Building an adaptive Web server with a meta-architecture: AISF approach

Junichi Suzuki† Yoshikazu Yamamoto†

†Department of Computer Science,
Faculty of Science and Technology,
Keio University, Yokohama, 223-8522, Japan.
{suzuki, yama}@yy.cs.keio.ac.jp

Abstract

As the Web is becoming ubiquitous environment, Web servers are faced with a diversity of requirements. This paper addresses how Web servers can meet a wide range of requirements, and proposes an adaptive framework called AISF (Adaptive Internet Server Framework) which employs a meta-architecture to make the Web server adaptable and configurable. We are interested in applying the meta-architecture to Web servers, where it has had little acceptance to date. AISF provides a set of fine-grained metaobjects which capture and express various aspects in Web servers. Web servers can dynamically adjust itself to meet any requirement by implementing it within the AISF programmable meta model. This paper also demonstrates that AISF can be used to meet every demand placed upon Web server consistently, and includes some proof-of-concept examples. Our approach aims to create adaptive Web servers beyond a static and monolithic servers.

1 Introduction

Today, the Web (World Wide Web) is a very popular mechanism for publishing or application front-end, and has widely used than most expectations. The explosive growth of the Web places increasingly larger and challenging demands on servers. For example, current Web servers are required to:

- connect with various software systems such as groupware, database management systems, mobile agent engine and TP monitors.
- integrate with the more generic distributed environments such as CORBA (Common Object Request Broker Architecture) and DCOM (Distributed Component Model).
- extend their functionality, e.g., by introducing additional network protocols or content media types.
- change its execution policies, e.g., in connection management or request handling, as a single policy cannot be the best solution in every situation.
- change the execution environment, as modern Web servers are deployed on ATM networks or embedded within the electronic devices like routers, printers or copiers.

As such, a Web server is required to be flexible and adaptable to meet a wide range of demands. Unfortunately, most existing Web servers are monolithic in that they provide a fixed and a limited set of capabilities. The usual solution that allows existing web servers to meet this diversity

of requirements, as described above, involves shutting down the system, adding some components that fulfill the new requirement, integrating them with existing components, and restarting the system. This solution, however, is often difficult or expensive (i.e. time consuming), or does not allow the server to dynamically change to meet new requirements (e.g. execution policies and configurations). This is why any requirements cannot be anticipated at the Web server's development phase. Therefore, the "scrap-and-build" solution, whereby the system is rebuilt from scratch, is often taken. Coupled with the increasing role on the Internet/Intranet, we need Web servers which can reuse and select feasible requirements without being locked in a single one.

Consequently, what most Web servers lack is *adaptability* to enable system evolution. The general issue on the system adaptability is that system designers cannot know or predict all possible uses of the system, because an service or configuration that is appropriate at one point in time is not always useful at another point, and the system cannot evolve transparently.

In this paper, we address this problem and describe how our work aims to make Web servers adaptable. For a highly adaptive Web server, we propose an adaptive framework for Web servers called AISF (Adaptive Internet Server Framework), which employs a meta-architecture. We consider the use of the meta-architecture as a mechanism for specifying the structure and behavior of an open-ended and flexible system that can be dynamically adapted and extended. We outline the design of AISF meta model and discuss how we can consistently exploit AISF for the increasingly diverse requirements using some applications. Our work differs from other efforts in that our framework can express various aspects of Web servers, and can support a wide range of requirements.

The remainder of this paper is organized as follows. Section 2 introduces the meta-architecture and reflection, and Section 3 outlines comparisons with related work. Section 4 describes the AISF conceptual model and design principles. Section 5 presents some examples of the use of AISF. We conclude with a note on the current status of the project and some future work in Section 6 and 7.

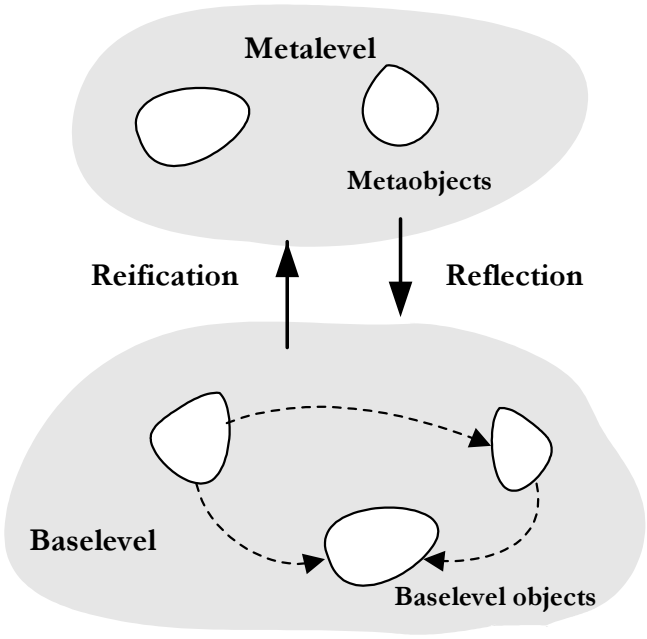

Figure1 Constructs found in a typical meta-architecture.

2 Meta-architectures and Reflection

Meta-architecture introduces the notion of "object/metaobject separation". In general, a *metaobject* (or *metalevel object*) is an object that contains information about the internal structure and/or behavior of one or more objects (*baselevel object* or *baseobject*). So, metaobjects can track and control certain aspects (i.e. structure and/or behavior) of baseobjects. While a set of metaobjects is called the *metaspace* or *metalevel*, a set of baseobjects is called the *baselevel*.

Reflection is the ability of a program to manipulate as data something that represents the state of the program [1] [2] and adjust itself to changing requirements. The goal of reflection is to allow a baseobject to reason about its own execution state and eventually alter it to change its meaning, during its own execution.

In contrast to reflection, *reification* [3] is the process to make something accessible which is not normally available in the baselevel (i.e. programming environment) or is hidden from the programmer. To supervise the execution of a baseobject, it has to be explicitly reified into the corresponding metalevel. A set of interfaces with which a baselevel object accesses its metalevel is

called *Metaobject Protocols* or *MOPs*. The relationship among the constructs described above is illustrated in Figure 1.

Reflection was originally introduced by 3-Lisp [4] [5] and has been studied within various programming languages such as 3KRS [2], CLOS [6], ABCL/R [7], Smalltalk [8] [9], OpenC++ [10], Iguana [11], and MetaJava [12]. The intention of these reflective languages is the introduction of a general mechanism to extend the language itself. It has been shown that reflection eases (1) *extensibility*, e.g., a reflective language can extend its syntax and semantics for problem domains which were not originally considered, (2) *compatibility*, e.g., backward compatibility between the new additional and existing definitions, and (3) *efficiency*, e.g., the implementation strategy can be varied to optimize the behavior.

Recently, reflection has been applied to more generic system design, other than programming languages, such as agent engines [13], object-oriented databases [14], parallel computing [15], operating systems [16] [17], class libraries [18], object request brokers [19], distributed type systems [20] and hypermedia systems [21].

3 Related Work

Few meta-architectures have been applied to Web servers. A rare example is Zypher. Zypher [21] [22] [23] is a framework for the open hypermedia system, which intends to provide three kinds of tailorability:

- Domain-level tailorability: is needed to deliver a hypermedia system for a specific application domain.
- System-level tailorability: aims to deliver services that affect the overall behavior of the hypermedia system.
- Configuration-level tailorability: aims to provide coordination among the system components, which can be adapted without changing their internal organization.

To allow system level tailorability, Zypher provides three metaobjects: `Path`, `Session`, `Hypertext`. Path deals with navigation in the hypermedia system, Session controls the presentation, and Hypertext is responsible for storage management. Zypher also incorporates a meta-metaobject: `HypermediaContext`, to allow configuration-level tailorability.

Baselevel objects interact with these objects to define and/or modify the behavior of the system. However, Zypher's metalevel is too coarse-grained to adapt to unanticipated behavior, anything other than predefined behavior. The author claims that it's possible to extend the metalevel by adding/removing metaobjects or instantiating multiple instances of each metaobject [22]. Unfortunately, there is no description of how to manage the metalevel components, e.g., how to add/remove metaobjects and coordinate them with existing objects, which may cause modification of the meta-metaobject. In contrast, the AISF metalevel is a set of fine-grained metaobjects, which can scale well, even for unanticipated behavior at all levels, as described in Section 4.

4 Applying a meta-architecture to Web servers

4.1 AISF conceptual framework

AISF is a framework dedicated to Web servers, which captures and expresses a wide rage of aspects of Web servers. The scope of AISF includes the full range of metalevel and a part of the baselevel, by providing a set of fine-grained metaobjects and supplemental utility objects to help writing both the base and metalevel. Since AISF is based on a pure object model, which has no elements that are not objects, both the base and metalevel consists of a series of objects. Implemented within the AISF programmable metalevel, a Web server can dynamically adjust itself so that it is executed in the best condition along with a given requirement.

Though modern Web servers provide some extension mechanisms like CGI (Common Gateway Interface), server-side APIs and server-side scripting, their extensibility is restricted within the application-level. In contrast, AISF provides a uniform and consistent 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, even unanticipated ones, without breaking the framework. In other words, AISF plays a role of a generic "change absorber" for

Web servers.

The advantages of the Web server based on a meta-architecture can be summarized as follows:

- **Separation of concern**: In conventional Web servers, the system's basic mechanisms are mixed and complicated by policy algorithms. This makes it difficult to understand, maintain and validate the program. The separation of the reflective facilities from the basic mechanisms allows the reusability of feasible policies [6] [24]. This separation allows system designers and application programmers to focus on their problem domain, which leads to more structured and more easily maintained systems.
- **Adaptability and Configurability**: The metalevel keeps the baselevel open [6]. New requirements can be implemented without changing the baselevel application code. Not only can the system designers profit from the metalevel, but also the system users or administrators who create or replace metaobjects to tailor the system for their specific demands. This eliminates the "scrap-and-build" solution to system development.
- **Transparency**: Transparency between the metalevel and baselevel reduces the constraints between both. Thus, changes in the metalevel can be achieved with relatively minor modifications to baselevel objects within the original system.

4.2 Design of the AISF model

To structure the AISF metalevel, we have isolated the decomposed services and entities by looking for the events which occur during the execution of Web servers. Then, we have specified each one as a metaobject. In this fashion, the Web server can be described with a collection of metaobjects. The AISF metalevel has a set of fundamental metaobjects called *base metaobjects*, which specify the Web servers' aspects and defines their default behavior. In our initial implementation, AISF provides seven primitive base metaobjects; `SysController`, `Initializer` `Acceptor`, `RequestHandler`, `Protocol`, `Logger`, and `ExecManager` (Figure **??**). The basic responsibilities of each are listed below:

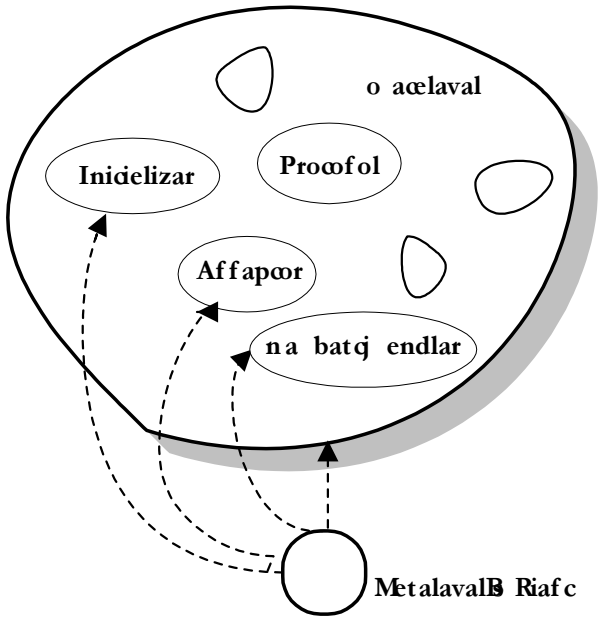

Figure2 A baseobject and its corresponding metaobjects (base metaobjects). Every baseobject has its own group of metaobjects (metalevel) and explicitly interact with its metaobjects to define and/or modify its own capability.

- `SysController`
  – keeps track of the system status and maintains the configuration within the overall system. Once the AISF metalevel is created, this object is instantiated at first. `SysController` is an active object executed on a thread and a root in the thread hierarchy.
- `Initializer`
  – initializes the network infrastructure along with the current configuration. The typical task is to create a socket(s) to accept incoming requests and `Acceptor` to wait for them.
- `Acceptor`
  – waits for and accepts incoming requests. Once it obtains a request, it asks for a `RequestHandler` to process the request along with the kind of request or communication protocol.
- `RequestHandler`
  – deals with requests passed by `Acceptor`. It is created when an `Accepter` accepts a request (i.e. on the per-request basis), or resides permanently.

- `Protocol`
  – defines the protocol specific information. It exists on a per-protocol basis and is used by a `RequestHandler`.
- `Logger`
  – records the log of access to the Web server.
- `ExecManager`
  – is used to execute any external entities like CGI scripts.

These base metaobjects express the typical aspects of Web servers. There can be more and less base metaobjects depending on requirements. As introduced later in this paper, AISF provides the dynamic mechanism to organize the metalevels.

Figure 2 depicts the relationship between a baseobject and its metalevel, which has its own group of metaobjects. Each group is represented with a metaobject `MetaSpace`, which is an entry point from the baselevel to metalevel. Any baseobjects access the `MetaSpace` when communicating with their metalevel(s). The `MetaSpace` is somewhat similar to the metaobject Reflector in Apertos operating system [16]. All the objects defined in AISF has the method `metalevel()` by default and can use it to refer to its metalevel (represented by an instance of `MetaSpace`), as follows:

```
aMetaSpace = aBaseobject.metalevel()
aMetaSpace...
```

AISF eliminates the explicit level-shifting from baselevel to metalevel and the reflective functions found in other reflective systems [4] [5]. This is similar to the approach that CodA [25] introduced. Metaobjects are the same as any other objects in the system, and the AISF metalevel is just another application. Reflective computation is executed by direct interaction with the desired metaobjects.

In AISF, every baseobject does not have to be attached to a metalevel or to metaobjects, but is dynamically attached at run-time; *lazy reification*. Every baseobject can be reified with either of the methods `reify()` and `fullyReify()`, which all baseobjects have:

```
aBaseobject.fullyReify()
aMetaSpace = aBaseobject.metalevel()
```

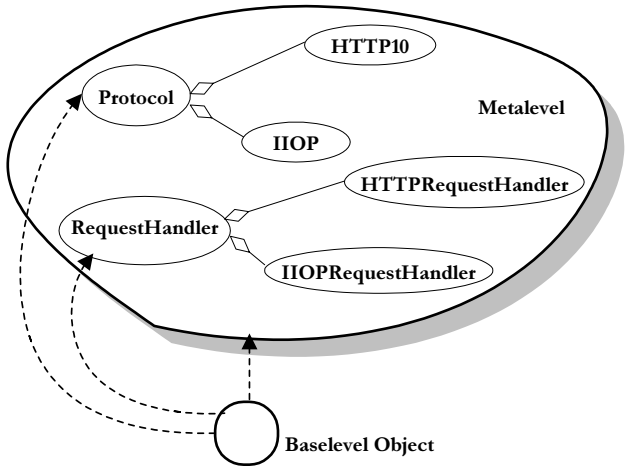

Figure3 The relationship between a base metaobject and concrete metaobjects, based on object composition.

```
aMetaSpace...
```

Once a baseobject is reified, it gets to be aware of its metalevel and the corresponding metaobjects are then instantiated. While the method `fullyReify()` creates all the default base metaobjects, the method `reify()` can specify the reified base metaobject as a parameter:

```
aBaseobject.reify("Acceptor")
aBaseobject.reify("RequestHandler")
aMetaSpace = aBaseobject.metalevel()
aMetaSpace...
```

As such, the number and type of instantiated metaobjects depends on what the users wish to do. Reflective computation, which incurs a performance overhead, occurs only when or where a baseobject needs to know (i.e. reify) its aspects which the corresponding metaobjects express.

As shown in Figure 3, a base metaobject has relationships with *concrete metaobjects* to provide the concrete behavior of the metaobjects. For example, `RequestHandler` has associations with `HTTPRequestHandler` and `IIOPRequestHandler`, which define the way to handle HTTP and IIOP requests, respectively. Also, `Protocol` is associated with `HTTP09`, `HTTP10`, `HTTP11` and `IIOP`, which define the protocol specific information of HTTP version 0.9, 1.0, 1.1 and IIOP, respectively. The relationship between a base metaobject and a

concrete metaobject is designed using object composition rather than inheritance, which is typically used to modify an object's behavior incrementally, because inheritance imposes so-called "early and permanent binding" between an object and its method's semantics. Early binding means that the system designer has to choose the method's semantics in instantiating the object or its derived objects. Permanent binding means that the system users cannot alter the method's semantics at run-time, once the object is created. This kind of discussion is well known as the "class inheritance versus object composition" argument in the object technology community. It is recognized that object composition should be favored over class inheritance, since object composition decouples the hardwired behavior and provides more run-time flexibility [26].

For example, `RequestHandler` delegates the method invocation of `handleRequest()`, which is called to process incoming requests, to the concrete metaobject that defines the behavior for the specific protocol (e.g. `HTTP10RequestHandler` or `HTTP11RequestHandler`). Also, `RequestHandler` can switch the delegate object at run-time. This allows the Web server to dynamically change the underlying protocol without a system shutdown.

Since AISF offers customizable functionality as a collection of metaobjects, users only have to assemble the appropriate metaobjects together to fulfill a new requirement. Thus, a scheme that organizes and coordinates the metalevel is required to compose metaobjects effectively. To achieve this concept, AISF employs the notion of metaobject/meta-metaobject separation in addition to the object/metaobject separation. A *meta-metaobject* is the metaobject for a base metaobject and exists on a per-metaobject basis. It knows the structure of a base metaobject and controls its behavior.

Figure 4 shows a typical interaction between a baseobject, a base metaobject and meta-metaobjects. Meta-metaobjects are named with the prefix `mm`; `mmRequestHandler` is a meta-metaobject of `RequestHandler`. Meta-metaobjects know the current configuration or status of a corresponding base metaobject. For example, `RequestHandler` can ask which concrete metaobject is currently selected for handling requests by calling the method `current()` of its meta-metaobject; `mmRequestHandler`:

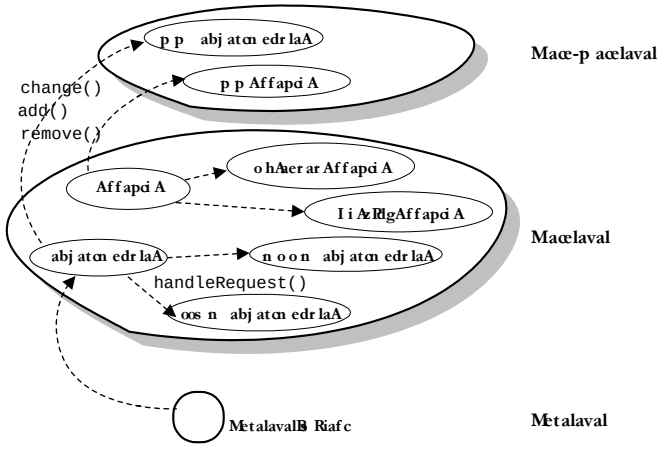

Figure4 Interactions among baselevel, metalevel and meta-metalevel. A base metaobject interacts with the corresponding meta-metaobject for refering to its current status and changing its configuration.

```
aMetaMetaRh = aRequestHandler.metalevel()
aMetaMetaRh.current()
```

With a meta-metaobject, `RequestHandler` can dynamically change the concrete metaobject to which the task `handleRequest()` should be delegated by calling the method `change()` of `mmRequestHandler`. To add or remove concrete metaobjects, a base metaobject invokes the method `add()` or `remove()` of its meta-metaobject. As such, meta-metaobjects allow the incremental modification in the metalevel. A concrete use of meta-metaobjects is demonstrated in Section 5.1.

5 Applications

AISF has been applied to the following four requirements. In this section, we explain the first example at length and others more briefly.

5.1 Changing Request Handling Policy

Most Web servers are currently built based on the "forking" model for a multi-process server. Such a server creates a new process for every client-server connection. This approach is simple to implement and manage, but does not scale very

well in high-load situations. Creating a new process is so expensive on any operating systems that it is actually noticeable to end users. For example, when you browse a HTML page with 10 inline images, you establish 11 connections and create 11 processes on the remote Web server. Thus, a lot of clients accessing the Web server and/or media-rich contents mean a lot of processes and the longer latency. Also, each process requires the allocation of a certain amount of memory space. Even though most operating systems incorporate techniques that make this much less costly than it could be, the result is still very wasteful. The server that is simultaneously accessed by 100 clients requires hundreds of megabytes of real memory in order to respond smoothly. Some Web servers like Apache provide a solution that partly overcome these drawbacks, the "process pool", which pre-creates a fixed number of processes. Incoming requests are attached to each process in turn. Although this alleviates the performance problem due to process creation, up to the pre-determined number of processes, it does not scale well.

An alternative solution is the "multi-threaded server", which avoids the problems involved in a "forking server" by using threads. With threads, multiple concurrent execution is achieved within a single process and thread creation/deletion is relatively cost effective. However, the multi-threaded server is hard to maintain and port into different environments. A "Thread pool" is used to precreate a fixed number of threads and attach each one in turn to an incoming request. This is an analogous approach to the process pool for forking servers.

In addition, a single-threaded server with I/O multiplexing capability has been proposed. It multiplexes I/O channels to incoming requests within a single thread (process). Running as a single process, there is no per-client process (or even thread) creation/destruction overhead and no context switching overhead. As well as these advantages, memory requirements are also lowered. The drawback in this kind of server is that the number of connections, which the server can establish simultaneously, is limited to the maximum number of file descriptors per single process (this number depends on the underlying operating system.), and that even a small error can cause the system down. The advantages and disadvantages of the above request handling policies are summarized below:

- Single-threaded with I/O multiplexing
  - Resource saving.
  - Less overhead.
  - Highly portable.
  - Less connections.
  - Fault sensitive.
- Process per request
  - Simple model.
  - Portable.
  - Much overhead.
  - Resource intensive.
- Process pool
  - Alleviates the overhead.
  - Requires mutual exclusion.
- Thread per request
  - Much faster than forking server.
  - Not portable.
- Thread pool
  - Alleviates the overhead.
  - Requires mutual exclusion.

In our initial implementation, AISF supports three policies; a process per request, a thread per request and a single thread with I/O multiplexing, and the threaded server is default. AISF allows Web servers to change the policy at run-time, using the metaobjects `SysController`, `Acceptor` and the utility object `Queue`. For example, a Web server can start as the single-threaded server with I/O multiplexing and then change itself into the threaded server, in case the work load (i.e. the access rate) of the server goes over the predefined threshold. The code in Appendix A shows the behavior of a baseobject, which configures the initial policy and then changes it. When the method `change()` of the meta-metaobject is executed like:

```
aMetaMetaAcceptor.change(ThreadedAcceptor)
```

`SysController` instantiates a new concrete metaobject `ThreadedAcceptor` and utility object `Queue` to store the requests temporally. Next, `SingleThreadedAcceptor` forwards the incoming requests to the `Queue` without processing them and processes only the existing re-

quests that have arrived before the change in policy. When all the existing requests are processed, the `SingleThreadedAcceptor` signals this fact to the `SysController`. Then, `SysController` dispatches the socket object to the `ThreadedAcceptor` and starts the acceptor's internal loop. Through the above process, the Web server becomes a threaded server and all the requests are processed on a thread. These synchronization issues are opaque for baseobjects so that the baselevel program is kept simple.

5.2 Changing Protocols

While most Web servers are based on HTTP, some modern servers like Apache, Netscape Enterprise Server and Microsoft Internet Information Server have now introduced additional protocols. Also, W3C (World Wide Web Consortium), which is the standardization body for Web technologies, is working on the next generation protocol. In near future, Web servers will be required to add emerging protocols. Currently, AISF defines three kind of protocols; HTTP version 0.9, 1.0 and 1.1. A baselevel application can alter the protocol with the metaobjects `SysController`, `RequestHandler` and `Protocol`. AISF can adopt any additional protocols that Web servers are underlying on, using the AISF consistent interfaces.

5.3 Adding an application level components

In addition to the above applications, which are examples of modification in the system's low level services, AISF can also capture the higher application-level requirements. We have described the CGI and Server-side scripting capability within AISF. A Web server can dynamically add and remove such components using the metaobjects `Syscontroller`, `Acceptor` and `ExecManager`.

5.4 Configuring the minimum set of functionalities

Changing the system's execution environment, rather than changing the functionality, is also important for system adaptability. For example, as touched on above, Web servers are now embedded into electronic devices such as routers, printers and copiers to show statistics and configuration information.

We have tested to configure the minimum set of functionalities for the execution environments like the above embedded Web servers. In such an environment, the Web server should be small and lightweight due to the limited resources (e.g. memory and disk space). AISF allows a Web server to configure itself for the minimum set of functionalities, which includes only processing HTTP requests without logging, keeping the system status or running CGI program.

5.5 Further proposals for applications

We are investigating to introduce the additional protocols or configuration settings, and working on the further applications for the following environments: a multimedia environment where the AISF metalevel represents the continuous media handling and streaming, a Web-CORBA integrated environment where a Web server is built on top of an object request broker, and a CSCW environment where the software design documents are shared within the distributed development team.

6 Current Project Status and Future Work

AISF has initially been implemented using Python programming language on a Pentium 120MHz PC with 48MB RAM running Windows NT. The initial AISF implementation includes seven base metaobjects, seven meta-metaobjects, twelve concrete metaobjects and seven supplemental objects. We are investigating the implementation of AISF with other reflective languages to illustrate that the design of AISF itself does not depend on the programming language. We are now evaluating some other reflective languages.

As for the AISF metalevel, we are experimenting with an additional mechanism to coordinate metaobjects. Currently, AISF incorporates the notion of metaobject/meta-metaobject separation to carry out this function, as described in Section 4.2. However, composing and coordinating metaobjects so as to introduce a new system behavior requires precise knowledge of the interfaces and implementations of all the metaobjects in the metalevel. Few methodologies have been proposed for metaobject composition. Currently, the

major experiment with AISF aims to develop a framework that enables the relationship between the baselevel and metalevel or among metaobjects to be more configurable, and to hide the detail of such coordination process away from baselevel. We are also measuring the costs of reflective computation in AISF.

As for AISF applications, the examples given above show unique and reasonable ways of implementing behaviors, which are not difficult to do without AISF, though they are quite simple. We are working on the additional applications touched in Section 5.5, to demonstrate the power of AISF and improve the model.

Further information on the project status can be obtained from
http://www.yy.cs.keio.ac.jp/ suzuki/project/aisf/.

7 Conclusion

This paper addresses how Web servers can meet diversing requirements, proposes a solution which make them adaptable and configurable by employing a meta-architecture, and illustrates that our framework is suitable to deploy an adaptable Web server beyond a static and monolithic server of today. Adding a meta-architecture to a Web servers opens its execution environment to a variety of requirements and allows it to continually evolve.

A Sample code

```
aServer.reify("SysController")
aServer.reify("Acceptor")
aMetaSpace = aServer.metalevel()
anAcceptor
  = aMetaSpace.getObject("Acceptor")
aMetaMetaAcceptor
 = anAcceptor.metalevel()
aMetaMetaAcceptor.
 change(SingleThreadedAcceptor)
.....
aSysController
 = aMetaSpace.getObject("SysController")
# If the server's load is
#  beyond the threshold
If(aSysController.loadStatus() > threshold):
 # a concrete metaobject is changed
 if(aMetaMetaAcceptor.current().
     isInstanceOf(SingleThreadedAcceptor)):
    aMetaMetaAcceptor.
     change(ThreadedAcceptor)
```

References

[1] D. Bobrow R. Gabriel J. White. CLOS in Context -The Shape of the Design Space. In A. Paepcke, editor, *Object-Oriented Programming- The CLOS Perspective*, page Chapter 2. MIT Press, 1993.

[2] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA '87*, pages 147–155, 1987.

[3] D. P. Friedman and M. Wand. Reification: Reflection without metaphysics. In *Symposium on LISP and Functional Programming*, 1984.

[4] J. des Rivieres and B. C. Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984.

[5] B. C. Smith. Reflection and semantics in lisp. In *Proceedings of ACM POPL '84*, pages 23–35, 1984.

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

[7] H. Masuhara S. Matsuoka T. Watanabe and A. Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 127–147, 1992.

[8] B. Foote. Reflective Facilities in Smalltalk-80. In *OOPSLA'89*, 1989.

[9] F. Rivard. Smalltalk: a Reflective Language. In *Reflection'96*, 1996.

[10] S. Chiba. A metaobject protocol for C++. In *OOPSLA95 Proceedings*, 1995.

[11] B. Growing and V. Cahill. Meta-Object Protocols for C++: The Iguana Approach. In *Proceedings of Reflection '96*, pages 137–152, San Francisco, USA, 1996.

[12] J. Kleinoder and M. Golm. Metajava: An Efficient Run-Time Meta Architecture for Java. In *International Workshop on Object-Orientation in Operating Systems (IWOOOS'96)*, 1996.

[13] T. Nishigaya. Design of Multi-Agent Programming Libraries for Java, 1997. available at http://www.fujitsu.co.jp/hypertext/free/kafka/paper/.

[14] Object Design Inc. *ObjectStore User's Guide r3.0*, 1994.

[15] L. H. Rodriguez Jr. Coarse-Grained Parallelism Using Metaobject Protocols. Master Thesis, MIT, 1991.

[16] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings OOPSLA '92*, 1992.

[17] C. Zimmermann and V. Cahill. It's your choice - on the design and implementation of a flexible metalevel architecture. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems, IEEE*, 1996.

[18] IBM. *SOMobjects Developer Toolkit ver. 2.0*, 1993.

[19] Java Reflective Broker. http://andromeda.cselt.it/users/g/grasso/

[20] S. Crawley S. Davis J. Indulska S. McBride and K. Raymond. Meta-meta is better-better! In *Interna-*

tional Working Conference on Distributed Applications and Interoperable Systems (DAIS'97), 1997.

[21] S. Demeyer P. Steyaert and K. D. Hondt. Zypher: the Role of Meta-Object Protocols in Open Hypermedia Systems. In *Hypertext'97*, 1997.

[22] S. Demeyer. Zypher: Tailorability as a Link from Object-Oriented Software Engineering to Open Hypermeria. Ph.D Thesis, Vrije University, 1996.

[23] S. Demeyer. The Zypher Meta Object Protocol. In *the 2nd Workshop on Open Hypermedia Systems - Hypertext'96*, 1996.

[24] W. L. Hursch and C. V. Lopes. Separation of Concerns. Technical report, NU-CCS-95-03, Northeastern University, 1995.

[25] J. McAffer. Engineering the Meta Level. In *Proceedings of Reflection '96*, 1996.

[26] E. Gamma R. Helm R. Johnson and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.