A Flexible and Efficient Application Programming Interface (API) for a Customizable Proxy Cache

Vivek S. Pai, Alan L. Cox, Vijay S. Pai, and Willy Zwaenepoel
iMimic Networking, Inc.
2990 Richmond, Suite 144
Houston, TX 77098

Abstract

This paper describes the design, implementation, and performance of a simple yet powerful Application Programming Interface (API) for providing extended services in a proxy cache. This API facilitates the development of customized content adaptation, content management, and specialized administration features. We have developed several modules that exploit this API to perform various tasks within the proxy, including a module to support the Internet Content Adaptation Protocol (ICAP) without any changes to the proxy core.

The API design parallels those of high-performance servers, enabling its implementation to have minimal overhead on a high-performance cache. At the same time, it provides the infrastructure required to process HTTP requests and responses at a high level, shielding developers from low-level HTTP and socket details and enabling modules that perform interesting tasks without significant amounts of code. We have implemented this API in the portable and high-performance iMimic DataReactor™ proxy cache[1]. We show that implementing the API imposes negligible performance overhead and that realistic content-adaptation services achieve high performance levels without substantially hindering a background benchmark load running at a high throughput level.

1 Introduction

As the Internet has evolved, Web proxy caches have taken on additional functions beyond caching Internet content to reduce latency and conserve bandwidth. For instance, proxy caches in schools and businesses often perform content filtering, preventing users from accessing content deemed objectionable. Caches in content distribution networks (CDNs) may perform detailed access logging for accounting purposes, pre-position popular items in the cache, and prevent the eviction of certain items from memory or disk storage. Support for these features has been added to caches developed in both academia and industry.

However, a proxy cache designer can not foresee all possible uses for the proxy cache and thus cannot include all features required by application implementers. While some developers of major systems such as CDNs have added their desired functionality to open-source caches, these developers are then burdened by the sheer volume of source code (over 60,000 lines in Squid-2.4 [18]). Additionally, their changes will likely conflict with later updates to the base proxy source, making it difficult to track bug fixes and upgrades effectively. Consequently, such *ad hoc* schemes erode the separation of concerns that underlies sound software engineering. The application developer should not have to reason with the details of the cache in order to add functionality. Instead, the developer should be able to write in standard languages such as C using standard libraries and system calls as appropriate for the task at hand.

One approach to enable such value-added services is to locate those functions on a separate server that communicates with the cache through a domain-specific protocol. The Internet Content Adaptation Protocol (ICAP) adopts this approach, allowing caches to establish TCP connections with servers that modify requests and responses to and from clients and origin servers [8]. On each HTTP request and response that will be modified, an ICAP-enabled proxy constructs a request which consists of ICAP-specific headers followed by the complete HTTP headers and body in chunked format. The proxy then collects a response from the ICAP server providing a complete set of modified HTTP headers and body. In addition to the TCP/IP socket overhead for communicating with the external service, such a protocol also adds overhead to parse the protocol headers and chunked data transfer format and to encapsulate HTTP messages within the protocol. Further, current implementations of ICAP locate value-added services on a separate server machine, even if the host CPU of the cache is not saturated.

An alternative approach is to use an application programming interface (API) that allows user modules to be directly loaded into the core of the cache and run ser-

[1] DataReactor is a trademark of iMimic Networking, Inc.

vices either on the cache system or on a separate server as desired. This paper presents an API that enables programming of a proxy cache after deployment. This API turns a previously monolithic proxy cache into a programmable component in a Web content delivery infrastructure, cleanly separating new functionality from the cache's core. At the same time, the API allows extremely fast communication between the cache and the user modules without the need for TCP connections or a standardized parsing scheme. Specifically, the API provides the infrastructure to process HTTP requests and responses at a high level, shielding developers from the low-level details of socket programming, HTTP interactions, and buffer management. This API can also be used to implement the ICAP standard by creating a dynamically loaded module that implements the TCP and parsing aspects of ICAP. The API extends beyond the content adaptation features of ICAP by providing interfaces for content management and specialized administration.

Several technological trends have made single-box deployment of API-enabled proxy servers more attractive. Among these are more available processing power and better OS support for high-performance servers. Proxy software in general has improved in efficiency while microprocessor speeds have increased. Recent benchmarks have shown that a 300 MHz 586-class processor is sufficient to handle over 7Mbps of traffic, enough for multiple T-1 lines [15]. Current microprocessors with two architectural generations of improvement and a clock rate that is 5-8 times higher will have significant free CPU for other tasks. Proxy servers running on general-purpose operating systems have met or exceeded the performance of appliances running customized operating systems. As a result, the proxy server has become a location that handles HTTP traffic and has the capacity and flexibility to support more than just caching.

Unlike some previously proposed schemes for extending server or proxy functionality, the API presented in this paper uses an event-aware design to conform to the implementation of high-performance proxy servers. By exposing the event-driven interaction that normally occurs within proxies, high performance implementations can avoid the overhead of using threads or processes to handle every proxy request. We believe that this performance-conscious approach to API design allows higher scalability than previous approaches, following research showing the performance advantages of event-driven approaches to server design in general [13].

We have implemented this API in the iMimic DataReactor, a portable, high-performance proxy cache. We show that implementing the API imposes negligible performance overhead and allows modules to consume free CPU cycles on the cache server. The modules themselves achieve high performance levels without substantially hindering a background benchmark load running at high throughput. While the API style is influenced by event-driven server design, the API is not tied to the architecture of any cache, and it can be deployed more widely given systems that support standard libraries and common operating system abstractions (e.g., threads, processes, file descriptors, and polling).

The rest of this paper proceeds as follows. Section 2 describes the general architecture of the system and the design of modules that access the API. Section 3 discusses the API in more detail. Section 4 describes sample modules used with the API and discusses coding issues for these modules. Section 5 provides a more detailed comparison of the API with ICAP. Section 6 describes the implementation of the API in the iMimic DataReactor proxy cache and presents its performance for some sample modules. Section 7 discusses related work, and Section 8 summarizes the conclusions of this paper.

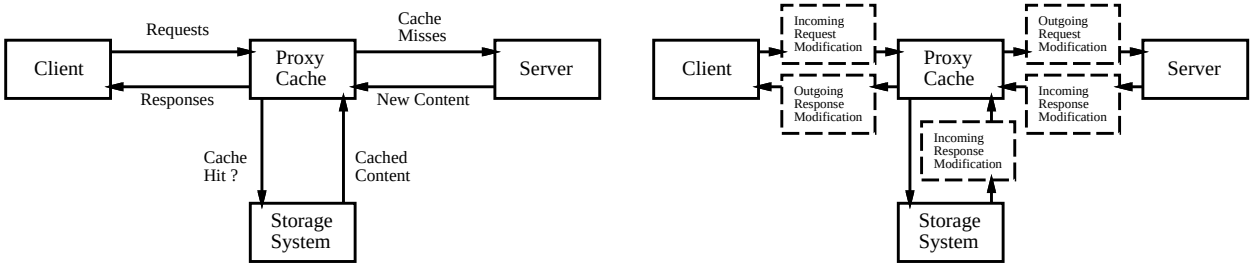
2 Structure of the API

The underlying observation that shapes the structure of the API is that a high-performance API should adopt the lessons learned from the design of high-performance server architectures. As a result, we use an event-driven approach as the most basic interaction mechanism, exposing this level directly to modules as well as using it to construct support for additional communication models based on processes or threads. By exposing the event-driven structure directly to modules, the API can achieve high scalability with minimal performance impact on the proxy. This approach dispenses with multiple thread contexts or multiple processes, enabling the scalability gains previously observed for server software [13]. The rest of this section describes how the API integrates with HTTP processing in the cache and how customization modules are designed and invoked.

2.1 Integration with HTTP Handling

Figure 1 illustrates the flow of request and response content through the proxy. Clients send requests to the proxy, either directly or via redirection through an L4/7 switch. The proxy interprets the request and compares it with the content it has stored locally. If a valid, cached copy of the request is available locally, the proxy cache obtains the content from its local storage system and returns it to the client. Otherwise, the proxy must modify the HTTP headers and contact the remote server to obtain the object or revalidate a stale copy of the object fetched earlier. If the object is cacheable, the proxy cache stores a copy of it in order to satisfy future requests for the object.

These various interactions between the proxy, the



(a) Original proxy server

(b) API-Enabled proxy server

Figure 1: Original and modified data transfer paths in a proxy server

client, and the server provide the basis for the primary functions of the API. In particular, the API provides customization modules with the ability to register *callbacks*, special functions with defined inputs and outputs that are invoked by the cache on events in the HTTP processing flow. These callback points are the natural processing points within an event-driven server; by exposing this structure, the API implementation can support modules with high scalability and low performance impact. Examples of such HTTP processing events include the completion of the request or response header, arrival of request or response body content, logging the completion of the request, and timer events.

As these interactions are common to all proxy servers, all available proxy software should be able to provide the hooks needed for implementing this API. A full implementation of this API is *http-complete*, in that it can support any behavior that can be implemented via HTTP requests and responses; this level of completeness is necessary to allow modules to perform content adaptation without limits. In addition to these hooks, the API also provides other mechanisms to control policy/behavioral aspects of the cache not covered within the scope of HTTP (e.g. prefetching content, request logging, server selection in round-robin DNS).

Modules register a set of function pointers for various events by providing a defined structure to the underlying proxy core. This structure is shown in Figure 2, and contains non-`NULL` entries for all of the callback functions of interest to the module. The module may specify `NULL` values for callback points for which it elects to receive no notification. Note that the events corresponding to the receipt of request or response body data may be triggered multiple times in an HTTP transaction (request-response pair), allowing the module to start working on the received data immediately without waiting or buffering.

The API tags each transaction with a unique identifier that is passed to each callback as the first argument. This allows the module to invoke multiple interactions with the proxy for a single HTTP transaction while recognizing which parts of the transaction have already taken place. Additionally, the API allows the module to pass back a special "opaque" value after completing each callback. This opaque value is not interpreted by the cache itself, but is instead passed directly to the next API callback function for this transaction. The opaque value typically contains a pointer to a data structure in the module specific to this transaction. The API invokes the `dfp_opaquefree` callback function when the transaction is complete, allowing the module to clean up the underlying structure as desired. The client IP address is also an input to each HTTP-related callback, allowing for different decisions depending on the source.

If a module is no longer interested in event notifications for a particular transaction, it may return a special response code recognized by the API. This allows modules to cease further effort on a transaction if, for example, request or response headers indicate that the module's service is not applicable. Once a module returns that response code, the API will not invoke any more callbacks for that transaction, but will invoke callbacks for other transactions.

2.2 Module Design

API modules are precompiled object files that are either dynamically linked into the proxy or are spawned in a separate address space. For security reasons, clients of the proxy cannot install modules into the proxy. Modules are trusted software components that must be installed by an administrator with the authority to configure the cache.

Modules export a set of standard entry points that are used by the proxy cache to invoke methods in the module in response to certain events affiliated with HTTP processing. The internal design of modules is not restricted; they can spawn other programs, invoke interpreted code, or call standard library and operating system APIs without impediment. The latter are particularly useful, for ex-

```
typedef struct DR_FuncPtrs {
  DR_InitFunc *dfp_init;              // on module load
  DR_ReconfigureFunc *dfp_reconfig;   // on configuration change
  DR_FiniFunc *dfp_fini;              // on module unload
  DR_ReqHeaderFunc *dfp_reqHeader;    // when request header is complete
  DR_ReqBodyFunc *dfp_reqBody;        // on every piece of request body
  DR_ReqOutFunc *dfp_reqOut;          // before request goes to remote server
  DR_DNSResolvFunc *dfp_dnsResolv;    // when DNS resolution needed
  DR_RespHeaderFunc *dfp_respHeader; // when response header is complete
  DR_RespBodyFunc *dfp_respBody;      // on each piece of response body
  DR_RespReturnFunc *dfp_respReturn; // when response returned to client
  DR_TransferLogFunc *dfp_logging;    // logging entry after request done
  DR_OpaqueFreeFunc *dfp_opaqueFree; // when each response completes
  DR_TimerFunc *dfp_timer;            // periodically called for maintenance
  int dfp_timerFreq;                  // timer frequency in seconds
} DR_FuncPtrs;
```

Figure 2: Structure provided by modules to register callback functions for specified events.

ample, for communicating with other systems via TCP/IP connections to provide services desired by the module.

Multiple modules can be active, and modules can be dynamically loaded and unloaded. Cascading multiple modules allows developers to combine services such as content filtering with image transcoding for a wireless business environment or site monitoring and content-preloading for a content delivery network. The ability to dynamically load and unload modules allows policies such as deactivating content filtering outside of normal work hours while still using image transcoding. The module programmer or deployer must specify the order of invocation for multiple modules so that data arrives as expected and interactions remain sensible.

2.3 Execution Models

The API supports modules executing in several formats, including processes, threads, and callbacks. The module sees the same interfaces in all cases, but the underlying implementations may differ. Since all of the models present the same interface, module developers are free to change the model used as the performance or flexibility needs of their modules change.

Processes – The most flexible model for an API module is to use a (Unix) process. In this manner, all module processing takes place in a separate address space from the proxy core, and the module is at liberty to use any operating system interfaces, run other programs, communicate across the network, perform disk operations, or undertake other slow or resource-intensive operations. A process may be single-threaded or multi-threaded. Using the process model enables the trivial use of multiprocessors and, with appropriately written modules, the ability to harness clusters of machines in a network. The flexibility of the process model implies more overhead in the operating system, including extra memory for storing the process state and more CPU overhead when the OS-supplied interprocess communication mechanisms are invoked to exchange information between the proxy core and the module. However, these communication costs are relatively minor for modules that perform significant processing.

Threads – Threads provide a higher-performance alternative to processes. In this model, the module spawns multiple threads in the proxy's address space. Each thread requires less overhead than a full process and the use of shared memory allows higher performance communication between the module and the proxy cache. Like the process model, threads can also trivially take advantage of multiprocessors. However, since the threads share the address space with the proxy cache, they must be careful not to corrupt memory or invoke system calls that affect the state of the proxy itself.

Callbacks – Callbacks are the lowest overhead mechanism for content adaptation, since the module is directly linked into the proxy cache's address space and invoked by the proxy cache state machine. As a result, callback overhead is comparable to a single function call. Since callbacks are performed synchronously in the proxy, the module's routines should not perform any blocking operation such as opening files, waiting on network operations, or synchronously loading data from disk. Nevertheless, callbacks may invoke nonblocking network socket operations and use polling functions provided by the API to determine when data is available for them. Modules that fit these criteria may use callbacks for the highest perfor-

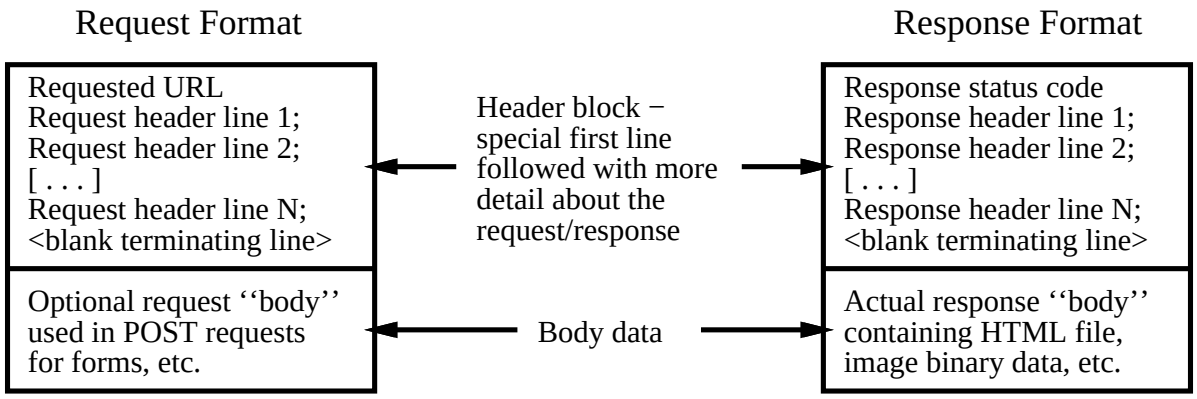


Figure 3: Structure of HTTP requests and responses

mance. Modules using callbacks should also be careful to avoid corrupting memory or performing stray pointer accesses, since corrupting memory can affect the running of the proxy cache.

3 API Functions

This section describes the functions provided by the API. This section does not aim to be a full manual, but rather describes the reasons for specific functions and the types of tasks supported by the API.

3.1 Content Adaptation Functions

The content adaptation functions allow modules to inspect and modify requests and replies as they pass through the proxy cache. Since any portion of the transfer can be inspected and modified by the modules, the API provides a powerful mechanism to use the proxy cache in a variety of applications. For example, the proxy cache can be used as a component of a Content Distribution Network (CDN) by developing a module that inspects the user's requested URL and rewrites it or redirects it based on geographic information. In mobile environments where end users may have lower-resolution displays, an API module could reduce image resolution to save transmission bandwidth and computation/display time on low-power devices. Such modular solutions would consist of far less code and better defined interactions than modifications to open-source code.

As explained in Section 2, the mechanisms for the API are closely integrated with the processing of requests and responses in the HTTP protocol. In the HTTP protocol, requests and responses have a well-defined structure, shown in Figure 3. Each consists of a header block with a special request/response line followed by a variable number of header lines, and then a variable-length data block.

The content adaptation interfaces allow modules to specify entry points that are called when the proxy handles the various portions of the requests and responses. Some of the entry points are called only when a particular portion is complete (e.g., when all of the response headers have been received), while others are called multiple times (e.g., as each piece of body data is received on a cache miss). For example, a module could provide routines that examine and modify the first line of each request and the full contents of each response that pass through the proxy.

The API includes header processing routines that allow searching for particular HTTP headers, adding new headers, deleting existing headers, and declaring that the current set of headers is completed and may be sent on. These routines provide a simple and flexible interface for the module to customize the HTTP-level behavior of the request, in isolation or in conjunction with transformation of the content body. This approach also insulates the modules from the details of the HTTP protocol, shifting the burden of providing infrastructure onto the proxy. Table 1 lists the routines provided for header and body manipulation.

The cache may also store modified content, allowing it to be served without requiring adaptation processing on future requests. This can be accomplished by registering interest in the arrival of the response header, and modifying the cache control header appropriately, causing the proxy cache to store the modified content if it would not be stored by default. When content adaptation is performed based on features of the client's request, the module can use the HTTP `Vary` header to indicate multiple variant responses cached for the same URL.

3.2 Content Management Functions

The API's content management features allow modules to perform finer-grained control over cache content than a proxy cache normally provides. These routines are ap-

Header Processing	DR_HeaderDelete DR_HeaderFind DR_HeaderAdd DR_HeaderDone	Modules can examine and modify the multiple request/response header lines as they arrive from the client/server, or as they are sent out from the proxy.
Body Modification	DR_RespBodyInject	Modules change the content body during notifications by changing return values, or at other times using this explicit call.

Table 1: Content adaptation functions

propriate for environments where the default content management behaviors of the proxy cache must be modified or augmented with information from other sources. The routines in this area fall into three broad categories: content freshness modification and eviction, content preloading, and content querying. The related functions are shown in Table 2.

Server accelerators and surrogates in content distribution networks benefit from finer-grained control over content validation to improve response time, reduce server load, or provide improved information freshness. In particular, surrogates normally achieve the maximum benefit when the origin servers set large expiration times to reduce the frequency of revalidation. However, the proxy may not see the most recent content if the underlying content changes during this time. When an external event (such as a breaking news item) occurs that voids the expiration information, the content management routines can update the proxies by invalidating cached content and fetching the newest information. The content management routines thus allow programmed, automatic control for exceptional situations while still using the regular mechanisms of the proxy cache under normal circumstances.

The content management routines also allow programmatic and dynamic preloading of objects into the cache, in addition to the content prepositioning support that already exists in many proxy caches. Preloading modules may be coupled to information sources outside of the proxy rather than just based on timer information. For example, a service provider running a network of caches could create a custom module to inject documents into caches as their popularity increases in other parts of the network.

Finally, the content management controls can also be used to provide "premium" services to portions of cached content. For example, the content management controls can be used to periodically query the cache and refresh the pages of premium customers, changing their cache eviction behavior.

Even though these functions change the material stored in the cache, the API introduces no security issues because its functions are invoked directly from trusted modules running on the same machine. However, the modules themselves may need to implement a security policy, especially if they take commands from external sources. The modules may use standard library functions to provide the mechanisms for this security; the API need not provide mechanisms since none of the encryption or authentication required is specific to the proxy cache environment.

3.3 Customized Administration

The API provides notifications that can be used to augment the administrative and monitoring interfaces typically available in proxy caches through Web-based tools, command-line monitoring, and SNMP. The event notification for logging allows the proxy to extract real-time information and manipulate it in a convenient manner in order to feed external consumers of this information. For example, content distribution networks and web hosting facilities may use real-time log monitoring to provide up-to-date information to customers about traffic patterns and popularity on hosted web sites, or to notice high rates of "page not found" errors and dynamically create pages to redirect users to the appropriate page. While all of this information can be obtained in off-line post-processing of access logs, the API allows deployers to implement custom modules that meet their specific needs for realtime analysis and action.

The administrative interfaces of the API can also be used to monitor networks of proxies, and to combine this information with other sources in Network Operation Centers (NOCs). For example, if a company is running a network of proxies and notices an unusually low request rate at one such proxy, this information can be combined with other information to help diagnose congested links or overloaded systems beyond the company's control. Since the proxy is functioning normally, one isolated measurement provides less reliable information than programmatic and comparative measurement of different systems.

3.4 Utility Functions

Table 3 describes utility functions provided by the API. The DR_FDPoll family of functions allow callback-

Naming & Identification	`DR_ObjIDMake`
	`DR_ObjIDRef`
	`DR_ObjIDUnref`
		Modules use internal identifiers for content management functions rather than requiring full URLs in all calls. These functions are used to create/destroy the identifiers

Existence & Lifetime Handling	`DR_ObjQuery`
	`DR_ObjFetch`
	`DR_ObjValidate`
	`DR_ObjExpirationSet`
	`DR_ObjDelete`
		For querying the existence/property of objects and loading, refreshing, and deleting them as needed

Cache Injection	`DR_ObjInject`
	`DR_ObjInjectStart`
	`DR_ObjInjectBody`
	`DR_ObjInjectDone`
		Objects can be manually added into the proxy without having the proxy initiate a fetch from a remote server. The object can be injected all at once, or in pieces.

Data Reading	`DR_ObjRead`
	`DR_ObjReadStart`
	`DR_ObjReadPart`
	`DR_ObjReadDone`
		Objects in the cache can be loaded into memory allocated by the module without being requested by the client. The object can be read all at once or in pieces.

Table 2: Content management functions

External Communication	`DR_FDPollReadNotify`
	`DR_FDPollReadClear`
	`DR_FDPollWriteNotify`
	`DR_FDPollWriteClear`
	`DR_FDPollClose`
		Modules may communicate externally using sockets, even when using the callback mechanism. In this case, they use asynchronous operations and use these functions to have the main proxy notify them of events on their sockets.

Custom Logging	`DR_StringFromStatus`
	`DR_StringFromLogCode`
	`DR_StringFromLogTimeout`
	`DR_StringFromLogHier`
		These functions provide the standard text strings used for the extended log format.

Configuration	`DR_ConfigOptionFind`	User modules invoke this function to extract information from the cache configuration file.

Table 3: Utility functions

based modules to use nonblocking socket operations. These functions allow external communication without blocking and without requiring modules to implement their own event management infrastructure; instead, such modules can simply utilize the underlying event mechanisms of the OS and host proxy. In particular, the `ReadNotify` and `WriteNotify` calls register functions that should be invoked when an event (either data available to read or space available to be written, respectively) takes place on a given file descriptor. The `ReadClear` and `WriteClear` functions indicate that the previously registered function should no longer be called, and the `Close` function indicates that the file descriptor in question has been closed and all currently registered notifications should be deregistered. All of these functions can be implemented using `poll`, `select`, `kevent`, `/dev/poll`, or other OS-specific mechanisms; their only requirement is that the underlying OS support the notion of file descriptors and some form of event notification, which are common to all standard systems.

The custom logging functions provide canonical names for the codes passed by the proxy to module functions for log notifications. `DR_ConfigOptionFind` allows the user module to extract desired information from the cache configuration files.

4 Sample Modules

To experiment with the API, both from a functional aspect as well as from a performance perspective, we developed some modules that use various aspects of the interface, including a module that implements ICAP. We were pleased with the simplicity of module development and the compactness of the code necessary to implement various features. Initial development and testing of each module required from a few hours to a few days. More detail about each module's behavior and implementation is pro-

Module Name	Total Lines	Code Lines	Semicolons	# API call sites
Ad Remover	175	115	51	4
Dynamic Compressor	387	280	126	11
Image Transcoder (+ helper)	391 + 166	309 + 118	148 + 54	10
Text Injector (+ helper)	473 + 56	367 + 32	170 + 8	12
Content Manager	675	556	289	56
ICAP client	1024	719	321	15

Table 4: Code required to implement sample services

vided below. Table 4 summarizes information about the code size needed for each module. Since the modules are freed from the task of implementing basic HTTP mechanisms, none of them are particularly large. The "Total Lines" count includes headers and comments, the "Code Lines" count removes all blank lines and comments, and the "Semicolons" count gives a better feeling for the number of actual C statements involved. All modules use the callback interface, with some spawning separate helper processes under their control.

Ad Remover – Ad images are modified by dynamically rewriting their URLs and leaving the original HTML unmodified. On each client request, the module uses a callback to compare the URL to a known list of ad server URL patterns. Matching URLs are rewritten to point to a cacheable blank image, leading to cache hits in the proxy for all replaced ads. To account for both explicitly-addressed and transparent proxies, the module constructs the full URL from the first line of the request and the `Host` header line of the request header. On modified requests, the `Host` header must be rewritten as well, utilizing the `DR_Header*` functions. Other uses for this module could include replacing original ads with preferred ads.

Dynamic Compressor – This module invokes the `zlib` library from callbacks to compress data from origin servers and then caches the compressed version. Clients use less bandwidth and the proxy avoids compressing every request by serving the modified content on future cache hits. This module checks the request method and `Accept-Encoding` header to ensure that only `GET` requests from browsers that accept compressed content are considered. The response header is used to ensure that only full responses (status code 200) of potentially compressible types (non-images) are compressed. The header is also checked to ensure that the response is not already being served in compressed form and is not an object with multiple variants (since one of those variants may already be in compressed form). Using the `DR_Header*` functions, the outbound response must be modified to remove the original `Content-length` header and to insert a `Vary` header to indicate that multiple versions of the object may now exist.

Image Transcoder – All JPEG and GIF images are converted to grayscale using the `netpbm` and `ijpeg` packages. Since this task may be time-consuming, it is performed in a separate helper process. The module buffers the image until it is fully received, at which point it sends the data to the helper for transcoding. The helper returns the transcoded image, or the original data if transcoding fails. The module kills and restarts the helper if the transcoding library fails, and also limits the number of images waiting for transcoding if the helper can not satisfy the incoming rate of images. The module uses the `DR_FDPoll*` functions to communicate with the helpers, the `DR_Header*` functions to modify the response, and the `DR_RespBodyInject` function to inject content into an active connection.

Text Injector – The main module scans the response to find the end of the HTML `head` tag, and then calls out to a helper process to determine what text should be inserted into the page. The helper process currently only responds with a text line containing the client IP address, but since it operates asynchronously, it could conceivably produce targeted information that takes longer to generate. The module passes data back to the client as it scans the HTML, so very little delay is introduced. For reasons similar to the case of the Image Transcoder module, the `DR_FDPoll*`, `DR_Header*`, and `DR_RespBodyInject` functions are all invoked.

Content Manager – This demonstration module accepts local `telnet` connections on the machine and presents an interface to the `DR_Obj*` content management functions. The administrator can query URLs, force remote fetches, revalidate objects, and delete objects. Object contents can also be displayed, and dummy object data can be forced into the cache. The module uses the `DR_FDPoll*` family of functions to perform all processing in callback style even while waiting on data from network connections.

ICAP Client – This module implements the ICAP 1.0 draft for interaction with external servers that provide

value-added services [8]. The module must encapsulate HTTP requests and responses in ICAP requests, send those requests to ICAP servers, retrieve and parse responses, and send forth either the original HTTP message or the modified message provided by the ICAP server. All of this processing can be implemented through callbacks with the assistance of the polling functions for network event notification. Implementing ICAP as a module rather than an integrated part of the proxy core is particularly appropriate as ICAP specifications continue to evolve.

5 Comparison with ICAP

This section discusses key points of comparison between our API and the Internet Content Adaptation Protocol (ICAP) draft specification [8]. We do not seek to undertake a detailed quantitative performance comparison since the ICAP standard and implementations are still evolving. We instead focus on differences in functionality and mechanism.

Functionality. ICAP provides services for content-adaptation, while the API additionally provides services for content-management and customized administration. As a result, ICAP provides a useful component for content delivery, but cannot enable the more detailed management infrastructure for a content delivery network without changes to the underlying cache architecture. In contrast, the API provides services to push, query, invalidate, or modify data while it is in the cache, and also provides features for real-time monitoring and integration with statistics. Some of these features may even inform content-adaptation; for example, real-time monitoring can potentially provide additional information beyond cookies alone (e.g., frequency of connections from a client IP address or authenticated user) that may lead to different transcoding behavior or object freshness policy.

Additionally, the polling functions of Section 3.4 encapsulate the low-level details of concurrency management. These polling functions enable the API modules to efficiently use the underlying OS and cache features for socket I/O to a variety of network services (including external ICAP servers) while avoiding the programming difficulty of implementing such an event notification state machine directly.

Mechanism. The primary differences in content adaptation mechanism between ICAP and the API stem from the communication methods used. ICAP invokes all communication by having the cache initiate contact with the service through a TCP/IP socket. In contrast, the API allows the cache to directly invoke functions registered to provide a service. While current ICAP implementations locate the value-added services on a separate server, the API allows for the use of either a separate server or a cache-integrated module. The latter is particularly valuable as processor speeds continue to accelerate faster than all other parts of the system, enabling substantial additional services beyond caching without saturating the CPU. The API is also sufficiently flexible to implement ICAP as a module rather than part of the proxy core.

ICAP allows servers to statically inform proxies that HTTP data for certain file extensions should not be passed to them, that others should be sent for *previewing*, and that others should be always sent in their entirety. For those HTTP requests and responses that must be previewed, an ICAP-enabled proxy constructs a preview message consisting of ICAP-specific headers followed by the complete HTTP headers and some arbitrary amount of the HTTP body. The ICAP server then indicates whether or not the proxy should continue sending body data for modification. If so, or if the file extension indicates that this request should always be sent rather than previewed, the proxy must send the entire set of HTTP headers and body. The server will then respond either with an indication that no modifications will take place or with a complete set of modified HTTP headers and body. The primary goal of previewing is to allow the service to act upon a message by reading the HTTP headers, but ICAP requires the proxy to construct ICAP headers and encapsulate the HTTP headers on a preview, after which it must parse a response from the ICAP server. In contrast, the API allows for more direct header examination with `DR_HeaderFind`, requiring no higher-level ICAP wrapper headers.

Services should also have an easy mechanism to decide that they have no further interest in an HTTP message. ICAP provides no mechanism to continue past the preview and then stop adaptation before seeing the full body. The API allows the service to dynamically turn off interest in further callbacks for a transaction at any point in the headers or body. This difference could affect the text injector module of Section 4, since the text injection process might finish at any arbitrary point in the body.

In short, while ICAP can provide a variety of useful content-adaptation features, the API presented here exposes an interface that provides a superset of these functions while also enabling low-overhead coordination with service modules (including ICAP itself).

6 Implementation and Performance

In this section, we describe the implementation of the API in the iMimic DataReactor proxy cache and conduct a series of experiments to understand various performance scenarios related to the API. We measure the impact of adding API support into the DataReactor, the various costs of API features, and the performance of some

of the sample modules that use the API.

6.1 The iMimic DataReactor Proxy Cache

Evaluating the impact of the API is heavily dependent on the quality of the underlying platform. A slow proxy will mask the overhead of the API, while a fast one will more easily expose the additional latency resulting from the API. The iMimic DataReactor Proxy Cache is a commercial high-performance proxy cache. It supports the standard caching modes (forward, reverse, transparent) and is portable, with versions for the x86, Alpha, and Sparc architectures and the FreeBSD, Linux, and Solaris operating systems. It has performed well in the vendor-neutral Proxy Cache-Offs, setting records in performance, latency, and price/performance [15, 16, 17].

We test forward-proxy (client-side) cache performance using the Web Polygraph benchmark program running a modified version of the Polymix-3 workload [16]. We use this test and workload because it has the highest number of independently-measured entries of any web proxy benchmark, and it heavily stresses proxy server performance. For the sake of time, we shorten our performance tests to use a 2 hour load plateau instead of four hours, and fill the disks only once before all tests rather than before each test. These changes shorten the load phase of the Polygraph test to roughly 6 hours instead of 10.5, and avoiding a separate fill phase reduces the length of each test by an additional 10-14 hours. The primary performance impact is a 3–4% higher hit ratio than an official test because of a smaller working set and data set. We call this modified test PMix-3.

Polygraph stresses various aspects of proxy performance, particularly in network and disk-related areas. It uses per-connection and per-packet delays between the proxy and simulated remote servers to cause cache misses to have a high response time. Likewise, it generates data sets and working sets that far exceed physical memory, causing heavy disk access. Polygraph stresses connection management by scaling the number of simulated clients and servers with the request rate.

The test system runs FreeBSD 4.4 and includes a 1400 MHz Athlon, 2 GB of memory, a Netgear GA-620 Gigabit Ethernet NIC, and five 36 GByte 10000 RPM SCSI disks. All tests use a target request rate of 1450 $\frac{\text{requests}}{\text{second}}$. This throughput compares favorably with other high-end commercial proxy servers and is over a factor of ten higher than what free software has demonstrated [16]. At this rate, the proxy is managing over 16000 simultaneous connections and 3600 client IP addresses. Given the fixed request rate, this test demonstrates any latency differences in the various test scenarios. (Polygraph also shows some run-to-run randomness in the offered workload, leading to additional minor variations.)

6.2 API Implementation Overhead

To understand the performance overhead of implementing the API in the DataReactor, we start with a standard DataReactor platform, incrementally add features, and test the result. Overheads from implementing the API result in increased hit and miss response times, since throughput is kept constant. Table 5 lists the results for these tests.

The various columns of Table 5 are as follows: "Baseline" is the standard DataReactor software without API support. "API-Enabled" is the same software with API support, but without any modules loaded. "Empty Callback" adds a module with all notifications specified, but with no work done in any of them. "Add Headers" adds extra headers to all inbound/outbound paths on the proxy, so four extra headers will be introduced on each transaction. "Body + Headers" additionally copies the response body of each reply and overwrites the response body with this copy.

The "API-Enabled" test shows that implementing the API adds virtually no overhead on cache hits and only a small overhead on cache misses. Actually installing a module causes a slight slowdown on hits and misses due to the extra calls needed. Due to the extremely small hit times, this effect appears as a 5% increase on hit time. On cache misses, where most of the time is spent waiting on the remote server, the overhead is less than one-tenth of one percent. These low overheads confirm the premise that an explicitly event-aware API design can enable an extensible proxy with minimal performance impact.

We also observe that using the features of the API, such as adding headers or modifying the body, generates low overhead. Adding headers introduces some extra delay on misses, but even modifying the full body does not generate any significant spike in response times. The hit times for "Body + Headers" show a 6% increase over the "Empty Callback", which translates into a cumulative 11.5% increase versus the baseline. However, in absolute terms, the increase is less than 2.5ms, or less than 1% of the overall response time.

6.3 Performance Methodology

We construct several tests to assess the performance of some of our content-adaptation modules, both on their own and in terms of their impact on the overall system. However, we cannot rely solely on Pmix-3 to generate the load, since this workload does not generate realistic content for the objects in its test. Without realistic content, measuring the performance of some of our content adaptation modules would be meaningless. For example, the Image Transcoder module would fail to perform any transcoding, and would return the images unmodified.

	Baseline	API Enabled	Empty Callback	Add Headers	Body + Headers
Throughput (reqs/sec)	1452.87	1452.75	1452.89	1452.62	1452.84
Response time (ms)	1248.99	1248.95	1251.25	1251.98	1250.14
Miss time (ms)	2742.53	2743.18	2744.33	2745.07	2746.98
Hit time (ms)	19.82	19.86	20.87	20.85	22.10
Hit ratio (%)	57.81	57.81	57.76	57.74	57.85

Table 5: Performance tests to determine overhead of implementing API

	Baseline	Ad Remover	Images 25 Trans/s	Images Max Trans	Max Trans nice 19	Compress 75 obj/s	Compress 95 obj/s
Throughput (reqs/s)	1452.87	1452.72	1452.65	1452.73	1452.68	1452.73	1452.88
Response time (ms)	1248.99	1248.87	1256.60	1277.76	1250.69	1252.24	1258.34
Miss time (ms)	2742.53	2743.55	2753.47	2778.09	2744.60	2745.63	2752.63
Hit time (ms)	19.82	20.42	23.21	43.30	20.15	23.44	28.69
Hit ratio (%)	57.81	57.81	57.74	57.80	57.78	57.81	57.78

Table 6: Background Pmix-3 benchmark performance when run simultaneously with content adaptation modules

Since transcoding is a more CPU-intensive process than rejecting non-image objects, the real performance impact of the transcoder could not be measured.

For the image transcoding and dynamic compression tests, we extend the Polygraph simulation testbed with a non-Polygraph client and server to generate requests and serve real objects as responses. The new server also generates only non-cacheable responses so that the modules must be invoked on each response. The content adaptation modules identify responses from the "real" server and only consider those responses as candidates for transcoding. While this approach generates some extra load on the module versus screening out all Polygraph client requests early, we feel our approach will yield more conservative performance numbers. We also continue running a Pmix-3 test against the same proxy at the same time, and keep the Pmix-3 request rate the same as in the earlier tests for an accurate comparison.

6.4 Module Performance

Table 6 shows the performance effects of the various content adaptation modules. The "Baseline" column shows our baseline performance with no API support. The "Ad Remover" column shows the performance of the Ad Remover module examining Polygraph traffic. The next three columns show proxy performance when the image transcoder is running in different scenarios. The final columns show the Dynamic Compressor serving a certain rate of compressed objects.

The Ad Remover tests show virtually no degradation in performance. This result is not surprising, because most of this module's work consists of inspecting request headers, which is computationally cheap. This module only rewrites headers on matching URLs, and this workload does not have any URL matches.

The Image Transcoder tests show how this module can affect the overall performance of the proxy, but also how a simple change can eliminate almost all of its negative impact. Since all transcoding is performed in a helper process, we show several scenarios for this module to gain a better understanding of how it behaves. On an idle machine, the transcoder can process JPEGs of size 8 KBytes at a rate of roughly 110 per second. During the load plateau of Pmix-3, most of the CPU is utilized serving regular traffic, and less time is available to the transcoder. At this point, if we run the transcode client in infinite-demand mode, we achieve an average of 30 transcodes/sec, with a range of 20-38. When this occurs, the proxy CPU has no idle time. Transcoding at 25 reqs/sec shows an 11ms increase in miss time and a 3ms increase in hit time. When the client runs in infinite-demand mode, miss times increase by 36ms while hit times rise by 23ms.

The transcoder's negative side effects on Pmix-3 traffic suggest that the proxy and helper are competing for the CPU. This competition can be almost completely eliminated by changing the process scheduling priority (the "nice" value) of the helper to 19, giving it the lowest priority of the system. With this change, the helper runs only when the CPU is idle. For the infinite-demand workload, queues between the proxy and the helper process never become overly long since further requests are delayed until earlier responses complete. As a result, the transcoder processes all requests made to it and the system is work-

conserving. Since the system is work-conserving and the CPU has idle time available, the priority change for the helper process only affects the scheduling of the helper but does not otherwise affect its throughput. With this simple change, the Pmix-3 performance numbers return to values only slightly worse than the base proxy.

On an idle machine, the dynamic compressor module can satisfy approximately 400 compressions per second with the input data as an 8 KByte text file of C source code. When run in combination with Pmix-3, the dynamic compressor is shown with two different workloads: compressing 75 objects per second and 95 objects per second. The system supports the lighter compression workload with very little impact on the hit or miss response time of the background Pmix-3 traffic. The heavier compression workload leads to about 10 ms increase in both miss and hit time relative to the baseline performance; however, even this still leads to less than 1% degradation of mean response time. No substantially higher rate is possible because the CPU is saturated when the Pmix-3 load plateau occurs simultaneously with 95 compressions per second.

These performance results show that the API can enable content-adaptation services to consume spare CPU cycles on the proxy cache without interfering substantially with the performance observed by transactions for unmodified content.

7 Related Work

Sections 1 and 5 discuss the Internet Content Adaptation Protocol (ICAP) and compare it with the API presented here. This section discusses other related work.

The event-aware nature of our API is clearly motivated by previous research on event-driven servers [3, 19], particularly by work showing their scalability benefits versus traditional multi-threaded or multi-process servers [13]. Through the use of dynamic loading of modules coupled with an event-driven proxy core, our implementation achieves performance comparable to adding existing states into a event-driven server.

The TransSend/TACC proxy [6] performs content adaptation using a system akin to Unix pipes, where thread-based modules receive a stream of bytes from the main proxy. In comparing the relevant section of that work, we find differences in architecture and coverage. By exposing an event-aware API, modules can choose to avoid the overhead of threads or processes, yielding higher scalability. In terms of coverage, since our API is specifically designed for caching proxy servers, it contains content management and utility functions not present in other APIs.

Commercial proxy caches by Inktomi and Novell have previously announced APIs. No public documentation of functionality or performance is available for the Inktomi API. The Novell Filter Framework provides a content adaptation system for Novell Border Manager and Volera Excelerator [12]. Filter modules are supported using only a callback model. Additionally, the system appears to be tightly integrated with the operating system kernel because standard libraries for memory management such as `malloc` are not available; instead, all memory allocation and management must take place through kernel-style memory chains. Filter Framework was never fully implemented, and has now been discontinued.

Many academic studies and commercial products have been based on modifying the source code of the Harvest cache and its successors such as Squid [3, 18]. However, if these source code changes are not integrated into the public releases of the proxy, the groups maintaining the modified proxy must track the public releases to incorporate bug fixes, performance improvements, and new features. In contrast, changes to an API-enabled proxy server only affect modules if the API specification changes.

Research in content adaptation has often shown the difficulty in modifying proxy behavior. For example, Chi et al. describe a proxy server that modifies Squid to compress incoming data objects, but keeps the original content-length header intact [4]. That work tests the proxy with a modified client that ignores the content-length header. In an API-based solution, deleting or changing headers is a simple task since the API provides the needed infrastructure.

The ad insertion proxy developed by Gupta and Baehr uses special header lines that provide information about what parts of an HTML document are ads that can be replaced by the proxy in cooperation with the origin server [7]. Their non-caching proxy was developed specifically for this purpose. The same system could be developed with an API-enabled proxy with much less effort, as the ad replacement module could use the same special headers to communicate with cooperating servers without modifying the infrastructure for managing other HTTP headers.

Various researchers have examined the issue of content management, often to address the limitations of the HTTP protocol's handling of object expiration/staleness or to take advantage of regional proxies. The PoliSquid server develops a domain-specific language to allow customization of object expiration behavior [1]. The Adaptive Web Caching project uses proxies in overlapped multicast groups to push content and perform other optimizations related to object placement [5]. Likewise, the approach proposed by Rabinovich et al. uses routing/distance information to determine when proxies should contact neighbors versus when they should request objects directly [14]. In all of these cases, a content-management

API would reduce development work of the customizers and would allow them to focus on their policies and improvements rather than the underlying mechanism.

Researchers and companies have also examined mechanisms for extending proxy server functionality using Java. The Active Cache project associates with each cacheable document a Java applet that is invoked when a proxy accesses the document [2]. Likewise, the JProxyma proxy uses Java plug-ins for performing content adaptation [9]. We believe that the API we propose can enable either approach; in particular, the use of helper processes in sample modules such as the transcoder shows that extended services can effectively launch external programs for their API interactions.

Component-based software architectures are rapidly gaining popularity in various domains of the computer industry. For instance, the applications in office productivity suites, such as Microsoft Office or the public-domain Koffice, all follow the component-based paradigm, exporting a set of APIs to other applications [10, 11]. The reason for this growing popularity is identical to the one that caused us to develop an API for proxy caches: providing the ability to control an application without having to modify it.

8 Conclusions

The need for Web proxy caches to provide a wide-ranging and growing set of services motivates systems that allow development of customization modules while shielding developers from the details of the underlying cache. This paper describes an Application Programming Interface (API) for proxy caches that allows them to become programmable components in sophisticated content delivery infrastructures. Using this API, functionality can be added to the proxy cache after it is deployed, without needing third-party source code modifications that may be difficult to maintain or needing external servers to run these services even if the cache CPU is not fully utilized. The API supports content adaptation, content management, and customized administration, and can be used to implement the ICAP Internet draft. We have demonstrated the power and ease of use of the API with some examples that show interesting tasks being performed with small amounts of code.

Although the API is independent of any particular proxy cache core, we have discussed the implementation and performance of the API in the iMimic DataReactor proxy cache. Because the API design follows the event-driven structure of a typical proxy, implementing the API has no significant effect on the performance of the proxy cache. Further, our results show two realistic content-adaptation services achieving high performance levels without substantially hindering a background benchmark load run at a high throughput level. This allows such services to consume free CPU cycles on the cache system itself, an increasingly available commodity as processor speeds accelerate faster than other portions of the system and as small-scale multiprocessors become more common.

References

[1] J. Barnes and R. Pandey. Providing Dynamic and Customizable Caching Policies. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS 99)*, Oct. 1999.

[2] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 373–388, 1998.

[3] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[4] C. Chi, J. Deng, and Y. Lim. Compression proxy server: Design and implementation. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS 99)*, Oct. 1999.

[5] S. Floyd, V. Jacobson, and L. Zhang. Adaptive web caching. In *Proceedings of the Second International Web Caching Workshop (WCW '97)*, 1997.

[6] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[7] A. Gupta and G. Baehr. Ad insertion at proxies to improve cache hit rates. In *Proceedings of the 4th International Web Caching Workshop*, Apr. 1999.

[8] ICAP Protocol Group. ICAP: the Internet Content Adaptation Protocol. Internet draft, June 2001.

[9] Intellectronix LLC. Jproxyma. http://www.intellectronix.com/jpro/aboutjpro.htm.

[10] KOffice Project. Koffice. http://www.koffice.org/.

[11] Microsoft Corporation. Microsoft Office. http://www.microsoft.com/office/.

[12] Novell Developer Kit. *Filter Framework*, Jan. 2001.

[13] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference*, pages 199–212, June 1999.

[14] M. Rabinovich, J. Chase, and S. Gadde. Not all hits are created equal: Cooperative proxy caching over a wide area network. In *Proceedings of the Third International Web Caching Workshop (WCW '98)*, June 1998.

[15] A. Rousskov, M. Weaver, and D. Wessels. The fourth cache-off. Raw data and independent analysis at http://www.measurement-factory.com/results/, Dec. 2001.

[16] A. Rousskov and D. Wessels. The third cache-off. Raw data and independent analysis at http://www.measurement-factory.com/results/, Oct. 2000.

[17] A. Rousskov, D. Wessels, and G. Chisholm. The second IRCache web cache-off. Raw data and independent analysis at http://cacheoff.ircache.net/, Feb. 2000.

[18] D. Wessels et al. The Squid Web Proxy Cache. http://www.squid-cache.org.

[19] Zeus Technology Limited. Zeus Web Server. http://www.zeus.co.uk.