A SMART HTTP COMMUNICATOR: SMACH

Yosuke Murakami, Yusuke Takada and Makoto Oya
Graduate School of Information Science and Technology, Hokkaido University, N14, W9, Kita-ku, Sapporo, 060-0814, Japan

Abstract: Most developments and applications of current eBusiness systems focus on large enterprise systems. HTTP based full implementation of the Internet/Web protocol stack is used in such environment. On the other hand, small businesses or community works require simpler and smaller implementation not assuming big computer resource. This requirement exists also in other business-like environments such as robot communication, mobile devices and ubiquitous environment. Essential requirements for HTTP infrastructure are (1) to support HTTP protocol, and (2) to enable peer-to-peer communication. Currently existing infrastructures such as Apache and Mozilla, however, support more and do not fit to smaller and simpler applications. They were originally developed considering Web document access applications and have large functionality unnecessary for eBusiness communication. To solve this problem, we developed a new open source middleware essentially supporting HTTP and having both server and client feature called SMACH (Smart Communicator for HTTP). SMACH is designed based on three policies: small, bidirectional, and HTTP protocol focused. It supports minimum but enough HTTP communication function and symmetric APIs for server and client applications, having multi-threading based architecture with persistent TCP/IP connection pursuing maximum performance in small memory environments. SMACH can communicate with major existing products including Apache, IIS, Mozilla and IE, achieving high performance and small executable size. This paper provides its development policy, function, architecture, and results of interoperability tests and performance evaluation.

Key words: HTTP, Web service, equal distributed system, bidirectional communication, Inter-organizational communication

1. INTRODUCTION

Most developments and applications of current eBusiness systems focus on large enterprise systems including complex trade systems, supply chain management systems, sales force systems. HTTP[1] based full implementation of the Internet/Web protocol stack is used in such environment. On the other hand, small businesses or community works require simpler and smaller implementation not assuming big computer capability. This requirement exists also in other business-like environments such as robot communication, mobile devices and ubiquitous environment.

eBusiness basic protocol is HTTP[1]. Essential requirements for the infrastructure are (1) to support HTTP protocol, and (2) to enable peer-to-peer communication. Currently existing infrastructures such as Apache and Mozilla, however, do not fit to smaller and simpler applications. They were originally developed mainly for Web document access applications and having large functionality for document processing such as HTML handling and rendering, which makes whole system too large for smaller applications. In addition, each infrastructure supports either server function or client function, resulting application development difficult, because eBusiness applications have both server and client functions as their nature.

To solve the above problem, we developed a new open source middleware essentially supporting HTTP and having server and client feature called SMACH (Smart Communicator for HTTP). This paper provides its development policy, function, architecture, and results of interoperability tests and performance evaluation.

SMACH is designed based on three policies: compact build, bidirectional, and HTTP protocol focused. It supports minimum but enough HTTP communication function and symmetric APIs for server and client applications, having multi-threading based architecture with persistent TCP/IP connection pursuing maximum performance in small memory environment. SMACH can communicate with major existing products including Apache[4], IIS, Mozilla, IE, and language clients such as Java. Client is 36-65% faster than Java client library, server is 16-60% faster than Apache/CGI. Size of executable is half of thttpd [5] and 1/26 of Apache.

2. DEVELOPMENT POLICY

SMACH was designed and developed based on the following policies:

- To minimize its functionality enough to support HTTP protocol execution.
- To prepare enough APIs for application programs to implement higher-layer protocol and functions, e.g., document processing, file handing, SOAP processing, messaging.
- To Support both client and server function.
- To prepare symmetrically designed APIs for client and server, which make implementation of bidirectional communication easier.
- To comply with latest specification, HTTP/1.1.

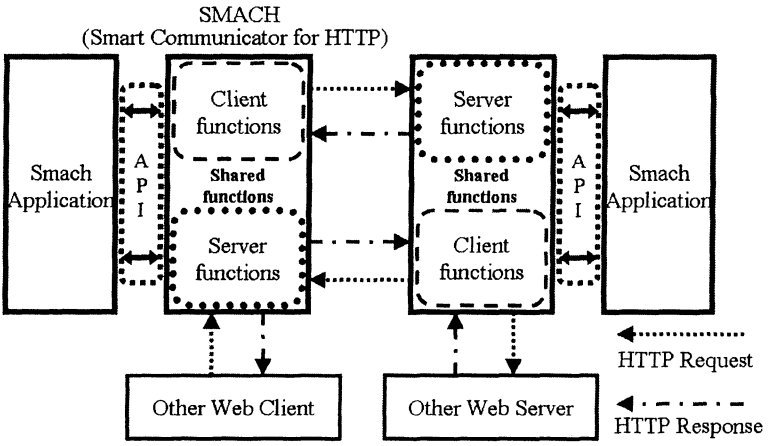

Figure 1. SMACH Overview

Figure 1 shows overview of SMACH. SMACH has client and server function and communicates using HTTP/1.1. Applications use SMACH functions through using originally designed APIs. In addition, SMACH client and server can communicate with existing HTTP client and server software. Client function will be used by library in three patterns:

- Execution as library function
- Execution as SMACH child process
- Execution as another process

3. FUNCTION AND API

3.1 Functions

Table 1 shows functions supported by SMACH. Following functions are excluded based on the development policy.

- Responding a file specified by URL when client requests it.
- Rendering and displaying HTML documents.

Table 1. Client and Server Function

Function	support
Server	
parsing request line and header field	*supported*
setting status line and header field	*supported*
getting file specified by URL	*API*
CGI	*API*
BASIC Authentication	*supported*
Logging	*supported*
Client	
setting request line and header field	*supported*
parsing status line and header field	*supported*
rendering the HTML file	*API*
Common	
Content negotiation	*supported*
Persistent Connection	*supported*
Chunked Encoding	*supported*
Cache	*API*
Cookie	*API*

3.2 Application Program Interface

This section mentions about two major design results, symmetry between client APIs and server APIs, and protocol hiding.

3.2.1 Symmetry between client APIs and server APIs

SMACH realizes both HTTP client and HTTP server functions in single software. Sending and receiving messages in HTTP communication using symmetric APIs. *Table 2* shows major APIs.

Table 2. SMACH APIs

Client API
API for creating request message (excerpt)
sma_set_request_line(void *h, int method, char *url)
sma_set_req_content_language(void *h, char *language)
API for sending request message
sma_send_request(void *h, char *body, int size)
API for receiving response message
sma_receive_response(void *h, char *body, int size)
API for parsing response message
sma_get_status_code(void *h)
sma_get_res_content_type(void *h, char *language);
Server API
API for receiving request message
sma_receive_request(void *h, char *body, int size)
API for parsing request message
sma_get_abs_path(void *h, char *abs_path)
sma_get_req_content_type(void *h, char *language);
API for creating response message (excerpt)
sma_set_status_line(void *h, int method, int status_code)
sma_set_res_content_language(void *h, char *language)
API for sending response message
sma_send_response(void *h, char *body, int size)

3.2.2 Hiding Protocol and Memory Management

1. Hiding message form

HTTP message is consisted of startline, header field, and body field. They have to follow the message form defined in HTTP/1.1. This form is hidden from the application using SMACH API. The Application can create HTTP message if it passes only data to SMACH. In parsing message, applications need not to parse HTTP messages, but only to get values.

2. Hiding message flow

Application is hidden from actual data flow. This enables the application to send and receive messages at any time in any size.

3. Hiding memory management

The data used in sending and receiving message don't have to be managed by the application. Memory allocation and free memory is managed by SMACH. So memory management is hidden from application.

Followings are examples of client and server application using API (attaching prefix, "sma_").

- Example Of Client Application

```
int client_application(void) {
   void *handle;
   int size = 1024;
 char res_body[1024];
 char req_body1[] = "<?xml version = ...>";
 char req_body2[] = "<soapenv:Envelope...>";
 char req_body3[] = "</soapenv:Envelope>";
 char uri[] = "www.smach.com";
 char language[10] ;
   /*establishing connection*/
sma_bind(&handle, uri, 80);
   /*initializing*/
   sma_initialize(handle);
   /*process for request message*/
   /*setting request line*/
   sma_set_request_line(handle,POST,"/service");
   /*setting request header*/
   sma_set_req_content_language(handle, "ja");
   /*setting request body*/
   sma_send_request(handle,  body1, strlen(body1));
   sma_send_request(handle,  body2, strlen(body2));
   sma_send_request(handle,  body3, strlen(body3));
   sma_send_request(handle,  NULL, 0);
   /*process for response message*/
   /*getting response header*/
   sma_get_res_content_type(handle, language);
/*getting response body*/
   while(1) {
rval = sma_receive_response(handle, body, size);
       if (rval == 0) break;
   }
/*closing connection*/
sma_close(handle);
return (0);
}
```

- Example of Server Application

```
int server_application(const void *handle) {
   int rval = 0;
   char language[10];
int size = 1024;
char req_body[1024];
char res_body1[] = "<?xml version = ...>";
char res_body2[] = "<soapenv:Envelope...>";
char res_body3[] = "</soapenv:Envelope>";
   /*process for request message*/
   /*getting request header value*/
sma_get_accept_language(handle, &language);
   /*getting request body message*/
   while(1) {
rval = sma_receive_request(handle, body, size);
       if (rval == 0) break;
   }
   /*process for response message */
   /*setting status line*/
   sma_set_status_line(handle, 200);
   /*setting response header */
   sma_set_res_content_language(handle, "ja");
   /*setting response body message*/
   sma_send_response(handle, res_body1,strlen(res_body1));
   sma_send_response(handle, res_body2,strlen(res_body2));
```

```
    sma_send_response(handle, res_body3,strlen(res_body3));
    sma_send_response(handle, NULL, 0);
    return(0);
}
```

4. SMACH ARCHITECTURE

4.1 Architecture Design

Architecture is designed on the following policies.

- Compact Size

By galvanizing functions from HTTP/1.1 (RFC2616), and separating them for client and server, we picked out common processes. They enable SMACH to cut off redundant processes between modules, and realize compact size.

- Loose Coupling Between Modules

Modules are designed not to influence intricately, but to have simple relation each other. They are also designed to be loosely coupled.

On the basis of these, we designed as *Figure 2*. The above is client architecture, and the below is server architecture. *Table 3* shows roles of each module.

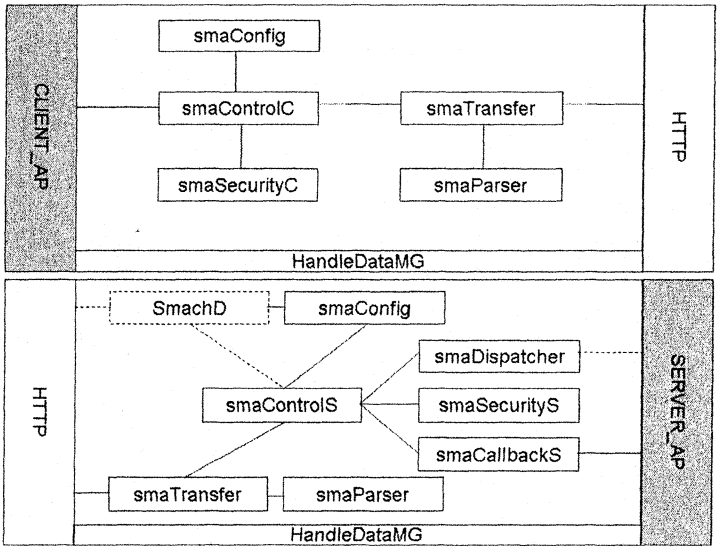

Figure 2. Architecture

Processes of both client and server system is controlled by smaControlC/S. This control part becomes the hub, and makes it easy to enhance when new processing is added. HTTP has symmetrical characters in sending and receiving, and structure of HTTP message. Therefore common processing becomes possible, and SMACH client and server are implemented by them. Other parts that can't be shared are used in original modules.

Table 3. Module

Client Module	
SmaControlC	Client Protocol Control
SmaSecurityC	Process Of security
Server Module	
SmachD	Establishment of TCP Connection, Forking Thread
SmaControlS	Server Protocol Control
SmaDispatcher	Application Dispatch
SmaCallbackS	Control of Callback
SmaSecurityS	Process Of security
Common Module	
smaTransfer	Control Of sending and Receiving
smaParser	Parser
smaConfig	Setting Up Configuration
HandleDataMG	Getter And Setter of data

4.2 Module Implementation

- Server Multiplexing

To respond to two or more requests, the server is multiplexed. There are three ways to realize multiplexing.

- Using process
- Using thread
- Using both process and thread

If something goes wrong with the parent process or thread, all threads come under the influence of the problem. So threads could compromise stability. But using thread has following merits.

- small consumption of memory
- short time in starting and stopping

SMACH uses thread from small and lightweight point of view. *Figure 5* shows the mode of multiplex processing. SmachD forks smaControlS as thread. Therefore smaDispatcher, smaSecurityS, smaCallbackS, smaTransfer, smaParser, HandleDataMG are multiplexed.

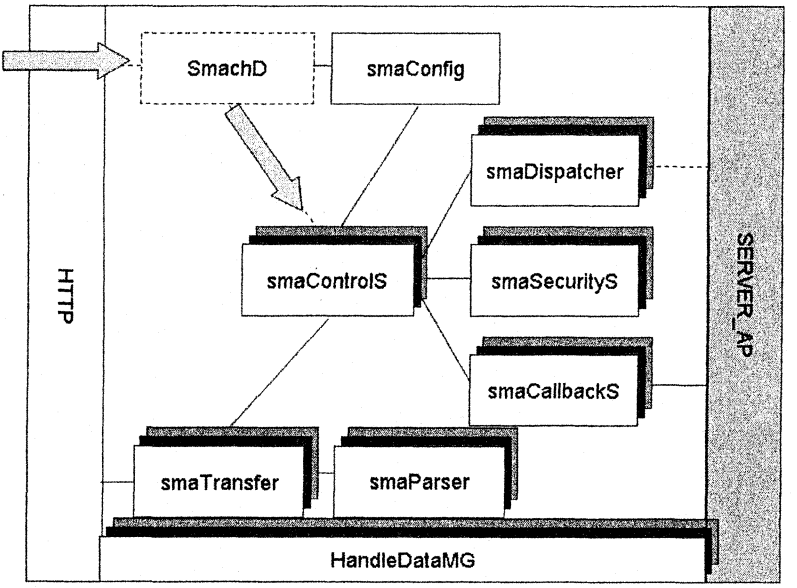

Figure 3. Server Multiplexing

- Callback from application

Server application which is implemented on SMACH Server is called from SMACH as library. But Server application needs to use SMACH inner functions to respond. SMACH uses Callback functions to resolve this problem. SMACH server passes the Function Vector including pointer to SMACH Server API when smaDispatcher calls for Server application. It realizes callback. smaCallbackS makes Function Vector, and treats callback processes. *Figure 4* shows callback process.

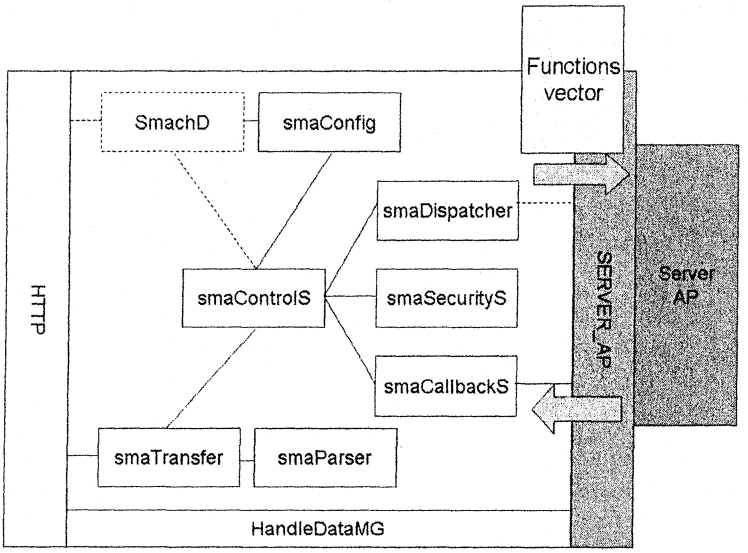

Figure 4. Callback Process

- Persistent connection

SMACH supports persistent connection[6]. SMACH can realize closing TCP connection when "Connection: Close" Header is attached. This enables SMACH Server to kill the thread in appropriate time, so this achieves small size structure.

5. EVALUATION

This section shows evaluation results of developed SMACH. The following configuration was used:

Server:

- OS: Fedora Core 3(2.6.10-1.760-FC3)
- Memory : 384MB
- CPU: Athlon(tm) XP1700 + 1466.601MHz

Client:

- OS : Redhat 9(2.4.20-8)
- Memory : 192MB
- CPU: Pentium III Coppermine (597.416MHz)

Fedora Core 3 and Redhat 9 were elected as OS based on the fact that SMACH chiefly targeted Linux, which is widely used well.

5.1 Interoperability Test

The following three tests were done.

TEST1: Connected experiment of client AP and server AP that uses SMACH API.

TEST2: Connected experiment of client AP that uses SMACH API and existing server (Apache, IIS).

TEST3: Connected experiment of existing client (Mozilla, Internet Explorer, and JAVA HTTP class) and server AP that uses SMACH API.

Table 4. Interoperability

	Client	Server	Result
TETS 1	SMACH –C+ AP	SMACH-S + AP	Connected
TEST 2	SMACH-C + AP	Apache	Connected
	SMACH-C + AP	IIS	Connected
TEST 3	Internet Explorer	SMACH-S + AP	Connected
	Mozilla	SMACH-S + AP	Connected
	Java HTTP Class	SMACH-S + AP	Connected

The connectivity of SMACH could be proven from the result of TEST1.Moreover; the connectivity with existing Server/Client can be proven from the result of TEST2 and TEST3.

5.2 Performance

5.2.1 Client performance

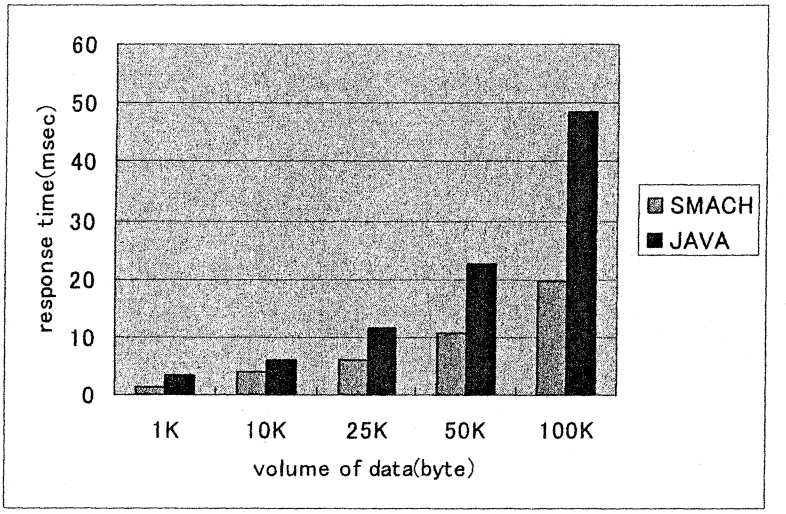

Figure 5. Client Performance

Figure 5 shows the result of measuring the performance with Java. The server is Apache. The performance of SMACH is 36-65% faster than that of Java. Java encodes data to Unicode once. I speculate that it becomes an overhead. Moreover, Java should have Java-VM, so there is a not suitable part for small equipment.

5.2.2 Server performance

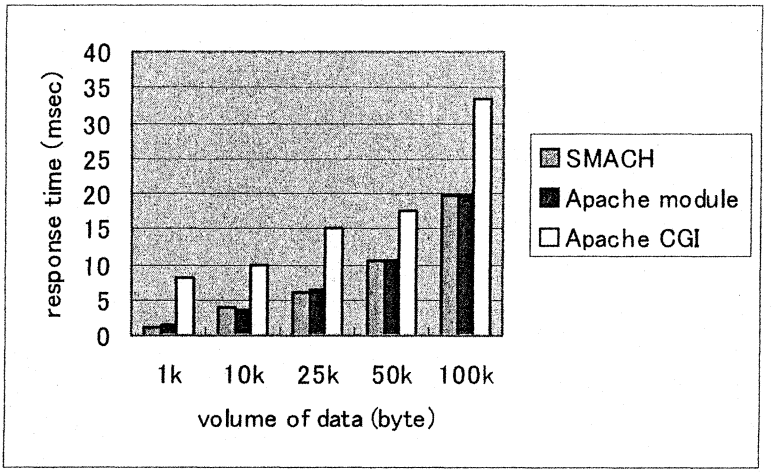

Figure 6. Server Performance

Figure 6 shows the result of measuring the performance of the server. The client is SMACH. The performance of SMACH is 16-60% faster than Apache CGI, and at the same level as the Apache modules. The overhead of CGI start cannot be disregarded when comparing it with Apache CGI because SMACH AP is called from SMACH statically. But it is a comparison result profitable because generally CGI is used when server AP is made with Apache. A general engineer has a difficult part by the AP development as the Apache module. Therefore, it can be said that there is an advantage in which SMACH is used from the comparison result with the Apache module.

5.3 Program Size

The comparison among the number of source files, the number of source lines, and the size of the execution file were done for Apache and thttpd. *Table 5* shows the result. Because the function of SMACH, Apache, and thttpd is different, SMACH is not necessarily simply smallest. However, it can be said that SMACH was able to achieve the target policy, considering containing both the client and the server.

Table 5. Performance Comparison Concernig Small Size

Software	Version	Source files	Source lines	exe (byte)
SMACH	1.0b	20	6697	46,385
thttpd	2.25b	21	11129	85,596
Apache	2.0.53	608	264505	1,222,809

6. CONCLUSIONS

SMACH has been designed based on three policies: small build, bidirectional, and HTTP protocol focused. It supports number of, yet enough HTTP communication function and symmetric APIs for server and client applications. It has multi-threading based architecture with persistent TCP/IP connection pursuing maximum performance in small memory environments. Interoperability test shows SMACH can communicate with major existing products including Apache, IIS, Mozilla, IE, and language clients such as Java. As for it performance, client is 36-65% faster than Java client library, the server is 16-60% faster than Apache/CGI. The developed implementation has achieved enough compactness HTTP infrastructure used in small eBusiness systems.

REFERENCES

1. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
2. Fielding, R., Gettys, J., Mogul, J., Frystyk, H. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, January 1997.
3. Berners-Lee, T., Fielding, R. and H. Frystyk, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.
4. The Apache Team, "The Apache Software Foundation" http://www.apache.org/
5. Jef Poskanzer. "thttpd - tiny/turbo/throttling HTTP server. Acme Laboratories", May 2005. http://www.acme.com/software/thttpd/
6. Joe Touch John Heidemann and Katia Obraczka, "Analysis of HTTP Performance", 1998
7. Andrew S. Tanenbaum., Maarten van Steen, "Distributed Systems"
8. Takada, Y., Murakami, Y. and Oya, M., "Design and Development of SMACH (Smart Communicator for HTTP)", IEICE/IPSJ Information Technology Letters, vol.4, 2005 (to appear). (in Japanese)
9. Murakami, Y., Takada, Y. and Oya, M., "Architecture of SMACH (Smart Communicator for HTTP)", IEICE/IPSJ Information Technology Letters, vol.4, 2005 (to appear). (in Japanese)