Connection Conditioning:
Architecture-Independent Support for Simple, Robust Servers

KyoungSoo Park and Vivek S. Pai
Department of Computer Science
Princeton University

Abstract

For many network server applications, extracting the maximum performance or scalability from the hardware may no longer be much of a concern, given today's pricing – a $300 system can easily handle 100 Mbps of Web server traffic, which would cost nearly $30,000 per month in most areas. Freed from worrying about absolute performance, we re-examine the design space for simplicity and security, and show that a design approach inspired by Unix pipes, Connection Conditioning (CC), can provide architecture-neutral support for these goals.

By moving security and connection management into separate filters outside the server program, CC supports multi-process, multi-threaded, and event-driven servers, with no changes to programming style. These filters are customizable and reusable, making it easy to add security to any Web-based service. We show that CC-aided servers can support a range of security policies, and that offloading connection management allows even simple servers to perform comparably to much more complicated systems.

1 Introduction

Web server performance has greatly improved due to a number of factors, including raw hardware performance, operating systems improvements (zero copy, timing wheels [29], hashed PCBs), and parallel scale-out via load balancers [9, 11] and content distribution networks [2, 14]. Coupled with the slower improvements in network price/performance, extracting the maximum performance from hardware may not be a high priority for most Web sites. Hardware costs can be dwarfed by bandwidth costs – a $300 system can easily handle 100 Mbps of Web traffic, which would cost $30,000 per month for wide-area bandwidth in the USA. For most sites, the performance and scalability of the server software itself may not be major issues – if the site can afford bandwidth, it can likely afford the required hardware.

These factors may partly explain why the Apache Web server's market share has increased to 69% [17] despite a decade of server architecture research [8, 12, 13, 18, 30, 32] that has often produced much faster servers – with all of the other advances, Apache's simple process pool performs well enough for most sites. The benefits of cost, flexibility, and community support compensate for any loss in maximum performance. Some Web sites may want higher performance per machine, but even the event-driven Zeus Web server, often the best performer in benchmarks, garners less than 2% of the market [17]

Given these observations and future hardware trends, we believe designers are better served by improving server simplicity and security. Deployed servers are still simple to attack in many ways, and while some server security research [6, 21] has addressed these problems, it implicitly assumes the use of event-driven programming styles, making its adoption by existing systems much harder. Even when the research can be generalized, it often requires modifying the code of each application to be secured, which can be time-consuming and error-prone.

To address these problems, we revisit the lessons of Unix pipes to decompose server processing in a system called Connection Conditioning (CC). Requests are bundled with their sockets and passed through a series of general-purpose user-level filters that provide connection management and security benefits without invasive changes to the main server. These filters allow common security and connection management policies to be shared across servers, resulting in simpler design for server writers, and more tested and stable code for filter writers. This design is also architecture-neutral – it can be used in multi-process, threaded, and event-driven servers.

We demonstrate Connection Conditioning in two ways: by demonstrating its design and security benefits for new servers, and by providing security benefits to existing servers. We build an extremely simple CC-aware Web server that handles only one request at a time by moving all connection management to filters. This approach allows this simple design to efficiently serve thousands of simultaneous connections, without explicitly worrying about unpredictable/unbounded delays and blocking. This server is ideal for environments that require some robustness, such as sensors, and is so small and simple that it can be understood within a few minutes.

Despite its size, this server handles a broad range of workloads while resisting DoS attacks that affect other servers, both commercial and experimental. Its performance is sufficient for many sites – it generally outperforms Apache as well as some research Web servers. Using the filters developed for this server, we can improve the security of the Apache Web server as well as a research server, Flash, with a tolerable performance impact.

2   Background

All server software architectures ultimately address how to handle multiple connections that can block in several places, sometimes for arbitrarily long periods. Using some form of multiplexing (in the OS, the thread library, or at application level), these schemes try to keep the CPU utilized even when requests block or clients download data at different speeds. Blocking stems from two sources, network and disk, with disk being the more predictable source. Since the client is not under the server's control, any communication with it can cause network blocking. Typical delays include gaps between connecting to the server and sending its request, reading data from the server's response, or sending subsequent requests in a persistent connection. Disk-related blocking occurs when locating files on disk, or when reading file data before sending it to the client. Of the two, network blocking is more problematic, because client may delay indefinitely, while modern disk access typically takes less than 10ms.

The multi-process servers are conceptually the simplest, and are the oldest architectures for Web servers. One process opens the socket used to accept incoming requests, and then creates multiple copies of itself using the `fork()` system call. The earliest servers would fork a new process that exited after each request, but this approach quickly changed to a pool of pre-forked processes that serve multiple requests before exiting. On Unix-like systems, this model is the only option for Apache version 1, and the default for version 2.

Multi-threaded and event-driven servers use a single address space to improve performance and scalability, but also increase programming complexity. Sharing data in one address space simplifies bookkeeping, cross-request policy implementations, and application-level caching. The trade-off is programmer effort – multi-threaded programs require proper synchronization, and event-driven programs require breaking code into non-blocking portions. Both activities require more programmer effort and skill than simply forking processes.

While these architectures differ in memory consumption, scalability, and performance, well-written systems using any of these architectures can handle large volumes of traffic, enough for the vast majority of sites. A site's choice of web server likely depends on factors other than raw capacity, such as specific features, flexibility, operating system support, administrator familiarity, etc.

3   Design

Using a pipe-like mechanism and a simple API, Connection Conditioning performs application-level interposition on connection-related system calls, with all policy decisions made in user-level processes called filters. Applying the pipe design philosophy, these filters each perform sim-
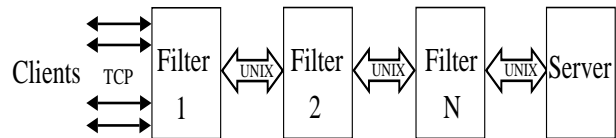

Figure 1: Typical Connection Conditioning usage – the server process invokes a series of filters connected to each other and the server via Unix-domain sockets. The first filter creates the actual TCP listen socket that is exposed to the clients. Clients connections are accepted at this filter, and are passed via file-descriptor passing through the other filters and finally to the server process.

ple tasks, but their combination yields power and flexibility. In this section, we describe the design of Connection Conditioning and discuss its impact on applications.

3.1   General Overview

Connection Conditioning replaces the server's code that accepts new connections, and interposes one or more filters. This design, shown in Figure 1, connects the filters and the server process using Unix-domain sockets. The TCP listen socket, used to accept new connections from clients, is moved from the server to the first filter. If we replaced the clients with standard input, this diagram would look like a piped set of processes spawned by a shell.

While modeled on Unix pipes, Connection Conditioning differs in several domain-specific respects. The most important difference is that rather than passing byte streams, the interface between filters as well as between the filter and server process is passing an atomic, delimited bundle consisting of the file descriptor (socket) and associated request. Using Unix-domain sockets allows open file descriptors to be passed from one process to another via the `sendmsg()` system call. While requests are passed between filters, the server's reply is written directly to the client socket.

Passing the client's TCP connection, rather than proxying the data, provides several benefits. First, the standard networking calls behave as expected since any calls that manipulate socket behavior or query socket information operate on the actual client socket instead of a loopback socket. Second, latency is also lower than a proxy-based solution, since data copying is reduced and the chance of any filter blocking does not affect data sent from the server to the client. Third, performance is also less impaired, since no extra context switches or system calls are needed for the response path, which transfer more data than the request path. Finally, the effort for using CC with existing server software is minimized, since all of the places where the server writes data back to the client are unaffected. Also unmodified are systems like external CGI applications, to which the server can freely pass the client's socket, just as it would without CC.

This approach allows filters to be much simpler than servers, and to be written in different styles – all of the parsing and concurrency management normally associated with accepting requests can be isolated into a single protocol-specific filter that is usable across many servers. Removing this complexity allows each filter and the server to use whatever architecture is appropriate. Programmers can use threads, processes, or events as they see fit, both in the server and in the filters. For simple servers and filters, it is even plausible to not even have any concurrency and handle only one request at a time, as we demonstrate later in the paper. This approach is feasible with Connection Conditioning because all connection management can be moved into the filters.

Note that the filters are tied to the number of features, not the number of requests, so a server will have a small number of filters even if it has many simultaneous connections. In practice, we expect that most servers will use 4 filters. Filter 1 will manage connections and take steps to reduce the possibility of denial-of-service attacks based on exhausting the number of connections. Filter 2 will separate multiple requests on a single connection, and present them as multiple separate connections, in order to eliminate idle connections from using server resources. Filter 3 will perform protocol-specific checks to stop malformed requests, buffer overflows, and other security attacks. Filter 4 can perform whatever request prioritization policy the server desires.

Filters are generally tied to the protocol, not the application, allowing filters to be used across servers, and encouraging "best practices" filters that consolidate protocol-related handling so that many servers benefit from historical information. For example, the "beck" denial-of-service attack [26] exploits a quadratic algorithm in URL parsing, and was discovered and fixed on Apache in 1998. The exact same attack is still effective against the thttpd server [1], despite being demonstrated in a thttpd-derived server in 2002 [21]. The beck attack is worse for thttpd than Apache, since thttpd is event-driven, and the attack will delay all simultaneous connections, instead of just one process. Thttpd is used at a number of high-profile sites, including Kmart, Napster, MTV, Drudge Report, and Paypal. Using CC, a single security filter could be used to protect a range of servers from attacks, giving server developers more time to respond.

CC filters are best suited to environments that consist of request/response pairs, where no hidden state is maintained across requests, and where each transaction is a single request and response. In this scenario, all request-related blocking is isolated in the first (client-facing) filter, which only passes it once the full request has arrived. Intermediate filters see only complete requests, and do not have to be designed to handle blocking. If the server's responses can fit into the outbound socket buffer, then any remaining blocking in the server may be entirely bounded and predictable. In these cases, the server can even handle just a single request at a time, without any parallelism. All of the normal sources of unpredictable blocking (waiting on the request, sending the reply) are handled either by CC filters or by the kernel. This situation may be very common in sensor-style servers with small replies.

To handle other models of connection operation, like persistent connections, the semantics of filters can be extended in protocol-specific ways. Since persistent connections allow multiple requests and responses over a single connection, simply passing the initial request to the server does not prevent all future blocking. After the first request is handled, the server may have to wait for further requests. Even if the server is designed to tolerate blocking, it may cause resources, such as processes or threads, to be devoted to the connection. In this case, the server can indicate to the filter that it wants the file descriptor passed to it again on future requests. Since the filter also has the file descriptor open, the server can safely close it without disconnecting the client. In this manner, the client sees the benefits of persistent connections, but the server does not have to waste resources managing the connection during the times between requests.

3.2 Connection Conditioning Library

To implement Connection Conditioning, we provide a library, shown in Figure 2. One function replaces the three system calls needed to create a standard TCP listen socket, and the rest are one-to-one analogues of standard Unix system calls. The parameters for most calls are identical to their standard counterparts, and the remaining parameters are instantly recognizable to server developers. We believe that modifying existing servers to use Connection Conditioning is straightforward, and that using them for new servers is simple. Any of these calls can be used in process-based, threaded, or event-driven systems, so this library is portable across programming styles. This library also depends on only standard Unix system calls, and does not use any kernel modifications, so is portable across many operating systems. The library contains 244 lines of code and 89 semicolons. Its functions are:

cc_createlsock – instantiates all of the Connection Conditioning filters used by this server. Each filter in the NULL-terminated array `filters[]` is spawned as a separate process, using any arguments provided by the server. Each filter shares a Unix-domain socket with its parent. The list of remaining filters to spawn is passed to the newly-created filter. The final filter in the list creates the listen socket that accepts connections and requests from the client. The server specifies all of the filters, as well as the parameters (address, port number, backlog) for the listen socket, in the cc_createlsock call. The server process no longer needs to call the

```
int cc_createlsock(struct in_addr sin_addr,
                   in_port_t sin_port,
                   int backlog,
                   char *filters[]);
int cc_accept(int s, struct sockaddr *addr,
              socklen_t *addrlen);
ssize_t cc_read(int fd, void *buf,
                size_t count);
int cc_close(int fd, int closeAllFilters);
int cc_select(int n, fd_set *readfds,
              fd_set *writefds,
              fd_set *exceptfds,
              struct timeval *timeout);
int cc_poll(struct pollfd fds[],
            nfds_t nfds, int timeout);
int cc_dup(int oldfd);
int cc_dup2(int oldfd, int newfd);
pid_t cc_fork(void);
```

Figure 2: Connection Conditioning Library API

socket/bind/listen system calls itself. The return value of cc_createlsock is a socket, suitable for use with cc_accept. Our filter instantiation differs from Unix pipes, since the server instantiates them, instead of having the shell perform the setup. This approach requires much less modification for existing servers, and it also avoids conflicts with stdin/stdout.

cc_accept – this call replaces the accept system call, and behaves similarly. However, instead of receiving the file descriptor from the networking layer, it is received from the filter closest to the server. The file descriptor still connects to the client and is passed using the sendmsg() system call, which also allows passing the request itself. The request is read and buffered, but not presented yet.

cc_read – when cc_read is first called on a socket from cc_accept, it returns the buffered request, and behaves as a standard read system call on subsequent calls. The reason for this behavior is because the socket is actually terminated at the client. If any filter were to write data into the socket, it would be sent to the client. So, the filters send the (possibly updated) request via sendmsg when the client socket is being passed.

In multi-process servers, with many processes sharing the same listen socket, the atomicity of sendmsg and recvmsg ensures that the same process gets both the file descriptor and the request. If requests will be larger than the Unix-domain atomicity limit, each process has its own Unix-domain socket to the upstream filter, and calling cc_accept sends a sentinel byte upstream. The upstream filter sends ready requests to any willing downstream filter on its own socket.

cc_close – since the same client socket is passed to all of the filters and the actual server, some mechanism is needed to determine when the socket is no longer useful. Some filters may want to keep the connection open longer than the server, while other filters may not care about the connection after passing it on. The cc_close call provides for this behavior – the server indicates whether only it is done with the connection or whether it and all filters should abandon the connection. The former case is useful for presenting multiple requests on a persistent connection as multiple separate connections. The latter case handles all other scenarios, as well as error conditions where a persistent connection needs to be forcibly closed by the server.

cc_select, cc_poll – these functions are needed by event-driven servers, and stem from transferring the request during cc_accept. Since the request is read and buffered by the CC library, the actual client socket will have no data waiting to be read. Some event-driven servers optimistically read from the socket after accept, but others use poll/select to determine when the request is ready. In this case, the standard system calls will not know about the buffered request. So, we provide cc_select and cc_poll that check the CC library's buffers first, and return immediately if buffered requests exist. Otherwise, they simply call the appropriate system calls.

cc_dup, cc_dup2, cc_fork – These functions replace the Unix system calls dup, dup2, and fork. All of these functions affect file descriptors, some of which may have been created via cc_accept. As such, the library needs to know when multiple copies of these descriptors exist, in order to adjust reference counts and close them only when the descriptor is closed by all readers.

While the CC Library functions are easily mapped to standard system calls, transparently converting applications by replacing dynamically-linked libraries is not entirely straightforward. The cc_createlsock call replaces socket, bind, and listen, but these calls are also used in other contexts. Determining future intent at the time of the socket call may be difficult in general.

4 Evaluation

Our evaluation of Connection Conditioning explores three issues: writing servers, CC performance, and CC security. We also examine filter writing, but this issue is secondary to developers if the filters are reusable and easily extensible. We first present a simple server designed with Connection Conditioning in mind, and then discuss the effort involved in writing filters. We compare its performance to other servers, and then compare the performance effects of other filters. Finally, we examine various security scenarios, and show that Connection Conditioning can improve server security.

4.1 A Simple Server

To demonstrate the simplicity of writing a Web server using Connection Conditioning, we build an extremely simple Web server, called CCServer. Using this server, we test whether the performance of such a simple system would be sufficient for most sites. The pseudo-code for the main loop, almost half the server, is shown in Figure 3. This listing, only marginally simplified from the actual source code, demonstrates how simple it can be to build servers using Connection Conditioning. The total source for this server is 236 lines, of which 80 are semicolon-containing lines. In comparison, Flash's static content handling and Haboob (not including NBIO) require over 2500 semicolon-lines and Apache's core alone (no modules) contains over 6000. Note that we are not advocating replacing other servers with CCServer, since we believe it makes sense to simply modify servers to use CC.

CCServer's design sacrifices some performance for simplicity, and achieves fairly good performance without much effort. Its simplicity stems from using CC filters, and avoiding performance techniques like application-level caching. CCServer radically departs from current server architectures by handling only one request at a time. The only exception is when the response exceeds the size of the socket buffer, in which case CCServer forks a copy of itself to handle that request. Within limits, the socket buffer size can be increased if very popular files are larger than the default, in which case one time cache miss in the main process is also justified – with the use of the zero-copy sendfile call, multiple requests for a file consume very little additional memory beyond the file's data in the filesystem cache. Parallelism is implicitly achieved inside the network layer, which handles sending the buffered responses to all clients.

CCServer ignores disk blocking for two reasons: decreasing memory costs means that even a cheap system can cache a reasonably large working set, and consumer-grade disk drives now have sub-10ms access times, so even a disk-bound workload with small files can still generate a fair amount of throughput. To really exceed the size of main memory, the clients must request fairly large files, which can be read from disk with high bandwidth. It is possible to build degenerate workloads with thousands of small-file accesses, but using a filter that gives low priority to heavy requestors (described in Section 4.2) will limit the performance degradation that other clients see.

The only obvious denial-of-service attack we can see in this approach is that an attacker could request many large files, causing a large number of processes to exist, and could make the situation worse by reading the response data very slowly. This situation is not unique to CCServer – any server, particularly threaded or process-oriented servers, are vulnerable to these attacks. All

```
char *filters[] = {"flt_prior", "flt_persist",
                   "flt_request", NULL};
char request[MAXREQUEST];
int s, c;

s = cc_createlsock(INADDR_ANY, SERV_PORT,
                   BACKLOG, filters);

while ((c = cc_accept(s, NULL, NULL)) >= 0) {
  bool is_child  = false, send_body = true;
  off_t offset   = 0;
  fileinfo file;

  cc_read(c, request, sizeof(request));
  file = parse_and_openfile(request);
  send_header(c, file.size);

  set_sendbufsize(c, SENDBUFSIZE);
  if (file.size > SENDBUFSIZE) {
    /* let a child process send the body */
    if (cc_fork() != 0) send_body = false;
    else is_child = true;
  }

  if (send_body) /* send the body */
    sendfile(c, file.fd, &offset, file.size);

  cc_close(c);
  close(file.fd);

  if (is_child) return 0;
}
```

Figure 3: Pseudo code of the really simple CCServer

of these techniques can be handled similar to how they would be handled in other servers. We could set process limits in the shell before launching the server, in order to ensure that too many processes are not created. To handle the "slow reading" attack, we could split the sendfile into many small pieces, and exit if any piece is received too slowly. With CC filters, we could use a filter that places low priority on heavy requestors, which would reduce the priority of any attacker.

All of the other concerns that one would expect, such as how long to wait between a connection establishment and the request arrival, how long to keep persistent connections open, etc., are handled by filters outside the server. Normally, all of these issues would cause a server that handled only one request at a time to block for unbounded amounts of time, and would necessitate some parallelism in the server's architecture, even for simple/short requests.

4.2 Filters

We have developed filters that implement different connection management and security policies. We find filter development relatively straightforward, and that the basic filter framework is easy to modify for different purposes. Common idioms also emerge in this approach, leading us to believe that filter development will be manageable for the programmers who need to write their own.

	Total (Semicolons)
Packaging	687 (248)
Persistence	+76 (+26)
Priority	531 (211)
Slow Read	587 (212)

Table 1: Line counts for filters – the persistence filter is conditionally-compiled support in the packaging filter, so its counts are shown as the extra code for this feature. The other filter line counts include the basic framework, which is 413 lines of source, and 152 semicolons.

We have found two common behavioral styles for filters, and these shape their design. Those that implement some action on individual requests, such as stripping pathname components or checking for various errors, can be designed as a simple loop that accepts one request, processes it, and passes it to the next filter. Those that make policy decisions across multiple requests are conceptually small servers themselves.

These filters are an important aspect of the system, since they are key to preserving programming style while enhancing security. In traditional multi-process servers without Connection Conditioning, making a policy decision across all active requests is difficult enough, but it is virtually impossible to consider those requests that are still waiting in the accept queue. Since the number of those requests may exceed the number of processes in the server, certain security-related policy decisions are unavailable to these servers.

The filters, in contrast, can use a different programming style, like event-driven programming, so that each request consumes only a file descriptor instead of an entire process. In this manner, the filters can examine many more requests, and can more cheaply make policy decisions. We use a very simple event-driven framework for the policy filters, since we are particularly interested in trying to implement policies that can effectively handle various DoS attacks. To gain cross-platform portability and efficiency, we use the libevent library [19], which supports platform-specific event-delivery approaches (kqueue [15], epoll, /dev/poll) in addition to standard `select` and `poll`. Our filters include:

Request packaging – this filter is often the first filter in any server. It accepts connections, reads the requests, and hands complete requests to the next filter. By making the filter event-driven, it can handle attacks that try to starve the server by opening thousands of connections without sending requests. The filter is only limited by the number of file descriptors available to it, and we implement some simple policies to prevent starvation. Any connection that is incomplete (has not sent a full request) before a configurable timeout is terminated, and if the filter is running

out of sockets, incomplete connections are terminated by network address. This filter maintains a 16-ary tree organized by network address, where each node has a count of all open connections in its children. The filter follows the path with the heaviest weights, ensuring that the connection it terminates is coming from the range of network addresses with the most incomplete connections.

Persistent connection management – while persistent connections help clients, they present connection management problems for servers, so this filter takes multiple requests from a persistent connection and separates them into individual requests. When the server is done with the current request, it closes the connection, and this filter re-sends the next request as a new connection. Since the filters keeps a socket open, the server closing a persistent connection is only a local operation, and is not visible to the client. We expect that this filter would be the second filter after the request packaging filter.

Recency-based prioritization – this filter acts as a holding area after the full request has arrived. It provides a default policy that makes high-rate attacks less effective, without requiring any feedback or throttling information from the server. As a side-effect, it also provides simple fairness among different users. This would be the last filter before the server. This filter basically accepts all requests coming from the previous filter, and then picks the highest-priority request when the server asks for one. The details of this approach are described in Section 4.2.1.

Slow read prevention – this filter limits the damage caused by "slow read" clients, who request a large file and then keep the connection open by reading the data slowly. In a DoS scenario, if a client could keep the connections open arbitrarily long, the prioritization filter alone would not prevent it from having too many connections. This filter explicitly checks how many concurrent connections each client has, and delays or rejects requests from any client range that is too high. We currently set the defaults to allowing no more than 25% of all connections from a /8 network range, 15% from a /16, and 10% from a /24. This approach limits slow-read DoS, but can not fully protect against DDoS. Still, any security improvement is a benefit for a wide range of servers.

We have also developed other more specialized filters, such as ones that look for oddly-formatted requests, detect and strip the beck attack, etc. Line count information for the filters described above are presented in Table 1.

4.2.1 The Recency Algorithm

To handle rate-based attacks coming from sets of addresses, we use an algorithm that aggregates traffic statistics automatically at multiple granularities, but does not lose preciseness. We break the network address into 8 pieces of 4 bits each. We use an 8 deep 16-ary tree, with
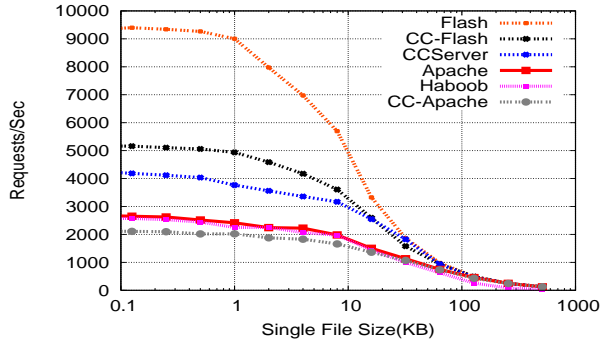
Figure 4: Single File Requests/sec

Figure 5: Single File Throughput

an LRU list maintained across the 16 elements of each parent. Each node also contains a count to indicate how many requests are stored in its subtrees. The tree is lazily allocated – any levels are allocated only when distinct addresses exist in the subtree. When a new request arrives, it is stored in the tree, creating any nodes that are needed, and updating all counts of requests. When it is time to provide the next request to be serviced, we descend each level of the tree, using the LRU child with a nonzero request count at each level. The request chosen by this process is removed, all counts are updated, and all children along the path are moved to the ends of their respective LRU lists. Subtrees without requests can be pruned if needed.

If an attacker owns an entire range of network addresses, a low-frequency client from another address range will always take priority in having its requests serviced or its incomplete connections kept alive. Even if the low-frequency client is more active than any individual compromised machine, this algorithm will still give it priority due to the traffic aggregation behavior. At the same time, the aggregation does not lose precision – if even a single machine in the attacker's range remains uncompromised, when it does send requests, they will receive priority over the rest of the machines in that range.

4.3 Performance Evaluation

Though performance is not a goal of Connection Conditioning, we evaluate it so that designers and implementors have some idea of what to expect. While we believe it is true that performance is generally not a significant factor in these decisions, it would become worrying if the performance impact caused any significant number of sites to reject such an approach. As we show in this section, we believe that the performance impact of Connection Conditioning is acceptable.

4.3.1 Testbed and Servers

Our testbed servers consist of a low-end, single processor desktop machine, as well as an entry-level dual-core server machine. Most of our tests are run on a $200 Microtel PC from Wal-Mart, which comes with a 1.5 GHz AMD Sempron processor, 40 GB IDE hard drive, and a
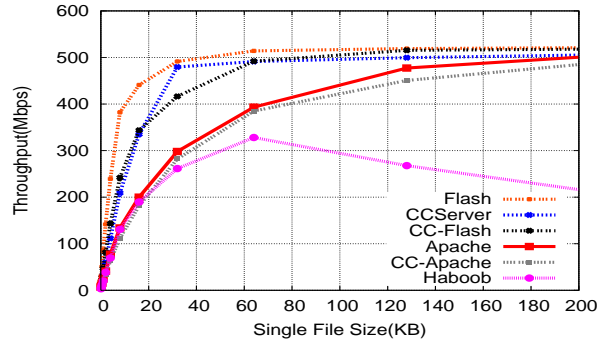
built-in 100 Mbps Ethernet card. We add 1 GB of memory and an Intel Pro/1000 MT Server Gigabit Ethernet network adapter, bringing the total cost to $396. Using a Gigabit adapter allows us to break the 100 Mbps barrier, just for the sake of measurement. The dual-core server is an HP DL320 with a 2.8 GHz Intel Pentium D 820, 2 GB of memory, and a 160GB IDE hard drive. This machine is still modestly priced, with a list price of less than $3000. Both machines run the Linux 2.6.9 kernel using the Fedora Core 3 installation. Our test harness consists of four 1.5 GHz Sempron machines, connected to the server via a Netgear Gigabit Ethernet switch.

In various places in the evaluation, we compare different servers, so we briefly describe them here. We run the Apache server [3] version 1.3.31, tuned for performance. Where specified, we run it with either the default number of processes or with higher values, up to both the "soft limit," which does not require recompiling the server, and above the soft limit. The Flash server [18] is an event-driven research server that uses *select* to multiplex client connections. We use the standard version of it, rather than the more recent version [23] that uses sendfile and epoll. The Haboob server [32] uses a combination of events and threads with the SEDA framework in Java. We tune it for higher performance by increasing the filesystem cache size from 200MB to 400MB. CCServer is our simple single-request server using Connection Conditioning. CC-Apache and CC-Flash are the Apache and Flash servers modified to use Connection Conditioning. In all of the servers using Connection Conditioning, we employ a single filter unless otherwise specified. Since the CC Library currently only supports C and C++, we do not modify Haboob. All servers have logging disabled since their logging overheads vary significantly.

4.3.2 Single-File Tests

The simplest test we can perform is a file transfer microbenchmark, where all of the clients request the same file repeatedly in a tight loop. This test is designed to give an upper bound on performance for each file size, rather than being representative of standard traffic. The results of
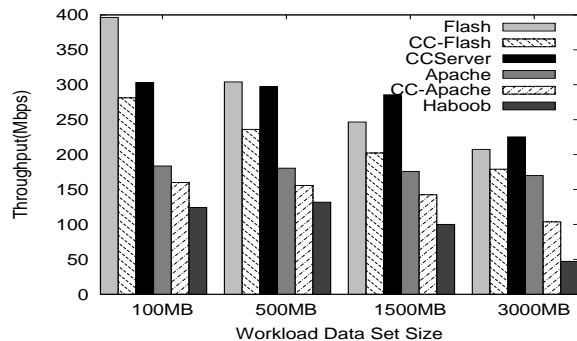
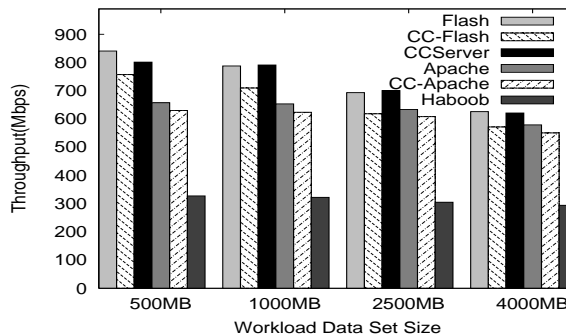Figure 6: SpecWeb99-like workloads on the Microtel machine

Figure 7: SpecWeb99-like workloads on the HP server

this test on the Microtel machine are shown in Figures 4 and 5 for request rate and throughput, respectively. The relative positions of Flash, Apache, and Haboob are not surprising given other published studies on their performance. Performance on the HP server is higher, but qualitatively similar, and is omitted for space.

The performance of CCServer is encouraging, since this would mean that it should have acceptable performance for any site using Apache. Any performance loss due to forking overhead once the response size exceeds the socket buffer size is not particularly visible. This server is clearly not functionally comparable to Apache, but given the use of multiple processes in request handling, we are pleased with the results.

Using Connection Conditioning filters with other servers also seems promising, as seen in the results for CC-Apache and CC-Flash. Both show performance loss when compared to their native counterparts, but the loss is more than likely tolerable for most sites. We investigate this further on a more realistic workload mix next.

4.3.3 More Realistic Workloads

While the single-file tests show relative request processing costs, they do not have the variety of files, with different sizes and frequency distributions, that might be expected in normal Web traffic. For this, we also evaluate these servers using a more realistic workload. In particular, we use a distribution modeled on the static file portion of SpecWeb99 [28], which has also been used by other researchers [23, 30, 32]. The SpecWeb99 benchmark scales data set size with throughput, and reports a single metric, the number of simultaneous connections supported at a specified quality of service.

We instead use fixed data set sizes and report the maximum throughput achieved, which provides a broader range of results for each server. We maintain the general access patterns – a data set contains a specified number of files per directory, with a specified access frequency for files within each directory. The access frequency of the directories follows a Zipf distribution, so the first directory is accessed N times more than the Nth directory.

The results of these capacity tests, shown in Figures 6 and 7, show some expected trends, as well as some subtler results. The most obvious trend is that once the data set size exceeds the physical memory of the machine, the overall performance drops due to disk accesses. For most servers, the performance prior to this point is roughly similar across different working set sizes, indicating very little additional work is generated for handling more files, as long as the files fit in memory. CCServer performs almost three times as well on the HP server as the Microtel box, demonstrating good scalability with faster hardware.

The performance drop at 3 or 4 GB instead of 1.5/2.5 GB can be understood by taking into account SpecWeb's Zipf behavior. Even though a 1.5 and 2.5 GB data sets exceed the physical memories of the machines, the Zipf-distributed file access causes the more heavily-used portion to fit in main memory, so this size has a mix of in-memory and disk-bound requests. At 3GB, more requests are disk-bound, causing the drop in performance across all servers. The HP machine, with a larger gap between CPU speed and disk speed, shows relatively faster degradation with the 4GB data set. Though CCServer makes no attempt to avoid disk blocking, its performance is still good on this workload.

In general, the results for the CC-enabled servers are quite positive, since their absolute performance is quite good, and they show less overhead than the single file microbenchmarks would have suggested. The main reason for this is that the microbenchmarks show a very optimistic view of server performance, so any additional overheads appear to be much larger. On real workloads, the additional data makes the overall workload less amenable to caching in the processor, so the overheads of Connection Conditioning are less noticeable.

4.3.4 Chained CC Filters

Inter-process communication using sockets has traditionally been viewed as heavyweight, which may raise concerns about the practicality of using smaller, single-purpose filters chained together to compose behavior.

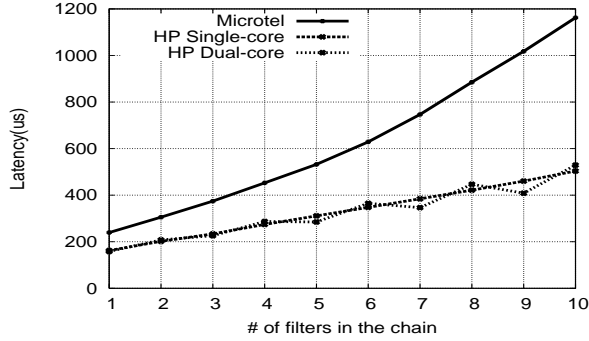To test the latency effects, we vary the number of filters

Figure 8: Latency versus Number of Filters

Figure 9: Throughput versus Number of Filters

used in CCServer, and have a single client issue one request at a time for a single 100-byte file. All of the filters except the first are dummy filters, simply passing along the request to the next filter. These results, shown in Figure 8, show that latency is nearly linear in the number of filters, and that each filter only adds 34 μseconds (HP) or 94 μseconds (Microtel) of overhead. Compared to wide-area delays of 100ms or more, the overhead of chained filters should not be significant for most sites.

The performance effects of chained filters are shown in Figure 9, where an in-memory SpecWeb-like workload is used to drive the test. Given the near-linear effect of multiple filters on latency, the shape of the throughput curve is not surprising. For small numbers of filters, the decrease is close to linear, but the degradation slows down as more filters are added. Even with what we would consider to be far more filters than most sites would use, the throughput is still well above what most sites need.

CCServer performs better on the HP server than the Microtel box on both tests, presumably due to the faster processor coupled with its 1MB L3 cache. The dual-core throughputs scale well versus the single-core, indicating the ability of the various filters in the CC chain to take advantage of the parallel resources. While enabling both cores improves the throughput in this test, it does not improve the latency benchmark, because only one request progresses through the system at a time. The sawtooth pattern stems from several factors: some exploitable parallelism between the clean-up actions of one stage and the start-up actions of the next, SMP kernel overhead, and dirty cache lines ping-ponging between the two independent caches as filters run on different cores.

4.4 Security Evaluation

Here we evaluate the security effects of Connection Conditioning, particularly the policy filters we described in Section 4.2. Note that some of these tests have been used in previous research [21] – our contribution is the mechanism of defending against them, rather than the attacks.

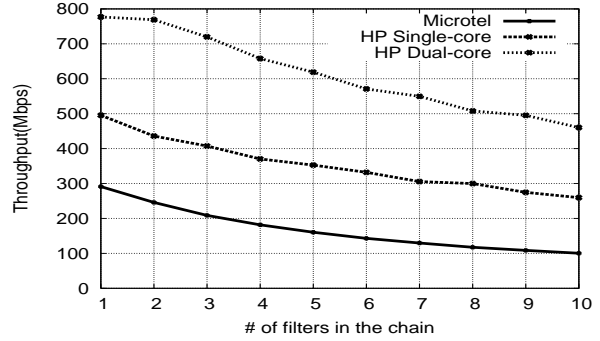Our primary reason for selecting these tests is that they are simple but effective – they could disrupt or deny service to a large fraction of Web sites, and they do not require any significant skill. Each attacking script requires less than 200 lines of code and only a cursory knowledge of network programming and HTTP protocol mechanics. Some of these attacks would also be hard to detect from a traffic viewpoint – they either require very little bandwidth, or their request behavior can be made to look like normal traffic. We focus on the Apache server both because its popularity makes it an attractive target, and because its architecture would normally make some security policies harder to implement. All results are shown for only the Microtel box because these tests focus primarily on qualitative behavior.

4.4.1 Starvation Test – Incomplete Connections

To measure the effect of incomplete connections on the various servers, we have one client machine send a stream of requests for small files, while others open connections without sending requests. We measure the traffic that can be generated by the regular client in the presence of various numbers of incomplete connections. These results are shown in Figure 10, and show various behavior for the different servers. For the process-based Apache server, each connection consumes one process for its life. We see that a default Apache configuration takes only 150 connections, at which point performance drops. Apache employs a policy of waiting 300 seconds before terminating a connection, so at this limit, throughput drops to 0.5 requests/second. Though Flash and Haboob are event-driven, neither have support for detecting or handling this condition. Flash's performance slowly degrades with the number of incomplete connections, and becomes unusable at 32K connections, while Haboob's performance sharply drops after 100 incomplete connections. Flash's performance degradation stems from the overheads of the `select` system call [4].

With the CC Filters, all of these servers remain operational under this load, even with 32K incomplete connections. Since the filter terminates the oldest incomplete connection when new traffic arrives, it can still handle workloads of 1800 requests/sec for CC-Apache, and
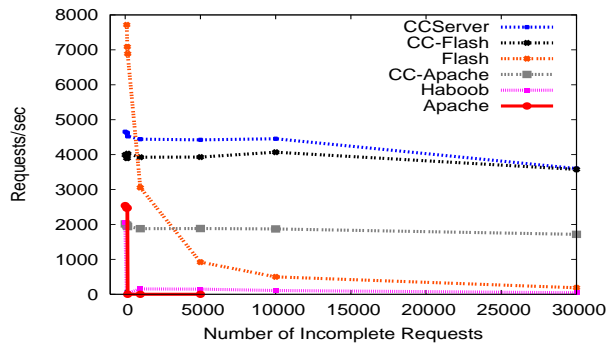
Figure 10: Number of Incomplete Connections Handled

Figure 11: Latency versus Number of Other Clients

3700 requests/sec for CCServer and CC-Flash. This test demonstrates the architecture-neutral security enhancement that Connection Conditioning can provide – both a multi-process server and an event-driven server handle this attack better with Connection Conditioning than their own implementation provides.

4.4.2 Prioritization Test

Though the request packaging filter closes connections in a fair manner, the previous test does not demonstrate fairness for valid requests, so we devise another test to measure this effect. The test consists of a number of clients requesting large files from a default Apache, which can handle 150 simultaneous requests. The remaining requests are queued for delivery, so an infrequent client may often find itself waiting behind 150 or more requests. The infrequent client in our tests requests a small file, to observe the impact on latency.

The results of this test, in Figure 11, show the effect on latency of the infrequently-accessing client. The latency of the small file fetch is shown as a function of the number of clients requesting large files. Without the prioritization filter, Apache treats the request in roughly first-come, first-served order. When the total number of clients is less than the number of processes, the infrequent client can still get service reasonably quickly. However, once the number of clients exceeds the number of server processes, the latency for the infrequent client also increases as more clients request files.

With CC-Apache and the prioritization filter, though, the behavior is quite different. The increase in the number of large-file clients leads to a slight increase in latency once all of the processes are busy. After that point, the latency levels again. This small step is caused by the infrequent client being blocked behind the next request to finish. Once any request finishes, it gets to run, so the latency stays low.

Performing this kind of prioritization in a multiple-process server would be difficult, since each connection would be tied to a process. As a result, it would be hard for the server to determine what request to handle next. With
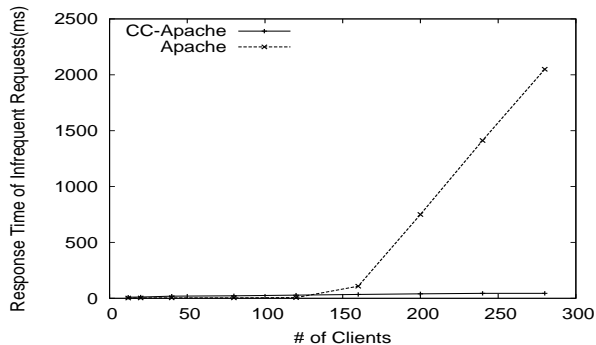
Connection Conditioning, the filter's policy can view all outstanding requests, and make decisions before the requests reach the server.

4.4.3 Persistence Test

Persistent connections present another avenue for connection-based starvation, similar to the incomplete connection attack. In this scenario, an attacker requests a persistent connection, requests a small file, and keeps the connection open. To avoid complete starvation, any reasonable production-class server will have some mechanism to shut down such connections either after some timeout or under file descriptor pressure.

Implementing a self-managing solution is tied to server architecture, complicating matters. While detecting file descriptor pressure is cheap in event-driven servers, they are also less vulnerable, since they can utilize tens of thousands of file descriptors. In contrast, multi-process servers are designed to handle far fewer simultaneous connections, and determining that persistent connection pressure exists requires more synchronization and inter-process communication, reducing performance. The simplest option in these circumstances is to provide administrator-controlled configuration options regarding persistent connection behavior as Apache does. However, the trade-off is that if these timeouts are too short, they make persistent connections less useful, while if they are too long, the possibility of running out of server processes increases.

Figure 12 shows the effect of an attacker trying to starve the server via persistent connections. We use Apache's default persistent connection timeouts of 15 seconds and 150 server processes. An attacker opens multiple connections, requests small files, and holds the connection open until the server closes it. For any closed connection, the attacker opens a new connection and repeats the process. We vary the number of connections used by the attacker. We also have 16 clients on one machine requesting the SpecWeb99-like workload with a 500 MB data set size. We show the throughput received by the regular clients as a function of the number of slow persistent connections. On Apache, the throughput drops beyond 150 persistent
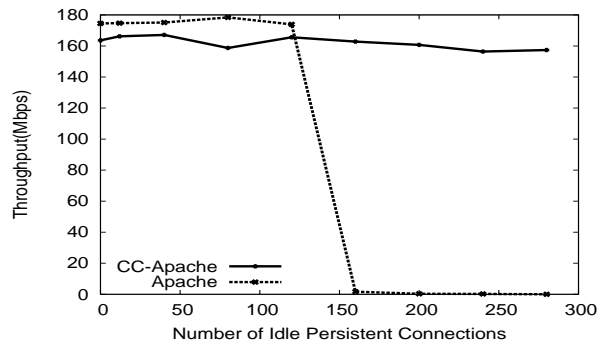
Figure 12: Throughput under Persistent Connection Limits

connections, but CC-Apache shows virtually no performance loss. Its maximum performance is lower than standard Apache due to the CC filters, but it supports more open connections. Apache's server processes never see the waiting periods between requests. This support only required modifying 8 lines in Apache.

5 Discussion

In this section, we discuss some alternatives to Connection Conditioning, some of the objections that may be raised to our claims, and possible deployment questions.

5.1 Novelty and Simplicity

Our contribution in Connection Conditioning is the observation that Unix pipes can be applied to servers, providing all of the benefits associated with text processing (simplicity, composability, and separation of concerns) while still providing adequate performance. In retrospect, this may seem obvious, but we believe that Connection Conditioning's design and focus on adoptability are directly responsible for its other benefits. Our approach allows vastly simpler servers with performance that approaches or even exceeds the designs introduced in the past few years. Particularly for small servers, such as sensors, our approach provides easy development with a broad range of protection, something not available in other approaches. We make no apologies for building on the idea of Unix pipes – given the option to build on a great idea, we see no reason to develop new approaches purely for the sake of novelty.

CC also provides the ability to incorporate best practices into existing servers, without having to start from a clean slate. Given the state of today's hardware, someone designing a server from scratch may develop a design similar to Connection Conditioning. However, even many research servers, with no compatibility constraints, have become increasingly complicated over time, rather than simpler. We consider the ability to support existing servers like Apache while still allowing new designs like CCServer to be a contribution of this work.

5.2 Rich Web Server APIs

Several servers provide rich APIs that can be used to inspect and modify requests and responses – Apache has its module format, Microsoft developed ISAPI, Netscape developed NSAPI, and Network Appliance developed ICAP. Any of these could be used to protect their host server from attacks like the beck attack mentioned earlier. However, we believe Connection Conditioning can provide protection from a separate class of attacks not amenable to protection via server APIs. These attacks, such as the "incomplete connection" starvation attack, waste server resources as soon as the connection has been accepted, and these connections are accepted within the framework of the server. Particularly for process-based servers, the resources consumed just by accepting the connection can be significant. By moving all of the inspection and modification outside the server, Connection Conditioning provides protection against this class of attack. Even event-driven servers can expend more state than Connection Conditioning – in our request prioritization example, we may want to select from tens of thousands of possible connections, particularly when we are under attack. The richness of the server's internal API does no good in this kind of example, since the server may not even be able to accept all of the connections without succumbing to the attack.

Some of CC's other benefits, such as relieving the server of the work of maintaining persistent connections, cannot be done inside all servers without architectural changes. The persistent connection attack we have shown is particularly effective, since regular servers would have to have global knowledge of the state of all requests in order to detect it. With CC, no server re-architecting is required, since this work can be done easily in the filters.

5.3 Security

We have seen that CC protects servers against several DoS attacks, and that it enables other types of protocol-specific security filters. Given how little bandwidth some of these attacks require, and given Apache's wide deployment, we feel that CC can provide an immediate practical security benefit. From a design standpoint, using CC with filters can also provide other benefits – privileged operations, such as communicating with authentication servers or databases, can be restricted to specific policy filters, moving sensitive code out of the larger code base of the main server. These filters, if designed for re-use, can also be implemented using best practices, and can be more thoroughly tested since wider deployment and use with multiple servers is more likely to expose security holes. We admit that some of these benefits will be hard to quantify, but we also feel that some of them are self-evident – moving code out of a large, monolithic server code base and executing it in a separate address space is likely to restrict the scope of any security problem.

5.4 Scope

While our evaluation of Connection Conditioning has focused mostly on Web servers, we believe CC has a fairly broad scope – it is suitable to many request/reply environments that tend to have relatively short-duration "active" periods of their transactions. Our focus on Web servers is mostly due to pragmatism – Web servers are widely deployed, and they provide ample opportunities for comparisons, so our evaluation of CC can be independently assessed. In addition to the server protection offered by CC, we also hope to use it in developing lightweight, DoS-resistant sensors for PlanetLab. We run several sensors on PlanetLab for providing status information – CoMon provides node-level information, such as CPU load and disk activity, while CoTop provides account-level (slice-level) information, such as number of processes and memory consumption. While these tools all use HTTP as a transport protocol, they are not traditional Web servers. By using CC for these tools, we can make them much more robust while eliminating most of their redundant code.

CC is not suitable for all environments, and any server with very long-lived transactions may not gain simplicity benefits from it. Video server match this profile, where a large number of clients may be continuously streaming data over long-lived connections for an hour or more. In this case, CC is no better than other architectures at providing connection management. In all likelihood, this case will require some form of event-driven multiplexing at the server level, whether it is exposed to the programmer or not. CC can still provide some filtering of requests and admission control, but may not be a significant advantage in these scenarios. This example is distinct from the persistent connection example we provided earlier – the difference is that with persistent connections, the long-lived connections may be handling a number of short-lived transfers. In that case, CC can reduce the number of connections actually being handled by the server core.

6 Related Work

While this paper has argued that performance-related advances in server design are of marginal benefit to most Web sites, some classes of servers do see benefit from many advances. Banga and Mogul improved the *select()* system call's performance by reducing the delay of finding ready sockets [5]. They subsequently proposed a more scalable alternative system call [7], which appears to have motivated *kqueue()* on BSD [15] and *epoll()* on Linux. Caching Web proxy servers have directly benefited from this work, since they are often in the path of every request from a company or ISP to the rest of the Internet. Any mechanism that reduces server latency is desirable in these settings. Examining the results from the most recent Proxy Cache-off [22] suggests that vendors are in fact interested in more aggressive server designs. In these

environments, CC may not be the best choice, but many ISPs still use the low-performance Squid proxy, so CC's overhead may be quite tolerable in these environments.

The method of filters we present is very general and allows customizable behavior. The closest approach we have found in any other system is the "accept filter" in FreeBSD, which provides an in-kernel filter with a hard-coded policy for determining when HTTP requests are complete. However, it must be specifically compiled into the kernel or loaded by a superuser. This approach resulted in opening the possibility of denial-of-service attacks on the filter's request parsing policy [10], which would have prevented the application from processing any requests. It would also be unable to handle some of the other starvation attacks we have covered in this paper. Similarly, IIS has an in-kernel component, the Software Web Cache, to handle static content in the kernel itself. While this approach can use kernel interfaces to improve scalability, its desirability may depend on whether the developer is willing to accept the associated risks of putting a full server into the kernel. For some of the cases we have discussed, such as developing simple, custom sensors that use HTTP as a transport protocol, in-kernel servers may provide little benefit if the infrastructure cannot be leveraged outside of its associated tasks.

Some of our security policies are shaped by work on making event-driven servers more responsive under malicious workloads [21]. We have attempted, as much as possible, to broaden these benefits to all servers, with as little server modification as possible. We believe that our recency-based algorithm is an elegant generalization of the approach presented in the earlier work.

While many of our evaluations have used Apache, both because of its popularity and because of the difficulty of performing certain security-related operations in a multi-process server, we believe our approach is fairly general. We have shown that it can be applied to Flash, an event-driven Web server. We believe that it is broadly amenable to other designs, including hybrid thread/event designs such as Knot [30]. While we tried to demonstrate this feasibility, we were unable to get the standard Knot package working in the 2.6 or 2.4 Linux kernel. We believe Connection Conditioning would benefit a system like Knot most by preventing starvation-based attacks. The higher-performance version of Knot, Knot-C, uses a smaller number of threads to handle a large number of connections, possibly leaving it open to this kind of attack. In conjunction with CC Filters, only active connections would require threads in Knot.

Some work has been done on more complicated controllers for overload control [25, 31], which moves request management policy inside the server. If such an approach were desired in Connection Conditioning, it could be done via explicit communication between the filter and servers,

using shared memory or other IPC mechanisms. However, implementing such schemes as filters has the benefit of leaving the design style of the filter up to the developer, instead of having to conform to the server's architecture. Having the filter operate in advance of the server's accepting connections has the possibility of reducing wasted work. Servers would still be free to enforce whatever internal mechanisms they desired.

Similarly, resource containers [6] have been used to provide priority to classes of clients in event-driven and process-based servers. This mechanism can be used to provide a specified level of traffic to friendly clients even when malicious clients are generating heavy traffic. This approach depends on early demultiplexing in the kernel, and forcing policy decisions into the kernel to support this behavior. We believe that resource containers can be used in conjunction with Connection Conditioning, such that livelock-related policies are moved into the kernel with resource containers, and that the CC Filters handle the remaining behavior at application-level.

Finally, a large body of work exists on some form of interposition, often used for implementing flexible security policies. Some examples of this approach include Systrace [20], which can add policies to existing systems, Kernel Hypervisors [16], which can generalize the support for customizable, in-kernel system call monitoring, and Flask [27], an architecture designed to natively provide fine-grained control for a microkernel system. While some of CC's mechanisms could be implemented using system call interposition, the fundamental concerns of CC differ from these projects since filters in CC are trusted, and are logically extending the server, rather than viewing the server in an adversarial context. In this vein, CC is more similar to approaches like TESLA [24], that are designed to extend/offload the functionality of existing systems. Combining CC with TESLA, which provides session-layer services, would be a logical pairing, since their focus areas are complementary. The reason for not using some form of system call interposition in the current CC is that some decisions are simpler when made explicitly – for example, a purely interposition-based system may have a difficult time detecting all uses of the common networking idiom of `socket/listen/accept`, especially if other operations, such as `fork()` or `dup()`, are interleaved. Making CC calls explicit greatly simplifies the library.

7 Future Work

The next step for Connection Conditioning would be to add kernel support for the interposition mechanisms, while still keeping the server and filters in user space. We intentionally keep the filters in user space because we believe that the flexibility of having them easily customizable outweighs any performance gains of putting them in the kernel. We also believe that by moving only the mechanisms into the kernel, Connection Conditioning can be used without requiring root privilege.

The general idea is to allow the server to create its listen socket, and then have a minimal kernel mechanism that allows another process from the same user to "steal" any traffic to this socket. The first filter would then perform connection passing to other filters using the standard mechanisms. However, when the final filter wants to pass the connection to the server, it uses another kernel mechanism to re-inject the connection (file descriptor and request) where it would have gone to the server. In effect, the entire filter chain is interposed between the lower half of the kernel and the delivery to the server's listen socket. Such a scheme would be transparent to the server, and could operate without any server modification if the ability to split persistent connections into multiple connections is not needed. Otherwise, all of the other CC library functions could be eliminated, with only cc_close exposed via the API. Some extra-server process would have to launch all of the filters, and indicate which socket to steal, but this infrastructure is also minimal.

For closed-source servers where even minimal modifications are not possible, this approach may be the only mechanism to use Connection Conditioning. However, since our current focus is on experimentation, the library-based approach provides three important benefits: it is portable across operating systems and kernel versions, it requires less trust from a developer wanting to experiment with it, and it is easier to change if we discover new idioms we want to support. At some point in the future, after we gain more experience with Connection Conditioning, we may revisit an in-kernel mechanism specifically to support closed-source servers.

8 Conclusions

While server software design continues to be an active area of research, we feel it is worthwhile to assess its chances for meaningful impact given the current state of hardware and networking. We believe that performance of most servers is good enough for most sites, and that advances in simplifying server software development and providing better security outweigh additional performance gains. We have shown that a design inspired by Unix pipes, called Connection Conditioning, can provide benefits in both areas, and can even be used with existing server software of various designs. While this approach has a performance impact, we have demonstrated that even on laughably cheap hardware, this system can handle far more bandwidth than most sites can afford.

Connection Conditioning provides these benefits in a simple, composable fashion without dictating programming style. We have demonstrated a new server that is radically simpler than most modern Web servers, and have

shown that fairly simple, general-purpose filters can be used with this server and others to provide good performance and security. The current implementation runs entirely in user space, which gives it more flexibility and safety compared to a kernel-based approach. However, a kernel-space implementation of the mechanisms is possible, allowing for improved performance while retaining the flexibility of user-space policies.

Overall, we believe that Connection Conditioning holds promise for simplifying server design and improving security, and should be applicable to a wide range of network-based services in the future. We have demonstrated it in conjunction with multi-process servers as well as event-driven servers, and have shown that it can help defend these servers against a range of attacks. We are investigating its use for DNS servers, which tend to prefer UDP over TCP in order to reduce connection-related overheads, and for sensors on PlanetLab, which use an HTTP framework for simple information services. We expect that both environments will also prove amenable to Connection Conditioning.

Acknowledgments

References

[1] ACME Laboratories. thttpd. http://www.acme.com/thttpd.

[2] Akamai Technologies Inc. http://www.akamai.com/.

[3] Apache Software Foundation. Apache HTTP Server Project. http://httpd.apache.org/.

[4] G. Banga, P. Druschel, and J. C. Mogul. Better operating system features for faster network servers. In *Proceedings of the Workshop on Internet Server Performance*, June 1998.

[5] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[6] G. Banga, J. C. Mogul, and P. Druschel. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[7] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for unix. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, 1999.

[8] A. Chankhunthod, P. B. Danzig, C. Neerdaeles, M. F. Schwartz, and K. Worrell. A hierarchical Internet object cache. In *Proceedings of the USENIX Annual Technical Conference*, January 1996.

[9] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. In *Sixth International World Wide Web Conference*, April 1997.

[10] FreeBSD Project. Remote denial-of-service when using accept filters. http://www.securityfocus.com/advisories/4159.

[11] Germán Goldszmidt and Guerney Hunt. NetDispatcher: A TCP Connection Router. Technical report, IBM Research, Hawthorne, New York, July 1997. RC 20853.

[12] J. Hu, I. Pyarali, and D. C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proceedings of the IEEE GLOBECOM '97*, November 1997.

[13] P. Joubert, R. King, R. Neves, M. Russinovich, and J. Tracey. High-performance memory-based web servers: Kernel and user-space performance. In *Proceedings of the USENIX Annual Technical Conference*, June 2001.

[14] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, 1997.

[15] J. Lemon. Kqueue: A generic and scalable event notification facility. In *FREENIX Track: USENIX Annual Technical Conference*, Boston, MA, June 2001.

[16] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Proceedings of the 13th Annual Computer Security Applications Conference (ACSAC '97)*, San Diego, CA, 1997.

[17] Netcraft Ltd. Web server survey archives. http://news.netcraft.com/archives/web_server_survey.html.

[18] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference*, June 1999.

[19] N. Provos. libevent. http://www.monkey.org/ provos/libevent/.

[20] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, 2003.

[21] X. Qie, R. Pang, and L. Peterson. Defensive Programming: Using an Annotation Toolkit to Build DoS-Resistant Software. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA USA, December 2002.

[22] A. Rousskov, M. Weaver, and D. Wessels. The fourth cache-off. Raw data and independent analysis. http://www.measurement-factory.com/results/.

[23] Y. Ruan and V. Pai. Making the "Box" transparent: System call performance as a first-class result. In *USENIX Annual Technical Conference*, Boston, MA, June 2004.

[24] J. Salz, A. C. Snoeren, and H. Balakrishnan. TESLA: A transparent, extensible session-layer architecture for end-to-end network services. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems(USITS '03)*, Seattle, WA, 2003.

[25] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. In *18th International Teletraffic Congress (ITC 2003)*, August 2003.

[26] M. Slemko. Possible security issues with Apache in some configurations. http://www.cert.org/vendor_bulletins/VB-98.02.apache.

[27] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, Washington, DC, 1999.

[28] Standard Performance Evaluation Corporation. SPEC Web Benchmarks. http://www.spec.org/web99/.

[29] G. Varghese and A. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proceedings of the 11th Symposium on Operating System Principles (SOSP-11)*, Austin, TX, 1987.

[30] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for Internet services. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP-19)*, Lake George, New York, October 2003.

[31] M. Welsh and D. Culler. Adaptive overload control for busy Internet servers. In *4th USENIX Conference on Internet Technologies and Systems (USITS'03)*, March 2003.

[32] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th Symposium on Operating System Principles(SOSP-18)*, Chateau Lake Louise, Banff, Canada, Oct. 2001.