Table of Contents

List of Figures

List of Tables

1. Introduction

As 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. The widespread availability of Web browsers means that a standardized presentation model and application protocol are now available for interaction among a multitude of systems, not just traditional Web sites on the Internet. Web browsers can provide a GUI interface to various client/server applications without a client application. Therefore, Web browsers have become the de facto standard user interface for a variety of applications. 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, monitor and control network devices over the Internet using a Web browser. Using a Web-based interface for management functions offers the user a means to interact with the device in a manner that is well-known. 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 [6], graphics and other features common to Web browsers [4]. An initial home page provides the central navigation point for the devices, and hyperlinks take the user to different sections of the interface. Information is provided to the user by simply retrieving pages, and information is sent back to the device using Forms that the user completes.

Figure 1 shows how Embedded Web Servers (EWSs) [1, 2, 3] are used for Web-based Management User Interface (WebMUI). By embedding a tiny EWS into products such as networking hardware, office equipment, and industrial products, users can remotely monitor, configure, and diagnose vital systems. Whatever products we develop, from the most sophisticated networking devices such as routers, switches and cache engines, to the smallest consumer electronics such as televisions and refrigerators, an EWS can be the fastest,

easiest way to get to market with Web-based management.

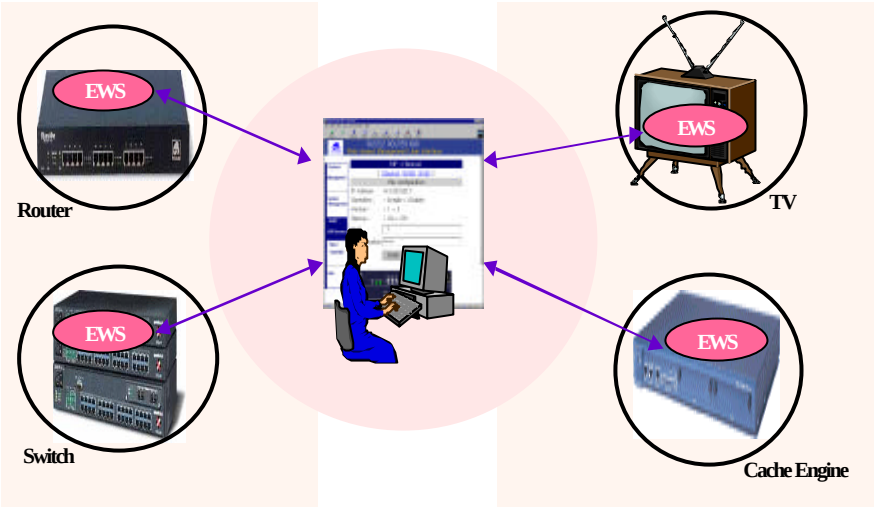

Figure 1. Web-based Network Management using EWS

EWSs have many aspects that are different from general Web servers. Web-based management user interfaces (WebMUIs) through embedded Web servers have many advantages: ubiquity, user-friendliness, low development cost and high maintenance. EWSs have functional requirements such as handling of an HTTP packet, examining a security, processing a file, and interfacing with an application program. In addition, EWSs have different nonfunctional requirements, such as low resource utility, high reliability and portability, for which general Web server technologies are unsuitable. Above all, due to resource scarcity of embedded systems it is important to make EWSs efficient and lightweight. There are also design issues such as HTTP [5] 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.

As mentioned before, EWSs have many advantages for WebMUI, and EWSs have other requirements and design issues different from general Web servers. Therefore, an efficient and lightweight embedded Web server is essential for Web-based network element management.

In this thesis, we present our work on developing 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 focusing on integration mechanisms of embedded management information. 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 this thesis is as follows. In Section 2, we present an overview of EWSs, and describe the EWS-WebMUI. In Sections 3 and 4 and 5, we present the EWS requirements, the EWS design and the implementation of our proposed EWS architecture, respectively. In Section 6, we evaluate POS-EWS's performance and explain our methods for optimizing POS-EWS. In Section 7, we briefly investigate the available offerings of EWS products focusing on their features and the approximate code size needed. In Section 8, we summarize our work and discuss possible future work.

2. Embedded Web Servers

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

2.1 General Web Server and Embedded Web Server

Web servers, also known as HTTP servers, are computer software systems that store and transmit information to Web clients. Generally, Web browsers include Internet Explorer and Netscape Navigator. The HyperText Transport Protocol (HTTP) [5, 7], which utilizes TCP/IP for its transport, is used by Web clients and servers to exchange hypertext information between the two. HTTP is a client driven protocol, simply meaning that all communications are initiated with the client. . To initiate a session, a Web client makes TCP open requests to the well-known port number 80. Once the Web server has accepted the request, it expects one of three basic operation commands (GET, HEAD and POST) and an object address. Object addressing is done using a Universal Resource Locator (URL) [8]. After processing the command, the server responds to the request with the header identifying the body content type, encoding, length and other supporting information, followed by the requested object body. Finally, the server closes the session by issuing a TCP close request.

A Web server is the repository for Web documents, storing the information and their display formats. A Web server handles requests and passes documents back to the browser. The browser performs the more difficult work of presenting the text, displaying graphics, generating sound or video and running Java applets. When a browser receives a file from a Web server, the server provides the MIME (Multipurpose Internet Mail Extension) [9] type of the file. The browser uses the MIME type to establish whether the file format can be read by

the browser's built-in capabilities or not, or whether a suitable plug-in application is required to read the file. Web servers use HTML [6] to represent hypertext documents on a Web browser without a helper application. The Web documents on the server can be static (e.g., stored as a read-only document) or created dynamically in response to parameters supplied by the clients. 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. Underlying this reduction of the resource requirements of the Web server is typically a portable set of code that can run on embedded systems with limited computing resources, and which can be utilized to serve the embedded Web documents to the Web browsers. This type of Web server is called an Embedded Web Server (EWS) [1, 2, 3]. 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. Embedded environments are often more limited in resources than non-embedded systems, and hence increased efficiency of resource usage is in order. Also, by limiting the functionality of certain embedded applications, security and reliability can be enhanced. Embedded Web operating system developers and tool developers should attempt to make their systems configurable so that functionality can be tailored to the application at hand.

For network devices, such as routers, switches and hubs, it is possible to place an EWS directly into the devices without additional hardware. As more devices (such as home appliances, manufacturing devices and medical instruments) are connected to the Internet, we believe that EWS technology will be essential to making Internet devices more manageable,

2.2 WebMUI and EWS-WebMUI

The rapid proliferation of Web-based management has made it clear that schemes using HTTP and standard Web browsers provide benefits to both users and developers. Several such Web management approaches have been proposed. Sun Microsystems is pushing its Java Management eXtension (JMX) [10] and Microsoft, Compaq and Intel are touting Web-based Enterprise Management (WBEM) [11]. However, both approaches are sufficiently complex that many small network devices would be unable to implement them.

Most Web-based management applications provide an interface to the status reporting, configuration, and control features of managed objects. This functionality requires dynamic and interactive contents as well as static contents because management information is either constantly changing or changed by an operator. Consequently, when a request arrives from a browser, the Web server determines the current value of the managed object, then inserts this information into an HTML document, ultimately returns it to the browser. If a request contains control information, the server passes the information to the management application, which in turn issues a control command according to the information. A user may configure the managed object's behavior interactively through the Web user interface.

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). Before the introduction of the Web, combinations of Command Line Interface (CLI) and Window-based Graphic User Interface (GUI) were employed in various ways for many years. CLI was implemented using a local serial or Telnet connection, which is cheap and easy to implement. As GUIs such as Macs, MS-Windows, and X-Windows proliferated, operators began to demand better management interfaces than Telnet. Manufacturers of embedded systems began to provide SNMP-based GUIs and they spent many man-years of development time accommodating the various GUI platforms [12]. By providing an EWS-WebMUI, enhanced user interfaces and low development cost can be achieved. There is no specialized management user program: one only has to run a Web browser on the client side and then input the proper URL of the embedded system to be managed.

Therefore, an EWS-WebMUI is a direct result of embedding a Web server, Web documents, and management applications into an embedded system. An EWS's major role is to receive requests for Web documents from a Web browser and return the requested document to the Web browser. The Web documents give a display form of management information, which is a collection of manageable data that is monitored or configured for managing an embedded system.

2.3 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 the development of the 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. In practice, this will result in the production of superior interfaces in a shorter time frame.

EWS-WebMUIs also have the advantage of a platform independent graphical user interface. The SNMP [13] management scheme usually consists of an SNMP based Network Management System (NMS). For providing ease of use most NMSs give users the possibility of using a graphical interface based on MS Windows or X-Window as opposed to the command line interface. This means most NMS users demand specific platforms, such as OS, or computer hardware, in order to install and execute the NMS. But an EWS-WebMUI does not demand any specific platform because Web browsers are available for virtually all computers.

While the EWS-WebMUI concept may appear straightforward and perhaps less than revolutionary, the implications are deeper than they may appear. 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 also no risk of the user having an outdated version of the user application software that does not support all the features of latest device [14]. 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, so that a device can receive an upgrade to its management interface, it is routinely possible to upload Web documents to the embedded

system from a remote location on the network. This feature makes it possible for developers to upgrade all devices over the network from one location. High maintenance for EWS-WebMUI is a direct result of ease of Web document development and upgrade from one location.

3. EWS Requirements

We illustrate EWS requirements that we must consider during development in this Section. Requirements are divided into two parts – functional requirements and nonfunctional requirements.

3.1 Functional Requirements

In order to implement an EWS, there are functional requirements to be considered. The EWS requires the following functions: handling the HTTP packet, interfacing management applications, processing a file system, and checking configuration and security.

The basic role of the EWS is processing HTTP packets, which is the role of an HTTP engine. The minimum requirement for an HTTP engine is that it must be compliant with HTTP specifications. The role of an HTTP engine is equal to that of general Web servers. Depending on the TCP/IP stack API available in a given embedded environment, the HTTP engine listens to a signal indicating a connection attempt. After accepting the connection, the server must serve the basic HTTP methods (GET, HEAD, POST). Finally, after serving the request, the HTTP engine is ready to receive another connection signal.

Unlike general Web servers, most Web pages generated by Internet appliances and embedded devices are dynamically generated. Whereas a general Web site will usually have vast stores of static pages, an embedded device will generate pages on demand. These pages display the device status, the results of sensors, or local data that have been captured by the device. To display the current device status, it is important to interface with the embedded management application program. Therefore, interface mechanisms with embedded management applications must be developed.

Many devices do not have disk space or permanent storage, yet we still wish to access and control them via the Web. For such devices, a method of storing Web pages in ROM is required. Embedded Web servers should be able to compile Web pages and then link the page object to the resulting executable image. General Web servers have no such requirement. Content storage on a general Web server system is quite trivial: URLs resolve to files, which are opened at service time and "poured" down a socket. But in an embedded system, served pages, graphics, sound, Java applets, etc. are most likely to reside in ROM or flash, to be partially or wholly copied to RAM. Only high-end systems are likely to find themselves stored on rotating media. For portability and resource scarcity, embedded Web servers assume the existence of a "virtual file system", that while not actually implemented on disk, emulates a sector-like block-level interface to the storage system.

Security is an important concern in network management in different applications for embedded Web server technology, especially ones that involve equipment configuration or administration. As the primary function of the Web server is to permit access and control, it is important to display the device's current status and provide operational feedback. Therefore, an EWS generally has a functionality of checking security and/or configuration. It is often desirable to limit access to this information to a specific set of users.

Simple authentication and access-control mechanisms are the preferred method to provide security for embedded Web servers. The authentication mechanism that is most widely supported by today's Web browsers and servers is basic authentication. This method is straightforward to implement, but suffers from the fact that both the user name and password are transmitted from the client to the server in a very lightly encoded form (the base-64 encode format). But this level of security may still be an unacceptable risk in some applications. The Internet Engineering Task Force (IETF) has proposed a new authentication mechanism, digest authentication [15], which resolves this weakness by having

the client transmit only a digest which has been computed over a hash of the user name, password, the URL named in the request, and the HTTP request method (GET, POST, etc.). The EWS has configuration environment variables such as the number of concurrent connections, socket port, own host name, root file path, default "index", inactivity timeout and time zone. These configuration variables determine the basic configuration of EWS.

3.2 Nonfunctional Requirements

The development of an EWS must take into account the relative scarcity of computing resources available to it. An EWS must meet the device's memory requirements and limited processing power. General purpose Web servers have evolved toward a multi-threaded architecture that either dedicates a separate thread to each incoming connection, or uses a thread pool to handle a set of connections with a smaller number of threads. Dedicating a single process or thread for every incoming connection is usually impractical in EWSs due to the memory overhead required and, in some cases, because of the lack of system support for multiple processes.

Regardless of the tasking model, when data is exchanged between modules the data is either copied into the local memory or a reference to the data is supplied. The latter is called a zero-copy transfer. While a copy transfer is simple to implement, a zero-copy transfer is more efficient and preferred for most embedded applications. Overcoming the memory limitation poses problems, but we are confronted with even grater difficulties in managing the impact of Web request servicing on the system CPU. In other words, can request processing be done in a way that allows the rest of the system to meet its real-time constraints? An EWS process, as a subordinate process for the main purpose of the device, must use as few CPU resources as possible in order not to interfere with the main task of the system. To minimize system resource usage,

an EWS can place restrictions on some parameters. For example, it is not necessary to support a very large number of connected users; usually only one to a half dozen users will be accessing the system at a time.

Generally, network devices require high reliability. As one embedded component of a network device, an EWS must also be highly reliable. Because it is a subordinate process, at the very least it must protect against propagation of internal failure to the whole system. An EWS needs to run on a much broader range of embedded system environments in terms of the facilities they provide, and with much tighter constraints on resources than mainstream computing hardware. It must be easily portable and should be transferable from one project to another. For this, source code that has already been ported and tested to a wide range of platforms is required.

It is vital for embedded Web servers to easily integrate. There are two parts to integrating a Web server:

- Integrating the Web server application into the Real Time Operating System.
- Integrating device access functions into the Web server.

Often device applications require that the Web server be integrated into an existing application or that the event loop of the Web server be accessible. To integrate device access functions into a Web server requires easy binding of existing device APIs to URLs. While Web documents can be quickly prototyped with readily available desktop authoring tools, the prototype must then be integrated into the system software. An EWS works with a fixed set of integrated Web documents that are usually frozen at the time the embedded system is manufactured, but then incorporates dynamic information of system software at run-time. For rapid development, an easy but powerful integration mechanism must be provided.

4. Design

In this section, we discuss the EWS design issues and present our EWS architecture and process structure.

4.1 Design Issues

Traditional Web servers are designed to serve static Web pages from high-end workstations with plentiful CPU resources. Embedded Web servers have different design considerations for which traditional technologies are unsuitable. The following issues must be considered.

4.1.1 Protocol Consideration

In May of 1996, the IETF published an informational memo (RFC 1945) to document the HTTP/1.0 protocol [7] as it was then being used, but there were sufficient problems with that protocol that it did not formally become a part of the Internet standards. Since that time, the HTTP/1.1 protocol [5, 16] has been developed to address the problems of HTTP/1.0. Two major improvements made in HTTP/1.1 are important to embedded system interface developers: explicit cache control and persistent TCP connection.

The improvement of explicit cache controls in HTTP/1.1 are designed to allow appropriate caching of responses, either by private caches within a browser or by intermediate systems such as proxy servers or cache severs. Both clients and servers can control when it is appropriate to cache a response or to use a cached response. For Web documents that do not change (such as logos and other embedded graphics, or pages containing “help” text) this is a great performance benefit because no more than a very brief request and response will be needed to confirm the validity of the cached copy. The mechanisms for

this in HTTP/1.0 were not sufficient, and what was included used timestamps, which presented a problem for embedded system implementations, which may not have a synchronized (or even any) clock. HTTP/1.1 caching can be controlled without a clock. EWS-WebMUI provides static and dynamic Web documents. Caching is desirable for static Web documents, but dynamically-generated Web documents must not be cached in order to retrieve up-to-date management information.

HTTP/1.0 uses each TCP connection for just one exchange - the browser sends a single request and the server sends the response and then closes the connection [17]. This creates both resource and performance problems because a number of requests are required for typical Web pages (one for the main page, one for each frame, and another for each embedded image). Because each TCP connection requires resources for buffering and maintaining the connection state, the memory requirements for the system increase. Moreover, TCP implementations maintain connection state information for two minutes after a connection is closed. Performance is reduced because of the extra round-trip delays required to set up each connection, and because the TCP congestion avoidance algorithms cause each new connection to artificially restrict throughput, increasing it only gradually to discover the maximum available bandwidth.

HTTP/1.1 also includes improvements in connection usage to allow multiple requests to be pipelined on a single connection, with a mechanism to encode each response as a series of chunks so that it is not necessary to buffer the entire response before transmitting it [18]. The result is better performance for the user at lower resource cost for the embedded system. It remains important that the embedded system be able to handle multiple simultaneous requests because most HTTP/1.1 browsers open two connections for a typical page, splitting the requests for the page content between them, and because there may be concurrent requests from different users. Figure 2 shows the difference between

HTTP/1.0 and HTTP/1.1 when processing a TCP connection request.

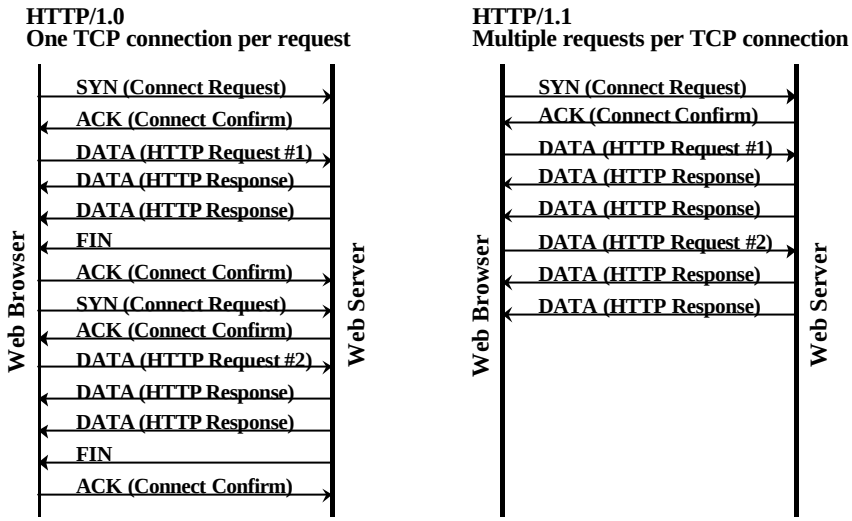

Figure 2. Comparison of TCP connection between HTTP/1.0 and HTTP/1.1

4.1.2 Embedded Application Interface Consideration

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) [19] found in many traditional Web servers. In this model, each URL 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.

While this approach is perhaps the easiest for embedded Web server developers to make, it is by far the most difficult for GUI designers to write: CGI scripts are tedious to produce. Once written, the display of the Web page can only be determined when the script is executed. In an embedded system,

this implies building an executable image, burning it into flash memory, and booting the device before the Web page can be viewed by a browser. Therefore, the CGI solution requires a long time for development, and is difficult to maintain.

Another solution is to use Server-Side Include (SSI) [6]. 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. For example, a proprietary scripting language could define an interface to invoke application functions used to generate the dynamic part of a Web page. Embedded system initialization code is used to register the server-side scripts with function calls. The code for a function call is invoked when server-side scripts are interpreted by the server.

SSI is easier to use than the raw interfaces of CGI solutions. However, interpreting scripts at runtime in an embedded system may impact system performance. Moreover, a significant amount of memory is required to maintain a database for mapping script names into embedded software functions and variables. Server-side scripts generally offer limited capabilities, resulting in a larger and more complex embedded application code.

In order to offload such substantial Web server processing from the embedded system at run time, a preprocessor tool can be used, which converts Web pages into C code. The C code is then compiled and linked with the embedded Web server and application software to produce a tightly integrated executable image. 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.

4.2 EWS Architecture

Based on the functional requirements discussed in Section 3, we have designed an EWS considering the aforementioned design issues. The overall EWS 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 3.

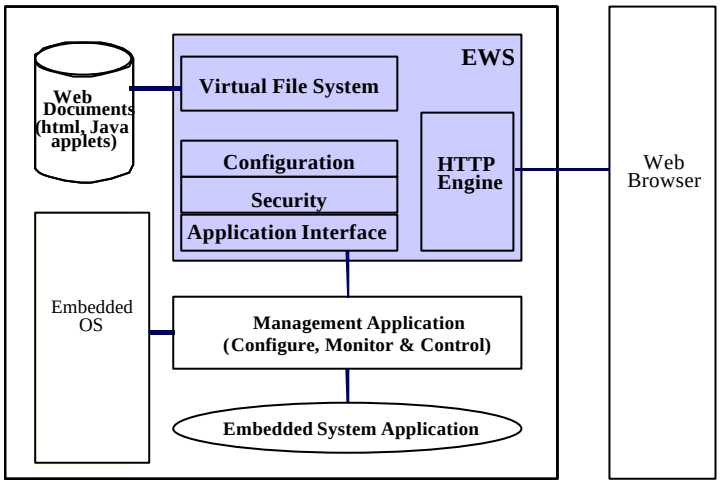

Figure 3. EWS Architecture

The functionality of an HTTP engine is explained in Section 3.1. Unlike general Web servers that start a new thread or process whenever a new connection is made, an HTTP engine normally supports multiple simultaneous users while running as a single process. The number of processes that the server requires can have implications on both RAM usage, due to the stack space per task, and CPU usage. An HTTP transaction process is explained using a state transition diagram in Section 4.3.

The application interface module in an EWS enables developers to add new management functionality by merging Web documents, created with any off-the-shelf Web authoring tool, with management application programs that generate specific dynamic management information. This module provides mechanisms for interacting with the embedded application and supports two application interface styles: CGI and SSI. Through the application interface, Web document development can be separated from management application development. The management application developer needs no longer to generate any part of a Web document through program logic. Also, Web document prototyping, involving different user interfaces with a completely deployed Web document, no longer requires code changes or recompilation.

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 for storing file information such as file size, last modified date, etc. The data structure for an HTML documents file requiring 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 and links them with the Web Server codes. The resulting structure dose not require a 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 4 illustrates the process of a Web server constructing a virtual file system.

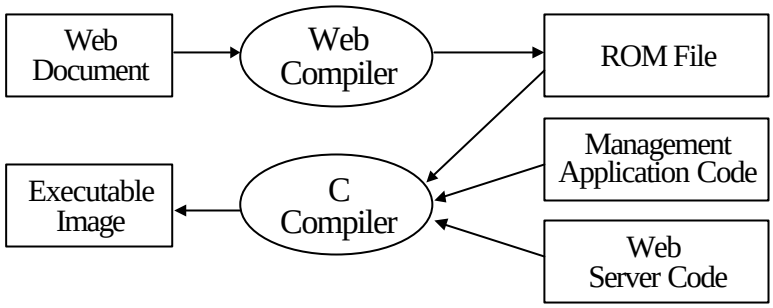

Figure 4. Process of a Web Server constructing a Virtual File System

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, 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.

4.3 EWS Process Structure

We designed our EWS as a finite state machine (FSM), which processes an HTTP request as a sequence of discrete steps. Figure 5 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 that uses a lightweight task structure, which 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 runable and moves the FSM one step if it is runable.

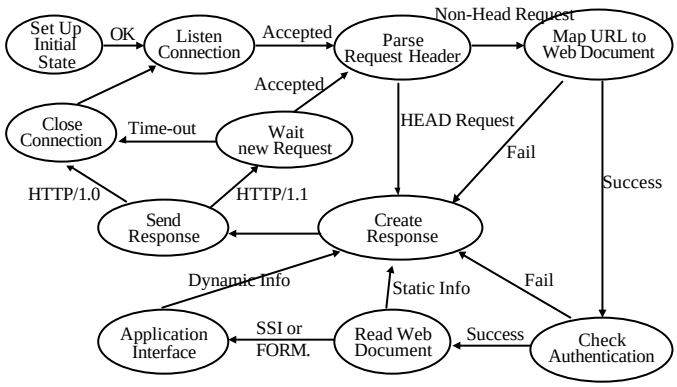

Figure 5. EWS Finite State Machine

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 the 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 shows HTTP/1.1 support.

?? *Close connection*: Close the TCP connection.

4.4 EWS Extended Architecture for EWS_WebMUI

As mentioned before, the only scheme of HTTP and HTML is client-driven. One of the side effect is that once a page is served to the Web browser it becomes static: it does not change even if management data have been altered on the server side. For a user seeking a device, which is dynamic, this is not very appealing. There are several methods to provide dynamic interfaces. Refresh buttons can be placed on pages with dynamic information, so the user can press them to reload the page from the server with updated information. This is obviously very cumbersome and not very appealing. This method can be

employed to periodically update the page a user is viewing by continuing to re-request and display it. The disadvantage of this method is that the entire screen will blank as the page is reloaded, and since communication can be slow this is usually fairly noticeable. Another method is server push: once a page that contains multi-part content has been loaded, it is possible to maintain a connection through which the server can continue to send updated data. For handling continuous data, a browser needs to use a plug-in for displaying graphics. The main disadvantage of a plug-in is that the software must be loaded and installed on the client computer.

To be useful for management application, pages will have to 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 [20] and/or CORBA [21, 22] 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) [13, 23] was created in the late eighties for the purpose of providing an industry standard protocol for communicating with network devices. The thought was that as networks grew there needed to be some common method for monitoring, controlling and receiving alerts from disparate devices. SNMP 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.

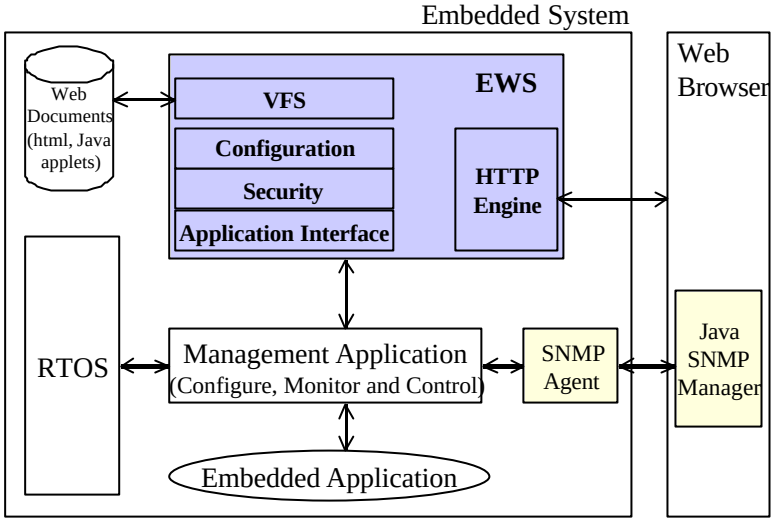

Figure 6. EWS Extended Architecture for WebMUI

The EWS extended architecture gives an integration platform. Figure 6 illustrates the EWS extended architecture for an EWS-WebMUI. The ultimate solutions are that the EWS-WebMUI as a user interface, and 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.

The code size of a Java SNMP applet is not too large to be embedded into the network device because SNMP is a simple (three basic massage 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 and appropriate responses made.

5. Implementation

We have implemented an HTTP/1.1 compliant embedded Web server based on the EWS design presented in the previous section. Specifically, we have implemented an efficient and lightweight EWS considering its requirements, such as low resource utility (CPU usage and ROM size). We call this system POS-EWS, which stands for POStech-Embedded Web Server. To demonstrate how POS-EWS works, we have applied POS-EWS to the element management of a commercial Internet router. Through this application, we propose an integration mechanism with embedded applications. The C programming language, commonly used in embedded systems, is used throughout the server implementation. We have implemented POS-EWS on the Xinu OS using the MPC 860 processor.

5.1 Features of POS-EWS

HTTP/1.1 is formally defined by the IETF document RFC-2068 [13]. 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 the duration of time that 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, due 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 in Section 4.2. Another application interface method is the FORM processing interface method [6]. 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 [24]. 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 the Web server. Cookies are a general mechanism which server side connections (such as CGI scripts) can use to both store and retrieve information on the client side of the connection. 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. Included in that state object is a description of the range of URLs for which that state is valid. Any future HTTP requests made by the client which fall in that range will include a transmittal of the current value of the state object from the client back to the server. This simple mechanism can be used in management applications.

5.2 POS-EWS Web Compiler

We have also developed a Web compiler [25] 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 6. 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 underscore.

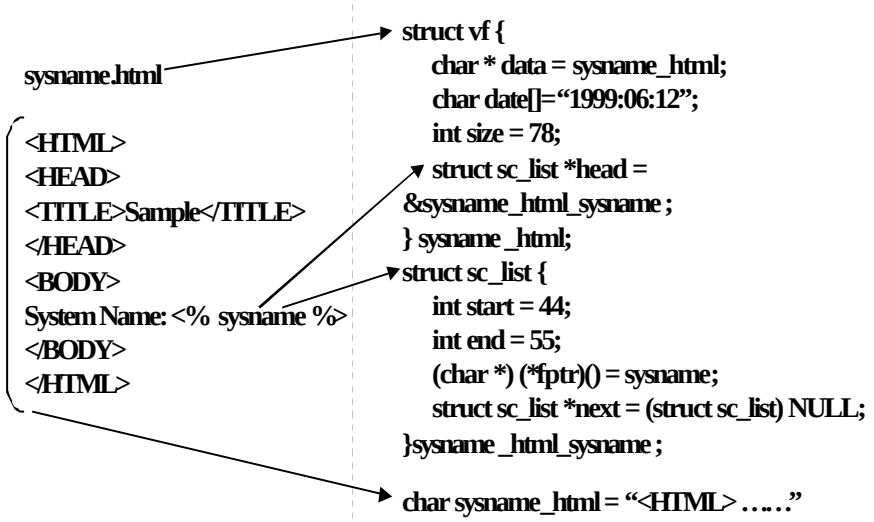

Figure 7. Virtual File System Code

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 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 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. What is more important is that 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* utility 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 that are 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. The library functions for the file interface are supported without any RTOS dependency.

5.3 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 8, 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 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 9 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 9 is Java applet which displays status of each port in real-time.

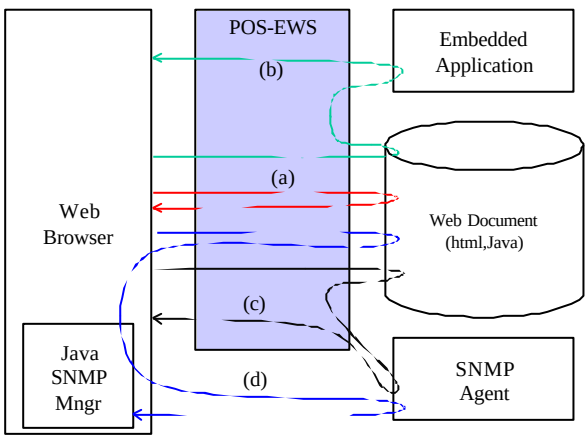

Figure 8. POS-EWS WebMUI Mechanism

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 9 shows the display result applied at menu.html of the WebMUI. Circles (a), (b), (c), (d) in Figure 9 show four different ways to retrieve data from an embedded system using the POS-EWS mechanism explained in Figure 8.

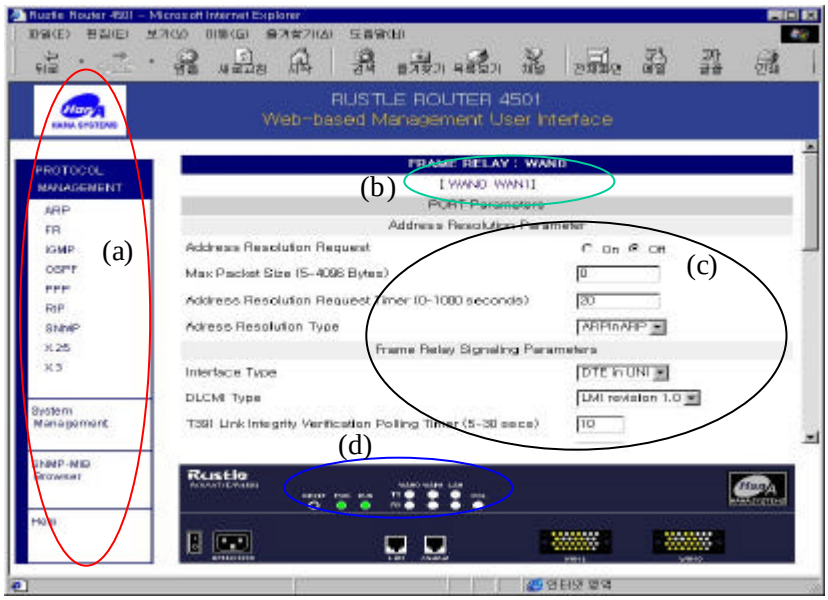

Figure 9. POS-EWS Application Interface Example

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.

6.1 Performance Metrics

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. Perception of this performance depends also on variables on the client side: the client platform and operating environment, and the Web client [26]. 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 the number of 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, along with the time it takes for a response 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.

- Connection latency is the time it takes to establish a connection.

- Request latency is the time it takes to complete the data transfer once the connection has been established.
- Network latency : transmission is determined by bandwidth and the physical limitations on speed that electrons can travel down a transmission medium. The distance between the client and server may be assumed to be fixed for the purposes of distributed networking. The speed of the transmission of electrons is a physical limitation, a function of the speed of light, and not subject to amendment.
- End-user latency is the sum of all latencies, including connection and request latencies, and network latency due to routers, gateways, etc.

?? 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.

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

In general, embedded systems must minimize the requirements for system resources, such as CPU and memory, since 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.

We select code size (memory footprint), run-time memory, CPU usage and maximum user connectivity (capacity of POS-EWS) as the performance elements of POS-EWS. The code size is approximately 30 Kbytes, the average run-time memory is 64 Kbytes, and the HTTP server has the lowest priority. 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 POS-EWS is so small that the performance issues are out of the question.

6.2 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, where a new process branches out 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 executed operations are significant, particularly since a typical Web request is very brief [27]. But the FSM supporting a 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/1.1 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 [7]. Requesting a Keep-Alive connection when GETing a file means the browser can reuse the connection to receive 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 [28]. 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.

7. 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. One capability designers should investigate is the server's tolerance to variations in the browser's command strings. The sequence of parameters in a command, for example, may vary from browser to browser. Handling such variations adds complexity to the server's code and increases the code size. An overly compact server may sacrifice this ability, and thus, may operate improperly with some browsers. On the other hand, if only one browser type connects to the server, the added complexity is unnecessary. All of the following products work with any standard browser, including Internet Explorer, Nets0cape Navigator or Communicator, Hot Java and Mosaic, on all platforms, allowing any network-connected browser user to easily access any device.

The table at the end of this section summarizes the options from the vendors detailed below.

7.1 Agranat - EmWeb

Agranat's EmWeb [29] takes a unique approach to dynamic content output and CGI-type input. Their HTML-to-C preprocessor allows for the insertion of proprietary tags containing C code or calls to C functions, the return values of which, usually text strings (char *), get inserted into the HTML stream served up to the client. This neat mechanism removes the need for SSI or costly "painting" of final HTML images in memory. The Agranat enhancement to HTML forms allows for straightforward data type specification in the code used process <INPUT> fields and for one to segregate the data into C structs.

Agranat also emphasizes the uniqueness of their approach, so much so that they have declared their intention to apply for a patent. They 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 using the HTML-to-C preprocessor approach. Like server-side scripting, this approach uses conventional Web authoring tools and browsers to support quick development and prototyping of Web pages. The Web pages are then enhanced with proprietary markup tags that encapsulate fragments of C source code. These code fragments provide simple and efficient interfaces to the embedded application software.

Also, the preprocessor tool compresses the Web pages, strips out the proprietary tags, parses HTML forms, and generates C code. The C code is then compiled and linked with the embedded Web server and application software to produce a tightly integrated executable image. 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 HTML-to-C preprocessor offloads substantial Web server processing from the embedded system. Furthermore, the embedded C code and form-parsing offer the greatest flexibility for developing embedded application code. This approach supports a small, efficient embedded Web server that increases system performance.

7.2 Allegro –RomPager

Bob Becker of Allegro positions his company's product as "rich in features with a very small footprint" and points to target applications in routers and hubs, as configuration tools, and in high-end printers, and smart interfaces. What is most interesting about RomPager [30] is its HTML compression method and the concomitant options for internationalization. The strategy of compressing

HTML, both static and dynamic, is to employ dictionary/token methods, wherein each HTML tag pair (e.g., <BODY> and </BODY>) is assigned a short or byte value for storage and is expanded only when served. Even highly adorned HTML tags, full of attributes could be compressed as templates. Allegro's RomPager takes this method one step further, offering options for assigning simple tokens to common text strings for storage and recalling the entire string from a dictionary when the page is served. This approach has the added benefit of supporting straightforward internationalization by substituting dictionaries for the same token values. RomPager also offers a feature to off-load storage from deeply embedded systems, allowing for redirection of GET requests for large files containing graphics or Java applets.

7.3 BVM – IntraScada Web Server

The IntraScada Web Server [31] is designed for OS-9 systems ranging from racked units with 68060 processors to embedded single board controllers using the CPU32 processor family. The target systems do not have to have a file storage system; if one is available, HTML pages and graphic images can be returned from files. Otherwise HTML pages with graphic images can be returned elsewhere on the network with only the dynamic data coming from the embedded system. It should be possible to install the server on existing control and data systems using the built-in facilities and support processes to interact with the present processes.

The server is able to produce a log, which, optionally, may show a combination of errors and activity for use during system debugging and proving. However it may be turned off in a finished system. The log level between 0 and 9 and a valid OS-9 path are set when the server is started. The code space required for disk based systems is typically less than 100Kbyte and the data space is 20Kbyte plus 28Kbyte for each connection.

7.4 Accelerated Technology – Nucleus WebServ

Nucleus WebServ [32] provides a utility to select files on a hard disk and convert them to a C file which contains the binary representation of the files and an associated directory. This serves as an embedded file system. The power of Nucleus WebServ is that it gives users the ability to intelligently use the files that WebServ is serving up to users. Nucleus WebServ also provides the ability to upload files from the Web browser.

7.5 QNX Software Systems Ltd - Voyager Web Server

Voyager Web Server [33] supports proxy server authentication and registers IP addresses so that clients can connect to the server if it is available. Designed to conserve system resources, Voyager Server uses DeviceSSL for building security into embedded systems. Computing resources such as the CPU, persistent storage and memory are in short supply on small computing devices. DeviceSSL therefore provides a critical success factor in designing software for embedded systems. DeviceSSL operates without the use of hard drives or traditional file systems, which are often completely lacking in embedded devices.

QNX provides full-featured and highly modular Voyager Web browser in addition to Web server. This browser supports all the current Internet standards – like JavaScript, HTML 3.2, frames, tables, proxy servers, SOCKS, server push, client pull, progressive image display, animated GIFs, JPEGs, FTP, basic and digest authentication, gopher and more.

7.6 Magma – Lava Family of Servers

The Lava server [34] product was one of the first to appear on the market and

is now, due to numerous upgrades, a state-of-the-art product. Magma CEO Omri Palmon points out that "Magma's first OEM contract was signed in 1995, and versions of Lava now run on many different OS platforms targeting 9 processors." Moreover, vendors of two popular RTOS resell Magma as part of their embedded offerings. When first encountered over two years ago, their focus was strictly on HTML. Currently, they have expanded their product line in several directions, with a lightweight product focused exclusively on serving Java applets, LavaApp; a high-end product called LavaPower; a product targeted at supporting on-line documentation needs, LavaDOC; an SNMP-oriented server called LavaSNMP; and a server for embedded PCs, LavaPC.

LavaSNMP is a Web server that creates Web functionality based on existing SNMP functionality. At the work of LavaSNMP, a user issues a Web request from his or her browser to the embedded system. LavaSNMP accepts the request, parses it, and sends back a Java applet. Then, the Java applet runs on the browser, enabling the browser to control the embedded system using SNMP messages. Consequently, LavaSNMP is a program that enables a manager to control an SNMP-managed embedded systems without requiring use of a management application. LavaSNMP provides a way to connect an SNMP-managed embedded system to the Web. And it also provides an interface to the Web without the cost of writing code or converting it.

7.7 Quiotix – QEWS

The Quiotix Embedded Web Server, QEWS [35], according to President Brian Goetz, "offers a highly modular architecture and a switchboard approach to CGI and security." Using a preprocessor/HTML compiler, the QEWS environment allows assignment of handlers to URLs in a virtual file system, with automatic generation of calls/entry points to extract input values from forms. Once development is complete, the entire application, including Web

pages, is automatically rehosted to the target environment without modification. And Quiotix's exclusive AutoSubSet feature automatically configures QUES so that only the features actually required by an application are present in the target environment. Another feature of this architecture is that the authentication security services that Quiotix offers can be applied to a hierarchy or tree of URLs just by encapsulating the handlers inside an authentication object.

The Quiotix product is extremely portable – it takes less than half a day to port QEWS to a LynxOS environment.

7.8 SpyGlass – MicroServer

SpyGlass is the best known of the companies herein, but is better known for its client browser product, Device Mosaic, than its MicroServer [36] embedded HTTP product. Paul Chapel of SpyGlass differentiates his company's offering in terms of "support, customization, and substance." The MicroServer product itself is substantial as well. Its somewhat larger footprint arises from a wide array of services, including basic and digest authentication security, an internal thread HTTP engine, and the SpyGlass Application Development Interface (ADI), a set of modules and tools for customization. SpyGlass also offers a thin server product for small, single-user applications that run in only 10K.

7.9 Web Devices (formerly CNiT) – Pico Server

The Web Devices Pico Server [37] was the first field to offer a broad range of security options, initially standard authentication and a proprietary triple DES scheme, and now SSL. The Pico Server offers encrypted PUSH of client files, PUSH technology, and a choice of conventional CGI or a plug-in interface with an API comparable to that of the Netscape server. In systems lacking a scheduler and/or a process address model, the Pico Server can be configured to

use the resident TCP/IP stack to queue request. Web Devices introduced its own embedded executive technology, PICOS, and also offers a scaleable embedded browser.

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.

Company & Product	OS supported	CPU supported	HTTP code size (version)	Features					
				SSI	VFS	Comp-iler	Compre-ssion	Security (encode)	Coo-kie
Agranat Systems, EmWeb	No OS	Any CPU with a C compiler	25kbytes (1.1)	O	O	O	Proprietary	Basic + Digest	
AllegroSoft, RomPager	Any RTOS, No OS	Any processor With an ANSI-C Compiler	10-40 kbytes (1.1)	O	File Sys.	O	Dictionary	Basic	O
BVM, IntraScada Web Server	OS-9	CPU32	< 100 kbytes (1.1)						
Accelerated Technology, Nucleus WebServ	Nucleus Plus	x86,68K,ARM, 683xx,SPARC, PowerPC,SH, H8/300H, TMS320C3x, MIPS,4x/5x/2x, Panasonic MN10200	40kbytes (1.0)	O			Proprietary	Basic (DES)	
Spyglass, MicroServer	LynxOS, QNX, OS-9, pSOS, VxWorks	Any CPU with a C compiler	35-110 kbytes (1.1)	O			None	Baisc + Digest + SSL	
QNX Software Systems Ltd, QNX Internet Toolkit	QNX real-time OS	x86, Pentium Pro, AMD Elan	106 kbytes (1.1)	O				Basic + Digest (encode)	
Magma, Lava	Any RTOS	Any CPU with a C compiler	15-40 kbytes (1.0)	O			Proprietary	Basic + SSL	
Quiotix, QEWS	pSOS, LynxOS, VxWorks	Any CPU with a C compiler	45-50 kbytes (1.0)	O	O		GZIP	Basic	
Web Device, Pico Server	LynxOS, Nucleus Plus, pSOS, VxWorks	Any CPU with a C compiler	15-30 kbytes (1.0)	O	O		ZIP-like	Basic + Digest + SSL	
POSTECH, POS-EWS	Real-time Xinu, pSOS	Any CPU with a C compiler	30kbytes (1.1)	O	O	O	GZIP/ CSS-Style	Basic + Digest (Base64)	O

Table 1. Comparison of EWS Products

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. 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 [30] 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.

8. Conclusion and Future Work

As the Internet continues to grow, the number of appliances and devices connected to the Internet will be exceed the number of users. Consequently, there is a strong drive for an embedded Web server to use in all of these devices. 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 equipment.

Embedded Web servers for Web-based network element management provide an administrator with a simple but enhanced and more powerful user interface without additional hardware. By embedding tiny Web servers into network devices, users can remotely monitor, configure, and diagnose vital systems. However these advantages may disappear without the application of embedded Web server technology.

In this paper, the basic technical concepts concerning efficient and lightweight embedded Web servers were outlined, 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, efficient and lightweight embedded Web server (called POS-EWS) based on the proposed architecture. 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. The EWS-WebMUI architecture provides easy integration platform with an SNMP agent. That is, POS-EWS provides easy and effective integration mechanisms for embedded management applications. We testified them through applying POS-EWS to management of a commercial router. This "webification" of network devices through EWSs 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 also plan to port POS-EWS to other CPUs and embedded OSs. Moreover, we want to make POS-EWS on chip for performance and efficiency. Another future work is to investigate methods for network management of devices equipped with EWSs.

References

[1] B. McCombie, "Embedded Web servers now and in the future," Real-Time Magazine, no.1 March 1998, pp. 82-83.

[2] A. Wilson, "The Challenge of embedded Internet," *Electronic Product Design*, January 1998, pp. 31-2, 34.

[3] I. Agranat, "Embedded Web Servers in Network Devices," *Communication Systems Design*, March 1998, pp. 30-36.

[4] C. Wellens, K. K. Auerbach, "Towards Useful Management," The Simple Times, 4(3):1-6, July 1996.

[5] W3C, "Hypertext Transfer Protocol-HTTP/1.1," Internet Draft draft-ietf-http-v11-spec-rev-06, HTTP Working Group, November 18, 1999.

[6] W3C, "HTML 4.0 Specification," Internet Draft REC-html40-19980424, HTML Working Group, April 1998.

[7] R. Fielding, "Hypertext Transfer Protocol - HTTP/1.0," RFC 1945, IETF HTTP WG, May 1996.

[8] T. Berners-Lee, "Uniform Resource Locators (URL)," Internet Draft, March 1994.

[9] N. Borenstein, N. Freed, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," RFC 1521, September 1993.

[10] Sun Microsystems Inc., "Java Management eXtension Architecture," June 1996.

[11] WBEM Consortium, "Web-Based Enterprise Management Proposal. HyperMedia Management Protocol Overview," Revision0.04. July 1996.

[12] C. Leidigh, "Implementing Web-Based Management of Networked

Devices," July 1997.

[13] J. Case, M. Fedor, M. Schoffstall, C. Davin, "The Simple Network Management Protocol (SNMP)," RFC 1157, May 1990.

[14] P. Mullaney, "Overview of a Web-based Agent". The Simple Times, Vol. 4, Number 3, July 1996.

[15] J. Franks, P. Hallam-Baker, J. Hostetler, P. Leach, A. Luotonen, E. Sink, L. Stewart, "An Extension to HTTP: Digest Access Authentication." RFC 2069. Northwestern University, CERN, Spyglass, Microsoft, Netscape, Open Market, January 1997.

[16] R. Fielding, "Hypertext Transfer Protocol - HTTP/1.1," RFC 2068 IETF HTTP WG, August 1996.

[17] J. Touch, J. Heidemann, K. Obraczka, "Analysis of HTTP Performance," USC/ Information Sciences Institute, June 1996.

[18] Henrik Frystyk Nielsen, et al., "Network Performance Effects of HTTP/1.1, CSS1, and PNG," W3C, June 1997.

[19] CGI, http://www.w3c.org/cgi.

[20] K. Arnold, J. Gosling, The Java Programming Language, Addison-Wesley, 1996.

[21] OMG, "The Common Object Request Broker: Architecture and Specification", Revision 2.0, July 1995, OMG TC Document.

[22] J. Y. Kong, J. W. Hong, "A CORBA-based Management Framework for Distributed Multimedia Services and Applications", Technical Report PIRL-TR-97-1, POSTECH, Korea, March 1997.

[23] Ching-Wun Tsai, Ruay-Shiung Chang, "SNMP through WWW," *International Journal of Network Management*, Vol. 8. 1998, p. 104-119.

[24] D. Kristol, L. Montulli, "HTTP State Management Mechanism," RFC 2109, February 1997.

[25] H. T. Ju, "POS-EWS Web Compiler for Supporting an Effective Application Interface", PIRL Technical Report, POSTECH, PIRL-TR-99-3, December 1999.

[26] James Rubarth-Lay, "Keeping the 400lb. Gorilla at Bay: Optimizing Web Performance," May 1996.

[27] V. Padmanabhan, J. Mogul, "Improving HTTP Latency," *Computer Networks and ISDN Systems*, v.28, pp. 25-35, December 1995.

[28] Edwards, Nigel, Owen Rees, "Performance of HTTP and CGI," ASNA.

[29] Agranat Systems, EmWeb, http://www.emweb.com/.

[30] AllegroSoft, RomPager, http://www.allegrosoft.com/.

[31] BVM IntraScada, Web Server, http://www.bvmltd.co.uk/.

[32] Accelerated Technology, Nucleus Embedded Software, http://www.atinucleus.com/.

[33] Spyglass, MicroServer, http://www.spyglass.com/.

[34] QNX Software Systems Ltd, QNX Internet Toolkit, http://www.qnx.com/.

[35] Magma Information Technologies, LAVA, http://www.magmainfo.com/.

[36] Quiotix Corporation, QEWS, http://www.quiotix.com/.

[37] Web Device Inc., Pico Server, http://www.webdevice.com/.

요 약

1990 년대 초반부터 관심이 집중되어 온 World Wide Web 은 인터넷을 대중화 시키고 다양한 멀티미디어 관련 응용 프로그램의 개발과 사용을 촉진시켰다. 이러한 웹 기술은 어떤 운영 체제나 플랫폼에서도 사용할 수 있고 문자에서 이미지, 음성과 영상까지, 통신망 상에 분산되어 있는 다양한 형태의 데이터들을 간단하면서도 강력한 방식으로 쉽게 제공 받을 수 있도록 함으로써 가장 각광 받는 기술 중의 하나로 주목되고 있다. 이런 대중화된 웹 기술을 network management 에도 적용할 수가 있다. 특히 network element management 에 적용하여 Web-based management user interface 를 제공할 수 있는데, 이것을 제공하기 위한 가장 직접적인 방법이 network device 에 Web server 를 내장 시키는 것이다.

작은 웹 서버를 네트워크 장비나 가정 용품, 사무 용품 등의 시스템에 내장 시킴으로써 관리자는 원거리에서 웹을 통해 시스템을 구성하고, 모니터링하고 제어할 수 있다. 이렇게 내장형 웹 서버를 사용하는 것은 사용자에게 친숙하고 값싸고 플랫폼에 자유로우며 네트워크가 준비된 그래픽한 user interface 를 제공한다. 또한 내장형 웹 서버 기술은 정교하고 사용하기 편한 그래픽한 인터페이스를 시스템에 빠르고 쉽게 넣는 것을 가능케 하므로 관리 시스템의 개발에 드는 시간을 줄이고, 유지 보수도 쉽게 이루어지게 한다.

본 논문에서는 이런 장점을 살려, POS-EWS 라는 작고 효율적인 내장형 웹 서버의 설계 및 구현 방법에 대해 고찰해 본다. 우리는 HTTP/1.1 Protocol 에 맞게 persistent connection 과 cache control 을 지원하며, Virtual File System 을 가지고 있으며, security 를 위한 security module 과 configuration module 을 가지고 있는 POS-EWS 를 개발하였다. 또한 이것은 내장형 시스템에서 가장 중요한 application interface 를 위한 SSI Style 을 지원하여 동적인 시스템의 정보를 제공한다. Embedded System 에 내장된 POS-EWS, Management Application 그리고 Embedded Application 은 실시간 운영 체제인 RTOS (Real Time Operating System)상에서 수행된다.

내장형 시스템의 적은 자원을 고려하여 POS-EWS를 탑재되기에 충분히 작은 사이즈로 만들었고, 시스템의 주 업무를 방해하지 않고 CPU utilization을 적게 하기 위하여 FSM 형태의 하나의 thread로 동작하면서 가장 낮은 수행 우선순위를 가지고 수행되게 만들었다. 개발된 POS-EWS를 상용 라우터에 적용하여 웹 기반의 망 관리 기능을 제공하도록 만들어 성능을 확인하였다. 또한 SNMP와 효과적으로 통합할 수 있는 방법도 제공하였으며 웹을 통하여 실시간 처리가 가능한 방법도 제시하였다. POS-EWS의 네 가지 내장 applications과의 통합 방법을 통해 관리되어야 할 모든 정보를 웹 기반으로 효율적으로 제공할 수 있다. 즉 작고 효율적인 내장형 웹 서버는 내장 시스템에 기능 장애를 야기하지 않고, 관리 기능을 제공할 수 있는 가장 직접적이고 효과적인 방법을 제공한다.