASK) | TOS;

        // recalculate checksum!

        // mark the packet as changed - needed by netfilter
        (*pskb)->nfcache |= NFC_ALTERED;
        return NF_ACCEPT;
}
```

Figure 11: Sample process_func() code

## 7 Conclusion

In this paper, the ITM infrastructure has been presented along with kernel-level and user-space traffic control applications written on top of it. We have presented code segments for a kernel-level differentiated service. We refer the reader to [2] for the implementation of a user-space overlay routing service. Yet another kernel-level traffic control application, the elastic-tunnel service [4], is currently under development using our *itmBench* API. The elastic-tunnel framework [4] provides soft bandwidth guarantees by establishing a virtual pipe made of as many as needed “coordinated” TCP connections between a source and a destination.

Our goal is to validate the flexibility and extensibility of our *itmBench* API design by developing a wide range of traffic management applications. We intend to provide a basic library so potential users will not have to start from scratch when writing new applications, but rather use a given set of basic capabilities.

Currently the list of registered modules (traffic control applications) is maintained as a simple FIFO queue. We intend to allow the registering application to specify its priority, so it can process packets of “interest” in a possibly non-FIFO order. This issue is part of our larger research agenda on the composition of larger traffic control applications from smaller ones. For example, a user might want differentiated service inside an aggregated virtual tunnel between two ITM boxes.

Finally, we intend to use our *itmBench* over Planet-Lab [12] so as to experiment with new overlay traffic management solutions.

## 8 Availability

Our ITM prototype is free software, available from `http://cs.bu.edu/groups/itm`. This work is a product of the *Internet Traffic Managers* project at the Computer Science Department of Boston University. This project is funded in part by the National Science Foundation under grant ANI-0095988 from the Special Projects in Networking program.

A related project that extends the *itmBench* programming framework to Internet applications other than traffic management and control can be found at `http://cs.bu.edu/groups/ibench`. This project is funded in part by the National Science Foundation under grant ITR ANI-0205294.

**Acknowledgment:** We would like to thank Rich West for his feedback.

## References

[1] S. Blake, D. Black, M. Carlson, E. Davies, Z.Wang, and W. Weiss. “An Architecture for Differentiated Services.” *IETF RFC 2475*, December 1998.

[2] Gali Diamant, Leonid Veytser, Ibrahim Matta, Azer Bestavros, Mina Guirguis, Liang Guo, Yuting Zhang, Sean Chen. “itmBench: Generalized API for Internet Traffic Managers,” Technical Report BUCS-2003-032, Computer Science Department, Boston University, December 16, 2003. *http://www.cs.bu.edu/techreports/*

[3] K. Fall. “A Delay Tolerant Networking Architecture for Challenged Internets.” In Proceedings of ACM SIGCOMM 2003, August 2003.

[4] Mina Guirguis, Azer Bestavros, Ibrahim Matta, Niky Riga, Gali Diamant, Yuting Zhang. “Providing Soft Bandwidth Guarantees Using Elastic TCP-based Tunnels,” Technical Report BUCS-2003-028, Computer Science Department, Boston University, December 2, 2003. *http://www.cs.bu.edu/techreports/ To appear in IEEE ISCC 2004.*

[5] Liang Guo and Ibrahim Matta. “The War between Mice and Elephants.” In Proceedings of ICNP’2001: The 9th IEEE International Conference on Network Protocols, Riverside, CA, November 2001. *http://www.cs.bu.edu/groups/itm/SATS/*

[6] Mark Handley, Orion Hodson, and Eddie Kohler, “XORP: An Open Platform for Network Research.” *http://www.xorp.org*.

[7] The ITM project web site, *http://www.cs.bu.edu/groups/itm*.

[8] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek, “The Click modular router,” *ACM Transactions on Computer Systems*, 18(3), August 2000, pages 263-297. *http://www.pdos.lcs.mit.edu/click/*

[9] Ibrahim Matta and Azer Bestavros. “QoS Controllers for the Internet.” In Proceedings of the NSF Workshop on Information Technology, March 2000.

[10] Measurement Studies of End-to-End Congestion Control in the Internet. *http://www.icir.org/floyd/ccmeasure.html*.

[11] The netfilter/iptables project, *http://www.netfilter.org*.

[12] The Planet-Lab project, *www.planet-lab.org*.