Dynamic Adaptation in the Web Server Design Space using OpenWebServer

Junichi Suzuki
Department of Computer Science,
Graduate School of Science and Technology,
Keio University
Yokohama City, 223-8522, Japan
+81-45-563-3925
suzuki@yy.cs.keio.ac.jp

Yoshikazu Yamamoto
Department of Computer Science,
Graduate School of Science and Technology,
Keio University
Yokohama City, 223-8522, Japan
+81-45-563-3925
yama@cs.keio.ac.jp

Abstract

The explosive growth of the Web requires servers to be extensible and configurable. This paper describes our adaptive web server, OpenWebServer that employs a meta-architecture. It supports dynamic adaptation of feasible design decisions in the web server design space by specifying and coordinating metaobjects that represent various aspects within the web server. We present some examples of system adaptation that change and tune configuration of concurrency, caching, logging, load balancing and fault tolerance. OpenWebServer can evolve continually beyond static and monolithic servers.

Keywords

Adaptive web server, Meta-architecture, Reflection, System adaptation, System evolution

1. Introduction

The explosive growth of the Web places larger and more challenging demands on servers. Its computing power and network bandwidth has increased dramatically. An effective design for web servers is highly required.

Current web servers must:

- Connect with various systems such as groupware, database management systems, mobile agent engines and transaction processing monitors.
- Integrate generic communication environments including CORBA (Common Object Request Broker Architecture) and DCOM (Distributed Component Object Model).
- Extend server functionality by introducing additional network protocols or content data types.
- Change server execution policies, e.g., optimize concurrency, connection management, request handling and cache management.
- Adapt to the execution environment, e.g. ATM networks, I/O subsystems such as RAID, and electronic devices such as network routers, printers or copiers.

Every user may not require the same functionality in a web server. Therefore, a web server should be flexible enough to meet a wide range of requirements on demand. Most current web servers, unfortunately, are monolithic. They provide a fixed and limited set of capabilities. It is typical to take the “scrap-and-build” approach for a given requirement, where the software is rewritten from scratch because it may be more economically feasible. A dynamically adaptable web server architecture based on reusable components is an attractive alternative for extensive and intrusive changes.

In this context, most web servers lack the adaptability to enable the system’s evolution. Designers cannot know or predict all possible uses of the system. A service or configuration that is appropriate at one point may not be useful later, and the system may not evolve transparently.

Our research vehicle for exploring the adaptability of the web server is OpenWebServer [1]. OpenWebServer is an adaptive web server based on the Adaptive Internet Server Framework (AISF) [2], an object-oriented framework that employs a reflective meta-architecture [3-5]. We consider the use of *Reflection* for specifying various aspects, e.g. structure and/or behavior, of an open-ended system that can be dynamically adapted. OpenWebServer contains metalevel(s) that specifies a wide range of aspects of web servers using fine-grained metaobjects. It is implemented within the programmable metalevel and can dynamically adjust itself so that it is executed in the best-tuned condition for a given requirement.

Though modern web servers provide some extension mechanisms like CGI (Common Gateway Interface) and server-side APIs, their extensibility is restricted within the

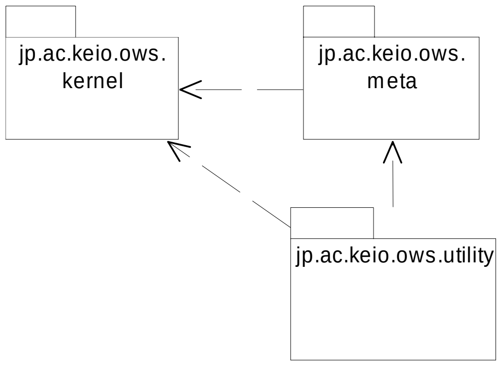

Figure 1: Dependencies between foundation packages in OpenWebServer

application level. In contrast, OpenWebServer provides a uniform platform where a variety of requirements can be specified from low-level services like the connection management, request handling and cache management into application-level services without breaking the single framework. In other words, the metalevel in OpenWebServer plays the role of a generic "change absorber" for the web server.

OpenWebServer is named after the *Open-Closed Principle* (OCP) [6], which states that software entities should be open for extension but closed for internal modification. This means we should design systems so that they can be changed by adding new code and not by changing working code. Thus, OCP encourages objects to be extensible by adding new objects via inheritance or composition not by modifying the object's internal code. OpenWebServer is intended to apply this principle in that the system can evolve by adding new metaobjects or coordinating them.

This paper addresses how OpenWebServer meets diverse requirements in the web server design space, and describes its adaptation capability to evolve continually beyond static and monolithic servers.

The remainder of this paper is organized as follows: Section 2 describes the metalevel design of OpenWebServer. Section 3 presents some examples of system adaptation using OpenWebServer. In Section 4 and 5, we conclude with a note on the current status of the project and present future work. Note that it is not possible to describe basics, history and benefits of reflection due to space limitation. They are mentioned in [1, 2] at large.

2. Metalevel design of OpenWebServer

This section describes the metalevel design of OpenWebServer. We present its Java version using software patterns. A pattern represents a recurring solution to a software development problem within a particular context [5, 7]. Patterns identify the static and dynamic collaborations and interactions between software

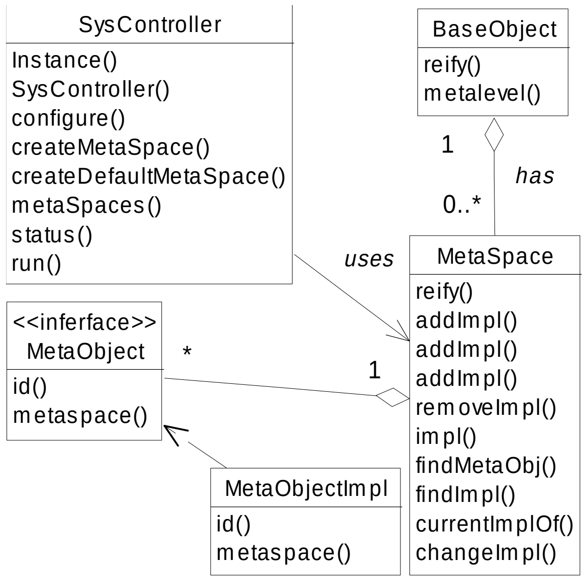

Figure 2: Classes in the `jp.ac.keio.ows.kernel` package.

components. In general, applying patterns to complex object-oriented concurrent applications can significantly improve software quality, increase software maintainability and support broad reuse of components and architectural designs [5].

OpenWebServer consists of three packages as depicted in Figure 1. The `jp.ac.keio.ows.kernel` package contains the foundation objects to construct and maintain the metalevel. The `jp.ac.keio.ows.meta` package contains a series of metaobjects, and is controlled by the `kernel` package. The `jp.ac.keio.ows.utility` package contains utility objects to support the writing of both the baselevel and metalevel. It is used by `kernel` and `meta` packages.

The `jp.ac.keio.ows.kernel` package is organized as shown in Figure 2, and contains the following objects:

- `SysController`

  Starts the system by creating appropriate metaspaces and metaobjects with a configuration file. This object is also responsible for stopping and resuming the system. It is an active object executed on a root thread in the thread hierarchy.

- `MetaSpace`

  Represents a metalevel, or metaspace. Multiple metalevels can exist within the system though it usually has a single metalevel. It references every metaobject and coordinates the interaction between them to meet a given requirement.

- `MetaObject`

Specifies the interfaces for all metaobjects. This is a base interface class for them. `MetaObjectImpl` and its subclasses provide the implementations of this object. The relationship between `MetaObject` and `MetaObjectImpl` is discussed in Section 4.2.

- `MetaObjectImpl`

  Provides an implementation for a `MetaObject`. Multiple implementations can be defined for a single `MetaObject`.

- `BaseObject`

  Specifies the interfaces needed to all baseobjects. This is a base class for them.

`SysController` is a *Singleton* object [7], since it has exactly one instance in the system. The *Singleton* pattern ensures a class has just one instance and provides controlled access to it [7]. The static `Instance()` method is used to instantiate or return the unique instance. The constructor is protected and called by `Instance()`. `SysController` creates one or more instances of `MetaSpace` on different threads using the method `createDefaultMetaSpace()` or `createMetaSpace()`. It holds a set of references to `MetaSpace`s.

`MetaSpace` represents each metalevel in OpenWebServer. It is an entry point from the baselevel to metalevel. Baseobjects can access `MetaSpace` when communicating with their metalevels. Every baseobject is derived from `BaseObject`, and every metaobject is derived from `MetaObject` (Figure 3).

Baseobjects do not have to be attached to a metalevel, but can be dynamically attached on demand. Attachment on demand is known as lazy reification. Each baseobject can be reified with its method `reify()`, and access its metalevel with the method `metalevel()`.Once a baseobject is reified, it becomes aware of its metalevel, and the corresponding metaobjects are then instantiated by the class `MetaSpace`. The number and type of created metaobjects depends on what the users wish to do.

The `jp.ac.keio.ows.meta` package consists of a collection of metaobjects. To specify metaobjects, we identified services and entities by looking for typical events or constructs found during the execution of web servers. Abstracting these events and constructs, OpenWebServer provides the following metaobjects by default:

- `Initializer`

  Initializes the network facility along with the current configuration. A typical task is to create one or more sockets according to the current communication protocol and concurrency policy and then instantiate an `Acceptor`.

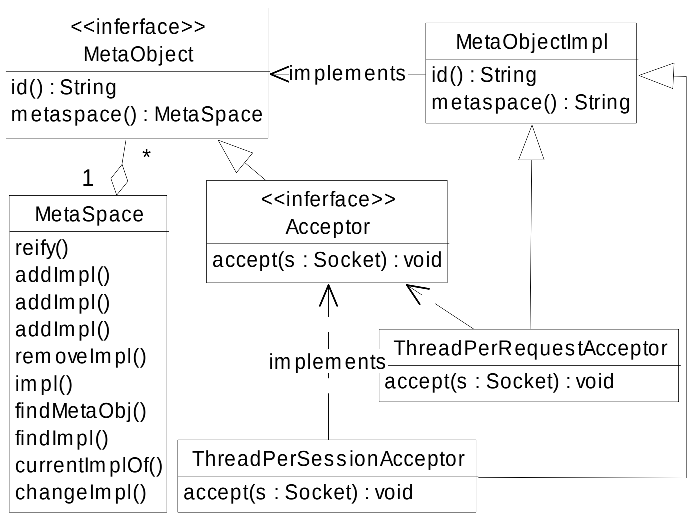

Figure 3: Classes in the `jp.ac.keio.ows.meta` package.

- `Acceptor`

  Waits for and accepts incoming requests. It encapsulates the different concurrency policies for simultaneous access. Once it obtains a request, it asks a `RequestHandler` to process the request given the current configuration, i.e., protocol and concurrency policy.

- `RequestHandler`

  Deals with requests from an `Acceptor`. It encapsulates the different policies for interpreting a request, based on the kind of request and target content. It is created on a per-request basis when an `Acceptor` accepts a request. It is executed on a separated thread or same thread that an `Acceptor` runs on.

- `Protocol`

  Defines protocol specific information on a per-protocol basis. It is referenced by a `RequestHandler`.

- `ContentFinder`

  Finds and obtains a target resource (e.g. HTML/XML files or data within a back-end repository) passed by a `RequestHandler`.

- `Cache`

  Caches target resources that have been accessed and retrieves them quickly. A `ContentFinder` uses it.

- `Logger`

  Records accesses to the web server. A `RequestHandler` calls it.

- `Redirector`

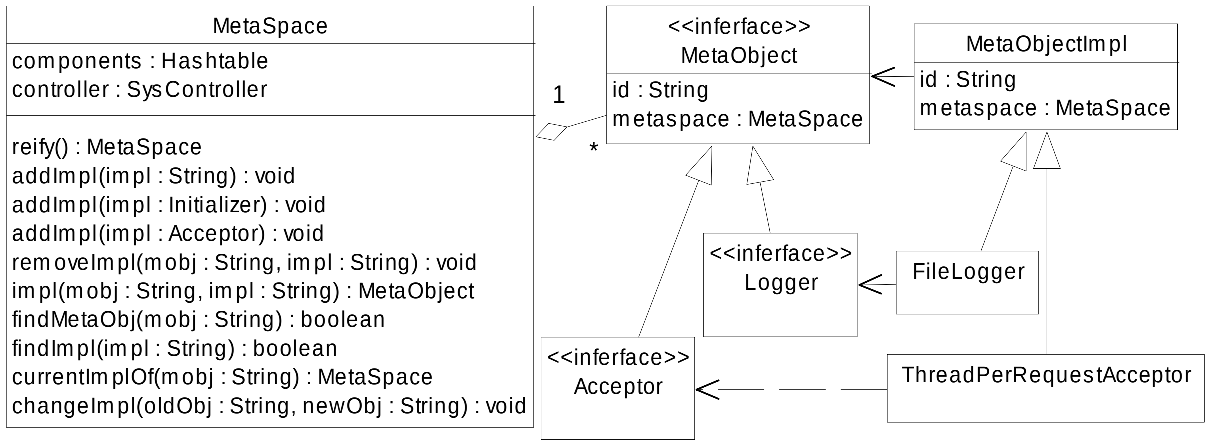
MetaSpace
components : Hashtable
controller : SysController
reify() : MetaSpace
addImpl(impl : String) : void
addImpl(impl : Initializer) : void
addImpl(impl : Acceptor) : void
removeImpl(mobj : String, impl : String) : void
impl(mobj : String, impl : String) : MetaObject
findMetaObj(mobj : String) : boolean
findImpl(impl : String) : boolean
currentImplOf(mobj : String) : MetaSpace
changeImpl(oldObj : String, newObj : String) : void
1
*
<<inferface>>
MetaObject
id : String
metaspace : MetaSpace
MetaObjectImpl
id : String
metaspace : MetaSpace
<<inferface>>
Logger
FileLogger
<<inferface>>
Acceptor
ThreadPerRequestAcceptor

Figure 4: A class structure of `MetaSpace`, metaobjects and their implementations based on *Mediator*. `MetaSpace` is an intermediary among metaobjects and encapsulates their interactions, instead metaobjects are connected each other directly.

Redirects an incoming request to another server, e.g. replicated server. `RequestHandler` uses it.

- `ExecManager`

  Executes external entities like CGI scripts.

These metaobjects represent typical aspects of web servers and define their interface and semantics. They are objects that affect the behavior of other objects, and has the following basic responsibilities [5]:

- Encapsulates system internals that may change.
- Provides an interface to facilitate modifications to the metalevel.
- Controls baselevel behavior.

As shown in Figure 3, all metaobjects are interface classes derived from `MetaObject` in the `kernel` package. Implementations of metaobjects are provided by implementation classes derived from `MetaObjectImpl`. Implementation classes are contained in the `meta` package, and provide different execution policies.

The relationship between metaobject and its implementations is based on *Bridge* design pattern [7]. The pattern explicitly decouples an interface and its implementation using object composition. OpenWebServer uses a variant of *Bridge*, which includes the interface-implementation relationship, provided by Java, between interface and its implementation. In our variant of *Bridge*, `MetaSpace` aggregates implementation objects so that it can change the metaobject's implementation dynamically. It can inspect the current implementation with the method `currentImpl()` and change it using `changeImpl()` (see Figure 3 and 4). *Bridge* avoids the permanent binding between metaobjects and their implementations, and allows changing the implementation at run-time, depending on the desired execution policy in the web server. It allows dynamic adaptation without system shutdown. Also, both the interface and the implementations should be independently extensible; changes in an implementation should have no impact on clients of the interface.

`MetaSpace` is a *Mediator* object [7], which encapsulates the interaction of a set of metaobjects and facilitates loose coupling among them by keeping the reference to every metaobject. It acts as an intermediary among the metaobjects, and coordinates interactions among a set of metaobjects. Each metaobject knows only `MetaSpace`, not any other metaobject, thereby reducing the number of interconnections. Figure 4 shows `MetaSpace` is an intermediary among `Acceptor` and `Logger`. Each metaobject has an attribute `metaspace` to refer to the metaspace it belongs to, and can call the method `curentImplOf()` of `MetaSpace` to get the current implementation of a desired metaobject. `MetaSpace` stores every metaobject and its implementations in the instance variable `components`, typed `Hashtable`. This variable has the mapping of a string entry (key) and `Vector` type variable. The former is used to assign the string name of metaobject, and the latter is for the sequence of implementation objects. The current implementation is assigned to the first element of the vector. The method `currentImplOf()` returns the first element, and `changeImpl()` changes the order of elements of the vector. To add an implementation object, the method `addImpl()` is used, which is prepared for every type of metaobject.

3. Adaptation of OpenWebServer

This section presents some examples of system adaptation using OpenWebServer.

Typical web servers spend most of the time for network and file I/O operations. Including file open operations, the time for them ranges between 80% to 90% [8]. By removing I/O, about 50% of the remaining sources of overhead is for operations to dispatch a HTTP request to a process or thread [8]. This result suggests we have to choose the right concurrency policy and avoid file accesses, if the server should be high-performance. For the system performance, OpenWebServer provides a wide range of policies for concurrency, caching and logging. These models are described in Section 3.1, 3.2 and 3.3, respectively.

The web server should also be robust and scale well for the high-workload condition. The typical approach is to make the server redundant. OpenWebServer provides the deployment options for fault tolerance and load balancing, described in Section 3.4.

3.1 Concurrency Models

OpenWebServer supports dynamic adaptation of concurrency policy. Currently, it defines a series of implementations for the `Acceptor` metaobject, and can tune the policy at runtime. It allows the following selections:

- Thread-per-request

  Is much faster than the forking policy. However, it is not portable for different platforms.

- Thread-per-session

  Is less resource intensive than the thread-per-request policy. It is significantly efficient in some particular conditions. However, it requires the server detects the client location for every request. It can be also performed using HTTP 1.1.

- Thread pool

  Reduces the overhead of the threading policy by pre-spawning a set of threads for the use in the future. It requires mutual exclusion.

- Single-threaded reactive I/O multiplexing

  Is conservative of resources and highly portable. It also has less overhead than the above policies by avoiding context switching and synchronization. However, it is fault-sensitive, and the number of simultaneous connections is limited.

The thread-per-request policy is implemented in `ThreadPerRequestAcceptor`, an implementation class of the `Acceptor` metaobject. This object creates an instance of `RequestHandler` and then executes it on a thread.

The thread-per-session policy maintains the established connections as long as possible and reuses them for the future data transfer. It can reduce the overhead to establish connections [9, 10]. This policy is effective in transferring media-rich documents. For example, when a document with 10 inline images is accessed, 11 connections are established and 11 threads (`RequestHandler`) are created on the server. The thread-per-session policy can potentially transfer this document with only one thread (`RequestHandler`). This policy is implemented in `ThreadPerSessionAcceptor`, an implementation class of the `Acceptor` metaobject. This object allows `RequestHandler`s to keep alive for the specified time, which is defined in a configuration file.

A thread pool is an object which holds a pre-created set of threads that can be dispatched to a task and then returned to the pool when they are no longer needed. This policy has the following advantages:

- Confines the elements of failure to create threads
- System resource use is bounded.
- Fast thread dispatching

OpenWebServer provides two different kinds of pools as utility objects: `FixedThreadPool` and `GrowableThreadPool`, which hold one or more idle `RequestHandler`s. The former creates a fixed number of threads when the pool is instantiated. If there is no available threads in the pool, `Acceptor` waits to dispatch an incoming request to a thread (`RequestHandler`) until an idle thread is returned. The latter object is initially populated with a fixed number of idle threads. However, a new thread is created and added to the pool, if all threads are busy, and if the maximum number of threads has not been reached yet. Also, any thread that has been in the pool longer than the time decay value without being dispatched to a task is automatically terminated, if the number of threads in the pool has not reached the initial thread count. The initial and maximum number of threads are defined in a configuration file.

The single-threaded reactive I/O multiplexing is a policy that consumes "request-arrived" events from connections and processes them synchronously within a single thread. The threading models suffer from context switching and synchronization overhead on single CPU platforms, while they scale well to multiple CPU platforms. Therefore, a single-threaded concurrent server is a good choice for a uni-processor platforms. It is implemented in `ReactiveAcceptor`, an implementation class of the `Acceptor` metaobject. This object follows the `Reactor` design pattern [11]. It has the following benefits:

- Lightweight and efficient

  This policy does not need context switching and synchronization among threads. It also lower the memory requirements.

- Improves portability

  This policy does not use threads and the interface of `ReactiveAcceptor` can be reused, independently of operating systems.

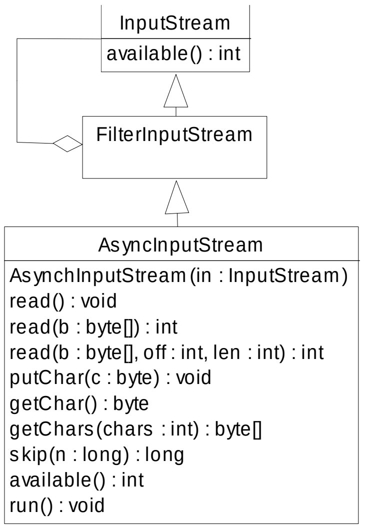

Figure 5: `AsyncInputStream` is used for the asynchronous I/O capability. It is plugged into `java.io.FilterInputStream`, based on the *Decorator* design pattern.

The liabilities are followings:

- Non-preemptive

  Serialized operations can not be preemptive while they are executed. Therefore, the long-duration operations, such as transferring large files, increase the system overhead.

- Error sensitive and limited number of connections

  The number of connections that the server can establish at same time is limited within the max number of file descriptors per a single process, which depends on every OS. Also, an even little error occurred in processing a request can cause the system down.

Consequently, this concurrency model is appropriate for the server that is low workload and transfers small files. Some studies shows that more than 80% of all requested files from a typical web server are actually small, smaller than 10kB [8].

This policy requires the `ReactiveAcceptor` must not block for handling any single connection at the exclusion of others, since this may significantly delay the responses to multiple clients. Common system calls for such an asynchronous I/O are `select` and `poll` for Unix [12], and `WaitForMultipleObjects` for Win32 [13]. The Java version of OpenWebServer provides an asynchronous I/O object, `AsyncInputStream`, plugged into the `java.io` package used for handling I/O streams (Figure 5). `AsyncInputStream` does not use the method `available()` of `java.io.InputStream`. This method returns the number of bytes that can be read from the stream without blocking. However, it does not work well with certain mechanisms like the network socket, and `read()` may not block if `available()` returns 0 [14]. `AsyncInputStream` reports the correct number of bytes that can actually be read asynchronously without using the derived `available()`. It can get available data in a non-blocking manner by spawning a thread that calls blocking `read()` exclusively. Figure 6 shows the sequence performing the single-threaded reactive model. An infinite loop executes the step 2 and 3 in this figure.

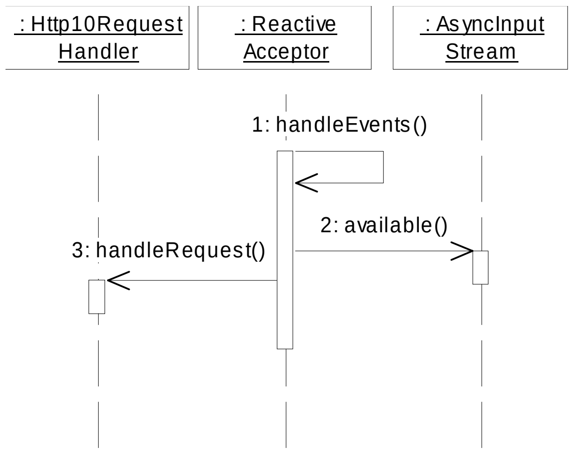

Figure 6: Sequence diagram of the single-threaded reactive I/O multiplexing policy. It follows the *Reactor* design pattern.

OpenWebServer allows changing concurrency policies at runtime. For example, it can start as a single-threaded reactive server, and then change itself into a thread-per-request server when the workload (i.e. the access rate) exceeds the predefined threshold. The workload is monitored by the `LoadManager` utility object, which tracks the system's status and calculates the statistics such as the average latency and throughput. The patterns to switch policies and their thresholds are defined in a configuration file.

In the process of changing concurrency policies, the `RequestQueue` utility object is used to store incoming requests temporally. Next, `ReactiveAcceptor` forwards the newly arrived requests to the `Queue` without processing them, and processes only the existing requests that have arrived before the change of policy. When all the existing requests are processed, the `ReactiveAcceptor` signals this fact to the `SysController`. Then, `SysController` dispatches the socket object to `ThreadPerRequestAcceptor`, and starts the acceptor's internal loop. Through the above process, the server becomes a threaded server and all the requests are processed on a different thread. These synchronization issues are opaque for baseobjects so that the baselevel program is kept simple.

3.2 Caching Models

Many servers transmit files via HTTP by reading them from the underlying file system and writing them to the TCP socket connected with clients. While this is easy to implement, it is not very efficient [12]. The main problem is that the data is copied twice: from the file system into memory, and again from the memory to network stream. As described earlier, avoiding file accesses increases system performance dramatically.

OpenWebServer provides the `Cache` metaobject that maintains pre-fetched files in memory and allows fast access to them. It holds the pairs of an absolute file path and actual open file. When the caching capability is enable, i.e. the Cache metaobject exists in the metalevel, `ContentFinder` uses it to find and return a target file to `RequestHandler`.

In general, the caching capability implements a policy to limit the size of cached entities. It determines which entity to cache and which to discard when the number of entities in the cache reaches the upper bound. There exists some caching policies for certain requirements [15, 16, 17], which have been studied in the OS research domain. OpenWebServer provides the following policies:

- FIFO (First-In, First-Out)

  Discards the oldest file, when the number of cached files reaches the upper bound. It is simple and easy to implement. However, it is not very optimal [15], because the lifetime of a cached file depends on the length of the cache size independently of access patterns.

- LRU (Least Recently Used)

  Discards the file that is least recently used from cache. It is known optimal, while it may be expensive [15].

OpenWebServer provides fixed and growable variants for each policy: `FixedFIFO`, `GrowableFIFO`, `FixedLRU`, and `GrowableLRU`. These policies are defined as implementation classes for the `Cache` metaobject. The growable models allow the cache size variable. It increases the size, if the current size reaches the current upper bound and if it does not reach the maximum size. The initial and maximum number of cached files can be determined according to the amount of available memory. These sizes are defined in a configuration file. OpenWebServer can also configures the cache disabled if the underlying environment has the limited size of memory.

3.3 Logging Models

Many servers record the access log in a file once a HTTP request is accepted. However, the logging-per-request is relatively expensive policy, because it opens a log file, writes down access information and then close the file whenever the server receives a request. Caching a log file is a workaround for this problem. It reduces overhead by avoiding the file I/O operation.

OpenWebServer provides the `logger` metaobject that specifies the logging policy. It prepares the following implementation objects for `logger`:

- `FileLogger`

  Logs the access information in a file whenever the server accepts a request.

- `BatchFileLogger`

  Logs the access information in a file in the batched manner.

Both loggers cache a log file by default, if the caching capability, described above, is enabled.

`BatchFileLogger` implements the policy of logging for multiple requests, which maintains a set of access logs in memory for a pre-defined period, and writes them into a file later. It is more efficient than the cached `FileLogger` by avoids writing an access log into a file.

3.4 Load Balancing and Fault Tolerance

An approach for improving performance of a high-workload server is to distribute the traffic to replicated back-end servers. It is a good idea because the back-end servers work well using the medium-scale hardware, and because they also increase fault tolerance, i.e. reliability and availability, by forming a cluster of servers.

OpenWebServer provides the deployment configuration that achieves load balancing and fault tolerance by managing two or more replicas of a web server. It can perform as a reverse proxy server [18, 19], which is a proxy for servers while a normal proxy server is a proxy for clients. A reverse proxy determines an appropriate back-end server to redirect a request and forwards back its response to a client, instead of processing the request (Figure 7). This clustering model delivers the following benefits:

- A set of replicated servers behave identically to the original single server. The difference is transparent to clients. Clients access a reverse proxy as if it is a normal web server. An entry point to the cluster remains single.
- A reverse proxy has the complete control over the delegation policy of incoming requests locally.

OpenWebServer implements the reverse proxy as an implementation class of the `RequestHandler` metaobject, `ReverseProxyRequestHandler`. It redirects incoming requests to the appropriate back-end server by rewriting the original URL and HTTP request/ response headers. When a target back-end server does not respond in a certain time, this request handler delegate the undelivered request to another server.

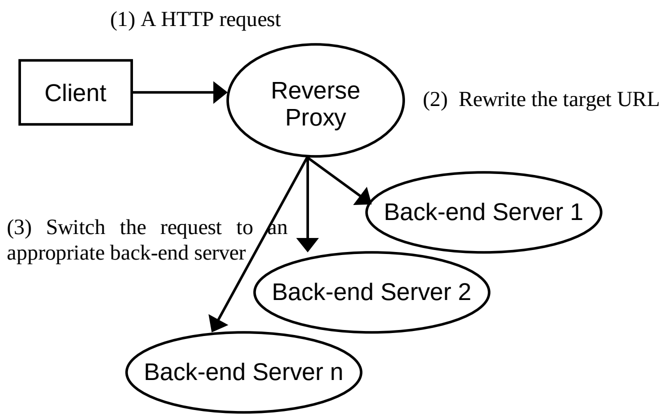

Figure 7: A web server cluster using a reverse proxy.

The `Redirector` metaobject defines the policy to delegate HTTP requests. The following implementations of `Redirector` are defined:

- `RandomRedirector`

  Redirects requests to a back-end server at random.
- `PriorityDrivenRedirector`

  Redirects requests with the priorities for back-end servers. Each priority is defined in a configuration file and changeable at runtime.
- `LatencyDrivenRedirector`

  Redirects requests with the average response time of each back-end server. It assigns a request to the most responsive server.

3.5 Document Personalization

OpenWebServer provides the document personalization capability as an application-level service. Personalization involves in fine-tuning contents and presentations based on the needs of the user. This service is performed with the Persona toolkit, which is an engine to mine a document's appropriate content and/or presentation suited to the context where the it is accessed. Both HTML and XML (eXtensible Markup Language) can be personalized with Persona [20, 21].

OpenWebServer prepares an implementations for the `ContentFinder` metaobject, `HtmlFinder` and `XmlFinder`, which find a requested HTML and XML document respectively. It also provides utility objects that parse HTML/XML documents, create their syntax trees, and retrieves elements and attributes in the parsed tree. Personalized documents are dynamically generated with the context of client-side environment, user profile and users' behaviors. The personalization for HTML documents is performed by re-authoring their contents. XML documents are personalized by either applying an appropriate XSL (eXtensible Stylesheet Language) stylesheet or re-authoring the document content.

4. Current Project Status and Future Work

The current OpenWebServer includes 9 metaobjects, 40 implementation objects and 52 utility objects. It was initially implemented with the Python programming language and, later, with Java. We are investigating other languages to illustrate that the architectural design of OpenWebServer does not depend on a specific language.

As for the metalevel in OpenWebServer, we are aggressively making metaobjects fine-grained. At present, we are dividing the metaobject `Acceptor` into different objects that deal with concurrency and I/O, as described in [22], because the current `Acceptor` is somewhat coarse.

We are also developing examples of system adaptation with OpenWebServer to demonstrate the power of its metalevel and improve it. We plan to introduce additional communication protocols such as HTTP 1.1, HTTP-ng, LDAP (Lightweight Directory Access Protocol) and SNMP (Simple Network Management Protocol). We also plan to provide real-time streaming functionality for continuous media using RTP (Realtime Transport Protocol) or RTSP (RealTime Streaming Protocol). New underlying environments of OpenWebServer are also planned including CORBA, embedded environments, and real-time operating systems.

5. Conclusion

This paper addresses how our adaptive web server can meet diverse requirements in the web server design space, and describes the advantage of its meta-architecture that makes the system adaptable and configurable. OpenWebServer makes its aspects open-ended for extension, and allows itself to continually evolve beyond the static and monolithic servers of today.

6. References

[1] J. Suzuki and Y. Yamamoto. OpenWebServer: an Adaptive Web Server using Software Patterns. In *IEEE Communications Magazine*, April 1999. to be appeared.

[2] J. Suzuki and Y. Yamamoto. Building an Adaptive Web Server with a Meta-architecture: AISF approach. In *Proceedings of SPA'98*, March 1998.

[3] G. Kiczales et. al. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[4] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA '87*, 1987.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal. *A System of Patterns: Pattern-oriented software architecture*. WILEY, 1996.

[6] B. Meyer. *Object-Oriented Software Construction 2nd edition*. Prentice Hall, 1998.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] J. C. Hu, S. Mungee and D. C. Schmidt. Principles for Developing and Measuring High-Performance Web Servers over ATM. In *Proceedings of INFOCOMM'98*, 1998.

[9] J. C. Mogul. The Case for Persistent-Connection HTTP. Technical Report, Digital Equipment Corporation Western Research Laboratory, May 1995.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Standards Track RFC 2068, January 1997.

[11] D. C. Schmidt. Reactor - An Object Behavioral Pattern for Event Demultiplexing and Event Handler Dispatching. In *Proceedings of the First Pattern Languages of Programs*, 1994.

[12] W. R. Stevens. *UNIX Networking Programming, First Edition*. Prentice Hall, 1990.

[13] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

[14] S. Oaks and H. Wong. *Java Threads*. O'Reilly, 1997.

[15] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[16] C. Maeda. A Metaobject Protocol for Controlling File Cache Management. In *Proceedings of International Symposium on Object Technology for Advanced Software (ISOTAS '96)*, 1996.

[17] M. Grand. *Patterns in Java*. WILEY, 1998.

[18] R. S. Engelschall. Load Balancing Your Web Site. Web Techniques Magazine, vol.3 issue 5, May, 1998.

[19] A. Nuotonen. *Web Proxy Servers*. Prentice Hall, 1997.

[20] J. Suzuki and Y. Yamamoto. Document Brokering with Agents: Persona approach. In *Proceedings of the Sixth Workshop on Interactive Systems and Software (JSSST WISS '98)*, December, 1998.

[21] J. Suzuki and Y. Yamamoto. Metadata Management in Personalizing Web Presentations. Submitted to *the Seventh International Conference on User Modeling (UM '99)*, 1999.

[22] J. C. Hu and D. C. Schmidt. Developing Flexible and High-performance Web Servers with Frameworks and Patterns. In *ACM Computing Surveys*, May 1998.