xOMB: Extensible Open Middleboxes with Commodity Servers

James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat
University of California, San Diego
{jwanderson,rbraud,rkapoor,gmporter,vahdat}@cs.ucsd.edu

ABSTRACT

This paper presents the design and implementation of an incrementally scalable architecture for middleboxes based on commodity servers and operating systems. `xOMB`, the eXtensible Open MiddleBox, employs general programmable network processing pipelines, with user-defined C++ modules responsible for parsing, transforming, and forwarding network flows. We implement three processing pipelines in `xOMB`, demonstrating good performance for load balancing, protocol acceleration, and application integration. In particular, our `xOMB` load balancing switch is able to match or outperform a commercial programmable switch and popular open-source reverse proxy while still providing a more flexible programming model.

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

Keywords

middlebox, application-layer switch, network processing pipeline

1. INTRODUCTION

Network appliances and middleboxes performing forwarding, filtering, and transformation based on traffic contents have proliferated in the Internet architecture. Examples include load balancing switches [6–8] and reverse proxies [3, 10, 16–18], firewalls [15, 31], and protocol accelerators [13, 16, 18]. These middleboxes typically accept connections from potentially tens of thousands of clients, read messages from the connections, perform processing based on the message contents, and then forward the (potentially modified) messages to destination servers.

Middleboxes form the basis for scale-out architectures in modern data centers, being used in three main roles. First, they perform *static load balancing* and possibly *filtering*, whereby they distribute (or drop) messages to server pools based on a fixed configuration. For example, a load balancing switch might forward requests to different server pools based on URL. Second, middleboxes perform *dynamic request routing and application integration*, where they execute service logic and use dynamic service state to forward requests to specific application servers and often compose replies from many application servers for the response. For example, front-end servers use object ids to direct requests to the back-end servers storing the objects. Third, middleboxes perform *protocol acceleration* by caching/compressing data and responding to requests directly from their caches. For example, services deployed across multiple data centers use middleboxes to cache content from remote data centers.

Unfortunately, the architecture of commercial hardware middleboxes consists of a mixture of custom ASICs, embedded processors, and software with, at best, limited extensibility. While firewalls, NATs, load balancing switches, VPN gateways, protocol accelerators, and other middleboxes perform logically similar functionality, they are individually designed by niche providers with non-uniform programming models. These boxes often command a significant price premium because of the need for custom hardware and software and their limited production volumes. Extending functionality to new protocols may require new custom hardware. Worse, expanding the processing or bandwidth capacity of a given middlebox may require replacing it with a higher-end model. Similarly, software reverse proxy middleboxes are specialized for specific protocols and, like their hardware counterparts, offer limited scalability and extensibility.

At first blush, software routers such as Click [28] or Route Bricks [21] may be employed to achieve extensible middlebox functionality. In fact, recent work [33] demonstrates the feasibility of such an approach with a middlebox architecture based on Click. However, these pioneering efforts are focused on per-packet processing, making them less applicable to the stream or flow-based processing common to the class of middleboxes we target in this work. For example, flow-based processing requires operating on a byte stream rather than individual packets and may require communication among multiple network elements to perform dynamic forwarding and rewriting. Further, they provide no specific support for managing and rewriting a large number of concurrent flows, instead focusing on high-speed pipeline processing of individual packets.

Hence, this paper presents `xOMB` (pronounced *zombie*), an eXtensible Open MiddleBox software architecture for build-

ANCS'12, October 29–30, 2012, Austin, Texas, USA.

ing flexible, programmable, and incrementally scalable middleboxes based on commodity servers and operating systems. xOMB employs a general *programmable pipeline* for network processing, composed of xOMB-provided and user-defined C++ modules responsible for arbitrary parsing, transforming, and forwarding messages and streams. Modules can store state and dynamically choose different processing paths within a pipeline based on message content. A control plane automatically configures middleboxes and monitors both middleboxes and destination servers for fault tolerance and availability. xOMB provides a single, unified platform for implementing the various functions of static load balancing/filtering, dynamic request routing, and protocol acceleration.

Several additional xOMB features add power and ease of programming to the simple pipeline model. *Asynchronous processing modules* and *independent, per-connection processing* efficiently support network communication (e.g., to retrieve dynamic state) as part of message processing. *Arbitrary per-message metadata* allows modules to store and pass state associated with each message to other modules in the pipeline. Finally, xOMB automatically manages client and server connections, socket I/O, message data buffers, and message buffering, pairing, or reordering.

We implement and evaluate three sample pipelines: an HTTP load balancing switch (static load balancing), a front end to a distributed storage service based on Eucalyptus [29] implementing the S3 interface [1] (dynamic request routing), and an NFS [32] protocol accelerator. Forwarding through xOMB presents little overhead relative to direct access to back-end servers. More importantly, xOMB scales its network and processing performance with additional commodity servers and provides transparent support for dynamically growing the pool of available middleboxes. We also compare the performance of our xOMB load balancing switch against a commercially available hardware load balancing switch and the leading open source reverse proxy. xOMB matches or outperforms the commercial switch and open source proxy in most cases, while providing a more flexible and powerful programming model.

2. OVERVIEW

Middleboxes can be defined as network elements that process, forward, and potentially modify traffic on the path between a source and a destination. With this generic definition, routers and switches can also be classified as middleboxes. In this paper, we focus on an *active middle box* ("middlebox" for short), a network device that performs programmable traffic processing based on entire packet contents rather than just on headers. We identify three key features differentiating middleboxes from traditional routers and switches: middleboxes i) understand the application semantics of network data, ii) may modify or even completely replace the contents of that data, and iii) despite being "in the middle," middleboxes may terminate connections and initiate new connections.

Although the xOMB design is general to a variety of middlebox applications, we primarily focus on their use in data centers in this paper as this deployment scenario stresses all aspects of our design and presents a particularly challenging use case, with strict requirements for performance and reliability. We begin by examining *load balancing switches* (LBSs), specialized middleboxes widely used to distribute requests among dynamic server pools in data centers.

2.1 Load Balancing Switches

Data center services rely on specialized hardware LBSs that serve as the access point for services at a particular data center, abstracting back-end service topology and server membership and enabling incremental scalability and fault tolerance of server pools. When a client initiates a request, rather than returning the IP address of an application server, the service instead returns the IP address of a LBS. Client packets must then pass through the LBS on the way to their ultimate destination.

LBSs may operate at the packet-header or packet-payload granularity. In the first case, they forward packets to back-end servers by transparently rewriting IP destination addresses in packet headers. Care must be taken to ensure that all packets belonging to the same flow are mapped to the same server. Many commodity switches provide basic hardware support for appropriate flow hashing to deliver such functionality. A straightforward design might employ OpenFlow [11] coupled with commercially available switches to map flows to back-end servers by monitoring group membership and load information. Hence, we focus on the more challenging (and commercially relevant) instance where LBSs must operate at the granularity of packet-payloads, performing arbitrary processing on application-level messages before forwarding them to appropriate back-end servers.

Performance optimizations, used either independently or in addition to protocol acceleration, are another example of LBS functionality. For example, an LBS will maintain persistent TCP connections to back-end webservers, re-writing HTTP 1.0 requests as 1.1 if necessary to use one of the existing connections. Switches may also implement connection collapsing, in which many incoming client TCP connections are multiplexed onto a small number of persistent TCP connections at the application servers. These functions provide several benefits. First, they eliminate the overhead of establishing new TCP connections and the delay for it to ramp up to the available bandwidth, reducing overall latency. Second, servers incur some degree of per-connection processing overhead, so if every client connection requires a connection from the middlebox to the back-end, then both the middleboxes and backends incur this cost. By "absorbing" the client connections through connection collapsing, the middlebox reduces back-end load and also reduces its own load by having fewer back-end connections to manage. Connection collapsing and request rewriting for persistent connections is one of the most popular uses for commercial LBSs [9, 30].

While some commercial LBSs can process byte streams, this capability is typically limited to a small set of protocols (e.g., HTTP) with restricted programming models. Further, as running at line speeds often requires specialized hardware employing custom ASICs, it is typically impossible for LBSs to perform message forwarding or rewriting based on state maintained across connections or to initiate RPCs to look up non-local state for dynamic request routing, requiring large-scale service providers to employ proprietary software solutions. The goal of our work is to address these challenges with a scalable, easy-to-program architecture built entirely from commodity server components.

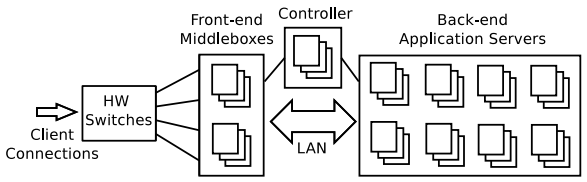


Figure 1: System Architecture

2.2 Architecture

Figure 1 shows the xOMB architecture. The major components include commodity hardware switches, our front-end software middleboxes, back-end application servers, and a controller for coordination. The software middleboxes communicate with each other, with the controller, and with *agent processes*—used for collecting statistics such as machine load on each back-end server—using an RPC framework.

Hardware switches. While our software middleboxes can perform application-level packet inspection to aid forwarding decisions, we can optionally leverage existing commodity hardware switches to act as the single point of contact for client requests. These commodity switches employ line-rate hashing to map flows to an array of our programmable software middleboxes [35].

Software middleboxes. Commodity servers with software middleboxes function as the front-end switches for xOMB. They parse, process, and forward streams of requests and responses between clients and the back-end application servers. xOMB flexibly supports arbitrary protocol and application logic through user-defined processing modules. Deployments can scale processing capacity by "stacking" xOMB servers either vertically (every server runs the same modules) or horizontally (servers run different modules and form a processing chain). Additionally, the middleboxes provide distributed failure detection and load monitoring.

Application servers. Application servers (e.g., webservers) form the back end of our system and process and respond to client requests according to the protocol(s) they are serving. These servers may be grouped into logical *pools* with related resources or functionality.

Controller. The controller provides a central rendezvous point for managing front-end and back-end configuration and membership. In addition to coordinating and storing server membership, the controller also stores a limited amount of service hard state. The controller may be implemented as a replicated state machine for fault tolerance.

3. DESIGN

The middleboxes form the core of the xOMB framework, with the primary goals of flexibility, programmability, and performance. They provide complete control over data processing, allowing them to work with any protocol, including proprietary, institution-specific ones. Our high-level approach to arbitrary byte stream processing is to terminate client TCP connections at the middlebox, execute the appropriate *modular processing pipeline* (pipeline) containing user-defined processing logic on an incoming byte stream, and then transmit the resulting byte stream over a new TCP connection to the appropriate back-end server. We leave extensions to message-oriented protocols such as UDP to future work. A separate control plane configures the middleboxes, manages membership, performs monitoring, and schedules and executes timers.

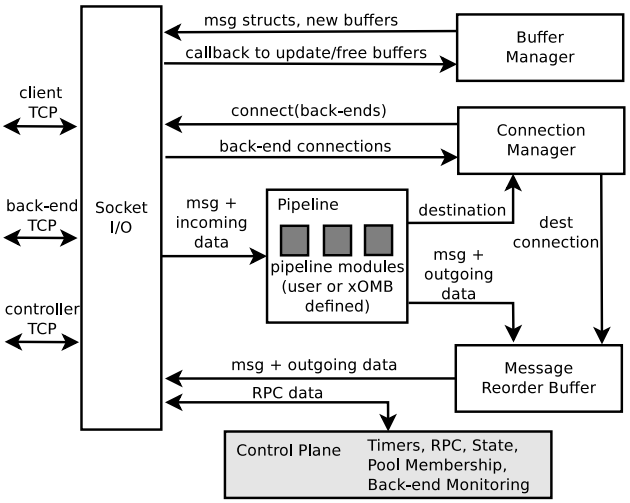


Figure 2: Anatomy of a xOMB Server

xOMB automatically handles low-level functionality necessary for high-performance processing, allowing the programmer to concentrate on the application logic. xOMB abstracts connection management, socket I/O, and data buffering. Additionally, xOMB targets *request-oriented* protocols, which comprise most of today's Internet services. In these protocols, request *messages* have exactly one logical response message, and responses should be returned in the order requested. xOMB tracks requests and buffers and reorders responses to meet this requirement. Although our design allows arbitrary data processing, our aim is to make handling common processing patterns and standard protocols as simple as possible (see §4.3.2).

Figure 2 shows an overview of the anatomy of a xOMB server. The user simply defines the processing modules (the dark gray boxes) that plug into the xOMB pipeline framework. In this section, we discuss the design of the xOMB core architecture, and we delve into the key implementation details in §4.

3.1 Pipelines and Modules

xOMB divides data stream processing into three logical stages: protocol parsing, filtering/transformation, and forwarding. Each stage is composed of an arbitrary collection of *modules* that dynamically determine the processing path of a message. As each module can process different messages independently and concurrently, we refer to the complete DAG of modules and any additional control-flow logic as the pipeline. Because requests and responses require different handling, middleboxes use separate pipelines for each direction.

3.1.1 API

Each pipeline module represents a single processing task, and modules may be composed of other modules. Modules may be either synchronous or asynchronous; asynchronous modules allow processing tasks to retrieve state over the network without blocking. A simple API consists of methods for initializing state, receiving failure/membership notifica-

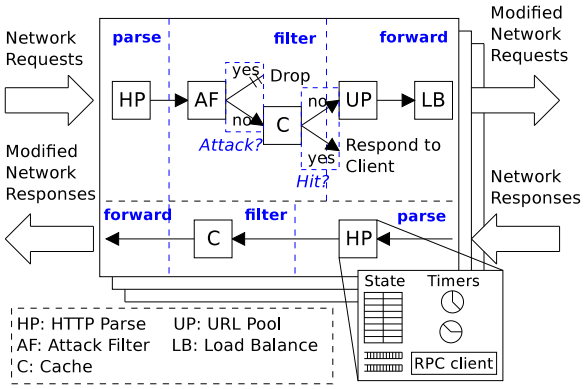


Figure 3: Example Pipeline

tions, and a `process()` method to execute the module's task. Middlebox programmers must only define a set of modules and the pipeline to link them together.

When a middlebox reads a chunk of data from a socket into a message buffer, it passes the buffer to the pipeline, which successively invokes module `process()` methods until either a module halts processing or every module has been executed. Modules may store soft state in memory, schedule timers, and, for asynchronous modules, make RPCs to retrieve or store shared persistent state. Modules pass processing results to other modules, or between requests and responses, through arbitrary *metadata* pointers associated with each message. For example, a parser module may set a data structure representing the parsed message as a metadata value.

`xOMB` assigns every connection its own pipeline object to increase throughput and simplify programming. The `xOMB` concurrency model uses *strands* [5] based on each connection, meaning that at most one thread will execute pipeline or I/O instructions for a given connection at a time. The advantage of this model over explicit locking is that it elegantly allows multiple cores to execute processing or I/O for different connections in parallel. Moreover, if one connection's pipeline makes an RPC (which is asynchronous), control will immediately transfer to another connection's pipeline until receiving the reply, further increasing throughput and utilization. Because many modules need to track per-message or per-connection state (e.g., the number of bytes parsed), per-connection pipelines have the additional advantage of simplifying programming by automatically giving each module private state. `xOMB` also provides support for modules storing state across connections (§3.1.3).

3.1.2 Example Pipeline: HTTP

Figure 3 shows an example request and response pipeline for processing HTTP traffic, broken into parse, filter, and forward stages.

Protocol Parsing. The protocol parsing stage transforms the raw byte stream into discrete application-defined messages and sets application-specific metadata in the message structure. On a parse error, the module sets a message flag to close the connection. The parser indicates when it has parsed a complete message along with the total bytes parsed. The middlebox reads these fields to create a new buffer pointing to any remaining bytes for the next message.

Parsers can be chained together to simplify the logic for different subsets of a complicated protocol. For example, parsing XML-RPC might use an HTTP parser followed by an XML parser. Our sample HTTP parser sets a metadata structure representing an HTTP request, including the protocol version, request method, path, headers, etc.

Filtering and Transformation. Filter modules perform arbitrary transformations on messages. Our example request pipeline has three filters illustrating common uses. First, an *AttackDetector* checks the request against a set of attack signatures using string matching expressions loaded from Snort [15] rules. If the message matches a rule, the filter sets the "drop connection" flag.

Next, the pipeline uses a *Cache* module for protocol acceleration. In the request pipeline, the module checks whether a cache entry exists for the path set in the HTTP metadata. If so, the module sets the message buffer pointers to the cached response and sets the message destination to the client. In the response pipeline, the cache module checks the response headers in the HTTP metadata and stores the response if permitted.

The final filtering step (omitted in Figure 3) performs HTTP version rewriting to allow middleboxes to maintain persistent connections to the back-end webservers, even when clients do not support them. If the HTTP metadata version is 1.0, the request pipeline module rewrites the headers to version 1.1 and adds the appropriate "Host" field. The response pipeline also uses a version filter that rewrites the response back to HTTP/1.0 for requests that were transformed. Similarly, because cached responses will always be version 1.1, the request pipeline sends these through a response version filter as well.

Forwarding. The forwarding stage sets the message destination based on metadata set by the previous stages. Forwarding can be as simple as selecting a back-end server from a pool to as complicated as computing the destination from a dynamically populated forwarding table. Response pipelines do not have forwarding modules; the middlebox automatically sends responses to the requesting client.

Our example request pipeline uses two forwarding modules. The *URLPool* module partitions back-end server pools based on the paths from the URL that they may serve. This module periodically reads configuration state from the controller that maps URL path prefixes to server pools. For example, paths beginning with "/image" go to one set of servers and paths beginning with "/video" go to another set. By using the HTTP metadata path, the module sets a metadata field with the pool for the longest prefix match. The *LoadBalance* module selects a destination server from the designated pool using a specified load balancing policy. We have implemented simple load balancers that use round-robin or random selection.

Figure 4 shows a sample module implementing random forwarding. The module sets the message destination to a random server in the specified pool. If the specified pool does not contain any servers, the module returns an error status that signals the pipeline logic to initiate alternate processing or close the connection.

3.1.3 Module State and Configuration

Integrating programmable middleboxes into complex distributed protocols requires that the middleboxes can access potentially large amounts of service state necessary for making forwarding decisions. We distinguish two kinds of

```
class RandomForwardModule : public Module {
 public:
  MessageStatus process(MessagePtr m) {
    // get membership from control plane
    MembershipSet members =
      Membership::getMembers(m->getPool());

    if (members.empty())
      // tell xomb to close connection
      return MessageStatus::Error;

    // set destination on message structure
    m->setDest(random(members)->addr());

    // tell xomb to start forwarding data
    return MessageStatus::Complete;

    // (Modules that need more data can
    //  return MessageStatus::ReadMore)
  }
};
```

Figure 4: A forwarding module implementing random forwarding to a pool of servers specified in the message metadata.

module state: configuration state and dynamic state. Configuration state specifies parameters such as rates, cache sizes, numbers of connections, etc., and any global state that changes infrequently. Examples of configuration state include the set of fingerprints used by the *AttackDetector* module or the path prefix to server pool mapping used by the *URLPool* module. xOMB uses the controller to manage all global configuration state, stored as a map from state name to opaque binary value. This map can be queried through an RPC interface by a middlebox when it starts, allowing modules to retrieve necessary configuration parameters. Optional metadata includes a version number and time duration, which tells modules how long they should use the current value before checking for a new version.

Dynamic state consists of any unique state that a module references or retrieves for each message. Consider an object store directory that maps billions of object ids to back-end servers. Modules typically cannot prefetch and store a complete copy of such forwarding state because the total state is too large, keeping a consistent copy would be too expensive, or both. xOMB modules may dynamically construct forwarding tables during message processing and may control the rate at which dynamic state is updated. Modules can retrieve required state on demand by making asynchronous RPCs to application services such as a back-end metadata server. The ability to build dynamic forwarding tables is a significant advantage afforded by the general programmability of xOMB middleboxes relative to less flexible callback-based models (§3.3).

Modules store global state—including both configuration and dynamic state—in memory that persists across connections. To allow modules to manage memory effectively, xOMB passes membership and failure notifications to every module so that they may discard unneeded state. Additionally, modules may set timers to perform periodic state maintenance to optimize storage or purge stale state.

3.2 Control Plane

Although pipelines and modules form the core of xOMB, a number of other components complete the system functionality and convenience.

3.2.1 Membership

xOMB middleboxes and back-end servers require the current server membership for various pools. As middleboxes and servers join and leave, updated membership must be disseminated efficiently. In addition to basic pool membership, xOMB must assign both back-end and middlebox servers to be monitored by one or more middleboxes. These assignments should remain balanced as middleboxes and servers are added and removed. Finally, we also require that there be no manual configuration for adding or removing servers—all membership pools and monitoring assignments must be updated automatically.

To achieve these requirements, the xOMB controller manages pool membership and monitors assignments. Although using a centralized controller may not scale to the largest systems, it is a simple solution and should be sufficient for thousands to tens-of-thousands of servers [22]. The controller may be implemented as a replicated state machine for high availability, or replaced with a coordination system such as [26].

When a new server comes online, its agent process makes a `join` RPC to the controller, registering itself as either a middlebox or an application server, and specifies the sets of pools to which it should be added and any pools for which it requires membership updates. The controller records this request and informs all servers who have registered interest in membership updates for that pool. When failure detectors inform the controller of a server failure, it similarly notifies all registrants.

3.2.2 Monitoring

Services must respond to failures and changing server loads. Front-ends such as xOMB implement this functionality as they direct requests to back-end servers: middleboxes can avoid sending requests to failed machines and shift traffic to less loaded servers. To effectively minimize service time for requests, the middleboxes need the current liveness and load status of all servers, generally including other middleboxes.

Each middlebox collects load information from a set of servers assigned by the controller at a configurable interval. The xOMB agent on each server reports machine-level information, such as load, CPU, network, and memory utilization, but application server agents may report more detailed application information, such as the number of active connections or operations per second.

3.2.3 Failure Detector

xOMB employs an active failure detector to quickly detect unresponsive servers. The controller assigns every middlebox a set of servers to monitor. Middleboxes ping each of their monitored servers at a configurable period. For more reliable failure detection, xOMB supports assigning multiple middlebox failure detectors for every server.

3.3 Design Discussion

While a general modular/pipeline approach is common in system design [28, 37], it represents a novel architecture for programmable middleboxes, which typically use a layered approach with protocol-specific callbacks [8, 17]. For example, a conventional middlebox may provide callbacks for processing a new TCP flow, part of an HTTP request (such as the URL), or a complete HTTP request.

The primary advantage of callbacks is that, as long as the

product supports your protocol, they make it straightforward to implement simple protocol-specific handling for a particular set of events. Because vendors tailor the set of callbacks to only specific supported protocols, they can provide a high level of integration for switch programmers, abstracting details such as protocol parsing and loading shared libraries—the programmer only needs to provide bodies of the desired event handlers. Additionally, the callbacks take as arguments the relevant fields for the event, eliminating the need for metadata objects attached to messages.

In contrast, xOMB modular pipelines provide four important advantages over callbacks. First, asynchronous modules allow message processing to perform RPCs to retrieve or store state over the network. The programmable middleboxes that we surveyed have the limitation that callbacks must run to completion and must not block, thus precluding this critical functionality for implementing dynamic request routing. Second, xOMB pipelines are more flexible because they are not limited to a fixed set of protocols or callbacks. Third, xOMB pipelines elegantly allow modules to pass arbitrary per-message state to other modules through the message metadata, enabling cross-module processing logic. While callbacks could provide similar functionality, this must be supported by the framework; current systems do not allow this and instead require setting global variables, a much more complicated and error-prone approach, and may limit processing parallelism if accessing these global variables requires locking. Finally, xOMB pipelines are potentially more efficient, because parsing modules only need to parse the minimal amount of bytes necessary to complete the desired processing. Furthermore, the pipeline can be programmed to immediately begin processing message fragments rather than waiting for the complete message, potentially reducing latency and overhead for large messages that can be processed with streaming logic.

xOMB pipelines can be structured to provide all of the convenience of callbacks while maintaining the above advantages. For example, we envision including parsers for popular protocols as part of the xOMB distribution. Further, pipelines can emulate a set of callbacks by using a series of modules with methods for each callback. These modules can wait for a desired event to occur and invoke the respective handler method with arguments from the message metadata.

4. IMPLEMENTATION

We now discuss xOMB's implementation in more detail.

4.1 Pipeline Libraries

To simplify pipeline programming, xOMB separates module and pipeline implementation from the xOMB C++ implementation. The user provides pipelines written in C or C++ as shared-object libraries linked with the required modules, also written in C/C++. xOMB allows users to define simple pipelines by just adding modules in the desired order to a pipeline module list. A xOMB server can load multiple pipelines, each associated with a separate port.

4.2 Control Module

The middlebox implements its portion of the control plane in a global control module. Upon startup, the control module first initializes the middlebox by creating threads for asynchronous I/O dispatch [5], and starts an RPC server to receive calls from the controller. Next, it dynamically loads specified pipeline shared-object libraries and begins listening for client connections on the data plane. Finally, it joins the controller and retrieves any global configuration parameters.

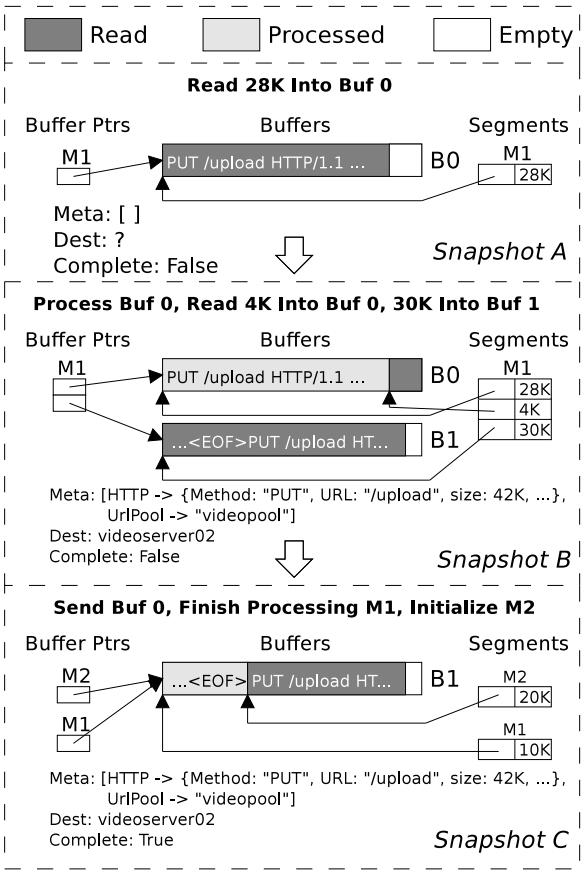


Figure 5: Message and Buffer State Snapshots

As described in §3.2, the middlebox receives server pool membership and monitoring assignments from the controller. The control module stores the pool assignments in a shared membership module and schedules timers for its monitoring tasks. A shared load monitor module queries servers for load information and stores the results. When the control module receives a failure notification from the controller, it notifies all pipelines so that the modules can update their state and respond appropriately. However, services such as health and load monitoring are optional. While we expect such functionality in many production environments, simpler deployments may not have this requirement.

4.3 Data Plane

The data plane listens for client connections on one or more ports, each of which has its own request and response pipeline. Although the xOMB architecture is general to any protocol, we focus on TCP-based protocols in this paper. Upon accepting a client TCP connection, the middlebox creates a client connection object that holds the client socket, the request pipeline, and a data structure to buffer and reorder responses. xOMB then creates a new message structure and buffer and reads data from the socket. We will illustrate pipeline processing with the state snapshots in Figure 5.

4.3.1 Messages and Buffers

The middlebox creates a message data structure for each

request and response to buffer message data during processing. Buffer management—how buffers are allocated, accessed, and copied—is a critical design detail for building a high performance middlebox. xOMB uses memory efficiently by avoiding user-level copying and freeing buffered data once it has been sent, even if the message is not complete. We use reference-counted buffer pointers to simplify memory management for data that persists across multiple messages.

Each message structure maintains a list of pointers to fixed-size buffers. Because buffers may not be full or may contain data for multiple messages, the message also has a list of *segments*—contiguous substrings in a buffer—each holding a pointer to the start of the segment (an offset into the buffer) and the segment length. The structure also holds fields for the total buffered data size and the number of bytes parsed, queued, and sent. The segment list and byte counts represent shifting windows of data to be processed and sent.

To maximize memory efficiency, the middlebox always fills every buffer by using scatter/gather I/O (reading into multiple buffers with one system call) and passing allocated but unused buffers to the next message. When a read completes, the middlebox adds segments to the message pointing to the newly read bytes. Snapshot A of Figure 5 shows message $M1$ after reading 28K into buffer $B0$.

Message processing proceeds one segment at a time. The pipeline returns one of three results: either the message is complete, an error occurred, or the pipeline needs more data. If the message is complete, the middlebox constructs a new message with any remaining unparsed buffer pointers and segments. On an error, the middlebox discards the message and closes the connection. If the pipeline returns incomplete, the middlebox performs another read.

Snapshot B of Figure 5 shows $M1$ after the middlebox has processed $B0$. The HTTP parser has set metadata representing the request, including the total request size of $42K$, *URLPool* has set the destination pool, and *LoadBalance* has set the destination server. Because $M1$ was not complete, the pipeline returned that it needed more, and the middlebox read another 30K into buffer $B1$.

Once the pipeline has determined the destination for the message, the middlebox sends any processed segments to the destination. As the middlebox reads and processes new buffers and segments, it simultaneously sends previous ones. By discarding buffers after sending them, the middlebox uses only a small amount of memory for each message, regardless of the total message size. The middlebox limits the amount of data buffered for any message by not reading on a connection while the total buffered size exceeds a threshold; the middlebox eventually closes connections for reads that take too long. Because message data typically will not fall on buffer boundaries, when the pipeline has processed a complete message, xOMB copies pointers for any buffered but unprocessed bytes from the completed message into a new message and invokes the pipeline with the new message before attempting another read.

Figure 5 C shows the state after processing $B1$ through the end of $M1$. While the pipeline processed $B1$, the middlebox concurrently sent $B0$ and then freed it. Because $M1$ is complete but $B1$ is not empty, the middlebox creates a new message $M2$ with initial buffers and segments as shown and empty metadata (not shown). The middlebox will process $M2$ before attempting to read more from the socket because

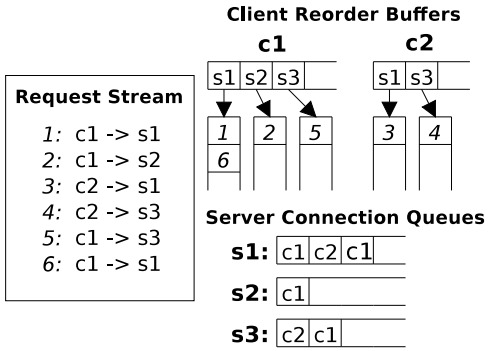


Figure 6: Reorder Buffer Example

$M2$ may be complete and, if so, processed before blocking on further data.

4.3.2 Connection Pool and Message Reordering

Most Internet protocols can be classified as either *request-oriented*, for which each logical request has one logical response, or *streaming*, where the byte stream cannot be separated into logical message boundaries. Although many request-oriented protocols, such as HTTP, have a one-to-one mapping between request and response messages, this is not always the case: for instance, NFS (see §6.4) may send multiple data fragment messages in response to a read request. xOMB relies on the parsing module to denote the logical message boundaries to when they span multiple protocol messages.

For streaming protocols, xOMB uses a unique server connection for each client connection and proxies the transformed pipeline data. For request-oriented protocols, xOMB maintains a pool of connections to back-end servers, with a configurable number of reusable connections per server. The connection collapsing performed by xOMB middleboxes can significantly increase back-end server efficiency by multiplexing a large number of client connections onto a small number of server connections (§6.2).

Because of connection collapsing, xOMB may interleave requests from different clients on server connections. Additionally, because xOMB may distribute pipelined client requests among different servers, the responses may arrive out-of-order. xOMB automatically demultiplexes, buffers, and reorders the responses to the clients, simplifying pipeline implementation. xOMB achieves this with a *reorder buffer*, data structure, illustrated with an example in Figure 6, that matches server responses with client requests based on their connections. As responses flow in, xOMB pairs the front of the server connection queue with the client reorder buffer, sending the message if they match and shifting the queues, or otherwise buffering the message (the vertical queues in Figure 6). xOMB limits buffered message data by only allowing a fixed number of pipelined requests per connection.

The example in Figure 6 shows six requests from two clients ($c1, c2$) distributed over three server connections ($s1 - s3$), and shows both kinds of reordering: 1) requests 1 and 3 from both $c1$ and $c2$ have been interleaved on connection $s1$, and 2) requests from both clients have been spread across different servers. For example, xOMB will buffer responses to $c1$ for messages 2 and 5 until the response for message 1 has been sent.

5. DISTRIBUTED OBJECT STORE

For a detailed example of dynamic request routing, we describe an object store service, xOS, based on Amazon's S3 [1]. To be interface compatible with S3, we used the unmodified Eucalyptus [29] *Walrus* storage components for our back-end application servers. However, as of the latest version (2.0), Eucalyptus does not support more than a single Walrus storage server. By using xOMB middleboxes together with a distributed metadata service, we transparently overcome this limitation while maintaining a unified, scalable storage namespace. We quantify xOS scalability in §6.3.

xOS hosts objects stored in *buckets* named by unique keys. Eucalyptus uses a single *cloud controller* to manage the authentication and metadata for all storage requests. Multiple storage servers will work independently when configured with the same Eucalyptus cloud controller, so we built a xOMB pipeline to consistently forward requests for a given user/bucket to the same storage server. xOS supports any middlebox processing requests for any bucket.

We implement $bucket \rightarrow server$ placement with a distributed metadata service that maps $\langle userid, bucket \rangle$ pairs to storage servers. Metadata servers subscribe to the storage server pool on the xOMB controller. Each metadata server is configured with a portion of a 160-bit key space, which it registers with the xOMB controller. The xOS forwarding module retrieves the key-space to metadata server mapping as part of its configuration state. Additionally, the forwarding module will update its metadata server configuration whenever it receives notification that the set of metadata servers has changed (via addition, removal, or failure).

The xOS request pipeline consists of the standard HTTP parse module followed by our xOS forwarding module. To process a request, the xOS forwarding module takes the SHA1 hash of the $\langle userid, bucket \rangle$ string parsed from the HTTP headers and URL. Using this hash, the module computes the metadata server responsible for that portion of the key-space and makes an RPC to retrieve the storage server. When a metadata server receives a lookup request, it either returns an existing assignment if found or otherwise chooses a new storage server and stores the assignment. The forwarding module caches these assignments to avoid subsequent lookups for the same bucket.

In our design, middleboxes maintain only soft state for their fowarding tables, so no middlebox recovery is necessary. Furthermore, middleboxes can update their bucket-to-server mappings lazily; if they attempt to forward a request to a failed server and receive a socket error, then they can contact the metadata servers to retrieve an updated mapping. When the storage servers are replicated (we did not implement this, as it was not the focus of this example), then storage server failure would be transparent to the client.

Basing xOS on distributed Walrus servers presents an additional challenge for the front-end middleboxes. The S3 interface contains a *ListBuckets* request to list all of a user's buckets. However, no single server contains this information. To support the complete interface, the xOS forwarding module recognizes the *ListBuckets* message and makes an RPC to each metadata server requesting all the user's buckets. Once the forwarding module has received all responses, it generates the HTTP and XML response by combining the separate bucket lists. Although not difficult to implement, we note that such functionality would be impossible in designs limited to event-handler callbacks.

6. EVALUATION

In this section, we evaluate the main xOMB design goals of scalability, performance, and programmability. For these experiments, we use servers with Intel 2.13 GHz Xeon quad-core processors and 4 GB DRAM running Linux 2.6.26. All machines have 1 Gbps NICs connected to the same 1Gbps Ethernet switch.

6.1 Programmability

First, we give a sense of xOMB's programmability. Table 1 shows the number of lines of C/C++ code for modules we implemented, as well as the xOMB core middlebox service framework (excluding the controller) for reference. The majority of the modules are short, although the HTTP parsing module includes 1700 lines of an HTTP parsing library [14], and the NFS parse and Cache modules both use code generated from the XDR protocol file. In addition, all of the pipelines use the default pipeline construction process, meaning they are only a handful of lines each.

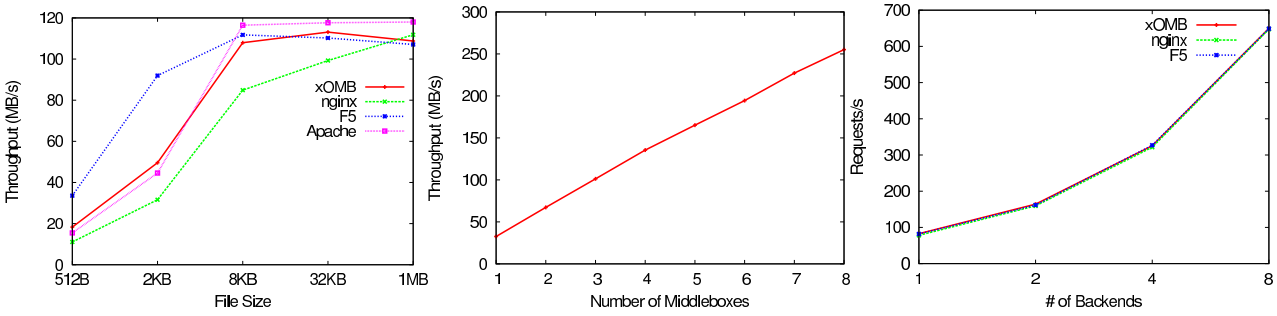
Module	Lines of Code
HTTP Parse	111 (+ 1699)
Round-Robin Forward	73
Random Forward	37
URL Pool Forward	31
HTTP Attack Filter	99
HTTP Version Filter	72
xOS Forward	277
NFS Parse	235 (+ 2906)
NFS Cache	413 (+ 602)
xOMB Core	4478

Table 1: Module Code Lengths

6.2 HTTP Performance

We first evaluate xOMB performance with HTTP pipelines. We used Apache 2.2.9 [2] running the MPM worker module, serving files of various sizes. All throughput measurements include the bytes transferred for HTTP headers. In addition, our HTTP client is pipelined, but limits itself to 10 outstanding requests. In these experiments, xOMB uses a basic HTTP pipeline configured to parse requests and forward them, using a "client sticky" round-robin forwarding module, across the available webservers. Our forwarding module ensures that all of a client's requests go to the same webserver, although each client is assigned to a webserver round-robin. In addition, we perform connection collapsing down to at most five connections from each middlebox to each webserver.

We compare our performance against Apache directly, the popular open-source reverse proxy nginx [10], and a programmable hardware switch from F5 Networks [8]. The F5 BIG-IP Local Traffic Manager 1600 (LTM) we used has an Intel 1.8 GHz E2160 dual-core processor and 4 GB DRAM and is running OS Version 9.4.8 and has a single 1 Gb/s NIC. We refer to the LTM simply as "F5" in our experiments. It is difficult to compare the processor employed in the F5 relative to our servers. While our machines are three years old and based on older microarchitecture, they do have four cores and a higher clock speed. One challenge with specialized hardware such as the F5 switch is integrating the latest processors and motherboards into a specialized



(a) xOMB Throughput (b) xOMB Front-end Scalabililty (using 1KB Files) (c) xOMB Back-end Scalabililty

Figure 7: Throughput and Scalability Comparisons

hardware and software environment, a downside endemic to non-commodity solutions.

Figure 7a shows a comparison of client throughput to Apache, through a single xOMB middlebox, a single nginx reverse proxy, and through the F5 switch. For each file size, we used 100 clients, which was enough to maximize throughput. Compared to Apache, xOMB does quite well, only losing out slightly at large file sizes due to the fact that the xOMB middlebox must allocate some of its 1 Gb/s of NIC bandwidth to forwarding the clients' requests to the back-end webserver. Nginx performs noticeably slower in almost all cases, only able to surpass xOMB slightly with 1 MB files. Finally, xOMB performs similarly to the F5 switch for larger files, but the F5 handily outperforms with smaller files.

Intrigued by the F5's impressive performance, we investigated further to determine how the F5 could outperform stand-alone Apache with small files since we had disabled caching. We wrote a very simple TCP proxy that accepts a client connection, parses request streams by looking for a lone CRLF, and simply copies the requests to a fixed destination connection. When running a single HTTP client through this proxy connected to Apache, we saw throughput increase by a factor of two-to-ten compared to the client connecting to Apache directly, depending on the file size. We saw this speedup regardless of whether the client pipelined requests or not. However, we saw zero speedup when we repeated this through a TCP proxy that did not parse requests. While we leave a more in-depth study to future research, our initial conclusion is that Apache appears to be sensitive to the way it receives streams of client requests, and for small files, parsing these requests is the limiting factor for throughput.

Although the presence of a single xOMB middlebox shows reduced performance compared to the F5 switch for small files, one of the key components of our design is the ability to scale well. Figure 7b shows client throughput when requesting 1 KB files with eight back-end webservers behind differing numbers of middleboxes. A single middlebox is not able to handle the extra capacity additional back-ends provide, but we see near-perfect scalability as we add middleboxes.

In our next experiment, clients request a simple CGI application that computes the SHA-1 hash of a 1 MB file on disk. Instead of being limited by middlebox processing capacity, we are now limited by back-end capacity. Figure 7c shows client throughput, now measured in requests per second, of 100 clients making requests to the CGI program with varying numbers of webservers behind a single xOMB middlebox. We see that xOMB, nginx, and the F5 switch are all able to achieve near-perfect scalability as back-end resources are added.

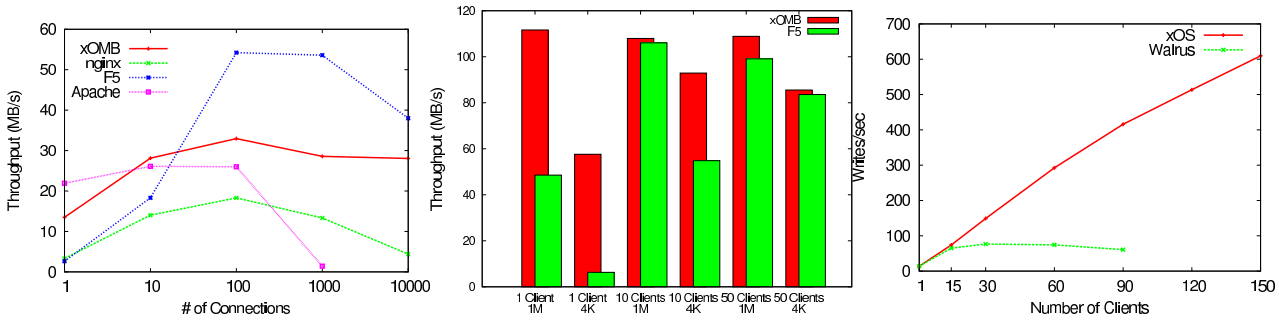
One of the most difficult aspects of middlebox design is performance under extremely high numbers of concurrent connections. We ran between 1 and 10,000 clients requesting 1 KB files against xOMB, nginx, the F5, and Apache, with the results shown in Figure 8a. We first note that Apache by itself does not perform well with 1,000 clients, and we could not get it to serve 10,000 concurrent clients at all. Both xOMB and nginx show similar curves, although we outperform nginx for all connection sizes. However, the F5 shows very unusual behavior. We see it is able to perform extremely well with between 100 and 1000 clients. Additionally, although it outperforms xOMB and nginx, we saw around 6,000 of the 10,000 clients' connections closed prematurely by the F5. We have been unable to determine the cause of this performance anomaly and leave further study of the F5 for future work.

Our final benchmark for HTTP traffic is a pipeline for filtering potential malicious requests (§3.1.2). We compare against the F5, which we programmed to perform the same checks. Although F5 offers a firewall module that can filter attack traffic, we chose to implement an attack filter for the F5 in their provided scripting language to both gain experience programming the F5 and to compare the exact same set of rules for both systems.

Our attack filter module loads 285 Snort [15] rules from the controller, each containing a regular expression to search for in the URL of a web request. We check each URL request against all 285 rules. Although we did not introduce any malicious requests in this experiment, we measure the performance hit of checking every request against all rules. Figure 8b compares xOMB's performance against the F5 running this attack filter against a single webserver. Not only does xOMB outperform the F5 across all tested numbers of clients with 1 MB and 4 KB files, but it sustains the throughput achieved when not running the attack filter with 1 MB files and comes within 10% for 4 KB files.

6.3 xOS Performance

Next, we evaluate the performance of xOS with dynamic request routing as described in §5. We ran two experiments,



(a) Connection Scaling Comparison (b) xOMB vs F5 Attack Throughput (c) xOS vs. Walrus Write Throughput

Figure 8: Throughput and Scalability Comparisons

one using a single instance of Walrus by itself, and the other using xOS. For both experiments, we used varying numbers of clients to repeatedly write 4 KB blocks to the storage system. We used the Amazon S3 curl client to make the write requests, which do not support pipelining. We had each client create and write to its own unique bucket, a workload that causes the middlebox to do the most dynamic request routing. For the xOS experiment, we used two metadata nodes, eight Walrus nodes, and a single xOMB middlebox.

The results of both experiments are shown in Figure 8c. We see the Walrus write throughput max out at around 74 operations per second. Throughput decreases as the number of clients increases past 30, and tests with more than 90 clients did not finish correctly as we began having connection issues with Walrus. In contrast, xOMB allows xOS clients to write to the eight Walrus backends in parallel, easily scaling past the capacity of a single Walrus machine.

6.4 NFS Acceleration

To evaluate xOMB in a different context than HTTP load balancing, we implemented an NFS protocol accelerator pipeline. Our basic accelerator, designed to speed up wide-area access to an NFS server, caches file attribute structures from lookups and file data from reads or writes. We did not attempt to write a comprehensive accelerator. Rather, our intent is to demonstrate that xOMB can effectively process diverse protocols with varying goals. A limitation of our implementation is the assumption that all client requests pass through the middlebox; it does not attempt to reconcile the cache with the server if some clients connect to the server directly. Some protocol accelerators increase write throughput by responding to the client immediately before forwarding the write to the server; we opted not to implement this.

The file attribute (getattr) and lookup calls form a significant portion of NFS traffic [32]. Typically, clients cache these attributes for a short duration (3 sec). Also, clients need to ensure that file attributes are up to date while serving reads from its local buffer cache. The large number of round trips due to attribute lookups causes significant performance degradation for NFS over a wide-area network. Having NFS clients connect to a xOMB middlebox on the same LAN, which in turn connects to the server, results in better response time for file system operations by caching file attributes and data.

We evaluate performance with the Postmark benchmark [12], modified to bypass the kernel buffer cache. For this experiment, we compare the performance of direct NFS versus xOMB when the clients connect over both a LAN and the an emulated wide-area link. We emulate a 100ms delay using NetEm [25]. To be a worst-case test for NFS, we mount the file system using synchronous writes. Postmark creates 60 files with sizes ranging from 1B to 10KB and performs 300 transactions. Table 2 shows the throughput, average operation latency, and total runtime results for the four runs. xOMB adds a small amount of latency on the LAN, as it must process all the NFS traffic and copy data into the cache. However, the xOMB cache significantly improves the performance of file lookups and read operations with a wide-area delay, completing the workload almost twice as quickly.

	LAN		100ms RTT WAN	
Operations	xOMB	NFS	xOMB	NFS
Read (KB/s)	48	50	3.2	1.6
Write (KB/s)	55	57	3.7	1.9
Create (ms)	1.9	0.3	201	204
Open (ms)	0.6	0.2	0.5	192
Remove (ms)	0.9	0.2	101	101
Read (ms)	0.6	0.2	0.5	100
Write (ms)	7.6	7.8	101	100
Total time (s)	**23**	**22**	**341**	**672**

Table 2: NFS Latency and Throughput Comparison

7. DISCUSSION

We are encouraged by our experience with xOMB and its evaluation. However, there are a number of important issues that require work for programmable middle boxes to be successful in general. We discuss these in turn below.

Performance. Our focus has been on scalability and extensibility. While our single node is reasonable (and in many cases superior to commercial product offerings), it will fundamentally be limited by our user-level implementation. One could imagine alternate, higher-performance xOMB implementations in kernel or even in programmable line cards. While reasonable for certain scenarios, we believe that such architectures will fundamentally limit the expressibility of the available programming model.

Load Balancing. Devising good load balancing algorithms is a difficult research and engineering challenge by itself. For evaluation, we implemented simple algorithms with no

claims of novelty. The goal of this work is not to innovate in developing better algorithms directly, but rather to provide a framework where it becomes easier to innovate in load balancing algorithms.

Debugging. All of our data processing modules have been of moderate complexity thus far, but debugging the behavior of a middle box is a challenge in general. `xOMB` facilitates pipeline debugging by allowing the programmer to enable progressively more verbose logging. Because `xOMB` is written in C++, programmers can leverage standard logging techniques, network monitoring (e.g., `tcpdump` and `wireshark`), and tools such as `gdb` to assist with debugging. More specialized middleboxes typically do not have the same rich set of open source tools or perhaps even the ability to log state over the network or to local disk.

Resource Allocation and Isolation. `xOMB` currently provides no support to isolate individual processing pipelines from one another or to isolate the processing of one flow from another. For example, a rarely-exercised code path could cause the entire pipeline to fail or to slow processing for concurrent flows. Similarly, an administrator may wish to allocate varying amounts of bandwidth or CPU resources to different flows or pipelines. A range of possible techniques are possible for delivering the necessary isolation, from heavyweight solutions employing entire virtual machines on a per-pipeline basis, to individual processes on a per-flow basis, to in-kernel queueing disciplines limiting the bandwidth available to any individual flow. We plan to explore these and other techniques [23] as part of our ongoing work.

8. RELATED WORK

Most closely related to our work in spirit are Click [28], RouteBricks [21], and CoMb [33]. Click provides a modular programming interface for packet processing in extensible routers. The principal difference between Click and `xOMB` is our focus on extensibility and programmability at the granularity of byte streams rather than individual packets. Our pipelined programming model focuses on efficiently parsing and transforming request/response based communication.

Like `xOMB`, RouteBricks also focuses on scaling network processing with commodity servers. Their work, like Click, focuses on routing and operates at the granularity of packets. They focus on the more extreme performance requirements of large-scale routers that may require terabits/sec of aggregate communication bandwidth and their in-kernel implementation and VLB-based load balancing delivers significant scalability. Middleboxes typically do not require quite the same level of bandwidth performance and while our architecture similarly scales with additional servers, our user-level implementation trades per-server performance for programmability and overall extensibility.

CoMb [33] shares the goal of using software middleboxes with commodity hardware, but emphasizes consolidation of middlebox hardware and simplifying network deployments, whereas `xOMB` focuses on programmability and extensibility. CoMb requires modular applications to be written in the Click framework with flow-level processing supported through a session reconstruction module. We see CoMb's goal of consolidation as complementary to ours and expect that `xOMB` middleboxes could be used with a CoMb controller.

Flowstream [24] also provides an architecture for middleboxes. It employs OpenFlow [11]-controlled hardware switches to separate traffic at flow granularity that is then forwarded to individual servers for further processing. By default these servers would perform network processing at packet granularity. As such, one could view `xOMB` as the architecture and programming model for extensible transformation of OpenFlow-forwarded byte streams in Flowstream.

There are many commercial hardware/software products for middlebox processing. F5 networks [8] provides popular load balancing switches. Pai et al. [30] performed some of the early academic work in this space. Bivio [4] focuses on deep packet inspection, while Riverbed [13] delivers protocol accelerators among other products. Each product typically focuses on a niche domain and provides a limited extensibility model. In particular, the F5 switch we evaluate uses the Tcl programming language. However, it is not able to support the full generality of our framework, for example with respect to making remote procedure calls or maintaining protocol-specific metadata and state. For instance, the F5 could not be employed to implement functionality for an entirely different protocol such as the S3 service (§5). In contrast, the goal of our work is to provide a unified framework and programming model for a range of traditional middlebox functionality.

Reverse proxies such as [3, 10, 16–18] aim to provide load balancing over a set of web servers. While this is similar in spirit to part of the functionality that `xOMB` supports, our architecture is much more general and can support arbitrary protocols. Most reverse proxies can only handle a static set of servers, protocols, and have fixed processing options, unlike `xOMB` which can handle dynamic membership and arbitrary processing. Nginx [10] is fairly extensible, although modules written for it must be compiled into the executable and not loaded dynamically like in `xOMB`. In addition, nginx does not support general connection collapsing of client requests, which can severely impact performance with large numbers of clients. Finally, RPCs for making dynamic routing decisions cannot be done with their callback model, although they do support passing arbitrary state between callbacks like `xOMB`.

Allman performed an early performance study of middleboxes [19]. He found that middleboxes are a mixed bag for performance, increasing or reducing performance under different circumstances. The study also found that middleboxes can reduce end-to-end availability, though typically availability remained at an acceptable 99.9%. One goal of `xOMB` is to develop a framework to increase the performance and availability of middleboxes.

One challenge with middlebox deployment is ensuring that flows are forwarded through an appropriate set of middleboxes based on higher-level policy. Dilip et al. [27] introduced an architecture to ensure such forwarding. OpenFlow provides a general mechanism to intercept flows and forward them through an appropriate set of middleboxes on the way to the destination. Ethane/SANE [20] is one instance of such an approach for enterprise network security and authentication.

DOA [36] is a delegation oriented architecture for more explicitly integrating middleboxes into the Internet architecture. One goal of DOA is to address the transparency issues introduced by non-extensible hardware middleboxes on evolving network flows. `xOMB` would ideally make it easier

for middleboxes to adapt to changing traffic characteristics. Similar to DOA, I3 [34] explicitly introduces indirection in data forwarding, this time at the overlay level using a DHT.

9. CONCLUSIONS

xOMB demonstrates a new design and architecture for building scalable, extensible middleboxes. We show that programmability need not come at the expense of performance; for instance, the xOMB implementation of a load balancing switch achieves performance comparable to a commercial load balancing switch. Beyond load balancing, we have shown how extensible middleboxes can be used to build scalable services by constructing dynamic forwarding tables based on application service state and the effectiveness of a xOMB protocol accelerator for NFS.

10. ACKNOWLEDGMENTS

We thank our reviewers for their feedback on the paper. This work is supported in part by the National Science Foundation (#CSR-1116079).

11. REFERENCES

[1] Amazon S3. http://aws.amazon.com/s3.
[2] Apache HTTP Server. http://httpd.apache.org.
[3] Apache mod_proxy. http://httpd.apache.org/docs/current/mod/mod_proxy.html.
[4] Bivio Networks. http://www.bivio.net.
[5] Boost Asio. http://www.boost.org/doc/libs/1_42_0/doc/html/boost_asio.html.
[6] Cisco Systems. http://www.cisco.com.
[7] Citrix Systems. http://www.citrix.com.
[8] F5 Networks. http://www.f5.com.
[9] F5 OneConnect Guide. http://www.f5.com/pdf/deployment-guides/oneconnect-tuning-dg.pdf.
[10] nginx. http://www.nginx.org.
[11] OpenFlow. http://www.openflowswitch.org.
[12] Postmark Benchmark. http://www.netapp.com/tech_library/postmark.html.
[13] Riverbed. http://www.riverbed.com.
[14] Ry's HTTP Parser. http://github.com/ry/http-parser.
[15] Snort. http://www.snort.org.
[16] Squid. http://www.squid-cache.org.
[17] Traffic Server. http://trafficserver.apache.org.
[18] Varnish Cache. http://www.varnish-cache.org.
[19] Allman, M. On the Performance of Middleboxes. In *IMC* (2003).
[20] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: Taking Control of the Enterprise. In *SIGCOMM* (2007).
[21] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *SOSP* (2009).
[22] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *SOSP* (2003).
[23] Ghodsi, A., Sekar, V., Zaharia, M., and Stoica, I. Multi-resource Fair Queueing for Packet Processing. In *Proc. SIGCOMM* (2012).
[24] Greenhalgh, A., Handley, M., Hoerdt, M., Huici, F., Mathy, L., and Papadimitriou, P. Flow Processing and the Rise of Commodity Network Hardware. In *ACM CCR* (2009).
[25] Hemminger, S. Network emulation with NetEm. In *LCA* (2005).
[26] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX ATC* (2010).
[27] Joseph, D., Tavakoli, A., and Stoica, I. A Policy-aware Switching Layer for Data Centers. In *Proceedings of ACM SIGCOMM* (2008).
[28] Morris, R., Kohler, E., Jannotti, J., and Kaashoek, M. F. The Click modular router. In *SOSP* (1999).
[29] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. The Eucalyptus Open-Source Cloud-Computing System. In *CCGrid* (2009).
[30] Pai, V. S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. Locality-Aware Request Distribution in Cluster-based Network Servers. In *ASPLOS* (1998).
[31] Paxson, V. Bro: a system for detecting network intruders in real-time. In *USENIX Security* (1998).
[32] Sandberg, R. Design and implementation of the sun network file system. In *USENIX* (1985), pp. 119–130.
[33] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., , and Shi, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. NSDI* (2012).
[34] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet Indirection Infrastructure. In *ACM SIGCOMM* (2002).
[35] Thaler, D., and Hopps, C. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991, 2000.
[36] Walfish, M., Stribling, J., Krohn, M., Balakrishnan, H., Morris, R., and Shenker, S. Middleboxes No Longer Considered Harmful. In *SOSP* (2004).
[37] Welsh, M., Culler, D., and Brewer, E. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev. 35*, 5 (2001).