Event-Driven Extraction of HTTP Payload From Concurrent Streams

Mingzhe Li[a,b], Jinlin Wang[a], Xiao Chen[a], Jun Chen[a]
[a] National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing 100190, China
Email: {limz, wangjl, xxchen, chenj}@dsp.ac.cn
[b] University of Chinese Academy of Sciences, Beijing 100190, China

Abstract— **Many network devices undertake tasks that involve extracting HTTP payload from a large number of data streams. In order to model such tasks and thus develop feasible approaches, this paper analyzes action characteristics of network byte stream reading and grammar of HTTP response message encoding. Based on this analysis, we propose a parsing algorithm for concurrent HTTP response streams, built upon stack buffer peeking operations and the extended finite state machine model. Implementation issues on parallel platforms are also studied in this paper. Our scheme is event-driven, does not require full buffering of the whole HTTP message, and saves one memory copy compared to naive static parsing method. Test results show that our method achieves better performance in terms of CPU and memory consumption.**

Index Terms— **HTTP response, chunked encoding, extended finite state machine**

I. Introduction

HTTP [1] is a most frequently used application-level network message exchange protocol [2]. Usually an HTTP client initate a TCP connection by which a request messages are sent to an HTTP server and response messages are returned to the client. A response message contains the requested object, which starts with a status line, serveral header lines, an empty line, and then comes the object itself. HTTP/1.1 version introduces chunked encoding, where the requested object is cut into data chunks, each of which is sent to the client in a streamlined fashion before later chunks gets ready. This is useful for a dynamically generated object whose size is not known until the whole object is completed created after considerable delay. In some cases compression is done before the object get transmitted. If chunked encoding is applied on compressed blocks [3], the sending process can be carried out simultaneously with the compression algorithm, thus the client may receive the first blocks before the whole compression process is finished.

Network nodes working as the client side maintain multiple HTTP sessions, and concurrently receive response data streams from which payload is extracted from chunked encoding in a protocol translating step. For instance, some gateways translates HTTP messages to a strictly MIME-compatible protocol which does not support chunked encoding [1]. Video streaming servers in a television broadcasting network would concurrently receive chunked encoded media data streams from a CDN(Content Distribution Network) and transform the data to a format recognizable by QAM systems [4] [5]. Proxy servers might also parse chunked encoding to make sure there is no more data passing through.

[1] mentions a naive algorithm for chunked encoding parsing, where network receive operations are not considered, which means that the whole message is downloaded into a continous buffer before parsing. However, the use of chunked encoding indicates a huge message size and an unaccetable buffer consumption. Even if unlimited memory resource is assumed, the non-streamlined nature of the process indicates that, during the rather long process of network reading, no parsed data is output, while during parsing step data bursts out [6]. This kind of jitter is inappropriate for some applications. Thus we consider intermixing network reading and protocol parsing in a finer grain to reduce buffering cost as well as to smooth output.

The two most frequently employed concurrency model for network applications are respectively multi-threading and event-driven [7] [8] [9]. In a multi-threading model each thread handles the processing of a single session, while in event-driven model streams share one executing context and occupy computation resource in a time-division fashion. Event-driven model provides better performance, flexibility and portability [10], and is adopted by this paper. To fully utilize CPU resources, each read operation on network sockets should be non-blocking, and each session ought to release its hold of CPU time when no more data is available for the moment. In this approach, however, a session might be switched out with a partially read message field, which complicated algorithm design. Besides, a highly concurrent network device would execute a lot of reading and parsing operations, making efficiency a crucial issue. Thus it is a challenging as well as rewarding to carefully design a correctly running and light-weighted algorithm for HTTP parsing.

The rest of the paper is organized as follows. Section II reviews the preliminaries including supporting concepts and techniques used for later discussion. In section III, we propose our light-weighted HTTP parsing algorithm. Performance evaluation is presented in section IV, and section V concludes this paper.

II. Preliminaries

To the best of our knowledge, no published research is devoted to chunked-encoding parsing in a highly concurrent scenario. To address this problem, this paper proposes a HTTP parser for concurrent response streams based on protocol peeking operations and the extended finite state machine(EFSM) model [11]. To better discribe our work, this section makes a brief discussion on EFSM, peeking operations and HTTP response message syntax.

A. Extended Finite State Machine

A finite state machine(FSM) [12] is a tool used to model finite number of object states and transitions between the states. This technique is widely employed in various fields including electronics, software engineering, logic calculus and linguistics. Specifically, FSM often plays a key role in work related to network protocol recognition and parsing [13]. Such a FSM can be expressed as a quintuple:

$$M = (Q, \Sigma, q_0, \delta, F) \tag{1}$$

where $Q = \{q_0, q_1, \cdots, q_n\}$ is a non-empty finite set of states, $q_0 \in Q$ is the initial state for the FSM, $\Sigma = \{\sigma_1, \sigma_2, \cdots, \sigma_m\}$ is a finite set of input alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function specifying effect of input $\sigma \in \Sigma$ on states. $F \subset Q$ is a set of accepting states often indicating a successfully matched pattern.

Such a model is suitable for protocol field matching tasks, but as data receiving work comes into play in our discussion, the FSM falls short of our need. To enhance the expression power of FSM, EFSM is proposed [14]. For convenience, we adopt the following definition of EFSM among those given in the bulk of literature [11] [15].

$$M = (Q, \Sigma, I, V, A, \Lambda) \tag{2}$$

where $Q(\neq \emptyset)$ is a set of states, and each $q \in Q$ can be atomic or composite. $\Sigma = \{\sigma_1, \sigma_2, \cdots, \sigma_m\}$ is a set of events. $I \subset Q$ is the set of initial states for each composite state. V is a set of variables globally shared by each $q \in Q$. A is a set of actions in response to events. $\Lambda : Q \times \Sigma \rightarrow Q \times A$ is a set of transitions, where each transition can be denoted as $\lambda \in \Lambda : (q, \sigma) \rightarrow (q\prime, a), \sigma \in \Sigma, a \in A$, meaning that σ triggers a transiton from q to $q\prime$ as well as an action denoted by a. This transition can also be represented in a diagram by an arc connecting two nodes standing for q and $q\prime$ respectively, labeled as σ/a. Q,Σ, I, V, A and Λ above are all finite sets.

B. Peeking Operation

As an HTTP message typically uses TCP as its transport layer vehicle, it can be viewed by the application level as a lossless and order-preserving byte stream:

$$\Theta = (b_1, b_2, \cdots, b_t, \cdots) \tag{3}$$

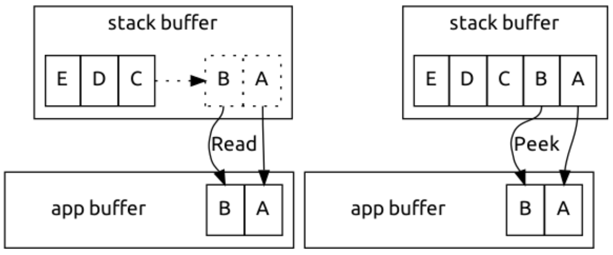

Figure 1. Peeking and Normal Reading

As the TCP stack must perform reliability guarantee measures, incoming packets will first be buffered into a sliding window and remain unavailable for some moments. It is not until a sequential range of octes are fully received and examined for correctness that the application is permitted accessibility to these data. We refer to the moment that a octet b_i is made available for application as $ReadyTime(b_i)$. Usually socket reading interfaces are designed in a way that applications can only fetch the ready octets in a sequential way. The above mechanism places a constraint on the octets: $\forall b_i, b_j \in \Theta, i < j$,

$$\begin{aligned} ReadyTime(b_i) &\leq ReadyTime(b_j) \\ ReadTime(b_i) &\leq ReadTime(b_j) \\ ReadTime(b_i) &> ReadyTime(b_i) \end{aligned}$$

When an application performs a read operation, byte count and destination buffer address must be specified. For a non-blocking reading, the number of successfully fetched bytes are often smaller than specified. By default, after a range of octets are read to application buffer, they are automatically deleted from stack buffer, so that the next read will get octets right after them. But if the read action is specified as a peeking operation, the fetched data remains in stack buffer. A following read, whether peeking or not, can still get the same data, as shown in Figure1. Socket APIs on Windows and the Unix family both support peeking operations. Although Java does not support peeking, the action can be implement by a programmer with an additional application-layer buffer [16]. Embedded systems that fail to provide support for peeking can also realize the function in a similar way with [16].

C. Syntax of HTTP response

To facilitate discussion in following sections, we describe syntax of HTTP response message in production rules as:

$$\Theta \to RLB \tag{4}$$
$$L \to N\mu_1 VeL | N\mu_1 Ve \tag{5}$$
$$B \to C\chi \tag{6}$$
$$C \to eFe\Omega C | eFe\Omega \tag{7}$$
$$F \to S\Gamma \tag{8}$$
$$\Gamma \to \mu_2 N \mu_3 V\Gamma | \epsilon \tag{9}$$
$$S \to hS|h \tag{10}$$
$$\Omega \to t\Omega | t \tag{11}$$

where Θ denotes the whole response message, R is the leading status line, L a number of header lines, B is the chunk-encoded body, and F is a marker field at the beginning of each chunk. ϵ represents an empty character string, e stands for carriage-return and line-feed string, i.e. 'CR LF', h denotes any hexadecimal character which can also be expressed as the regular expression `[0-9a-fA-F]`, and χ stands for the terminating string for chunked-encoding, i.e.'CR LF 0 CR LF'. μ_1, μ_2 and μ_3 each represents a character: ':',';' and '=' respectively. t is used to stand for any octet.

B contains at least one data chunk, and its termination is marked by χ. C stands for consecutive chunks, each starting with F and followed by e and then payload data Ω. In F, the optional Γ may consist of several extention fields. These fields will be ignored by the receiving side if not recognized. S represents the amount of payload data bytes in hexadecimal characters which should be equal to the actually size of Ω. N and V are character strings used here to denote a field name and its value respectively.

HTTP grammar can be expressed in ABNF notation [17], just like the SIP protocol [18]. Work has been done to generate SIP parser from general ABNF parser generators [19] [20], so it is reasonable to consider creating a HTTP parser in the same way. But extra effort is needed to evaluate the feasibility and performance of such an approach, and this paper is not going to further discuss this topic.

A naive HTTP response parser, as shown in Algorithm 1, can be easily conceived, which requires the whole message is buffered before parsing and will be refered to as a static parsing method hereinafter. During parsing, payload is extracted and output to another buffer.

The static parsing approach results in expensive buffering cost. Besides, the whole parsing step has to be scheduled as an atomic operation in multi-task scenarios, making it unsuitable for real-time systems and jitter-sensitive applications.

III. THE PROPOSED SCHEME

To overcome the high overhead and other issues of the static parsing method, we put forward a light-weight algorithm, which is based on the EFSM model that is extensively used in protocol parsing related tasks, and resorts to peeking operations to solve the broken-field problem.

Algorithm 1 Static Parsing

```
1: procedure SP($\Theta$)
2:     $p \leftarrow \Theta$                    ▷ p is a pointer
3:     $p \leftarrow parseR(p)$                 ▷ parse R
4:     $p \leftarrow parseL(p)$                 ▷ now p points to B
5:     while True do
6:         $l_\Omega \leftarrow ToInt(p)$
7:         if $l_\Omega = 0$ then
8:             return                           ▷ $\chi$ deteced
9:         end if
10:        $F \leftarrow matchF(p)$
11:        $p \leftarrow p + len(F)$
12:        $\Omega \leftarrow p \ldots p + l_\Omega$
13:        output $\Omega$
14:        $p \leftarrow p + l_\Omega$
15:    end while
16: end procedure
```

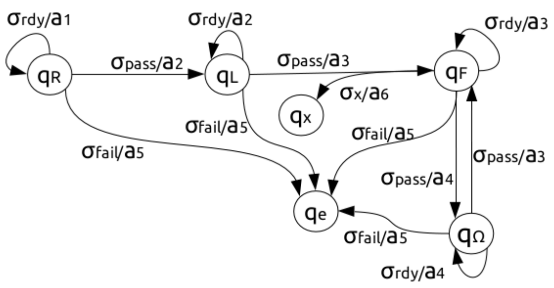

Figure 2. States Diagram for the Proposed Method

The following terms are used in our algorithm: Objects defined in formula (4) to formula (11) other than Ω will be refered to as metadata fields. $mread$ and $dread$ represent one normal read operation on metadata and payload data respectively. $s = peek(n)$ denotes that an application requests a peeking operation for a range of n bytes in a non-blocking way, and actually gets a content string, written as s, which can be empty. Both normal and peeking read interface require length and destination buffer specified as parameters. For $mread$ and $peek$, the destination parameter is specified as a temporary application-layer buffer, whereas $dread$ can output directly to an application-specific data port to save one memory copy operation.

According to the network bytes stream model depicted above, one read on a metadata field might only fetch it partially. The leftover part might not be available until a long time has passed, after which an extra reading effort will be responded with the suffixing bytes of that field. This brings in trouble for field recognization, and might involve text concatenation. As mentioned above, peeking operations can solve the problem neatly. Whenever a field is to be matched, a peeking operation is issued, specifying range length equal to the maximum length of that field. If the complete field is fetched, an $mread$ is issued to erase the field from stack; otherwise, the current session(data stream) would yield CPU leaving stack buffer the way it was as if no effort on the field had been made.

A. Design of EFSM

The procedure to receive and parse a single HTTP stream works as follows: First parse the status line. If the status code indicates success(such as 200, 206, et.al), work is continued; otherwise the procedure is exited as a failure. Nextly header lines are parsed from which application-specific useful infomation might be extracted. Then come a series of F and Ω pairs. In the end, χ is detected while expecting F, indicating end of message.

Integrating formula (2) and syntax of Θ, an EFSM can be devised to model the above procedure for single stream parsing:

1) A set of states $Q = \{q_R, q_L, q_F, q_\Omega, q_e, q_x\}$, where each element in Q corresponds to a state when R is to be parsed, L to be parsed, F to be parsed, Ω to be extracted, an erroneous state and a completed state, respectively.
2) $\Sigma = \{\sigma_{rdy}, \sigma_{pass}, \sigma_{fail}, \sigma_x\}$ is a set of events, in which σ_{rdy} is a data-ready message, σ_{pass} is generated whenever a pattern is matched, σ_{fail} indicates any error during parsing, and σ_x implies χ is detected.
3) $V = \{v_l, v_e\}$ where v_l stands for the amount of bytes left there for the current chunk, and v_e stores error information recorded before entering q_e.
4) A set of transitions, Λ, as shown in Figure 2, will be detailed later.
5) A set of actions, $A = \{a_1, a_2, \cdots, a_6\}$, is a collection of any action taken during state transitions.

Action a_1 is a routine to read and parse the status line. If HTTP status code is unexpected in a_1, σ_{fail} is thrown; otherwise, the status line is erased from stack by $mread$.

Algorithm 2 Action a_1

```
1: procedure $a_1$
2:     $s \leftarrow peek(16)$
3:     if $R \subsetneq s$ then
4:         Exit                              ▷ $R$ not ready
5:     end if
6:     if $statusBad(s)$ then
7:         Throw($\sigma_{fail}$)
8:     end if
9:     $l_R \leftarrow len(getR(s))$
10:    $mread(l_R)$                   ▷ erase $R$ from stack buf
11: end procedure
```

a_2 is a routine to read and parse header lines. A tricky part is to make sure whether all header lines are completely loaded into the temporary buffer for parsing. As suggested by formula (6) to (8), recognition of ee can be used as the criterion. Extraction of useful infomation is application-specific, and will not be discussed further in this paper. As with a_1, L should be erased from stack with $mread$ after completely fetched by $peek$.

a_3 acquires useful info from F. Length of F is to be estimated before peeking. Unless in some rare cases, F does not contain extension fields Γ and cannot occupy more than 16 bytes of space. If F is identified exceeding the estimated length, additional peeking attempts are to be made with longer length. Once F is completely fetched, a_3 calls $parse\Gamma$ to obtain info from Γ, and calls $parseS$ to compute payload length of the current chunk. Both $parse\Gamma$ and $parseS$ throw σ_{fail} on error.

Algorithm 3 Action a_3

```
1: procedure $a_3$
2:     $r_{len} \leftarrow 16$
3:     while True do
4:         $buf \leftarrow peek(r_{len})$
5:         if $len(buf) < r_{len} \wedge F \subsetneq s$ then
6:             Exit                          ▷ $F$ not ready
7:         end if
8:         if $F \subset s$ then
9:             break                         ▷ $F$ ready
10:        end if
11:        $r_{len} \leftarrow r_{len} * 2$
12:    end while
13:    $parse\Gamma(buf)$
14:    $v_l \leftarrow parseS(buf)$
15: end procedure
```

a_4 outputs Ω data until CPU slice for the current data stream is run out, or no more data available for the moment. a_5 handles σ_{fail} event, and a_6 deals with σ_x, both of which are application-specific.

Algorithm 4 Action a_4

```
1: procedure $a_4$
2:     $slice \leftarrow 0$
3:     while True do
4:         $buf \leftarrow dread(2048)$
5:         if $len(buf) \leq 0 \vee slice > maxSlice$ then
6:             Exit
7:         end if
8:         output $buf$
9:         $v_l \leftarrow v_l - len(buf)$
10:        $slice \leftarrow slice + len(buf)$
11:    end while
12: end procedure
```

Algorithm 5 Action a_5

```
1: procedure $a_5$
2:     $v_e \leftarrow errInfo()$
3:     goto $q_e$
4: end procedure
```

The above state machine is used to implement our HTTP response payload extraction module, where receiving and parsing of network data are interweaved.

Data fetched by a single read operation is immediately parsed, and thus a pair of read and parsing operations constitute one task slice as a unit for CPU scheduling. Slices further aggregate to compose a whole extraction task for one data stream. Payload is delivered to a data output port of this module. The nature of this output port is application-specific, which can be a packet queue for network transmission, or a buffer for disk writing. One single process can handle multiple stream, and CPU scheduling for the stream tasks is driven by events.

B. Implementation of the EFSM

Each data stream, which corresponds to a TCP connection and an HTTP session, possesses one EFSM designed above, and memory is allocated to hold the stream's working set, including V, q of EFSM and other application variables. State machines can be implemented with a matrix of action handlers, and each of the handlers is composed of code to execute for a specific state-event pair. For instance, in C, a state machine can be implemented with a two-dimension array of pointers to functions, each of which corresponds to an action and takes two variables representing state and event type respectively as its arguments [21]. For experienced programmers, such explicit form of a state machine is unnecessary, since the logics of the EFSM can be implicitly embodied by flow control statements, such as **if** and **switch** in C, with better efficiency.

Size of each task slice should be controlled within a desirable range. By limiting bytes count for each read operation, upper limit of a slice can be guaranteed. A non-blocking read might fetch nothing, resulting in an empty slice, which is wasteful since it achieves nothing but still costs a few computation cycles. By issuing a read operation only after a σ_{rdy} arrives on the same stream, empty slices can usually be avoided, and thus the lower limit of a slice can also be controlled.

Reckless implementation of a_4 might introduce a problem: if a $slice$ yields execution due to upper limit, and no more σ_{rdy} is generated for this task since all data is ready, this session will never obtain CPU again, leaving the last few bytes forever unread in stack. Approaches can be taken to deal with the situation:

1) Application provides a timer to periodically check and cleanse inactive streams.
2) Throw one σ_{rdy} when a_4 quits due to slice limit.
3) Set $\mathrm{maxSlice}$ to infinity so that a_4 does not quit due to slice limit.

C. Implementation of the event system

The application is required to provide an event system for the state machine, and keep checking if a new event is generated. Events are embodied by messages, including network messages and application level ones. Network messages are notifications of stack events like successful establishment of connections, arrival of packets, and network errors. The messaging mechanism is often provided by the specific stack interface or application framework (like ACE) in use, while in other cases the application developers have to build their own messaging functionality. For example, epoll is a widely used I/O multiplexing network programming interface [22], where one process or thread actively polls for all TCP connections for new events, and invokes appropriate callback functions for events that do arrive. Especially, when a data-ready event σ_{rdy} is generated, a corresponding handler is called, which locates the corresponding data stream and issues a read operation on it, and usually one transport-layer segment can be obtained. Epoll's active polling interface(`epoll_wait`) can be specified to return after some timeout, which gives the application a chance to poll for application level messages as well.

Such an event-driven multi-stream processing program can easily scale in a multi-processor, multi-process or multi-thread system, which are basically similar in term of scheduling paradigm. To implement our algorithm in a multi-processor system, each processor receives a subset of all messages. Measures must be taken to prevent two processor simultaneously accessing the same EFSM. One possible solution is to add a spin lock variable to set V and each processor has to acquire the lock before operating on the state machine. Other solutions exempt applications from explicitly dealing with syncronization by providing hardware-supported automatic syncronization control of multiple execution contexts, like in Cavium OCTEON [23]. As an optimization technique, an application might want to schedule events involving the same stream to the same hardware processing unit in order to achieve better cache friendliness [24].

IV. Performance Evaluation

Now consider performance of HTTP payload extraction algorithms, especially the efficiency to parse B, since the cost of R and L are relatively minor. We implemented and tested the static parsing algorithm(referred to as SP) and our proposed peeking-based algorithm(referred to as PEEK), using CPU consumption and memory overhead as metrics. The host machine that runs our test program is equipped with Intel Pentium Dual-Core E5700 CPU, 2GB RAM and Linux Kernel 3.8.0-32. Test programs are written in Python.

Different test cases are parameterized by different length of message. For each test case, an HTTP server generates HTTP response streams using chunked-encoding, with chunk size fixed to 1KB. Two HTTP clients run SP and PEEK respectively. Peeking operations are well supported in Linux both for Python and for C, by specifying a `MSG_PEEK` flag when invoking a read operation. Both clients output extracted payload into a fixed-size buffer, and when the buffer is full, additional payload is redirected to the beginning of the buffer covering old data. That means the output buffer is used in a recycled fasion, and payload is actually discarded. Server side and client side run on the same host, communicating with socket interface.

To measure CPU consumption, overall process time for data receiving together with parsing is collected. We use a Python module named Timeit to execute test cases repeated and calculate average running time. The memory overhead metric is represented by physical memory usage allocated by Linux. In Linux's /proc file system are process-related files containing information about the specific process. The ps command built on top of this file system can also server as a means to collect resource usage statistics, as adopted by this paper.

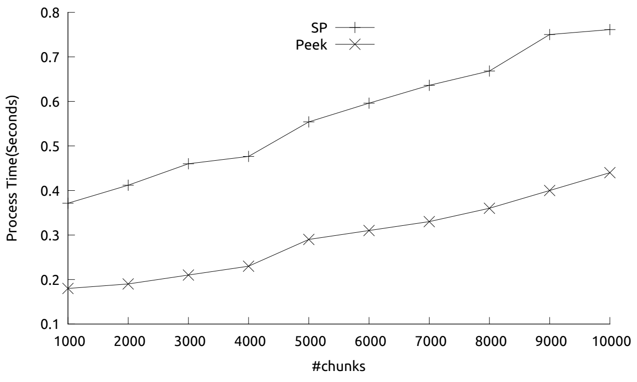

Figure 3. CPU Consumption

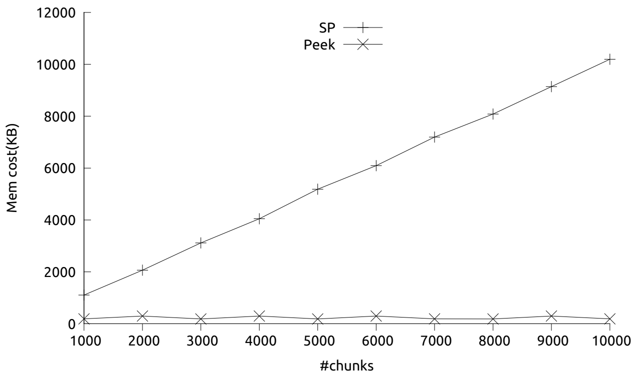

Figure 4. Memory Overhead

As shown in Figure 3, PEEK algorithm costs much less CPU time than SP to parse the same message stream. The gap widens almost linearly as message length grows. PEEK parses data right after fetching them leading to a cache-friendly a time locality characteristic. Besides, SP introduces an extra memory copy of data stream by loading a whole HTTP message into a parsing buffer. Actually, each payload byte is copied from NIC buffer to stack buffer, then to parsing buffer, and finally to output buffer. As for PEEK, the parsing buffer step is circumvented.

The result for memory cost measurement is shown in Figure 4. SP's memory cost grows linearly with message length, while PEEK's memory cost remains unchanged. The reason behind this is clear, as the parsing buffer of SP entails a $O(n)$ space complexity.

V. Conclusions

Many network devices are required to handle the parsing of highly concurrent HTTP response streams. Selection of parsing algorithms greatly affects system resource consumption, and thus influences concurrency capacity and transmission QoS. After analysis of network bytes receiving model and HTTP syntax production rules, this paper introduces an event-driven strategy to extract HTTP payload data concurrently. This method manages task slices for a data stream with an extended finite state machine, and solves the broken-field problem by stack peeking operations. Implementation details and optimization tips are also discussed in this paper. We evaluated the performance of the propsed strategy against a naive algorithm on a Linux host. Test results show that our approach significantly reduces computation and memory overhead.

Like HTTP, RTP and SIP are also widely used stream transmission protocols. In future we will investigate on concurrent parsing of RTP and SIP data streams.

Acknowledgment

This work is supported by the Key Technology Research on Architecture of General Technology and Standard and Service Systems of NGB (China's National Key Technology R&D Program under Grant No. 2012BAH02B01) and the Key Technology Research and Verification of the NGB Intelligent Access Gateway(Key Deployment Project of Chinese Academy of Sciences under Grant No. KGZD-EW-103-4). We are sincerely grateful to their support.

References

[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol –HTTP/1.1," 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2616.txt

[2] C. Huang, J. Wang, H. Deng, and J. Chen, "Mining Web Logs with PLSA Based Prediction Model to Improve Web Caching Performance," *Journal of Computers*, vol. 8, no. 5, pp. 1351–1356, May 2013.

[3] F. Ernawan, N. A. Abu, and N. Suryana, "Integrating a Smooth Psychovisual Threshold into an Adaptive JPEG Image Compression," *Journal of Computers*, vol. 9, no. 3, pp. 644–653, 2014.

[4] X. Yang, "An Alternative to Internet-The Next Generation Broadcasting Networks Standard," in *2010 2nd International Conference on Information Science and Engineering (ICISE)*, Hanzhou, 2010, pp. 1823 – 1826.

[5] A. Breznick, "A Switch in Time: The Role of Switched Digital Video in Easing the Looming Bandwidth Crisis in Cable," *Communications Magazine, IEEE*, vol. 46, no. 7, pp. 96–102, 2008.

[6] A. A. McEwan and I. F. Mir, "Age Distribution Convergence Mechanisms for Flash Based File Systems," *Journal of Computers*, vol. 7, no. 4, pp. 988–997, 2012.

[7] A. Erbad, N. Hutchinson, and C. Krasic, "DOHA: scalable real-time web applications through adaptive concurrent execution," in *Proceedings of the 21st international conference on World Wide Web*, 2012, pp. 161–170.

[8] X. Wang, “Technical Analysis of High-Capacity and Concurrency Server Groups,” in *E-Product E-Service and E-Entertainment (ICEEE), 2010 International Conference on*, 2010, pp. 1–4.

[9] D. Kimpe, P. Carns, K. Harms, J. M. Wozniak, S. Lang, and R. Ross, “AESOP: Expressing Concurrency in High-Performance System Software,” in *IEEE Seventh International Conference on Networking, Architecture, and Storage*. Ieee, June 2012, pp. 303–312.

[10] T. Y. Arif and R. F. Sari, “Throughput Estimates for A-MPDU and Block ACK Schemes Using HT-PHY Layer,” *Journal of Computers*, vol. 9, no. 3, pp. 678–687, 2014.

[11] V. Alagar and K. Periyasamy, “Extended Finite State Machine,” in *Specification of Software Systems*, 2nd ed., ser. Texts in Computer Science. London: Springer London, 2011, ch. 7, pp. 105–128.

[12] M. Yang, “State Assignment for Finite State Machine Synthesis,” *Journal of Computers*, vol. 8, no. 6, pp. 1406–1410, 2013.

[13] V. S. Alagar and K. Periyasamy, *Specification of software systems*, 2nd ed. London: Springer, 2011.

[14] K. T. Cheng and A. S. Krishnakumar, “Automatic Functional Test Generation Using the Extended Finite State Machine Model,” in *Proceedings of the 30th International Design Automation Conference*, ser. DAC ’93, 1993, pp. 86–91.

[15] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt, “Control dependence for extended finite state machines,” in *Fundamental Approaches to Software Engineering*. Springer, 2009, pp. 216–230.

[16] J. Heaton, “Peekable InputStream,” 2008. [Online]. Available: http://www.heatonresearch.com/articles/147/page2.html

[17] D. Crocker and P. Overell, “Augmented BNF for syntax specifications,” *RFC 5234*, 2008. [Online]. Available: http://tools.ietf.org/html/rfc5234

[18] J. Rosenberg and H. Schulzrinne, “SIP:session initiation protocol,” *RFC 3261*, 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3261.txt

[19] S. Tan, Y. Ge, and K. Tan, “Dynamically loadable protocol stacks-a message parser-generator implementation,” *Internet Computing, IEEE*, vol. 8, no. 2, pp. 19–25, 2004.

[20] T. Stefanec and I. Skuliber, “Grammar-based SIP parser implementation with performance optimizations,” in *Proceedings of the 11th International Conference on Telecommunications (ConTEL)*, 2011, pp. 81–86.

[21] P. der Linden, *Expert C programming: deep C secrets*. Prentice Hall Professional, 1994.

[22] “EPOLL,” 2012. [Online]. Available: http://man7.org/linux/man-pages/man7/epoll.7.html

[23] “Cavium Networks.” [Online]. Available: http://www.cavium.com/

[24] P. He, “Studying Flow Based Packet Scheduling and Transimission on Multi-core Processor,” Ph.D. dissertation, Graduate University of Chinese Academy of Sciences, 2010.

Mingzhe Li received his B.S. degree in automatic control from University of Science and Technology of China in June 2010. He is currently working towards his Ph.D. degree in signal and information processing at University of Chinese Academy of Sciences. His current research interest includes network processors and and network streaming media.

Jinlin Wang received his B.S. degree in mathematics from University of Science and Technology of China Institute of Technology in 1986 and his M.S. degree in acoustics from Institute of Acoustics, Chinese Academy of Sciences in 1989. He is the director and a Research Fellow in the National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing. He is also a professor in University of Chinese Academy of Sciences. His research interest includes digital signal processing, IP network technology and network streaming media.

Xiao Chen received his B.S. degree in software from Huazhong Institute of Technology in 1984 and his M.S. degree in acoustics from Institute of Acoustics, Chinese Academy of Sciences in 1989. He is a Research Fellow in the National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing. His research interest includes digital signal processing, multimedia networks and network communication.

Jun Chen received her Ph.D. degree in signal and information processing from Graduate University of Chinese Academy of Sciences in June 2004. She is Research Associate in the National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing. Her research interest is multimedia networks and information security.