

# **An efficient and lightweight embedded Web server for Web-based network element management**

By Hong-Taek Ju,\* Mi-Joung Choi and James W. Hong

---

---

*An Embedded Web Server (EWS) is a Web server which runs on an embedded system with limited computing resources to serve embedded Web documents to a Web browser. By embedding a Web server into a network device, it is possible to provide a Web-based management user interface, which are user-friendly, inexpensive, cross-platform, and network-ready. This article explores the topic of an efficient and lightweight embedded Web server for Web-based network element management. Copyright © 2000 John Wiley & Sons, Ltd.*

## **Introduction**

**A**s the World-Wide Web (or Web) continues to evolve, it is clear that its underlying technologies are useful for much more than just browsing the Web. Web browsers have become the *de facto* standard user interface for a variety of applications. This is because Web browsers can provide a GUI interface to various client/server applications without a client application. An increasing number of Web technologies can also be applied to network element management.

Web-based network element management gives an administrator the ability to configure and monitor network devices over the Internet using a Web browser. The most direct way to accomplish this is to embed a Web server into a network device and use that server to provide a Web-based management user interface constructed using HTML,<sup>5</sup> graphics and other features common to Web browsers.<sup>4</sup> Information is provided to the user by simply retrieving pages, and information is sent back to the device using forms that the user completes. Web-based management user interfaces (WebMUIs) through embedded Web servers have

---

*Hong-Taek Ju received his BS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 1989 and MS degree in Computer Science and Engineering from Pohang University of Science and Technology (POSTECH) in 1991. From 1991 to 1997, he worked at DAEWOO Telecom. Currently, he is a PhD candidate in the Department of Computer Science and Engineering, POSTECH. His research interests include distributed processing and network management.*

*Mi-Joung Choi received her BS degree in computer science from Ewha Womans University. She is currently a graduate student in the Department of Computer Science and Engineering, POSTECH. Her research interests include Web-based network management and policy-based network management.*

*James W. Hong is an associate professor in the Department of Computer Science and Engineering, POSTECH, Pohang, Korea. He has been with POSTECH since May 1995. Prior to joining POSTECH, he was a research professor in the Department of Computer Science, University of Western Ontario, London, Canada. Dr Hong received BSc and MSc degrees from the University of Western Ontario in 1983 and 1985, respectively, and PhD degree from the University of Waterloo, Waterloo, Canada in 1991. He has been very active as a participant, program committee member and organizing committee member for IEEE CNOM sponsored symposiums such as NOMS, IM, DSOM and APNOMS. For the last several years, he has been working on various research projects on network and systems management, which utilize Web, Java and CORBA technologies. His research interests include network and systems management, distributed computing and traffic engineering and planning. He is a member of IEEE, KICS, KNOM and KISS.*

\*Correspondence to: Hong-Taek Ju, DPNM Laboratory, Department of Computer Science and Engineering, Pohang University of Science and Technology, San 31, Hyojadong, Namgu, Pohang, Korea.  
Email: juht@postech.ac.kr

many advantages: ubiquity, user-friendliness, low development cost and high maintainability.

Embedded Web Servers (EWSs)<sup>1-3</sup> have different requirements, such as low resource utility, high reliability, security and portability, for which general Web server technologies are unsuitable. Above all, due to resource scarcity in embedded systems it is important to make EWSs efficient and lightweight. There are also design issues such as HTTP<sup>6,7</sup> and embedded application interface. In embedded Web server usage, Java applets can play an important role for making embedded Web servers truly useful for management applications.

In this paper, we present our research to develop an efficient and lightweight EWS for Web-based network element management. We first propose the architecture of an embedded Web server that can provide a simple but powerful application interface for network element management. We then present the design and implementation of POS-EWS, an embedded Web server that we have developed for Web-based network element management. Finally, we present the results of POS-EWS's performance and EWS optimization methods for making an efficient and lightweight EWS. There are many commercial EWS products on the market for Web appliances, but our work is a good example of making an efficient EWS suitable for Web-based network element management.

The organization of the paper is as follows. In the second section we present an overview of EWSs, and describe the EWS-WebMUI and EWS requirements. In the next two sections we present the EWS design and implementation of our proposed EWS architecture, respectively. In the fifth section we evaluate POS-EWS's performance and explain our methods for optimizing POS-EWS. In the sixth section we briefly investigate the available offerings of EWS products focusing on their features and the approximate code size needed. In the final section we summarize our work and discuss possible future work.

## Embedded Web Servers and Web-based Management User Interface

In this section, we briefly overview embedded Web servers, comparing them with general Web servers. Also, we describe the EWS-WebMUI and

EWS requirements that we must consider during development.

### —Embedded Web Server—

General Web servers, which were developed for general-purpose computers such as NT servers or Unix and Linux workstations, typically require megabytes of memory, a fast processor, a pre-emptive multitasking operating system, and other resources. A Web server can be embedded in a device to provide remote access to the device from a Web browser if the resource requirements of the Web server are reduced. The end result of this reduction is typically a portable set of code that can run on embedded systems with limited computing resources. The embedded system can be utilized to serve the embedded Web documents, including static and dynamic information about embedded systems, to Web browsers. This type of Web server is called an Embedded Web Server (EWS).<sup>1-3</sup>

EWSs are used to convey the state information of embedded systems, such as a system's working statistics, current configuration and operation results, to a Web browser. EWSs are also used to transfer user commands from a Web browser to an embedded system. The state information is extracted from an embedded system application and the control command is implemented through the embedded system application. In many instances, it is advisable for embedded Web software to be a lightweight version of Web software. For network devices, such as routers, switches and hubs, it is possible to place an EWS directly in the devices without additional hardware.

### —EWS-WebMUI—

*WebMUI and EWS-WebMUI*—The rapid proliferation of Web-based management makes it clear that schemes using HTTP and standard Web browsers provide benefits to both users and developers. Most Web-based management applications provide an interface to the status reporting, configuration, and control features of managed objects. Several such Web management approaches have been proposed thus far. Sun Micro-systems

is pushing its Java Management eXtension (JMX)<sup>8</sup> and Microsoft, Compaq and Intel are touting Web-based Enterprise Management (WBEM).<sup>9</sup> However, both approaches are sufficiently complex that many small network devices would find it very difficult to implement them.

By embedding a Web server, Web documents and management applications into an embedded system, a Web-based Management User Interface (WebMUI) can be provided directly to system administrators (an EWS-WebMUI). Therefore, an EWS-WebMUI is the direct result of embedding a Web server, Web documents and management applications into an embedded system. The Web documents give a display form of management information, a collection of manageable data that is monitored or configured for managing an embedded system.

---

---

***B***y embedding a Web server in a network device, the device can serve up Web documents to any Web browser.

---

---

*Advantages of EWS-WebMUI*—By embedding a Web server in a network device, the device can serve up Web documents to any Web browser. These Web documents become the GUI interface to the device. Consequently, few techniques need to be learned for management interface of the new device. Because Web documents can be displayed directly from files that may be edited with either ordinary text editors (for HTML) or specialized authoring tools, it is easy to quickly prototype the look and feel of a WebMUI. Alternatives can be explored and reviewed without ever actually embedding the interface into the system. If the mechanisms used to embed the interface are properly designed, changes made to the Web documents can be quickly imported to the embedded system with little or no change to the management application code. This translates into the potential for better, more useful interfaces in less development time.

EWS-WebMUIs also have the advantage of a platform independent graphical user interface. The SNMP<sup>10</sup> management scheme usually consists of an SNMP based Network Management System (NMS). Most NMSs give users the option of using

a graphical interface based on MS-Windows or X-Window as opposed to the command line interface. Most NMS users demand specific platforms, such as OS, or computer hardware in order to install and execute the NMS. By contrast, an EWS-WebMUI does not demand any specific platform because Web browsers are available for virtually all computers.

While the EWS-WebMUI concept appears straightforward and perhaps even commonplace, the implications are deeper than first appears. By placing the GUI within the device itself, the device is now self-contained and need not be matched with a corresponding version of a user management application program; the problems inherent in providing separate user interface software disappears; there is no risk of the user having an old version of the user application software that does not support all the features of latest devices; and users can upgrade some systems to the latest release without having to change the management software they use because the necessary part of upgrade is only the EWS-WebMUI. Consequently, there are no porting or distribution efforts for the user application program.

Additionally, it is usually possible to upload Web documents to the embedded system so that a device can receive an upgrade to its management interface from a remote location on the network. This feature makes it possible for developers to upgrade all devices over the network from the one point. High maintainability for EWS-WebMUI is a direct result of ease of Web document development and one point upgrade.

## Design

In this section, we present our design result that includes a functional architecture and a process structure of EWS.

### —EWS Architecture—

We have designed an EWS that consists of five parts: an HTTP engine, an application interface module, a virtual file system, a configuration module, and a security module. The design architecture of our EWS is illustrated in Figure 1.

The most important part of the EWS is an HTTP engine, which serves a client's request. The

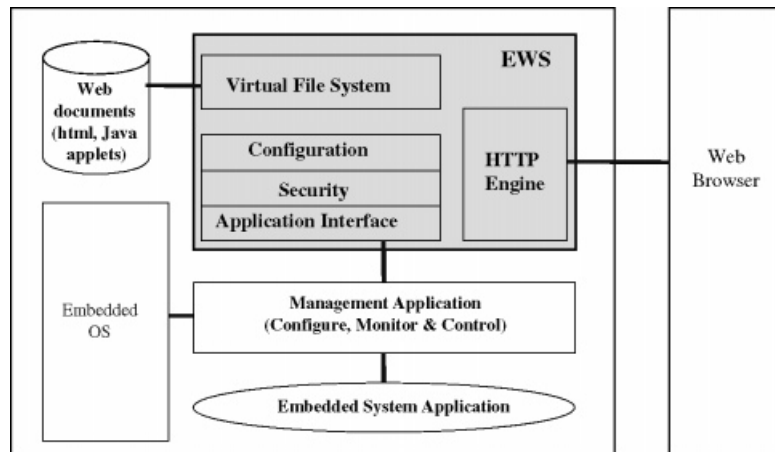


Figure 1. EWS architecture

minimum requirement for an HTTP engine is that it must be compliant with HTTP specifications. Unlike general Web servers that start a new thread or process whenever a new connection is made, normally an HTTP engine supports multiple simultaneous users while running as a single process. The number of processes that the server requires can impact on both RAM usage, due to the stack space per task, and CPU usage. Next, we explain an HTTP transaction process using a state transition diagram.

In an EWS, the application interface module enables developers to add new management functionality. With any off-the-shelf Web authoring tool, it can merge Web documents with management application programs to generate specific dynamic management information. This module provides mechanisms for interacting with the embedded application. Embedded Web server software must provide mechanisms for the embedded application to generate and serve Web pages to the browser, and to process HTML form data submitted by the browser. One possible solution is modeled after the Common Gateway Interface (CGI)<sup>15</sup> found in many traditional Web servers. In this model, each URL<sup>16</sup> is mapped to a CGI script that generates the Web page. In a typical embedded system, the script would actually be implemented by a function call to the embedded application. The application could then send raw HTML or other types of data to the browser by using an interface provided by the embedded Web server software.

Another solution is to use Server-Side Include (SSI).<sup>5</sup> With this approach, Web pages are first developed and prototyped using conventional Web authoring tools and browsers. Next, proprietary markup tags that define server-side scripts are inserted into the Web pages. The marked-up Web pages are then stored in the device. When a marked-up Web page is served, the embedded Web server interprets and executes the script to interface with the embedded application. In order to offload substantial Web server processing from the embedded system at run time, a preprocessor tool can be used. The preprocessor enables sophisticated dynamic Web-page capabilities by performing complex tasks up front and generating an efficient and tightly integrated representation of the Web pages and interfaces in the embedded system.

The virtual file system (VFS) provides the EWS with virtual file services, which are *file\_open* for opening the file, *file\_read* for reading the file, and *file\_close* for closing the file after reading. The file system has a data structure storing file information such as file size, last modified date, etc. The data structure for an HTML documents file needing dynamic information must store the pointer of the script and the function name called by the script. To construct this VFS we need a Web compiler. The Web compiler supports any format, such as Java, GIF, JPEG, PDF, TIFF, HTML, text, etc. It compiles these files into intermediate C-codes and then compiles & links them with the Web Server codes. The resulting structure does not require a

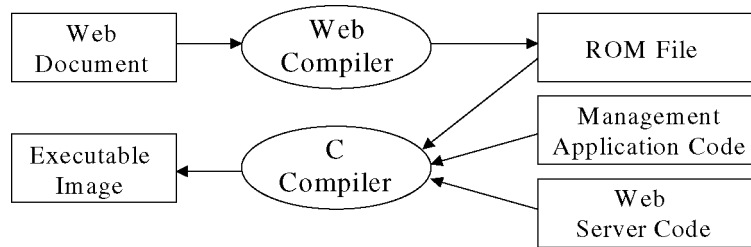


Figure 2. Process of a web server making a virtual file system

file system, yet the files are organized like in a file system—a virtual file system. The Web browser traverses this virtual file system just as if it were an actual file system. Figure 2 illustrates the process of a Web server making a virtual file system.

Security is an important concern in network management. Therefore, an EWS generally has a security and/or configuration module. Security is accomplished by defining security realms on a server and username/password access to each realm. When a request comes in for an object in a protected realm, the server responds with a response code of 401 (Unauthorized). This will force a browser to prompt the user for a username/password pair. The original object request will be resubmitted with the username/password, base-64 encoded, in the request header. If the server finds the login correct, then it will return the requested object, otherwise, a 403 forbidden response is returned. The configuration module provides the administrator with the functionality

to set the embedded Web server configuration from any standard Web browser. The configuration environment variables passed at startup define the number of concurrent connections, socket port, own host name, root file path, default 'index', inactivity timeout and time zone. Common usage of Web browsers makes it a more important matter to protect abnormal access to the sensitive information of network devices, especially those that involve equipment configuration or administration.

—EWS Process Structure—

We designed an EWS as a finite state machine (FSM), which processes an HTTP request in a sequence of discrete steps. Figure 3 shows the state transition diagram of the HTTP engine. In order to support multiple connections in a single thread environment, multiple finite state machines are run by a scheduling system which uses a lightweight

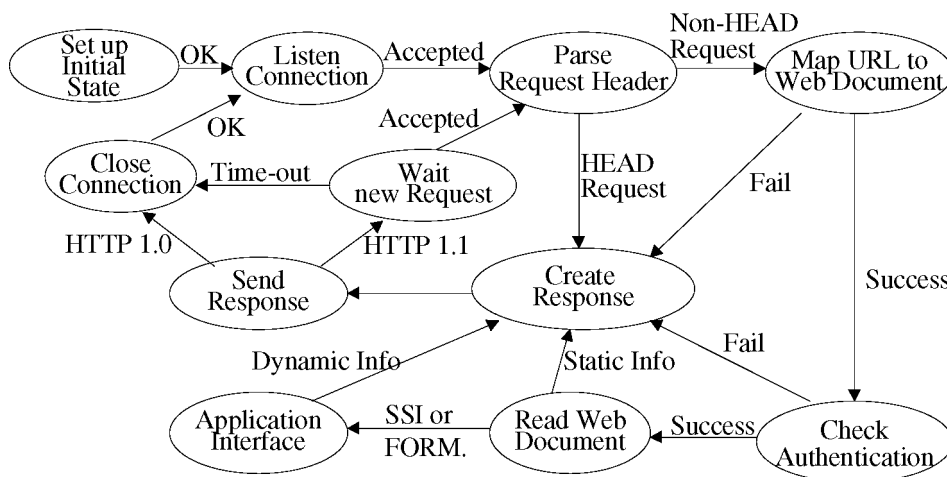


Figure 3. EWS finite state machine

task structure. It consists of a pointer to the function being run, a variable holding the state in the FSM, and a flag indicating whether the FSM can be run or blocked. The scheduling system allocates an available FSM for an accepted connection, checks each FSM to see if it is blocked or runnable and, if it is runnable, moves the FSM one step.

Each state in an FSM can check for the presence of data that is ready to be processed at the entry point; if none is ready, the FSM can block itself until data arrives. When data becomes available at the entry point, the FSM can then be unblocked so its handler can perform the task of state, and turn over the result to the next state by changing the state flag and pointer to the handler.

The following list describes the behavior of each state.

- *Set up Initial State*: Set up the task structure for an FSM. The task of this state is performed at the server initial time for all FSMs.
- *Listen Connection*: Check to see if any request is allocated to this FSM.
- *Parse Request Header*: Read the HTTP message, parse the HTTP header and store the parsing result.
- *Map URL to Web Document*: Determine type of application interface and store a pointer to the handler.
- *Check Authentication*: Force authentication of the user upon the URL and user name/password.
- *Read Web Document*: Read Web document from virtual file system.
- *Application Interface*: Call application function upon the URL.
- *Create Response*: Create HTTP response header.
- *Send Response*: Send HTTP header and Web document.
- *Wait new Request*: Wait for a new HTTP request from the same TCP connection if the received request says HTTP/1.1 support.
- *Close connection*: Close the TCP connection.

### —EWS Extended Architecture for EWS WebMUI—

As mentioned earlier, only the scheme of HTTP and HTML is client-driven. One side effect is that once a page is served to the Web browser it becomes static: it does not change even if management data

has been altered on the server side. For a user seeking a device, which is dynamic, this is not very appealing.

To be useful for management applications, pages must be constructed dynamically so that real-time data can be placed alongside static HTML in the same page. For common types of real-time data, such as traffic monitoring and CPU load, users want to see data displayed in a dynamic graphic form. This is where Java applets<sup>17</sup> and/or CORBA<sup>18,19</sup> objects come in. Java applets are automatically downloaded by a browser as separate applications that get used within an HTML page. Once the applet is loaded, it has control over where it gets its data and how to display or manipulate that data. Java applets by nature are cross-platform and will act the same within any browser.

Simple Network Management Protocol (SNMP)<sup>10,20</sup> is the most widely used management framework for managing network devices on the Internet. Its protocol is simple enough that it can be implemented in small platforms without much difficulty. Now most network devices are equipped with an SNMP agent. With integration of SNMP and the EWS-WebMUI, the advantages of EWS-WebMUI are preserved without the giving up the SNMP implementations.

The EWS extended architecture gives an integration platform. Figure 4 illustrates the EWS extended architecture for an EWS-WebMUI. The ultimate solution is to make the EWS-WebMUI a user interface to communicate with the network device via SNMP. Java implementation of SNMP mediates between an SNMP agent and a Web browser. The Java SNMP source code is written and compiled to produce a Java SNMP applet. This applet is stored in a network device and is transferred by the EWS to the browser over the network at run time. After loading on the JVM of a browser, the Java SNMP applet communicates with the SNMP agent in the network device and enables the administrator to control and monitor the network device through the browser, using SNMP messages. In addition to the Java SNMP applet, the network device in this scenario must store at least one HTML document containing reference to the applet. The HTML document is loaded into the Web browser and then the Web browser would automatically request the Java SNMP applet referenced by the previous HTML.

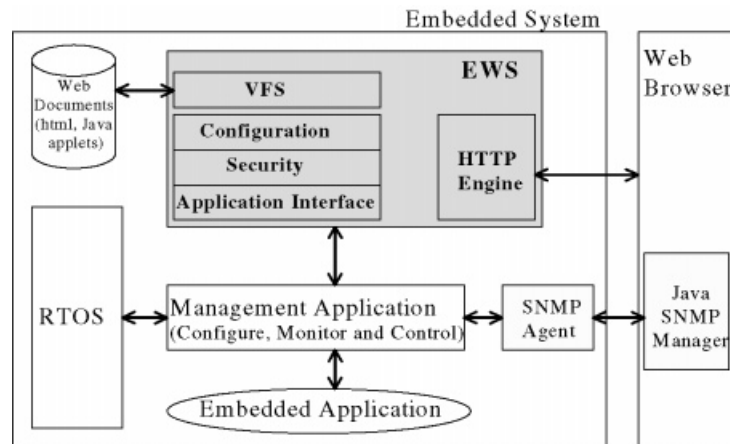


Figure 4. EWS extended architecture for WebMUI

The code size of a Java SNMP applet is small enough to be embedded into the network device because SNMP is simple (three basic message types and a simple message format) and light protocol (uses UDP as its transport protocol, and thus does not have connection setup and acknowledgement overhead). SNMP defines an alert message and traps, which can be directed toward one or more trap receiver stations. If a trap management application is implemented as the Java SNMP applet and loaded from the network device, traps can be collected and viewed together by the Java SNMP applet where appropriate responses will follow.

## Implementation

We have implemented an HTTP/1.1 compliant embedded Web server based on the EWS design presented in the previous section. We call this system POS-EWS, which stands for POSTech-Embedded Web Server. To demonstrate how POS-EWS works, we have applied our POS-EWS to the element management of a commercial Internet router. The C programming language, commonly used in an embedded system, is used throughout the server implementation. We have implemented POS-EWS on the Xinu OS using the MPC 860 processor.

### —Features of POS-EWS—

POS-EWS implements a subset of the HTTP features typically required for use in an embedded

system. To reduce the TCP connection resources, HTTP/1.1 permits a persistent TCP connection to be established for as long as the Web browser requires access to the server. For providing up-to-date dynamic information, the server needs to control the cache mechanism that is also included in HTTP/1.1. The cache control and persistent TCP connection is essential for an EWS, and POS-EWS supports these two features.

In an embedded software system, dedicating a unique process or thread to every incoming connection is usually impractical due to the memory overhead required and, in some cases, to the lack of embedded OS support for multiple processes. When developing POS-EWS, we approached the problem of supporting multiple connections in the context of a single thread by implementing a finite state machine, which processes a request as a sequence of discrete steps. With multiple finite machines in a single thread, several connections can be activated at once, where each state machine representing a specific connection is scheduled to process in a round-robin manner. POS-EWS imposes a deterministic scheduler for handling multiple finite state machines.

For an embedded system, which may not need the full features of a file system, POS-EWS uses a Virtual File System (VFS) which can provide a limited set of read-only files built into the ROM. The VFS can be used with or without a real file system. If a real file system exists, the VFS will forward the file request to it. Using the VFS generator, which is one component of the

POS-EWS preprocessor compiler, the compressed HTML file for use by the EWS is created. The file will be decompressed by the VFS prior to use by POS-EWS.

POS-EWS supports the SSI style application interface. A proprietary tag can be included in a Web page so that when the page is requested it will cause POS-EWS to execute the function specified in the Web page using the tag. The function returns string data directly to POS-EWS to be used as part of the requested Web document. This allows the inclusion of dynamic management data directly into a loading HTML document, such as the current time or communication port status. We implemented this interface style via a table of name and pointer to functions. The table is constructed from the POS-EWS preprocessing compiler using the construction method explained below. Another application interface method is the FORM processing interface method.<sup>5</sup> The HTML FORM keyword allows the browser to send input back to the server by issuing a POST HTTP message. This feature is useful if there are control commands or configuration settings that need to be sent to management applications. Upon receipt of a POST message, POS-EWS calls a function that parses input from the browser and performs an action based on what it found in the input. Like the SSI style interface, this type of interface is also implemented by a table and preprocessing.

POS-EWS also supports state management using HTTP cookies.<sup>21</sup> A cookie is a record that contains management data for a manager to set. It is stored

on Web browsers, and is sent to Web servers each time a manager sends a request to a Web server. Cookies are useful for having a Web browser remember some specific information which the Web server can later retrieve. A server, when returning an HTTP object to a client, may also send a piece of state information which the client will store. This simple mechanism can be used in management applications.

### —POS-EWS Web Compiler—

We have also developed a Web compiler<sup>22</sup> for constructing a virtual file system (VFS) and efficient SSI application interface. Interpreting scripts at run time results in full scanning for the HTML file, which may impact system performance. The Web compiler can offload POS-EWS's scanning results by recording the position of a tag with the HTML file in the VFS. The server reads the HTML file before the starting point of a tag, calls the script function and proceeds with reading the HTML.

An example of an HTML and a subset of a compilation result are shown in Figure 5. In this example, the content of *sysname.html* is converted into a character array by the name of *sysname\_html*, which is the result of simple conversion from file name to C language array name, i.e., changing the dot to an under-score.

The structure *vf* is a container for storing file information such as file size, last modified date, etc. The pointer value of the converted character

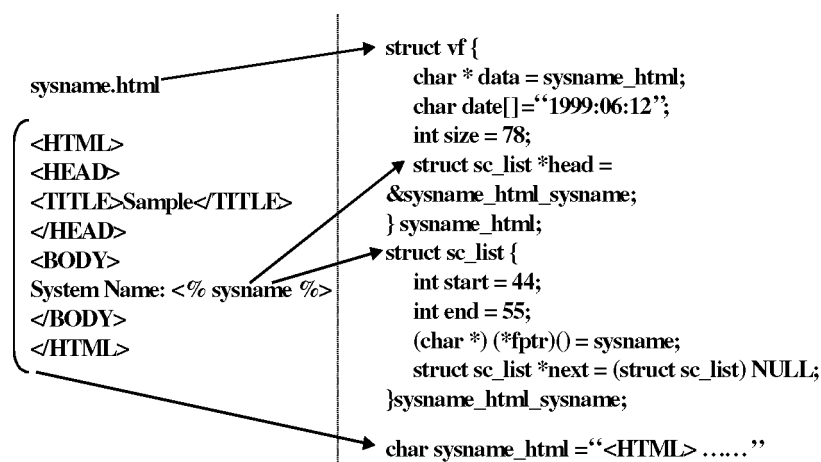


Figure 5. Virtual file system code

array, *data*, is one of the most important elements in the structure because this value is used to read real content by POS-EWS at run time. The structure *sc\_list* is used to make a linked list for script functions. The header pointer for the linked list is also one element in the *vf* structure. POS-EWS uses this pointer value for calling the script function. The structure *vf* has an additional variable for supporting the file interface functions, for example, file read pointer, file state flag, etc. With the file interface functions such as file open (*vf\_open*), file read (*vf\_read*) and file close (*vf\_close*), generated C codes become a complete virtual file.

Optionally, the Web compiler can also compress the Web documents. HTML is easily compressed as much as 50% with almost no run time memory required for decompression. HTTP/1.1 supports compressed file transfer from the Web server to Web browser. The Web document is stored in compressed form, transmitted directly, and decompressed by the Web browser. HTTP/1.1 can convey the information of compressed documents in the *Accept-Encoding* and *Content-Encoding* header fields. More importantly, it indicates what decompression algorithm will be required to remove the compression encoding. The following algorithms, as well as others, are registered in standard HTTP/1.1: *gzip* (generated by the GNU *gzip* program), and *compress* (produced by the common UNIX file compression program *compress*). Because the algorithms minimize the ROM space used, storing a reasonable size of Web documents on the

device has a negligible impact on embedded system resources. For POS-EWS, we have used the *gzip* algorithm to compress at preprocessing and decompress at run time.

The results of implementation can be summarized as follows: the POS-EWS Web compiler converts Web documents to be stored in the virtual file system to compressed C arrays as a virtual file. Then it creates a directory data structure in order to store the file information in the virtual file system. Library functions for the file interface are supported without any RTOS dependency.

### —POS-EWS Management Application Example—

Management information can be classified by the update period, direction of information flow or object of information source. From the viewpoint of update period, some management information changes dynamically, and some does not change at all. Furthermore, some information possesses real-time characteristics. Regarding the direction of information flow, some information can originate from a Web browser and go to a Web server and vice-versa.

As shown in Figure 6, there are four methods to retrieve data from an embedded system using POS-EWS. Method (a) is the most basic method to display data in a Web browser. POS-EWS reads data from a file and sends it to the browser. It

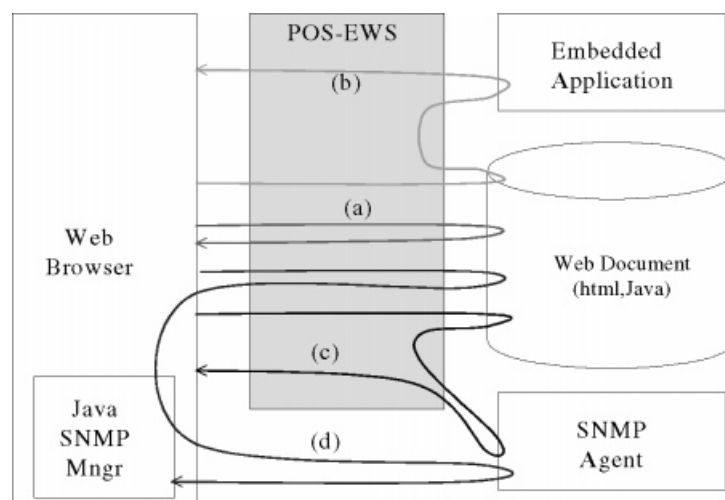


Figure 6. POS-EWS WebMUI mechanism

is suitable for static information like menu, image and so on. Method (b) is the second method, where POS-EWS reads the requested HTML file, calls an embedded application function in accordance with the script tags, replaces the tags of HTML with the result of the application function all format, and sends it to the browser. This method is suitable for showing dynamic information of the system. Circle (b) in Figure 7 shows the user-selectable port name retrieved by this method.

Method (c) is the same as (b) except for the information provider. POS-EWS retrieves an SNMP MIB value instead of an application function return value in replacing tags in HTML. It is suitable if management information is defined in SNMP MIB. It has the advantage of showing static and dynamic information of a system using an SNMP MIB database without additional SNMP traffic.

In method (d), POS-EWS sends the Java SNMP manager applets to the Web browser when requested, and the browser executes them. The Java SNMP manager continuously sends SNMP GET messages to the SNMP agent in the system for displaying real-time data. This method is suitable for showing real-time changes of system status and SNMP Trap information. Circle (d) in Figure 7 is

Java applet which displays status of each port in real-time.

For validation of our work on the design and implementation of an efficient and lightweight EWS, we have used our POS-EWS for the network element management of a commercial Internet router. Figure 7 shows the display result of the WebMUI. Circles (a), (b), (c), (d) in Figure 7 show four different ways to retrieve data from an embedded system using the POS-EWS mechanism explained in Figure 6.

## Performance Evaluation

We developed POS-EWS for Web-based network element management. In this section, we evaluate POS-EWS's performance in areas such as code size, run-time memory, CPU usage and connection capability. We also explain the methods used in optimizing our POS-EWS.

### —Performance Metrics—

The performance of a Web server is dependent upon a number of variables: the server hardware and operating environment, the server application,

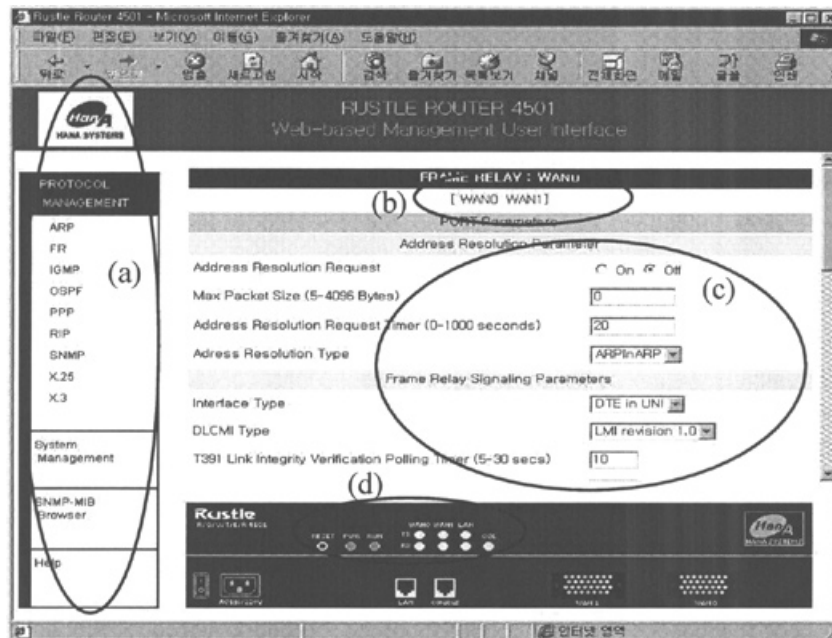


Figure 7. POS-EWS application interface example

---



---

***The performance of a Web server is dependent upon a number of variables: the server hardware and operating environment, the server application, the network protocol, and the network, hardware, bandwidth and traffic load.***

---



---

the network protocol, and the network, hardware, bandwidth and traffic load. Perception of that performance depends also on variables on the client side: the client platform and operating environment, and the Web client.<sup>23</sup> There are four metrics most often used to measure the capacity of general Web servers.

- Requests per second (rps or HTTP ops/sec), which is connections served or requests made per second.
- Throughput in bytes per second, which is dependent upon the bandwidth of the data pipe.
- Round-trip time (RTT), which is a measure of how long it takes for a packet to be sent from the client plus the time a response from the server takes to be received by the client, completing the request.
- Error rate, which is a measure of how many HTTP requests were lost or not handled by a server.

The two key elements of HTTP performance are latency and throughput.

- Latency is measured by the RTT and is independent of the object size.
- Throughput is a measure of how long it takes to send data, up to the carrying capacity of the data pipe. Improving throughput is simply a matter of employing faster, higher bandwidth networks.

Basically, the performance of a general Web server can be measured by timing how long a server takes to respond to a request and to count the number of bytes delivered per unit time. But EWSs have different performance metrics from general Web servers.

In general, embedded systems must minimize requirements for system resources such as CPU and memory: the embedded Web server is acting

as a secondary feature of the system and should avoid interfering with the system's main purpose as much as possible. The most important task of an embedded system is to perform mission-critical and real-time applications. An EWS is often the lowest priority service in the system, so end-users can wait hundreds of milliseconds for a response (an eternity compared to the low-latency requirements of many embedded real-time applications). Therefore, RTT and throughput are not important metrics for an EWS.

The performance element of POS-EWS are code size (memory footprint), run-time memory, the CPU usage and maximum user connectivity (the capacity of POS-EWS). The code size is approximately 30 Kbytes, the average run-time memory is 64 Kbytes, with the HTTP server having the lowest priority. The POS-EWS supports multiple, simultaneous HTTP transactions and multiple users. By evaluating the system performance we can determine how much an EWS impacts its embedded system. The code size of our POS-EWS is very small so that it puts little impact on the resource scarcity problems.

### —POS-EWS Optimization—

We implemented POS-EWS as a finite state machine (FSM) to improve its performance. We approached the problem of supporting multiple connections in the context of a single process and thread by implementing an FSM, which processes an HTTP request as a sequence of discrete steps. The 'process-per-connection' architecture, which forms a new process for each connection, while it goes by the stateless model of the HTTP scheme, is less than efficient. The time and resources required by the fork and exec operations are significant, particularly in light of the fact that a typical Web request is very short.<sup>24</sup> But the FSM supporting single thread is run by a small scheduling system that uses lightweight task structures. This makes the CPU usage and the memory footprint reasonable.

We also give our HTTP engine more improved performance following the HTTP standard. We implemented POS-EWS to keep TCP connections open and reuse them by the Keep-Alive option. Therefore, the cost of opening a new connection for each transaction can be eliminated by reusing

existing TCP connections. In this way the transaction time will be the time to send a request plus the RTT plus the processing time on the server plus the time to send the response. When the Web server keeps TCP connections open and does not close them at end of an HTTP exchange, the maximum number of available TCP connections will be reached. The Web server then closes the oldest idle connection first. HTTP version 1.1 specifies the optional use of the Keep-Alive connection.<sup>7</sup> Requesting a Keep-Alive connection when GETing a file means the browser can reuse the connection to get subsequent files from the server.

The POS-EWS Web compiler preprocesses and compresses Web pages and images into compilable ANSI-C code. This allows pages to be developed using standard HTML tools, and stored internally in an efficient format. The Web compiler makes it possible to minimize the application memory footprints through intelligent compression. Shared and nested pointer techniques are used for additional memory savings. The Web compiler reduces the processing time through preprocessing.

We used APIs to extend a server's functionality. Instead of having to parse the incoming amorphous stream of form data, POS-EWS uses options for receiving it, often preformatted and type converted into C struct fields, ready for program use. Using an API instead of a CGI has the advantage of integrating the extensions to the server within the server process.<sup>25</sup> This eliminates the need to go to the OS for communications between the server and the script. Such a C level interface can save much time and simplify CGI programming immensely.

## Related Work

In this section, we briefly investigate embedded Web server products, focusing on their features. Web servers can have a range of capabilities and still be http-compatible. A number of commercial EWS products have appeared on the market, each with its own particular value position. There are also many computer communities exploring small Web servers. They make a Web server the size of a matchbox. The Matchbox Server<sup>35</sup> is a single-board computer with a 16 MB RAM, and 16 MB flash ROM, big enough to hold a useful amount of Linux including the HTTP daemon. In addition, the iPic Web server<sup>36</sup> is the smallest Web server.

It is about the size of a mere match-head. The single chip computer runs the iPic web-server, the world's tiniest implementation of a TCP/IP stack and an HTTP web-server.

Table 1 compares the features of a number of commercially available Embedded Web Server implementations and our POS-EWS. Blank columns represent features not supported or we could not find appropriate information on them from the available literature. We summarize the offerings available and the approximate code size needed. This range does not necessarily reflect differences in code efficiency, however. Most EWSs offer small footprints lower than 100 Kbytes for low resource utility, dynamic content generation of SSI type mechanism, some kind of page compression, and options for security/authentication and porting layers to accommodate custom file system and TCP/IP stack.

All of the above products work with any standard browser, including Internet Explorer, Netscape Navigator or Communicator, Hot Java and Mosaic, on all platforms, allowing any network-connected browser user to easily access any device. As well, all serve multiple, simultaneous users. Processing multiple requests may be important if more than one user is to access the embedded system at the same time, or if the system is to report itself busy to potential users. A few support a Web compiler and Virtual File System (VFS), which are essential features for enhancing Web ability and efficiency. Only RomPager<sup>27</sup> and our POS-EWS clearly specify HTTP cookies for state management. POS-EWS has full features of EWS to support the functionality of EWS.

Also, our POS-EWS provides effective integration mechanisms into embedded management applications. POS-EWS offers four basic interface mechanisms for use between a management application and application of an embedded system and an embedded Web server or Web documents. A developer can choose an appropriate interface mechanism depending on the characteristics of management information or types of Web documents. We testified them through applying POS-EWS to management of a commercial router. The integration mechanisms into embedded management applications are important when an EWS is applied to network element management.

Company & Product	OS supported	CPU supported	HTTP code size (version)	Features					
				SSI	VFS	Compiler	Compression	Security (encode)	Cookie
Agranat Systems, EmWeb <sup>26</sup>	No OS	Any CPU with a C compiler	25 kbytes (1.1)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Proprietary	Basic + Digest	
AllegroSoft, RomPager <sup>27</sup>	Any RTOS, No OS	Any CPU with a C compiler	10-40 kbytes (1.1)	<input type="checkbox"/>	File Sys.	<input type="checkbox"/>	Dictionary	Basic	<input type="checkbox"/>
BVM, IntraScada, Web Server <sup>28</sup>	OS-9	CPU32	<100 kbytes (1.1)						
ccelerated Technology, Nucleus Embedded Software <sup>29</sup>	Nucleus Plus	x86,68K,ARM, 683xx,SPARC, PowerPC,SH, H8/300H, TMS320C3x, MIPS,4x/5x/2x, Panasonic MN10200	40 kbytes (1.0)	<input type="checkbox"/>			Proprietary	Basic (DES)	
Spyglass, MicroServer <sup>30</sup>	LynxOS, QNX, pSOS, OS-9, VxWorks	Any CPU with a C compiler	35-110 kbytes (1.1)	<input type="checkbox"/>			None	Basic + Digest + SSL	
QNX Software Systems Ltd, QNX Internet Toolkit <sup>31</sup>	QNX real-time OS	x86, Pentium Pro, AMD Elan	106 kbytes (1.1)	<input type="checkbox"/>				Basic + Digest (encode)	
Magma, Lava <sup>32</sup>	Any RTOS	Any CPU with a C compiler	15-40 kbytes (1.0)	<input type="checkbox"/>			Proprietary	Basic + SSL	
Quiofix, QEWS <sup>33</sup>	pSOS, LynxOS, VxWorks	Any CPU with a C compiler	45-50 kbytes (1.0)	<input type="checkbox"/>	<input type="checkbox"/>		GZIP	Basic	
Web Device, Pico Server <sup>34</sup>	LynxOS, Nucleus Plus, pSOS, VxWorks	Any CPU with a C compiler	15-30 kbytes (1.0)	<input type="checkbox"/>	<input type="checkbox"/>		ZIP-like	Basic + Digest + SSL	
POSTECH, POS-EWS	Real-time Xinu, pSOS	Any CPU with a C compiler	30 kbytes (1.1)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	GZIP/CSS-Style	Basic + Digest (base64)	<input type="checkbox"/>

Table 1. Comparison of EWS products

## Conclusion and Future Work

Web servers are already being built into many network devices today. In the near future, we can expect this trend to grow even further to home appliances, medical instruments and industrial equipments. Embedded Web servers for Web-based network element management provides an administrator with a simple but enhanced and more powerful user interface without additional hardware. Although without the management application specific aspect of embedded Web server technology these advantages may disappear.

In this paper, basic technical concepts on efficient and lightweight embedded Web servers were described and technical issues for their management application interface to network devices were explored. Also, we presented our design and implementation of an HTTP/1.1 compliant embedded Web server (called POS-EWS) based on the proposed architecture. The EWS-WebMUI architecture provides an easy integration platform with an SNMP agent. That is, POS-EWS provides easy and effective integration mechanisms for embedded management applications. This 'webification' of network devices has generated a new philosophy for network element management.

We also ported POS-EWS to the pSOS OS using MPC 860 processor. We plan to optimize our POS-EWS even further and make it more powerful in its application to home appliances, office equipment, and industrial products. We must check the reliability of our POS-EWS for porting to other embedded systems. This is important because POS-EWS is a part of embedded systems requiring reliability. We plan to measure quantitative performance data of POS-EWS. We also plan to port POS-EWS to other CPUs and embedded OSs. Future work involves investigating methods for network management of devices equipped with EWSs.

## References

1. McCombie B. Embedded web servers now and in the future. *Real-Time Magazine* 1998; No. 1, 82–83, March.
2. Wilson A. The challenge of embedded internet. *Electronic Product Design* 1998; 31–2, 34, January.
3. Agranat I. Embedded web servers in network devices. *Communication Systems Design* 1998; 30–36, March.
4. Wellens C, Auerbach KK. Towards useful management. *The Simple Times* 1996; 4(3): 1–6, July.
5. W3C. *HTML 4.0 Specification*. Internet Draft REC-html40-19980424, HTML Working Group, April 1998.
6. W3C. *Hypertext Transfer Protocol-HTTP/1.1*. Internet Draft draft-ietf-http-v11-spec-rev-06, HTTP Working Group, November 1999.
7. Fielding R. *Hypertext Transfer Protocol—HTTP/1.0*. RFC 1945, IETF HTTP WG, May 1996.
8. Sun Microsystems Inc. *Java Management API Architecture*, June 1996.
9. WBEM Consortium. Web-Based Enterprise Management Proposal. *HyperMedia Management Protocol Overview*. Revision 0.04. July 1996.
10. Case J, Fedor M, Schoffstall M, Davin C. The simple network management protocol (SNMP). RFC 1157, May; Mullaney P. Overview of a web-based agent. *The Simple Times* 1990; 4: No. 3, July, 1996.
11. Berners-Lee T. Uniform Resource Locators (URL). Internet Draft, March 1994.
12. Fielding R. Hypertext Transfer Protocol—HTTP/1.1. RFC 2068 IETF HTTP WG, August 1996.
13. Touch J, Heidemann J, Obraczka K. *Analysis of HTTP Performance*. USC/Information Sciences Institute, June 1996.
14. Frystyk Nielsen H. *et al.*, Network Performance Effects of HTTP/1.1, CSS1, and PNG. W3C, June 1997.
15. CGI, <http://www.w3c.org/cgi/>.
16. Berners-Lee T. Uniform Resource Locators (URL). Internet Draft, March 1994.
17. Arnold K, Gosling J. *The Java Programming Language*. Addison-Wesley: Reading, MA, 1996.
18. OMG. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, July 1995, OMG TC Document.
19. Kong JY, Hong JW. *A CORBA-based Management Framework for Distributed Multimedia Services and Applications*. Technical Report PIRL-TR-97-1, POSTECH, Korea, March 1997.
20. Tsai CW, Chang RS. SNMP through WWW. *International Journal of Network Management* 1998; 8: 104–119.
21. Kristol D, Montulli L. HTTP State Management Mechanism. RFC 2109, February 1997.
22. Ju HT. *POS-EWS Web Compiler for Supporting an Effective Application Interface*. PIRL Technical Report, POSTECH, PIRL-TR-99-3, December 1999.
23. Rubarth-Lay J. Keeping the 400lb. gorilla at Bay. *Optimizing Web Performance* May 1996.
24. Padmanabhan V, Mogul J. Improving HTTP latency. *Computer Networks and ISDN Systems*, December 1995; 28: 25–35.
25. Edwards N. Rees O. *Performance of HTTP and CGI*. ASNA.

26. Agranat Systems. EmWeb. <http://www.emweb.com/>.
27. AllegroSoft. RomPager. <http://www.allegrosoft.com/>.
28. BVM IntraScada. Web Server. <http://www.bvmltd.co.uk/>.
29. Accelerated Technology. Nucleus Embedded Software. <http://www.atinucleus.com/>.
30. Spyglass. MicroServer. <http://www.spyglass.com/>.
31. QNX Software Systems Ltd. QNX Internet Toolkit. <http://www.qnx.com/>.
32. Magma Information Technologies. LAVA. <http://www.magmainfo.com/>.
33. Qiotix Corporation. QEWS. <http://www.quotix.com/>.
34. Web Device Inc. Pico Server. <http://www.webdevice.com/>.
35. Stanford University. Matchbox Server. <http://wearables.stanford.edu/>.
36. University of Massachusetts. iPic Web server. <http://www-ccs.cs.umass.edu/~shri/iPic.html/>. ■

**If you wish to order reprints for this or any other articles in the *International Journal of Network Management*, please see the Special Reprint instructions inside the front cover.**