ModNet: A modular approach to network stack extension

Sharvanath Pathak and Vivek S. Pai
Princeton University

Abstract

The existing interfaces between the network stack and the operating system are less than ideal for certain important classes of network traffic, such as video and mobile communication. While TCP has become the de facto transport protocol for much of this traffic, the opacity of some of the current network abstractions prevents demanding applications from controlling TCP to the fullest extent possible. At the same time, non-TCP protocols face an uphill battle as the network management and control infrastructure around TCP grows and improves.

In this paper, we introduce ModNet, a lightweight kernel mechanism that allows demanding applications better customization of the TCP stack, while preserving existing network interfaces for unmodified applications. We demonstrate ModNet's utility by implementing a range of network server enhancements for demanding environments, including adaptive bitrate video, mobile content adaptation, dynamic data and image compression, and flash crowd resource management. These enhancements operate as untrusted user-level modules, enabling easy deployment, but can still operate at scale, often providing gigabits per second of throughput with low performance overheads.

1 Introduction

With the growing popularity of HTTP, TCP has emerged as the de facto transport protocol for many real-time network applications (e.g. streaming video servers [21, 7]). However, since the traditional TCP stacks lack any interfaces for explicit control over the buffered content and explicit feedback about the progress of transmission, it becomes difficult to adapt quickly to changing network conditions in order to provide the desired responsiveness. Evidently, the web server responses are largely oblivious to network conditions even though in the current world of mobile clients the variability of bandwidth and sudden variations in network conditions is the norm.

The lack of appropriate interfaces for exposing lower level network behavior means that applications have to rely on implicit feedback from the transmission of response. This implicit feedback based mechanism, however, does not allow the adaptation to be performed at finer granularity. Moreover, the lack of any control over buffered data means that there can be considerable change in network condition between when the adaptation was performed and when the content was transmitted over the wire. Similarly, the lack of any generic interface for easily deployable, user-level customization of TCP stack behavior requires any such changes to be implemented inside the kernel, and thus, hinders their wider adoption.

One way to address these issues is to devise custom protocols, but slow adoption and difficulties with upgrading middleboxes have limited its appeal. Similarly, implementing user-level protocol stacks over RAW sockets faces the compatibility restriction that some systems require superuser permission for RAW socket support. The long-term issues related to developing a modified network stack or a new protocol involve the extra overhead of maintaining the stack and taking advantage of improvements in the native OS stacks. For example, UDP-based applications will often have to re-implement many common TCP behaviors to be network-friendly, and will have to develop mechanisms to interact nicely with NATs, firewalls, etc.

In this paper, we propose ModNet, a system which provides new, richer interfaces to the traditional TCP stack. The key idea behind ModNet is to loosen the boundary between the network applications and operating systems and, as a result, widen the scope for enhancement of widely used network applications, such as web servers, proxy servers and multimedia streaming servers.

ModNet provides new interfaces to allow fine-grained feedback about the network condition and to allow greater control over the buffered content to network applications. ModNet also provides an interception mechanism through *network modules* that allows easy customization of socket layer behavior. Network modules also allow quick and wider deployment of new server behaviors, which would otherwise be hard-coded into specific implementations. At the same time, all the legacy applications remain unchanged, and the TCP stack behavior is unchanged for these applications. We demonstrate ModNet through several modules, that improve mobile content adaptation,

video rebuffering and flash crowd behavior.

2 ModNet Design

The main idea behind ModNet is to give applications more insight into the operation of the network stack, and the ability to delegate management of data transfer, so that they can proactively adapt to network conditions when possible, and react to changing conditions when necessary. We want to loosen the boundary between applications and the network stack, so that applications or their delegates can see what is happening to the data that they want delivered, and to act on that process as the conditions of delivery change. At the same time, we want to ensure that all of the mechanisms we provide are safe, are deployable, and are maintainable.

We focus our efforts on three key areas:

- **Delegation** – allow sockets to be intercepted by one or more modules that can manipulate socket contents, parameters, and timings. These modules may be invoked directly by the application, or they may be automatically attached to application sockets by a user with the appropriate permissions. They can be composed, so that each module performs a specific task, but that a collection of modules can perform more complicated actions. In this manner, content between the application and the network can be manipulated, and the modules can be reused across applications where appropriate.

- **Inspection** – allow an easy way for interested application or modules to observe lower-level network behavior for its connections, and use this information to adapt its behavior. Normally, when an application sends data over TCP, the data is buffered and the application has no way to track its progress. We want applications to see what is happening to the data inside the network stack, so that they can react appropriately for any future data they generate.

- **Revocation** – where possible, allow applications or modules to "undo" their past behavior by modifying the contents that they have handed to the network stack, as long as such modifications do not cause any consistency problems. In practice, this means that any *unsent* data in a socket buffer can be modified if needed, allowing applications the flexibility of large socket buffers but the responsiveness of smaller ones.

In keeping with the idea that ModNet should enhance the network stack rather than disrupting it, we focus on implementing these behaviors with as few changes to the network stack as possible. Naturally, existing applications can continue to operate as usual with no modifications, but the delegation mechanism can even allow modules to operate on the network activity of otherwise unmodified network applications. We are interested in efficiency to the extent that it does not affect programmability, so that easier-to-use options are preferable to the highest possible performance. We note that CPU throughput (especially through multiple cores) has comfortably outpaced wide area network bandwidth, so maximum efficiency is not the primary driver for most networking applications. However, we take steps to ensure that ModNet is as efficient as possible within our constraints.

2.1 Delegation

In ModNet, the preferred mechanism for delegation is the use of modules, which are standalone processes that logically divert the flow of a socket between the operating system and the process that created it. These modules can be chained together, and to the application, the presence or absence of modules should be as transparent as possible.

In modern event driven servers (e.g. Nginx) writing complex extension modules is not easy, mainly because any blocking call inside the module can block the server's event loop and thereby, severely hurt the performance and scalability of the server. This possibility would get worse as modules are chained together. ModNet 's delegation mechanism provides a generic extension mechanism which does not impose any such restrictions, and at the same time can be reused across applications without any extra effort. In addition, since the ModNet modules are standalone processes, they can have separate privilege levels, scheduling priorities and resource limits, which might be required for implementing critical system services.

The other alternative mechanism for implementing complex web server extensions is to use loopback proxy servers. However, proxies are not performance optimized for this usage, and moreover, are tailored to specific application level protocols. A performance comparison between the Nginx loopback proxy and ModNet's delegation mechanism is presented in §5.2.

We propose an interposition scheme that interposes on the sockets. This approach allows modules to examine, process, and modify the data being passed in both directions on sockets. To distinguish this approach from the existing interposition mechanisms, we term this technique *socket stealing*. This term also better describes the mechanism involved, which looks like stealing the endpoints of an existing socket and replacing them with the endpoints of the interposed module.

A schematic of the interposition mechanism for a chain of two modules, i.e. a composition of modules, is shown in Fig. 1. The application's socket $Sock_{real}$ is stolen

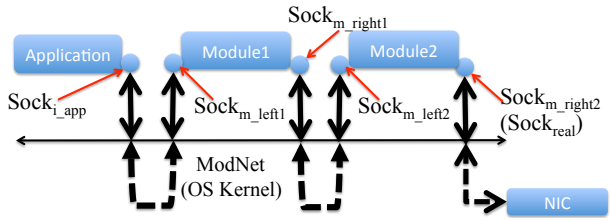

Figure 1: The architecture of ModNet's interposition mechanism for a chain of two modules. The $Sock_{real}$ is the application socket that is stolen. The black lines show the flow of data between various components.

and replaced by an *intermediate socket* $Sock_{i\_app}$, which is connected to another intermediate socket $Sock_{m\_left1}$. Since we have two modules in this case, a pair of connected intermediate sockets, namely $Sock_{m\_right1}$ and $Sock_{m\_left2}$, is created to join the two modules. To ease integration in the kernel, the sockets $Sock_{m\_left1}$ and $Sock_{m\_right1}$ are mapped to the file descriptor table of the first module. Similarly, $Sock_{m\_left2}$ and $Sock_{m\_right2}$ (i.e. $Sock_{real}$) are mapped to the file descriptor table of the second module. In general, we refer to the two sockets mapped to a module's file descriptor table as $Sock_{m\_left}$ and $Sock_{m\_right}$. The modules read the data from $Sock_{m\_left}$, optionally transform it, and write the final data to $Sock_{m\_right}$, and do the same for the reverse direction. This stealing is akin to dynamically adding bidirectional pipes within an existing network connection, and this simple interface can be used to implement a large class of network functions.

2.2 Inspection

ModNet allows inspection of progress in two ways, by examining content through the interposition mechanism, and by examining the status of connections. The interposition is an *active* inspection mechanism for modules that was described in §2.1, we describe the latter now.

Adaptive network applications, adapt their responses according to the network bandwidth, e.g. adaptive video streaming servers. An interface for the TCP state is desirable for estimating bandwidth when the clients do not explicitly provide a feedback about the network conditions, e.g. an adaptive web server [16]. We propose a generic, performance efficient interface for exposing the relevant per-socket information to the applications.

To examine connection status, ModNet allows modules and applications a fast, passive means of examining connection progress and status. The current interfaces for reading connection state for a socket (e.g. TCP state variables) are either not easy to use or are inefficient in terms of performance. For instance, in Linux one could use the tcpprobe module to read the socket state, but the interface is cumbersome, and expensive, going through the proc pseudo-filesystem. The other alternative, TCP_INFO socket option invokes a full system call, with no easy means of determining when an activity of interest occurs. For instance, when trying to use this status on each packet reception (e.g. in packet-pair [18] bandwidth estimation), the overhead of this polling can be significant.

The connection status tracking in ModNet extends the `mmap` (memory map) system call to socket file descriptors to implement shared readable control. The application or module receives a mapped memory region and typecasts it to the shared socket state structure. Among other fields, the shared socket state for a TCP socket contains the TCP state variables, the timestamps, sequence numbers and acknowledgment sequences for the two most recently received packets. We provide the measures for two packets because they might be needed for bandwidth estimation mechanisms such as packet-pair [18].

2.3 Revocation

The final mechanism in ModNet is for revocation, and allows the application or module to remove unsent content from socket buffers. With the growing popularity of HTTP, TCP has become the de facto transport protocol for many real time applications. However, the lack of interface for manipulation of socket buffer content in the current socket API makes it difficult to adapt effectively.

To better understand the need for revocation, consider the case of an HTTP based streaming video server that handles adaptive bitrate video. It receives requests from clients for fragments of a video, using the client's estimate of what bitrate it can handle. In normal operation, the server would prefer to have very large socket buffers, so that each `write` or `sendfile` system call it performs can write the associated content to the client without blocking. If it does have to block, it would prefer to block as few times as possible, for better performance. Normal socket buffer sizes might be in the range of 16KB∼128KB or as large as 1MB for high performance servers. Moreover, socket buffers are also a form of feedback control, and the application/module may wish to monitor data transfer performance and take action when the bandwidth drops. In this case, large socket buffers lead to long delays in the control loop of the application, increasing latency, and perhaps to long rebuffering times when bandwidth drops. Small socket buffers, however, not only increase the number of system calls per piece of content, but also run the risk of not meeting client bandwidth, if they are smaller than the bandwidth-delay product or if the associated application blocks or encounters scheduling delays when the socket empties. Ideally, we want large socket buffers when things are going well, and

short socket buffers when things are going poorly.

To achieve this effect, ModNet introduces a new system call, `modnet_yank`, which allows applications and modules to pull, or optionally, read a desired amount of data from the socket buffer. We describe the API details in §3.2. In the case of UDP, packets are synchronously transferred to the device queue, unless the application explicitly asks the OS to buffer them (e.g. using UDP_CORK option on Linux). Thus, `modnet_yank` is mostly applicable to TCP sockets. To ensure network consistency, it does not remove any data that has already been sent even if it has not been acknowledged.

3 ModNet API

ModNet has a simple and intuitive programming interface that provides better control and insight to network applications. The implementation of modules using ModNet API will be discussed in §4.2. Table 1 provides a concise overview of the ModNet API.

3.1 Delegation API

Since modules in ModNet can be standalone entities, some rendezvous mechanism is needed to have applications apply modules, and to this end, modules register themselves with the OS by name via the `modnet_register` system call. To steal file descriptors from a process, the module issues a `modnet_getsockets` system call and receives two file descriptors for each stolen socket, which correspond to the sockets $Sock_{m\_left}$ and $Sock_{m\_right}$ (see Fig. 1). If the module knows how many sockets the application will generate, it can call `modnet_getsockets` repeatedly, or it can register for a new EPOLL_STEAL event that ModNet delivers via the `epoll` event notification mechanism. Since the EPOLL_STEAL event is not tied to a file, the file descriptor argument should be a negative integer denoting the CPU mask. The CPU mask is used to specify the core affinity for socket stealing, which is discussed in §4.1.

Modules can be applied individually or in a chained fashion via the `modnet_apply` system call, which takes the module names and process ID. The process ID allows the application to specify that the module can be applied to itself, or to a target process, assuming they have the same owner. This mechanism can be generalized – for example, we have written a program that takes an application name and module name, and applies the module to any matching processes, or launches a new instance via `fork/exec`.

The `modnet_yield` system call allows a module to insert new modules into the chain adjacent to itself, and optionally remove itself from the chain. The system call takes two file descriptors corresponding to $Sock_{m\_left}$ and $Sock_{m\_right}$ sockets, and yields them to the specified chain of modules. The "operation" argument can be APPLY_LEFT, APPLY_RIGHT or REPLACE to instruct applying to the left of the module, right of the module or replacing the module in the chain, respectively. Naturally, the $Sock_{m\_right}$ (or $Sock_{m\_left}$) file descriptor is not required for APPLY_LEFT (or APPLY_RIGHT) operations. A usage example is the following: an HTTP filtering module that intercepts the web server connections, and based on the request invokes some other modules (e.g. Gzip compression, SSD swap, etc.) and removes itself. `modnet_yield` can also be used by applications (using the APPLY_RIGHT operation) to specify a chain of modules per-socket instead of using the `modnet_apply`, which specifies a fixed list of modules for all sockets. One can argue that the `modnet_yield` system call provides a sufficient mechanism for chain manipulation since each module is expected to be independent, and no module should modify remote portions of the chain.

3.2 Revocation API

The `modnet_yank` system call is used for yanking unsent content from socket buffers and modules. It can also be used for reading the existing data from socket buffers by specifying the "operation" argument as PEEK instead of the default, i.e. YANK. Peeking might be useful in case the application wants to make the revocation decision based on the data. For instance, an HTTP streaming server might want to find a legal video frame boundary before yanking. While the application is making the decision about what to yank, it might want to prevent transmission of any new data, e.g. the streaming video server might want to prevent sending the data at the boundary of video frame. Thus, we support locking the transmission as a side effect of `modnet_yank`. Once the application locks the transmission, it should also be able to unlock the transmission. In order to allow this control of the application over data transmission, `modnet_yank` takes a "lock" argument that can be YANK_LOCK, YANK_UNLOCK or YANK_NONE.

In the case of chained modules, a call to `modnet_yank` might require the succeeding modules in the chain to reconstruct the original data. We added the EPOLL_YANK_REQ event for instructing the modules to reconstruct data. Specifically, a `modnet_yank` call on a $Sock_{i\_app}$ socket or an intermediate $Sock_{m\_right}$ socket leads to an EPOLL_YANK_REQ event on the succeeding module's $Sock_{m\_left}$ socket. If the succeeding module has not registered an EPOLL_YANK_REQ event on the $Sock_{m\_left}$ socket, the call returns an appropriate error (e.g. EOPNOTSUPP). On receiving the EPOLL_YANK_REQ event, the succeeding module

System call	Description
*modnet_register(char *mod_name)*	registers the calling process using the specified module name, or fails if an existing registration has the same name. The unregister happens on the exit.
*modnet_apply(pid_t target, char *mod_names[], int num_mods)*	applies the named modules to the calling process or another process by pid, in the given order. Fails if either of the names do not have a corresponding active module.
modnet_getsockets(int left_fds[], int right_fds[], long cpu_mask)	gets pairs of sockets from module's steal queue and writes the two file descriptors corresponding to $Sock_{m\_left}$ and $Sock_{m\_right}$, in the array arguments, and optionally provides CPU affinity, discussed in §4.1
*modnet_yield(int left_fd, int right_fd, char *mod_names[], int operation)*	yields the pair of sockets with file descriptors $left\_fd$ and $right\_fd$ to the specified chain of modules. Fails if the file descriptor arguments are illegal or any of the module names do not have a corresponding active module. See §3.1 for details.
*modnet_yank(int fd, void *buf, int len, int operation, int lock, int flags)*	removes or copies upto the requested amount of unsent data from the socket buffer. See §3.2 for details.
*modnet_yankwrite(int fd, void *buf, int len)*	writes the supplied amount of data to the yank buffer of preceding socket in the chain. This system call is used by a module for returning the reconstructed data in case the preceding module/application in the chain calls a yank. See §3.2 for details.

Table 1: An overview of ModNet API

should reconstruct the original data, if required, perform the yank operation recursively and write back the reconstructed data by calling `modnet_yankwrite`.

The reconstructed data is held in the *yank buffer* of the preceding socket in the chain. The data might be partially written if the yank buffer is full. The size of yank buffer is a configurable system parameter. While the modules reconstruct the original data, the `modnet_yank` call might block or return immediately with an appropriate error (e.g. EAGAIN) depending on whether the YANK_DONTWAIT is set in the "flag" argument or not. The EPOLL_YANK event can be used for monitoring the readiness of yank. Note that the readiness of yank refers to the case when there is sufficient data in the yank buffer to satisfy the request.

3.3 Inspection API

The inspection API does not introduce any new system calls. The shared memory mechanism can be exposed by extending the `mmap` system call as described in §2.2. Since currently `mmap` is not supported for TCP sockets, this extension does not affect the existing systems.

In the case of chained modules the `mmap` for all the intermediate sockets returns the pointer to the shared socket state of the real socket. This allows seamless composition of modules. For instance, an image compression module would read the same network state variables regardless of whether a succeeding module has been applied to it.

4 Implementation

We have implemented ModNet on Linux, modifying 364 lines of existing kernel source code and adding 2758 new lines of code to implement the new system calls and behavior we have introduced into existing functions, header file modifications and modification of the `epoll` mechanism to support EPOLL_STEAL event, which is tied to the current process rather than a file, and the EPOLL_YANK and EPOLL_YANK_REQ events, which take an additional length parameter. Below, we describe the implementation of the major components of ModNet.

4.1 Socket Stealing

If a module has been applied to a process, ModNet intercepts the creation of any new network sockets by the process. Two intermediate sockets are created, which correspond to $Sock_{i\_app}$ and $Sock_{m\_left}$. The $Sock_{i\_app}$ is mapped to the file descriptor table of the application and the corresponding file descriptor is returned to the application. The original socket, i.e. $Sock_{real}$, and the intermediate $Sock_{m\_left}$ socket are added to the module's *steal queue*. In case of chained modules (e.g. Fig. 1), a pair of connected intermediate sockets is also created for every two adjoining modules in the chain. A module reaps the entries of its *steal queue* using the `modnet_getsockets` call. The intermediate sockets are developed on top of the local UNIX domain socket implementations.

To reduce the interposition overheads, we support batching and processor affinity for socket stealing. To amortize the costs of the `modnet_getsockets` system call, it returns the file descriptors for multiple stolen sockets in one call. Knowing that modules will often copy data, we want to allow the source and sink of the data to use the same processor cache. Borrowing the idea from Affinity Accept [23], we provide support for performing

all the processing for a stolen socket on the same core. For each module, a *steal queue* is maintained per core. Any stolen sockets are enqueued to the *steal queue* of the local core. The `modnet_getsockets` call takes a bitmask as an argument called *cpu_mask*, and returns sockets only from the *steal queues* of the specified cores. While implementing the modules, we pin a thread on each core and each thread calls `modnet_getsockets` with a mask that is "on" only for its local core.

To make the socket stealing mechanism as transparent to the original application as possible, we must ensure that operations intended for the original socket are actually received by the original socket. For example, if the application issues a `getpeername` system call, it would (in the absence of modules) expect to get information about the other TCP endpoint. To ensure this, all the socket system calls for the intermediate sockets, other than `recv`, `send`, `shutdown` and the event notifications (`epoll`) are directly translated to the corresponding $Sock_{real}$ socket. By translating a system call, we mean the effects and the return value of the system call on the intermediate socket will be identical to that of the same system call on the $Sock_{real}$ socket. The general file operations, like `close, dup, fcntl`, etc. have their usual semantics for both the intermediate sockets and the original socket. Since the stolen $Sock_{real}$ socket is the application's original socket, all the socket system calls have regular semantics for it.

4.2 Implementing Modules

Module implementations are very similar to proxies, only are simpler because they do not have to implement the whole protocol. Figure 2 shows a simplified psuedo-code for bi-directional forwarder module.

We have developed some sample modules for ModNet using the Libevent library [5] to provide scalability. A complete bi-directional forwarder module serves as the template for other modules, and is capable of handling roughly 80K connections per second (setup and teardown) with a moderately powerful 4 core server. This template is 440 lines of C code, excluding the Libevent library. Most of the code for forwarder module can directly be reused while implementing other modules. For example, the implementation of the adaptive gzip compression module §5.3 reuses this code with only 22 lines of changes.

5 Applications and Evaluation

We begin by characterizing the performance of ModNet's delegation framework through web server microbenchmarks in §5.2. The subsequent sections present evaluations of using ModNet to solve some important problems for network servers for emerging classes of Internet traffic. We evaluate an adaptive gzip compression

```
epfd = epoll_create(...)
ev.events = EPOLL_STEAL
epoll_ctl(epfd, EPOLL_CTL_ADD, -mask, &ev)

while (true):
  ev = epoll_wait()
  if (ev.events & EPOLL_STEAL):
   modnet_getsockets(fd_left, fd_right, mask)
   foreach (pair of fd_left,fd_right):
    other[fd_left] = fd_right
    other[fd_right] = fd_left
    ev.events = EPOLLIN | EPOLLRDHUP
    ev.data.fd = fd_left
    epoll_add(epfd,EPOLL_CTL_ADD,fd_left,&ev)
    /* similarly add events for fd_right */
    .....
  elif (ev.events & EPOLLIN):
    /* read from ev.data.fd, and write
    to other[ev.data.fd] (omits the code
    for handling the case where the
    write buffer is full) */
    .....
  elif (ev.events & EPOLLRDHUP):
    /* signal shutdown to other end, to abide
       by protocols that are sensitive to
       end of streams (e.g. HTTP) */
    shutdown(other[ev.data.fd], SHUT_WR)
```

Figure 2: A simplified psuedo-code for the event-based bi-directional forwarder module.

module for data (§5.3) and an adaptive JPEG compression module for images (§5.4), which handle the variable network conditions for mobile clients. We also present the evaluation of a socket buffer swap module (§5.5), which augments the total socket buffer space by offloading part of it to SSD, and optimizes resource consumption in case of a wide spectrum of client bandwidths. §5.6 contains the evaluation of a deduplication module that handles flash crowds better by reducing duplicate buffering of content across sockets. §5.7 evaluates the use of yanking socket buffers to improve the ability of an HLS (HTTP Live Streaming [21]) server to respond to network conditions.

5.1 Experimental Setup

All the machines used in the experiments are 3.5 GHz, 4-core Intel(R) Xeon(R) E3-1270 v3 processors with 8GB of DRAM. Hyperthreading is enabled for all the experiments. The server runs our modified Linux 3.13.5 kernel, while clients run standard Linux kernels.

Each machine has two NICs: a 10Gb NIC and a 1Gb NIC. Each machine has a 256 GB SSD drive as secondary storage, attached via a SATA-III (6Gbps) port. The SSD can sustain up to 100K IOPS and 90K IOPS, for reads and writes respectively, each of size 4KB.

We use the Linux in-kernel traffic shaper (TC) [6] for regulating link bandwidths in our experiments. To emulate the bandwidth characteristics of a real network, we

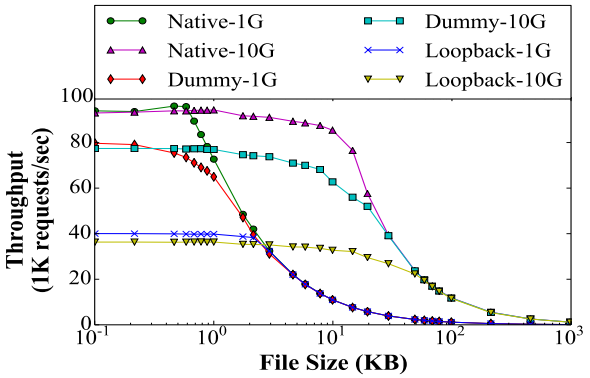

Figure 3: The throughput (thousands of requests handled per second) of Nginx, Nginx with dummy module applied to it, and Nginx behind a loopback proxy. The results are shown for both 1Gbit and 10Gbit NICs.

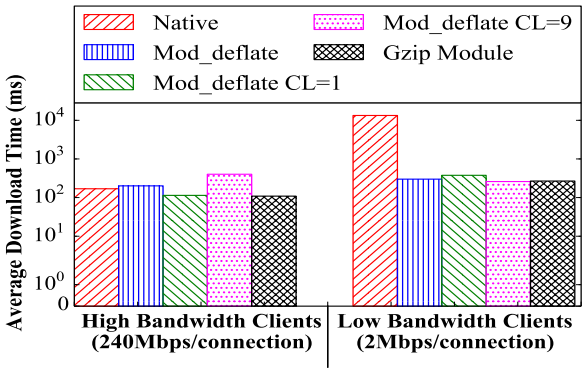

Figure 4: Performance comparison of adaptive gzip module with various configurations of mod_deflate (Apache's gzip compression implementation).

use bandwidth traces of a 3G mobile network [24] in some experiments. The summary of six different traces that we use in our experiments is provided in table 2.

5.2 Overheads of Delegation

In this section, we characterize the performance overheads of ModNet's delegation framework by studying the overheads of applying a dummy module to a web server. The dummy module simply forwards data in both directions. In our micro-benchmark, a large number of clients request the same static file repeatedly, for various file sizes. The workload is CPU bound for small files and network bound for larger files.

Figure 3 shows the number of connections handled per second for a single instance of the native Nginx Web server (Native) and for the case where the dummy module is applied to it (Dummy). For comparison, we also include measurements for a loopback Nginx proxy (Loopback) applied to the Nginx instance, both of which are running on the same machine. Experiments were performed separately for the 1Gbit and 10Gbit NICs, and the corresponding results are marked with suffixes "-1G" and "-10G", respectively, in the figures. The poor performance of loopback proxy can be attributed to the use of heavy-weight IP sockets for intermediate connection between proxy and servers, and, in general, not being as well-optimized as the module implementation for this specific usage.

To generate the workload, we used two client machines running a total of 400 concurrent clients. Non-persistent connections were used in order to fully expose the per-connection overhead of the module. For small files the workload is CPU bound and ModNet poses an overhead in the range of 15-25%, which is much lower than the 50% overhead of loopback proxy. As the file size increases, the workload becomes network bound and ModNet's overhead approaches 0, after which the module provides a throughput close to the network bandwidth (i.e. 10Gbps or 1Gbps).

In the interest of space we only discuss the conclusions from our experiment of the effect of chaining dummy modules on throughput. The throughput dropped by ∼13% per additional module for a 100 byte file, and we observed 52% and 75% throughput drop for chains with 5 and 10 dummy modules, respectively. However, with growing number of processor cores, WAN bandwidth is expected to be the bottleneck for common file sizes. Moreover, even with small files we were able to handle as large as 23K connections per second for a chain of 10 modules.

Trace Name	Duration (s)	Mean BW (bits/s)	BW CV
Trace img1	457	2.67 M	0.54
Trace img2	390	1.95 M	0.58
Trace img3	1036	1.51 M	0.60
Trace vid1	308	730 K	0.93
Trace vid2	619	735 K	0.85
Trace vid3	430	605 K	0.95

Table 2: Total trace duration (in seconds), Mean bandwidth of the trace (BW) and the coefficient of variation (CV) of the bandwidths for various 3G bandwidth traces used in the experiments.

5.3 Adaptive Gzip Compression

Many web servers and proxies implement run-time content compression, so clients can still save bandwidth even when the original content was not compressed. In these cases, run-time compression can introduce extra CPU overhead, which is reasonable for slower clients, but may be a problem with fast clients or when the server CPU becomes overloaded. We use ModNet's inspection facilities to obtain fine-grained information about the network

condition and adapt accordingly. Specifically, the adaptive gzip module periodically reads the socket state and drops the compression level for that transfer if its TCP congestion window is bigger than the socket buffer data, and raises the compression level otherwise.

For evaluating the adaptive gzip module, we perform a similar experiment as in the last section, i.e. §5.2. The number of clients was fixed to 40 for this experiment, because the compression process starts fully utilizing the CPU at that point. We used the monthly usage report of a personal Amazon EC2 account. These reports are good examples of large, dynamically generated and highly compressible content. The document size for uncompressed monthly report was 3MB and the compression ratio was in the range of 34-51.

Fig. 4 depicts the average download times for the following configurations: Apache without gzip compression (Native), Apache's gzip compression active with the compression level settings, default, i.e. 6 (Mod_deflate), 1 (Mod_deflate CL=1) and 9 (Mod_deflate CL=9), Apache with gzip compression disabled and our adaptive gzip module applied to it (Gzip Module). Note that the Y-axis is in log scale to accommodate large range of values.

To demonstrate the benefit of the adaptive behavior, we perform experiments with two classes of clients: (1) high bandwidth clients, where each client has a bandwidth of 240Mbps, and (2) low bandwidth clients, where each client has a bandwidth of 2Mbps, which is around the median of the mobile bandwidth samples we used. As shown in the figure, for high bandwidth clients, the bottleneck is compression speed, and thus, compression level 1 is almost 4X faster than compression level 9, 2X faster than the default case and 1.5X faster than no compression. However, for low bandwidth clients network bandwidth is the bottleneck and so the compression level 9 is almost 1.5X faster than compression level 1, 1.2X faster than the default compression level and 51X faster than no compression. The adaptive gzip module gives the optimal performance for both cases. Although this experiment is designed to emphasize the importance of inspection mechanism, it is worth noticing that the even with a high bandwidth as 240Mbps (i.e. 40 clients on a 10Gbps uplink) ModNet's modularization overhead does not have any visible effect on performance relative to that of the compression process, while we get extra flexibility by using a separate network module.

5.4 Adaptive Image Compression

Given the enormous variability in bandwidth of clients in the current internet infrastructure and the fact that more than 60% [2] of the transferred bytes for an average webpage are images, serving images at a fixed resolution may be suboptimal. Even serving image resolution based on device type may be problematic, since smartphones with wi-fi access may be faster than desktops using dial-up.

Website	Number of images	Total Size
BBC	24	940KB
IMDB	44	314KB
Pinterest	57	1082KB
Yahoo	20	282KB

Table 3: Characteristics of the Image datasets used

An adaptive approach could select from multiple statically compressed variants of the same image or could dynamically re-compress the images. We used dynamic recompression of images because it allows us to change the compression level on the fly based on a passive bandwidth estimate that is acquired as the connection progresses. Moreover, dynamic re-compression is suitable for transformational proxies, which have been argued to be better for incremental deployment and amortization of operating costs [13] (Google has already deployed a compression proxy that dynamically re-encodes images [2] and other content for the chrome browser).

We employed ModNet's inspection mechanism in order to obtain a fine-grained estimate of the client's bandwidth. We use a passive bandwidth estimation mechanism since it allows us to work with unmodified clients. Our estimation mechanism is based on the packet-pair [18] estimation, where we consider the pair of last two acknowledged packets if they were sent close enough and have similar sizes. Our bandwidth estimation mechanism works well for HTTP file downloads. We used ModNet's delegation framework for implementing the adaptive Jpeg module. This recompression could be performed at a server or at a performance-enhancing middlebox [30].

We use the JPEG image format for this module, since it is widely used and supported across browsers. Changing the image resolution dynamically for JPEG images is, however, not straightforward, because as per the JPEG specification [4], there is a single quantization matrix for each color component, and it precedes the whole scan data. We devised a new scheme where we zero out the higher coefficients to get a better run length encoding (RLE), and thus better compression ratio.

Fig. 5 and Fig. 6 show the total download time and average image quality for the four image datasets, for the native Nginx web server (Native) and with our adaptive JPEG module applied to it (Jpeg Module). We used the SSIM index [29] for estimating the image quality. The bandwidth is being shaped according to the 3G network traces (see Table 2 for details). Results suggest that the adaptive JPEG module keeps the download times reason-

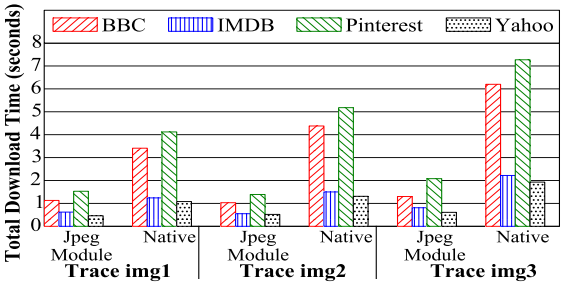

Figure 5: The load times for Nginx and Nginx with adaptive JPEG module

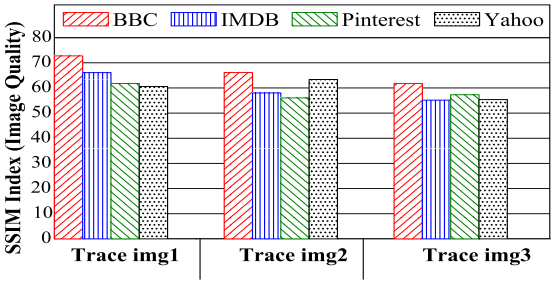

Figure 6: The average quality estimates (SSIM indices) for Nginx with adaptive JPEG module

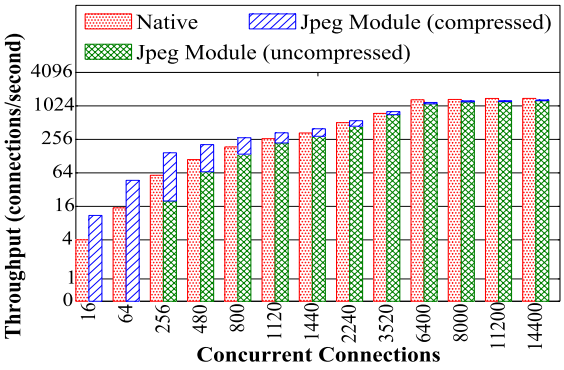

Figure 7: The throughput of Nginx with and without adaptive JPEG module for BBC image dataset. Note that Y-axis is in log scale.

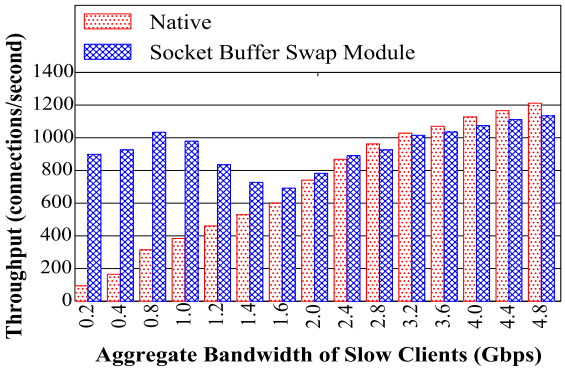

Figure 8: The performance comparison of dynamic content serviced by Apache with and without the socket buffer swap module.

able, regardless of how poor the client's network bandwidth is. It is worth mentioning here that the trade-off between the reduction in size vs. degradation in image quality is a policy question, and the results are shown for one specific policy that we used. For this experiment, we used only one active client. We now study the effect of concurrent connections.

For a large number of concurrent clients, the computational cost of re-encoding images becomes a matter of concern. However, since we only change the coefficients, we only have to handle the RLE step, and not the more expensive DCT processing. We estimated that doing an inverse DCT and a DCT for each image would require almost 3 times more computation.

To provide consistent throughput, the module only compresses a fraction of images if the CPU becomes the bottleneck. The throughput results for the adaptive JPEG module are shown in Fig. 7. The *Jpeg Module (compressed)* and *Jpeg Module (uncompressed)* correspond to the fraction of connections being serviced as compressed and uncompressed images, respectively, with the adaptive JPEG module applied to Nginx. Each connection involves a request for all the images in the BBC image data-set. The "Trace img2" (ref. table 2 for details) bandwidth trace is used for this experiment. The adaptive JPEG compression module provides up to 3 times more throughput when the server is not heavily loaded.

5.5 Swappable Socket Buffers

To demonstrate the flexibility of the ModNet approach, we demonstrate its behavior in managing network socket buffer space, irrespective of usage. A large socket buffer can increase performance by reducing the chances of an application getting blocked on socket writes. As an example, consider an Apache server handling PHP requests, which use a separate process per connection. These systems typically cap the number of processes to avoid overloading the server, but if too many slow clients access the server, the PHP processes may be blocked on writing to the clients, even if the responses have been generated. Increasing the socket buffer size can reduce this chance and free the application resources early, at the expense of increasing kernel memory consumption.

One solution to this problem is to swap socket buffers that are being drained too slowly, which reduces kernel memory usage while still allowing large socket buffers to free application resources. With the advent of flash storage devices, which support fast random reads, the sec-

ondary storage is a natural candidate for swapping this overflow content.

We used ModNet's delegation framework to implement a socket buffer swap module that takes care of socket buffering by yanking data from slowly-draining socket buffers and swapping it to the SSD once the socket buffer and the designated per-connection memory are full. Although swapping of excess content to secondary storage increases the throughput when the bottleneck is network, it can even degrade the throughput if disk becomes the bottleneck. In order to address this issue, we implement an adaptation mechanism to prevent swapping of new content once the disk load is high. The swap module decides whether or not to swap content by comparing the network bandwidth and the expected disk throughput, which is estimated based on the number of outstanding operations.

It is worth emphasizing here that ModNet's delegation framework makes it very easy to deploy such system-wide policies, and allows prioritizing the scheduling of such performance critical processes. Additionally, since the modules are standalone processes it also makes the process secure by running the module with appropriate privileges to the swap area. Moreover, the inspection mechanism allows us to evaluate network condition, and decide when to swap. Revocation is used for swapping existing data in case of sudden changes in conditions.

For this experiment, we used a mix of low and high bandwidth clients. The per client bandwidth for low bandwidth clients varies from $\sim$175Kbps (making the lower end of aggregate bandwidth 0.2Gbps) to $\sim$4Mbps (making the higher end of aggregate bandwidth around 4.8Gbps). The high bandwidth clients will collectively receive the residual bandwidth of the server's 10 Gbps link. We used 1200 low bandwidth clients and an equal number of high bandwidth clients, which repeatedly request a dynamically generated file of size 800KB.

For generating dynamic content, we use a PHP script that generates 800KB of data and use Apache's mod_php plugin to serve it. Fig. 8 shows the throughput of native Apache (Native) and Apache with the socket buffer swap module applied to it (Socket Buffer Swap Module). We set the maximum limit on Apache's worker processes to 512, because increasing the limit beyond that reduces its performance.

We see more than 9X improvement in throughput when there a number of clients with considerably lower bandwidth. As the bandwidths of these low bandwidth clients increase, their request rate also increases, because each request takes less time to finish. Therefore, the throughput starts dropping after a point because of the increased disk load. Note that it still performs better than blocking on the network, until the point where disk throughput becomes the bottleneck. Once the disk throughput becomes the bottleneck, the adaptation mechanism will try to prevent the further offloading of content to SSD; we see a slightly lower performance after this point because the adaptation mechanism is not perfect.

5.6 Deduplicating Socket Buffers

While caching mechanisms and CDNs can be used to handle flash crowds for static content, scalable server instances are required in case of dynamic content. With the growing complexity of web pages [10], the memory pressure of socket buffers can become a limiting factor for web server scalability. The socket buffer swap module (§5.5) can be used to reduce the memory pressure at the expense of higher disk I/O. In this section we describe the deduplication module, which reduces the memory pressure at the expense of higher CPU utilization.

The deduplication module exploits the fact that responses generated by web servers often contain large amounts of template material, such as in the case of dynamic content [14]. Web servers can avoid the duplicate buffering for static files by using the `sendfile` system call (or its equivalents). However, there is no easy mechanism to avoid this extra memory pressure for web proxies or web servers generating dynamic content. We implemented a deduplication module using ModNet's delegation framework that reduces duplicate buffering for servers. As argued in §5.5, ModNet modules greatly ease the deployment of such "OS-like" services.

We use Rabin fingerprinting to detect duplicate chunks, and share a single copy of the duplicated content by using Linux's vmsplice `vmsplice` system call. Fig. 9 shows the memory usage versus the number of concurrent connections for Nginx (Native) and Nginx with deduplication (Deduplication) module applied to it. All the connections request a dynamically generated file with the same template. We used the Yahoo homepage as the template and inserted scripts for portions we deemed would vary across downloads by different users. The size of the page was 346KB on an average, and the dynamic portion was less than 300 bytes. As can be seen in the graph, the memory pressure reduces by up to 7X for this experiment.

Note that varying the number of connections does not affect the throughput for this experiment because the bottleneck is network bandwidth throughout the range of this experiment. The average application throughput was close to 930Mbps for both with and without deduplication on the 1Gbps link. The average CPU utilizations were 31% and 63% for Native and Deduplication, respectively. However, the CPU was fully utilized for the 10Gbps link,

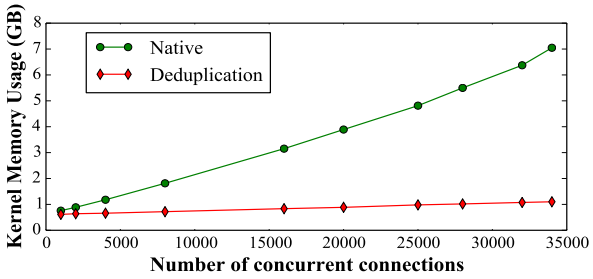

Figure 9: The kernel memory usages for Native Nginx and Nginx with deduplication module applied to it.

and the deduplication was only able to deliver ∼3.2Gbps of throughput.

The relative CPU overhead of deduplication increases as the content generation processing decreases. Therefore, we emulated zero generation time using static content in order to expose the maximum overhead; the average CPU utilizations for Native and Deduplication were 17% and 52% in this case. Thus, the maximum processing overhead of deduplication is around 200%. However, the fact that the deduplication module was able to sustain the 1Gbps link with a moderately powerful 4-core server makes this a highly practicable solution.

5.7 Video Streaming with Revocation

While the swapping and deduplication modules yanked and later re-inserted content into the socket buffer, another use of ModNet's revocation mechanism is to change unsent content in the socket buffer. We target adaptive bitrate video, where the client requests video fragments encoded at multiple bitrates. If the client chooses the wrong bitrate or if the network connection abruptly changes, the viewer can experience rebuffering. As discussed in §5.4, ModNet's shared state mechanism can be used for a packet-pair based passive bandwidth estimation.

We implement yank support in Mistserver [8], an open source HLS [21] server, so that the server can participate in the adaptive bitrate system. We implement two approaches – in the first, the server monitors bandwidth and truncates any in-progress transfer, leaving any content already in the socket buffer to be sent. In the other, not only does any remaining content get stopped, but any unsent data is also yanked from the socket buffer. Although truncation is not officially supported by the HLS [21] protocol, we have tried our implementation with two popular players, VLC and Quicktime, and both of them were able to play the video without any visible problems. In order to support persistent connections, we use the chunked encoding to produce HTTP responses of variable size. We use "Big Buck Bunny" [1] as the video clip for streaming. The duration of segments and encoding levels were chosen in accordance with best practices stated in the HLS documentation [3].

We serve video segments at the best bitrate encoding that can be sustained by the current estimated bandwidth. In case the bandwidth drops below the bitrate of the current encoding, we truncate the video segment. While truncating, we can use the `yank` system call to remove the pending socket buffer data at any legal boundary. The server prefixes any pending video fragment from the last segment in each response. The entire segment in a response is encoded at a bitrate that can be sustained by the bandwidth that was recorded at the end of transmission of the last segment.

Fig. 10 shows a plot of segment bitrates vs. time for the requested segments for a synthetic bandwidth variation data-set, using VLC as the client. Researchers have conducted similar studies in the past [9] for other adaptive HTTP-based video streaming protocols. Results are shown for the following three variants: (1) adaptive, which is the default HLS behavior, (2) truncate, which uses server-side adaptation and truncation of segments, and (3) yank, which uses server-side adaptation and employs yank to perform better truncation. The plot clearly demonstrates that the standard adaptive protocol reacts very slowly to steep bandwidth changes, a server-side truncation mechanism allows a relatively faster reaction, and using yank allows us to react almost instantaneously.

Fig. 11 shows the re-buffering durations, i.e. the periods when the player has no video segments to play, and the startup times, i.e. the period before the video playback begins, for the different bandwidth traces of a real 3G network [24] (see table 2 for details). Our version of VLC starts the playback as soon as it has downloaded one full segment; earlier versions used to start playback after downloading two full segments. Note that we have only shown the results for a small set of representative traces that exhibit some amount of re-buffering.

From this test, we see that simply using server-side adaptation to truncate ongoing segments and change bitrate can yield some improvement over client-side adaptation. However, being able to use ModNet's `modnet_yank` can reduce the rebuffering and startup time by as much as a factor of 2-5 for these traces. At the same time, the normal operation of the server is not impacted, since it can continue to use large socket buffers when the client's bandwidth estimation is correct.

6 Related Work

ModNet can be viewed as a combination of an interposition system and a proxy, although it has more network interaction than either of those systems. In this section, we discuss related systems that have not already been men-

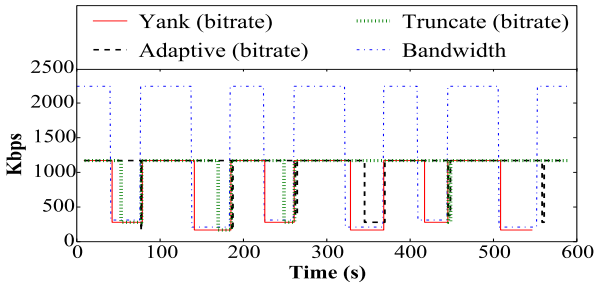

Figure 10: The bitrate of video segments for a synthetic bandwidth trace for all the three variants: adaptive, truncate and yank.

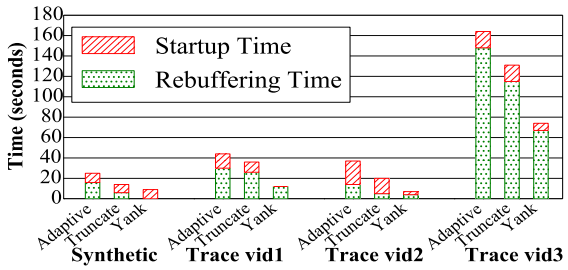

Figure 11: The rebuffering duration and startup times.

tioned in the paper.

Various kinds of proxies can provide some of the same kinds of behaviors we have shown in this paper. Fox et. al. [12] propose the use of transformational proxies to perform compression of content according to network bandwidth, screen size and other client's characteristics, but heavily rely on client-side support for bandwidth estimation. Krishnamurthy et. al. [16] propose characterization of clients based on network connectivity for adapting web server responses, which is a server-side response to similar work that was done amongst clients by Seshan et. al. [27]. We believe that ModNet's interface for exposing connection status information makes this process more direct, and can augment client-side estimation as shown in our adaptive video experiment.

Other proxy work has implemented portions of the work in ModNet. Rosu et. al. [25] propose a shared memory abstraction for exposing some socket state information to applications. However, their main intention behind doing so is to implement a fast select/poll mechanism at user level. Connection conditioning [22] also uses a chained series of services. However, their mechanism is specific to web servers, and only handles the request path, not responses. Furthermore, their implementation, which is purely in user space, is considerably different than ours, and demands application changes. Other loopback proxy approaches have much higher performance overheads, as we showed in the microbenchmark experiments.

Packet interception mechanisms, such as packet filtering [19, 20] or virtual network devices such as TUN/-TAP in Linux, might allow a user space daemon to intercept the packets and modify them (e.g. the Linux libnetfilter_queue [31] mechanism). Since the daemon intercepts individual packets, it is not suitable for connection-oriented processing. Specifically, doing things like multiplexing connections through read/write events or `mmap`-ing sockets to read the connection state will not be possible. Even if some connection tracking mechanism was to be used in conjunction, these would demand extra programmer effort.

Much work in general has taken place on user-level network stacks, with the goal of avoiding the long delay in kernel adoption. Some implementations [11, 28] allow some application-level flexibility, although the development and maintenance efforts may make them unattractive for many domains. The most successful user-level stack is arguably Click [15], which has shown that flexibility can be more desirable than raw speed, which has shaped some of our design choices. Not all user level approaches have implemented the full stack. Tesla [26] is a framework for transparently implementing session-layer services, such as compression, encryption, etc. Although Tesla is more specialized for session-layer services, ModNet's module framework is more generic.

In comparison to user-level stacks, some work has been done on entirely new protocols to avoid these problems, such as DCCP [17]. This protocol implementation provides a shared packet ring abstraction to allow manipulation of buffered data, which they call *late-data choice*. The problem with this approach, however, has been slow deployment at end hosts, limiting application adoption.

7 Conclusions

We presented the design and implementation of ModNet, which increases the flexibility of network stack by introducing a framework for delegating network stack management, inspecting connection progress, and revoking unsent content. We demonstrated a range of modules that allow dynamic control of data generation, socket buffer management, and server behavior, at time scales and granularities not easily achieved with existing interfaces. We believe that the small amount of kernel change introduced by ModNet is palatable, and the additional mechanism is small, general-purpose, and can be easily maintained, raising the chances that ModNet or something like it will have greater deployability than custom protocols or other approaches with higher barriers to adoption.

References

[1] Big Buck Bunny: http://www.bigbuckbunny.org/.

[2] Google Chrome data compression proxy: https://developer.chrome.com/multidevice/data-compression.

[3] HLS Best Practices: https://developer.apple.com/library/ios/technotes/tn2224/_index.html.

[4] JPEG Specifications: http://www.w3.org/Graphics/JPEG/itu-t81.pdf.

[5] Libevent: http://libevent.org/.

[6] Linux Advanced Routing & Traffic Control: http://www.lartc.org/manpages/tc.html.

[7] Microsoft smooth streaming:http://www.microsoft.com/silverlight/iis-smooth-streaming/.

[8] The Mistserver wiki: http://wiki.mistserver.org/.

[9] S. Akhshabi, A. C. Begen, and C. Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 157–168. ACM, 2011.

[10] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 313–328. ACM, 2011.

[11] D. Ely, S. Savage, and D. Wetherall. Alpine: A user-level infrastructure for network protocol development. In *USITS*, volume 1, pages 15–15, 2001.

[12] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 160–170, New York, NY, USA, 1996. ACM.

[13] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *Personal Communications, IEEE*, 5(4):10–19, 1998.

[14] D. Gibson, K. Punera, and A. Tomkins. The volume and evolution of web page templates. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 830–839, New York, NY, USA, 2005. ACM.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[16] B. Krishnamurthy and C. E. Wills. Improving web performance by client characterization driven server adaptation. In *Proceedings of the 11th international conference on World Wide Web*, pages 305–316. ACM, 2002.

[17] J. Lai and E. Kohler. Efficiency and late data choice in a user-kernel interface for congestion-controlled datagrams. In *Proc. SPIE*, volume 5680, pages 136–142, 2005.

[18] K. Lai and M. Baker. Measuring link bandwidths using a deterministic model of packet delay. In *ACM SIGCOMM Computer Communication Review*, volume 30, pages 283–294. ACM, 2000.

[19] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*, pages 2–2. USENIX Association, 1993.

[20] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the eleventh acm symposium on operating systems*, pages 39–51, 1987.

[21] R. Pantos. HLS RFC: http://tools.ietf.org/html/draft-pantos-http-live-streaming-12.

[22] K. Park and V. S. Pai. Connection conditioning: Architecture-independent support for simple, robust servers. In *NSDI*. USENIX, 2006.

[23] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European conference on Computer Systems*, pages 337–350. ACM, 2012.

[24] H. Riiser, P. Vigmostad, C. Griwodz, and P. Halvorsen. Commute path bandwidth traces from 3G networks: analysis and applications. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 114–118. ACM, 2013.

[25] M.-C. Rosu and D. Rosu. Kernel support for faster web proxies. In *USENIX Annual Technical Conference, General Track*, pages 225–238, 2003.

[26] J. Salz, H. Balakrishnan, and A. C. Snoeren. Tesla: A transparent, extensible session-layer architecture for end-to-end network services. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[27] S. Seshan, M. Stemm, and R. H. Katz. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, pages 1–18, 1997.

[28] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking (TON)*, 1(5):554–565, 1993.

[29] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13(4):600–612, 2004.

[30] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. *ACM SIGCOMM Computer Communication Review*, 41(4):374–385, 2011.

[31] H. Welte. The libnetfilter_queue home: http://www.netfilter.org/projects/libnetfilter_queue/index.html.