The Reverse C10K Problem for Server-Side Mashups

Dong Liu and Ralph Deters

Department of Computer Science
University of Saskatchewan
Saskatchewan, Canada
dong.liu@usask.ca, ralph@cs.usask.ca

Abstract. The original C10K problem [1] studies how to provide reasonable service to 10,000 simultaneous clients or HTTP requests using a normal web server. We call the following problem the reverse C10K problem, or RC10K — how to support 10,000 simultaneous outbound HTTP requests running on a web server. The RC10K problem can be found in scenarios like service orchestrations and server-side mashups. A server-side mashup needs to send several simultaneous HTTP requests to partner services for each inbound request. Many approaches to improving the performance and scalability of HTTP servers can be applied to tackle the original C10K problem. However, whether these approaches can tackle the reverse C10K problem needs to be verified. In this paper, we discuss the RC10K problem for server-side mashups, and propose a design that takes advantage of advanced I/O, multithreading, and event-driven programming. The results of analysis and experiments show that our design can reduce the resource requirements by almost one order of magnitude with the same performance provided, and it is promising to tackle the RC10K problem.

Keywords: HTTP, Mashup, Scalability, Performance, Client, C10K, RC10K.

1 Introduction

AJAX (Asynchronous JavaScript and XML) and mashup applications are efficient interfaces for consuming published services on the web. An AJAX or mashup page gets data from one or more services hosted on different servers [2]. If a mashup is generated by on-demand code on client agent (client-side mashup), requests are typically sent out by an API like XMLHttpRequest (XHR) of JavaScript. XHR acts as an HTTP client in such scenarios. The server hosting those AJAX and mashup pages is not responsible for requesting data from partner services, and all those computations are carried out on the client by code-on-demand. The situation is different when the HTTP request tasks of a mashup is executed on the server (server-side mashup, SSM for short), and the SSM server is responsible for fetching data from partner services by sending outbound HTTP requests through broker clients. The conceptual structure of an SSM server is shown in Fig. 1.

G. Feuerlicht and W. Lamersdorf (Eds.): ICSOC 2008, LNCS 5472, pp. 166–177, 2009.

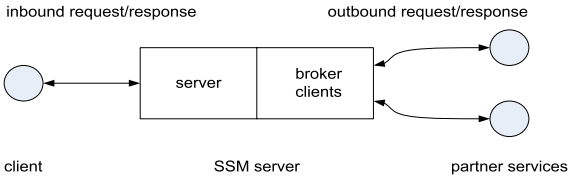


Fig. 1. The conceptual structure of an SSM server

HTTP brings two constraints [3] to the clients running on an SSM server:

(1) The basic message exchange pattern is request-response.
(2) An established connection is required for message transportation.

The constraints means a mashup server needs to maintain at least one connection for each outbound service consumption, and perform at least one request-response transaction on that connection. In order to reduce the service time of an inbound request, an SSM will launch parallel outbound requests. This yields simultaneous outbound connections and message exchanges. If there are N simultaneous inbound requests for an SSM server, and each request results in C parallel outbound requests, there will be CN simultaneous outbound connections and active outbound HTTP requests in the worst case. If an SSM server needs to handle several thousand simultaneous requests, then the server can have about $10,000$ or more simultaneous outbound HTTP requests to deal with. How an SSM server can support that many simultaneous outbound HTTP requests is what we called the reverse C10K problem, or RC10K for short.

The original C10K problem [1] studies how to provide reasonable service to $10,000$ concurrent clients using a normal server. Many approaches to improving the performance and scalability of HTTP servers can be applied to tackle the original C10K problem [1,4,5]. However, whether and how these approaches can tackle the RC10K problem of an SSM server are still open questions. This paper proposes a client design that adapts the approaches for the original C10K problem to this RC10K problem. The evaluation shows that our approach can effectively improve the scalability of an SSM server, and tackle the RC10K problem. The rest of this paper is structured as following. Section 2 discusses the approaches addressing the original C10K problem. An architectural design of SSM server's broker client is presented in Section 3. Section 4 evaluates the design by experiments. Related work is discussed in Section 5. Section 6 is the conclusions and future work.

2 The C10K Problem

The essential of the C10K problem is how to support a large number of inbound TCP connections and how to serve the concurrent inbound requests, which leads to two key design decisions of HTTP servers: I/O and concurrency strategy.

2.1 I/O

The input/output models can be divided into two groups: basic I/O and advanced I/O. Basic I/O is synchronous and blocking. An I/O operation is synchronous if the thread initializing the I/O operation cannot switch to other operation until the I/O operation is finished. A function or method is blocking if it does not return until its execution either successfully finishes or encounters an error. There are three ways to make advanced I/O [6]. The first approach is to construct a loop to keep trying an I/O option while catching I/O errors until it succeeds. This approach is called polling and it wastes CPU time. The second approach is I/O multiplexing uses `select()`-like system functions. Most operating systems support `select()`, and it is also supported by Java VM 1.4 and later versions. The descriptors of connections can be registered on a `selector`, which calls `select()` to check if there is any I/O event for each of the descriptors. So a thread initializing an I/O operation can delegate the job to a `selector` and switch to other job. Note that `select()` is blocking until one of the registered descriptors has an event or timeout happens. The third approach is asynchronous I/O (AIO), which is both asynchronous and non-blocking. Both I/O multiplexing and AIO enable a server to use very few threads to handle many concurrent connections. In practice, I/O multiplexing and AIO are applied for developing scalable servers [1].

2.2 Concurrency

Servers can roughly be classified into two categories, single-threaded and multithreaded, according to their concurrency model. Multithreading is favoured in many servers as a means to dealing with simultaneous requests, increasing the degree of concurrent processing, and making use of multiprocessors e.g. multicore processors [7]. This improves the performance of the platform compared to single-threaded implementations [8].

HTTP 1.1 [3] introduced persistent connections, which enables a client to send multiple requests through the same connection. Persistent connections improved the keep-alive connections in HTTP 1.0 [9]. By using persistent connections, the time to open or close connections can be saved and the number of parallel connections needed by a fixed number of requests can be reduced [10]. A connection can have two different behaviours — idle for a long period or busy with back to back requests. Polling of Web 2.0 applications is a typical cause of the second situation. It is tricky to optimize the usage of threads when allocating threads to connections in a multithreading server. Resources will be wasted if a thread is allocated for each connection and the connections are often idle. The strategy of allocating a thread for each request will also waste resource if the request has to be hold waiting for other messages or events and the thread cannot be switched to other job.

For a multithreading server programmed in Java or .Net, the number of threads is a measure of allocated resources, since it is directly related to CPU and memory consumption. It has been observed on .NET applications [11] and Java EE applications [12,13] that the response time of an application will increase

with respect to the number of active threads in an application server until the server is overloaded. Therefore, the concurrency model needs to:

(1) limit the maximum numbers of threads that can be spawned in the server, and
(2) keep the number of active threads as few as possible.

The first requirement results in a bounded thread pool. For the second requirement, the threads need to be programmed in an event-driven fashion, which yields a hybrid server architecture [14,15]. There are two different strategies for the second requirement — one is to make the waiting thread idle until the event for its job comes, and the other is to switch the thread to another active job and resume the current waiting job later on the corresponding events.

3 Design for the RC10K Problem

It is straightforward to apply the I/O and concurrency strategies of the server to the broker client. However, the client has two new problems that the server does not have. One problem is how to reuse the available connections in order to save connection open and close time and keep the concurrent opened connection number below the maximum number allowed on partner server. The other problem is how to embedded the client code into a mashup application easily.

A solution of the first problem is to introduce a component for connection management to the client. We propose a modular design shown in Fig. 2. Our design is inspired by the design of Jetty server [16], whose details can be found in Section 5. The client contains a connector that is responsible for connecting to a server and transporting messages to and from it. A thread pool provides worker threads to perform the client jobs. A connection manager creates, registers, and reuses connections. The connector and thread pool of the server can be reused by the client. The SSM server and the broker client can share the same thread pool instance for easy coordination of resource allocation.

The second problem can be addressed by introducing a message exchange object or structure, which has conceptually four components: destination, socket connection, request message, and response message. Its major behaviours are the state transitions during message exchanging and the actions driven by the transition events.

The state transitions of an HTTP client are shown in Fig. 3. A socket connection is initially closed when created by a client, and the connection will be established when a `SYN+ACK` from the server side is received. The client may retry connecting for several times before giving up if the connection attempts are either rejected or timeout. HTTP requests then can be sent through the established connection. The client will wait for the response when the request is completely sent. When the response comes, the client will first parse out the header then the message body. When the whole response is completed, the connection will become idle and ready for another request to be sent. Timeout may happen when the client is waiting for or getting a response. The client may try to resend the request several times before giving up.

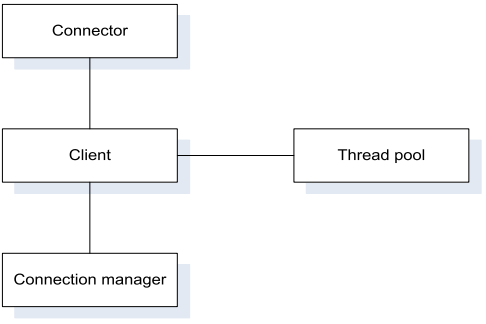
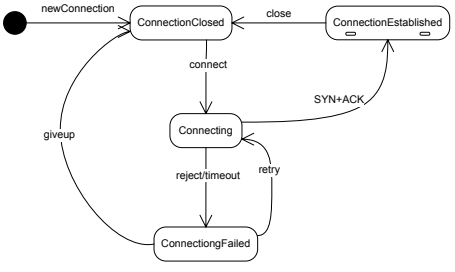
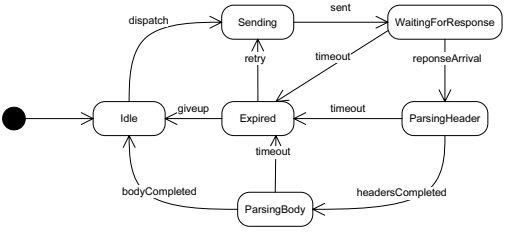


Fig. 2. The architecture of the proposed HTTP client



(a) State transitions of transportation.



(b) Detailed state transitions of the composite state `ConnectionEstabished`.

Fig. 3. A state transition diagram of an HTTP client

The important events and corresponding potential actions described in Fig. 3 are listed in Table 1. Some events happen on socket I/O level, and therefore the responsibility of capturing them can be allocated to the connector. These events can be programmed as the methods of exchange object. So a mashup application just needs to create an exchange object that implements the required methods. The major purpose of combining event-driven programming with multithreading is to enable asynchronous processing. Asynchronous processing means that, the processing of a request can be suspended when waiting for responses of outbound requests, and it can be resumed when those responses come or timeout happens. Asynchronous processing can save computation resource effectively.

Table 1. The important events and associated potential actions for an HTTP exchange

Event	Potential actions
On connection established	update the connections for the current IP socket address, prepare request message to be sent
On connection failed	handle the failure
On connection idle	update the connections for the current IP socket address
On waiting for response	switch the thread to other job
On headers completed	process the headers or wait for the body
On body completed	locate a thread for current request and process the message
On expired	handle the expiration

4 Evaluation

We want to verify if the design described in Section 3 can lower resource demand for the same workload compared with other design options. The number of active threads is used as the indicator of resource consumption. First we derive an analytic result and then use experiments to verify the analysis.

4.1 Analysis

How many threads are required in an SSM server for N concurrent requests? The number will be a linear function of N, the number of concurrent inbound requests, as the following equation.

$$n = N \times (1 + c) \,, \tag{1}$$

where 1 represents the demand by the thread allocated for the inbound request, and c is the number of threads required by outbound HTTP requests. Furthermore, c can be calculated by

$$c = \frac{\sum_{i=1}^{C} D_i}{S} \,,$$

where C is the number of outbound service consumptions for each inbound request, D_i is the time demand for ith outbound service consumption, and S is the average service time of an inbound service request. From Equation 1, there are two ways to decrease n — reducing either D_i or 1. D_i will become smaller by switching the threads to other jobs while it is waiting for outbound responses. Similarly, if a thread initially allocated to an inbound request can be switched to another job by asynchronous processing, the 1 can become smaller, and the average thread number will be

$$n = N \times (\frac{D}{S} + c) \,, \tag{2}$$

where D is the time demand for inbound service request. In order to have an inbound or outbound service request suspended and resumed later, the platform needs to support continuation-like mechanism [17,18].

4.2 Experiments and Results

In order to evaluate the design presented in Section 3, we carried out a series of experiments. The mashup server in the experiments are programmed in Java based on Jetty 7.0[1]. Note that implementation details are not the focus of this paper and implementations vary for various programming paradigms and languages.

The first aspect to be evaluated is how event-driven programming and asynchronous processing can optimize the usage of computation resources. We use two machines in this experiment, machine A is running Window XP SP3 and Java SE runtime environment 1.6 on 3.2GHz P4 CPU with Hyper-threading[2] and 2GB RAM, and machine B is running Window XP SP3 and Java SE runtime environment 1.6 on dual 3.2GHz Xeon CPU's with Hyper-threading and 2GB RAM. Both of them have the TcpIP parameter `TcpTimedWaitDelay`[3] set as 60 seconds. The two machines are connected with a router, and the connection speed is 100Mbps. JMeter[4] is running on machine A to simulate end users. The think time of simulated users is set to zero in order to make the number of concurrent requests in the intermediary service as close to the number of simulated users as possible. We chose JMeter for load generation because JMeter can control the exact number of concurrent running clients. A server-side mashup is running on machine B, and it consumes two other partner services to generate responses. The broker client uses `select()` type I/O. The two partner services are hosted by Jetty running on machine B as well. One partner service spends about 0.5 seconds for each request, and the other about one second. Both of them reply with tiny payloads. The resource requirements for running the two partner services are low enough for not interfering the mashup's performance. Fig. 4 shows the active thread number and throughput as functions of the number of simulated users in synchronous and asynchronous processing modes.

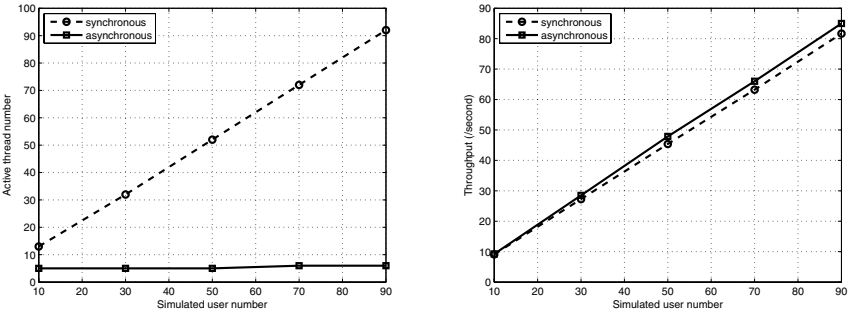
As shown in Fig. 4(b), the throughputs for synchronous and asynchronous processing are almost the same because most of the response time is spent on getting response from the partner services. There is a distinction of active thread number between two processing models shown in Fig. 4(a) The thread number for synchronous processing is about linear with respect to the number of simulated users. Note that the number of simulated users is very close to the number of simultaneous active requests due to zero think time and low network latency. This fits Equation (1) very well. On the contrary, the thread number for asynchronous processing is likely constant, which does not seem to accord to Equation (2) at first glance. In fact, the thread demand is still increasing with the user number in this case, but it is very slow because $\frac{D}{S} + c$ in Equation (2) is very small. Table 2 lists the values of D, S, and $\frac{D}{S}$ in the experiments. Asynchronous processing

[1] See `http://www.mortbay.org/jetty/`

[2] See `http://www.intel.com/technology/platform-technology/hyper-threading/index.htm`

[3] See `http://support.microsoft.com/kb/314053`

[4] See `http://jakarta.apache.org/jmeter/index.html`



(a) Active thread number (b) Throughput

Fig. 4. Active thread number and throughput as functions of the number of simulated users in synchronous and asynchronous processing modes

Table 2. The average time demand and service time for intermediary service request

Users number	10	30	50	70	90
D **(millisecond)**	0	1	1	1	2
S **(millisecond)**	1014	1012	1012	1015	1014
D/S	0	0.0010	0.0010	0.0010	0.0020

can drop the resource requirements of intermediary services dramatically when $\frac{D}{S}$ is far less than 1.

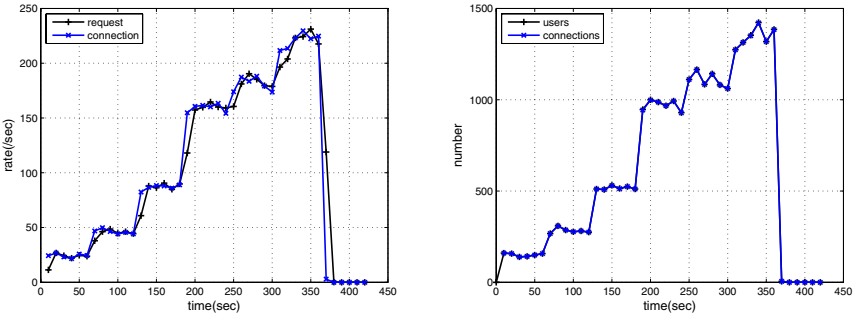
The second aspect to be evaluated is whether our design is sufficient for the RC10K problem, or how close it can be pushed to the target. In this experiment, the mashup server is running on machine B. The tested mashup consumes 3 partner services 2 times for each inbound request. The partner services are hosted by YAWS[5] on three other machines running Linux 2.6.24-19-server, and their hardware is exactly the same as machine B. Tsung[6] running on machine A is used to generate the load. All the machines are connected through 100 Mbps LAN. Tsung is choosed in this scenario because it can generate workload of high arrival rate, and therefore is able to simulate an open network environment. Machine B is specially tuned for the large number of inbound and outbound TCP connections. The TcpIP parameter `MaxUserPort` is set as 65534, `MaxFreeTcbs` as 10000, and `MaxHashTableSize` as 8192. The JVM is tuned for heap size and thread stack size, specially `-Xss64k -Xms1024M -Xmx1024M -XX:PermSize=256M -XX:MaxPermSize=256M` is used in this experiments.

Each consumption of a partner service takes averagely 5 seconds. This time is deliberately set to be large compared to normal services in order to obtain a large number of concurrent outbound connections for a certain arrival rate. Each test

[5] See `http://yaws.hyber.org/`

[6] See `http://tsung.erlang-projects.org/`

is composed of several phases back to back, and each phase takes one minute. Fig. 5 shows the throughput and concurrent users of the mashup server in one of the tests. The inter-arrival time (seconds) changes along different phases from 0.04 to 0.02, 0.01, 0.005, 0.004, and 0.003. The inter-arrival time is decreased gradually in order to warm up the SSM server and reach its maximum capacity gracefully.



(a) Throughput. (b) Concurrent users.

Fig. 5. The throughput and concurrent users along different phases of a test

The largest number of concurrent users that has been reached is about 1400 when the server still operates normally. When the inter-arrival time decreases to 0.002 seconds, the server becomes unstable. The largest number of concurrent outbound connections is about 8400 (1400×6). Although it is less than $10,000$, it is very close to the target. The same tests were performed on WSO2 mashup server 1.5.1 on which a mashup with the same logic was deployed. The WSO2 mashup server is overloaded when the inter-arrival time reached 0.1 seconds, that is, the server cannot even support 50 concurrent inbound requests and 300 concurrent outbound requests. The details about WSO2 mashup server can be found in Section 5.

5 Related Work

As discussed in Section 2, there have been many research and development activities related to the original C10K problem or the scalability issues of HTTP servers. There are few research literature addressing the scalability issues of HTTP client that can be seen in the RC10K problem. We expect there will be more and more research work related to this topic with the development of mashup and web service applications.

XMLHttpRequest (XHR) is probably the most popular HTTP client interface currently used for Web-based service consumption. XHR's interface was specialized by W3C [19] and its implementations vary a lot in different Web browsers.

Each XHR object requires an event listener that is normally a callback function describing the actions to be triggered on certain events. The function is called every time the `readyState` changes. `readyState` can be in one of the five states, namely request unsent, `open()` success, response header received, loading response body, and response done. The XHR also provides accesses of request header, request body, response status, response header and response body. XHR is a perfect example for high-level message exchange interface design. However, XHR specification does not address the aspects of connection and thread management.

HTTPCLIENT[7] is a Java-based open-source project at Apache. HTTPCLIENT depends on HttpCore NIO extensions[8] to support non-blocking I/O (NIO) and event-driven programming. HTTPCLIENT is used in some SSM servers like WSO2 Mashup Server[9]. HTTPCLIENT has a component for connection management, but does not provide thread pool. Without a thread pool, it will be difficult to reuse the available idle threads.

WSO2 Mashup Server is a platform that uses JavaScript as the language of representing and programming mashups. In other words, it exposes JavaScript functions as services. WSO2 Mashup server provides several hosted objects that ease common mashup operations like fetching feeds, scrapping web pages, and sending either HTTP or SOAP requests to other services. The WSRequest object[10] mimics the XHR interface, and is able to perform both synchronous and asynchronous requests. Due to the limitation of JavaScript and the underlying Mozilla Rhino JavaScript engine, the response of an asynchronous request can only be caught by using a `wait()` to hold the requesting thread, which make the asynchronous request consume more resources and run slow.

The tested SSM server of proposed architecture is developed on the basis of Jetty 7.0 and especially its Servlet 3.0 features[11]. Jetty is an open-source web server implemented in Java. Jetty provides dynamic content support through Servlet and JSP technologies. We leverage the client code base of Jetty with shared thread pool and asynchronous processing for the evaluation. Note that the official Jetty project focuses on HTTP server, and the client is just an 'extra' part. More efforts are needed to improve the client code base, and make an SSM server out of it.

6 Conclusions

The scalability and performance of the broker client in an SSM server directly affect server scalability and performance. To date, the scalability issues of HTTP clients have been overlooked in research. Although many approaches have been studied for improving HTTP servers' scalability like the C10K problem, whether

[7] See `http://hc.apache.org/httpclient-3.x/`
[8] See `http://hc.apache.org/httpcomponents-core/index.html`
[9] See `http://wso2.org/projects/mashup`
[10] See `http://wso2.org/project/mashup/1.5.1/docs/wsrequesthostobject.html`
[11] See `http://jcp.org/en/jsr/detail?id=315`

those approaches are effective for HTTP clients is still an open question. We formulate the reverse C10K problem in this paper, and propose an architectural design of the client that uses advanced I/O, multithreading, and asynchronous processing in order to tackle the RC10K problem.

The evaluation shows that our design can reduce resource requirements by almost one order of magnitude in order to achieve the same performance compared with other designs using synchronous processing. The evaluation also shows that the 10,000 simultaneous outbound connections is feasible in a normal web server setup and virtual machine environment like JVM. Currently, we are investigating how much better other languages like Erlang[12] can perform in the RC10K problem due to its features like light-weight process, no memory-sharing, and built-in message-passing.

References

1. Kegel, D.: The c10k problem. Web (September 2006), http://www.kegel.com/c10k.html
2. Ort, E., Brydon, S., Basler, M.: Mashup styles, part 1: Server-side mashups. Web (May 2007), http://java.sun.com/developer/technicalArticles/J2EE/mashup_1/
3. The Internet Society: Hypertext transfer protocol – http/1.1. Web (June 1999), http://tools.ietf.org/html/rfc2616
4. Darcy, J.: High-performance server architecture. Web (August 2002), http://pl.atyp.us/content/tech/servers.html
5. Barish, G.: Building scalable and high-performance Java Web applications using J2EE technology: Using J2EE Technology. Addison-Wesley, Reading (2002)
6. Stevens, W.R.: Advanced Programming in the UNIX Environment. Addison-Wesley Professional, Reading (1992)
7. Dollimore, J., Kindberg, T., Coulouris, G.: Distributed Systems: Concepts and Design, 4th edn. Addison-Wesley, Reading (2005)
8. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 7th edn. John Wiley & Sons, Chichester (2004)
9. The Internet Society: Hypertext transfer protocol – http/1.0. Web (May 1996), http://tools.ietf.org/html/rfc1945
10. Gourley, D., Totty, B., Sayer, M., Reddy, S., Aggarwal, A.: HTTP: The Definitive Guide. O'Reilly, Sebastopol (2002)
11. Hasan, J., Tu, K.: Performance Tuning and Optimizing ASP .NET Applications. Apress (2003)
12. Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., Tantawi, A.: An analytical model for multi-tier internet services and its applications. In: SIGMETRICS 2005: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pp. 291–302. ACM Press, New York (2005)
13. Haines, S.: Pro Java EE 5 Performance Management and Optimization. Apress (2006)

[12] See http://www.erlang.org/

14. Beloglavec, S., Heričko, M., Jurič, M.B., Rozman, I.: Analysis of the limitations of multiple client handling in a java server environment. SIGPLAN Not. 40(4), 20–28 (2005)
15. Li, P., Zdancewic, S.: Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. SIGPLAN Not. 42(6), 189–199 (2007)
16. Wilkins, G.: Jetty 6 architecture. Web (November 2006), `http://docs.codehaus.org/display/JETTY/Architecture`
17. Queinnec, C.: Inverting back the inversion of control or, continuations versus page-centric programming. SIGPLAN Not. 38(2), 57–64 (2003)
18. Gomez, J.C., Ramos, J.R., Rego, V.: Signals, timers, and continuations for multithreaded user-level protocols. Softw. Pract. Exper. 36(5), 449–471 (2006)
19. W3C: The xmlhttprequest object. Web (April 2008), `http://www.w3.org/TR/XMLHttpRequest/`