Available online at www.sciencedirect.com
SCIENCE DIRECT
Information and Software Technology 48 (2006) 471–483
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

An event-driven framework for inter-user communication applications

Chien-Chih Hsu, I.-Chen Wu*

Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu 300, Taiwan, ROC

Received 16 June 2004; revised 20 May 2005; accepted 24 May 2005
Available online 10 August 2005

Abstract

This paper presents an event-driven framework for inter-user communication applications, such as Internet gaming or chatting, that require frequent communication among users. This paper addresses two major blocking problems for event-driven programming for inter-user communication applications, namely output blocking and request blocking. For the former, an output buffering mechanism is presented to solve this problem. For the latter, a service requesting mechanism with helper processes is presented to solve this problem. The above two mechanisms are incorporated into the framework presented in this paper to facilitate application development. In practice, this framework has been applied to online game development.

Keywords: Inter-user communication applications; Event-driven programming; Concurrent programming; Framework; Threads

1. Introduction

With the rapid growth of the Internet, applications involving real-time communication among clients have become increasingly important. These applications include chat rooms such as Yahoo! Chat [44] and EFnet chat network [11], Internet games such as Yahoo! Games [45], Warcraft III [4], and Counter-strike [41], and present and instant messaging systems such as ICQ [17] and MSN Messenger [23]. Consider an example of chat room or game system. One user types a message and others then can read that message in real time. Since these applications involve inter-user communication, this paper calls them *inter-user communication applications*.

For inter-user communication applications, servers are often used to handle inter-user communication. For example, game servers receive player events (or messages) and then respond (or pass messages) to other players. For inter-user communication applications, server developers generally must consider the following criteria.

1. *Minimize the client response time.* If the response time is unexpectedly long, interactions may not evolve as expected or users may run out of patience.
2. *Ensure high server stability.* Server crashes cause all clients connected to that server to become disconnected.
3. *Support as many clients concurrently as possible.* For example, support thousands of players on a single server.

The first criterion is essential for server programming in inter-user communication applications. To respond to users as rapidly as possible, servers usually hold connections to clients. Servers thus must handle client messages (or events) from all connections concurrently and server developers must handle concurrent events carefully.

Two main programming models exist for concurrent event handling, namely threading and event-driven programming. Threading is a general-purpose technique for managing concurrency. The advantages of threading compared to event-driven programming include: (a) support of context switching among threads, and (b) support of scalable performance on multiple CPUs.

However, some developers and researchers [27,32] have also observed that threading has some drawbacks compared to event-driven programming. Note that Ousterhout [27] described the following drawbacks:

* Corresponding author. Tel.: +886 3 573 1855; fax: +886 3 573 3777.
E-mail addresses: jjshie@csie.nctu.edu.tw (C.-C. Hsu), icwu@csie.nctu.edu.tw (I.-C. Wu).

0950-5849/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2005.05.005

1. *Difficult to program.* Threads generally require synchronous mechanisms (e.g. locks) to access shared data safely. However, incorrect locking may cause deadlocks, making independent module design difficult. Besides, another problem that also increases programming difficulty is that several standard libraries are not thread-safe [24].
2. *Hard to debug*. For threading, it is difficult for developers to debug the code due to data and timing dependencies. Besides, another problem that also increases debugging difficulty is that thread stack sizes are normally limited [22,24], causing processes crash when stacks overflow. In contrast, in event-driven programming, the lack of context switching among event handlers makes it quite easy to debug the code by recording and then replaying the sequence of events.
3. *Difficult to achieve good performance.* Coarse-grain locking yields low concurrency, while fine-grain locking tends to increase lock operations and thus reduce performance.

Since inter-user communication applications are often used to facilitate heavy inter-user communication among numerous clients (say, thousands of players in a game system), it makes the above drawbacks even worse. The first two drawbacks imply that it is hard for threading to satisfy the second criterion (above) of the inter-user communication applications; and the third drawback indicates that it is hard for threading to satisfy the third criterion. Thus, for the application developers who are more concerned with the second and third criteria and less concerned with the two threading advantages (described above), the event-driven programming model becomes attractive. Hence, this paper is motivated to study and design an event-driven framework for inter-user communication applications. Note that a framework [13,33] is defined as a set of collaborative classes that enable developers to reuse the architecture and implementation of a generic program for a set of domain specific applications.

Our framework is based on event-driven programming (rather than threading) for the following reason. In inter-user communication applications, the above three drawbacks of threading (or the second and third criteria) are important as described above, while the two drawbacks of event-driven programming are less important because they can be ignored or alternatively can be solved in this paper. First, regarding the two drawbacks of event-driven programming, this paper ignores the one, namely not supporting scalable performance on multiple CPUs, because for most inter-user communication applications servers can be separated into several processes to achieve scalable performance. In the case of casual games, such as Chess and Bridge, servers can naturally be separated into several processes, e.g. one for each game. Even for most massive multiplayer online games (MMOGs), such as Ultima Online [12], the server system can use several processes each dealing with a single game scene. Second, this paper focuses on overcoming the other drawback of event-driven programming: the need to pay attention to the blocking problem in event handling.

This paper addresses two major blocking problems and presents solutions or guidelines. The two blocking problems are described below.

1. *Output blocking:* This problem occurs on sending messages to clients with corresponding full kernel buffers. The buffer generally becomes full when network traffic is jammed. This problem frequently is neglected at the start of server development.
2. *Request blocking:* This problem occurs when a server waits for responses after sending requests to other servers. For example, when a game server attempts to read several game records from a remote database server.

This paper presents solutions for the above two blocking problems. An output buffering mechanism is presented to solve the output blocking problem, while a service requesting mechanism is presented to solve the request blocking problem. Meanwhile, for the second problem, several system and library calls that may cause the problem are also identified. Both mechanisms are incorporated into the event-driven framework presented in this paper.

Practically, the event-driven framework has been used in the CYC game system [39] that provides players with casual games, such as Chess, Bridge, Mahjong, etc. Currently, the CYC game system has supported up to 10,000 concurrent players.

The rest of this paper is organized as follows. Section 2 reviews the event-driven programming model. Section 3 describes the output blocking problem and presents solutions. Section 4 then describes the request blocking problem and presents solutions. Section 5 presents our experiments by applying our framework to the CYC game system and some performance analysis. Finally, Section 6 summarizes our work.

2. Event-driven programming

This section reviews the event-driven programming model. In this model, applications wait for specific events and dispatch occurring events to appropriate handlers for processing. In networked applications, event-driven based servers generally handle both input and output events. Input events occur when sockets are ready to read, while output events occur when sockets are ready to write.

Most event-driven based servers in the Unix environment use the `select` system call [10,18,34,36] to demultiplex input/output events. The `select`-based event-driven model has been induced as the Reactor design patterns in [31,32]. In this pattern, the core component named

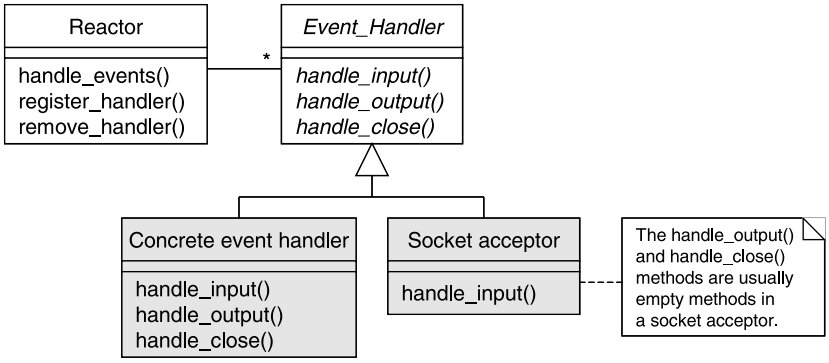

Fig. 1. Class diagram of the Reactor pattern.

`Reactor` waits for input/output events synchronously. When such events occur, the `Reactor` object identifies the handlers of these events and then invokes the appropriate methods of the handlers.

The Reactor pattern defines an event handler interface. Concrete event handlers implement the event handler interface to support application specific services. These concrete event handlers are registered with the `Reactor` object dynamically, and then are passively reacted to the occurrences of designated events. Application developers only need to implement concrete event handlers, when reusing the dispatching mechanism of the `Reactor` object.

Fig. 1 shows the class diagram of the Reactor pattern. Note that the shadowed classes are application-specific (that is, application developers must implement these classes only). The responsibilities of each class are detailed as follows.

- `Reactor` is the core component of event-driven applications and a process normally requires only one `Reactor` object. Applications can register and remove event handlers by calling `register_handler` and `remove_handler` of this object, respectively. The `handle_events` method in this object is invoked to run the event-handling loop, in which the `select` system call is used to wait for some specified input or output events synchronously. As events occur, the `Reactor` object dispatches those events to the corresponding event handlers.
- `Event_Handler` is an abstract event handler class that defines several hook methods [16,30]: `handle_input`, `handle_output`, and `handle_close`. Concrete event handlers are application specific event handlers that inherit the `Event_Handler` class. When an input (or output) event occurs for some concrete event handler, the `handle_input` (or `handle_output`) method of the handler is invoked to process the event. Before handler removal, the `handle_close` method is invoked for application specific termination operations.
- A socket acceptor is a special concrete event handler that is responsible for accepting new connections. The `handle_input` method of the socket acceptor accepts a new connection from a client (by calling `accept` in Unix), generates the corresponding concrete event handler for the client and then registers this handler with the `Reactor` object. Since a socket acceptor does not output data and hold state, its `handle_output` and `handle_close` are generally empty.

The above Reactor pattern forms a basis of the event-driven framework presented in the remaining part of this paper.

3. Output blocking problem and solution

This section discusses the output blocking problem for event-driven programming in inter-user communication applications. Section 3.1 describes the output blocking problem, and Section 3.2 then presents a mechanism for solving this problem.

3.1. Output blocking problem

In most TCP/IP implementations, each socket contains both send and receive buffers [10,36] in the kernel. Both buffers are tens of kilobytes in size in Unix. When the send buffer of a socket in the kernel is full, output to the socket is blocked, since by default all sockets are in the blocking mode [36].

Output blocking is a serious problem in inter-user communication applications. For example, in a game system, it is common for a server to receive one message from one player, say A, and then immediately send that message to a set of players, say including player B. However, if network traffic is jammed near or around player B (but not elsewhere), the server blocks due to the failure to send the buffer of the socket to B.

From our experience with the CYC game system [39], the output blocking problem seriously degrades the performances of inter-user communication applications, since network traffic may be jammed unexpectedly. A more serious situation is the following: if a client crashes or its

network wire is disconnected, the server may not detect the disconnection by default for approximately 9 min [35]. Furthermore, since a server for a game system usually serves thousands of players or more, it is easy to cause server blocking as above and result in slow responses to all clients.

3.2. Solution to the output blocking problem

Stevens (cf. Section 15.2 of [36]) demonstrated a simple buffering method when discussing non-blocking I/O. Since the buffer size is fixed to a small number, this method still cannot solve the output blocking problem when the message size is larger than the buffer size. Some event-driven based web servers, such as thttpd [1] and mathopd [5], used the `sendfile` system call [40] as well as non-blocking sockets to avoid the output blocking problem upon sending files. The Flash web server [28] also proposed a method that can avoid the output blocking problem for sending web pages. The above work solved the problem specifically for their own applications. In this paper, we propose a reusable framework for generally solving this problem.

In order to solve this problem in event-driven based servers, this paper presents a mechanism, called an *output buffering mechanism*, and incorporates it into our event-driven framework. This mechanism sets all the sockets to the non-blocking mode and extends event handlers to those with extra dynamic output buffers. The buffers are used to store unsent data that cannot be sent when the send buffers of sockets are full, as described above. Namely, the unsent data are stored into the extra output buffer when the socket send buffers are full, and the buffered data then are sent out whenever the send buffers have available space.

Fig. 2 shows the class diagram of the event-driven framework with the output buffering mechanism. This paper simply describes the extra classes and methods in this figure when compared to Fig. 1, as follows:

- `disable_output_handling` and `enable_output_handling` are two new methods added to the `Reactor` class. The former method disables handlers from handling output events, while the latter method enables handlers to handle output events. Output handling is initially disabled for all event handlers.
- `Buffered_Output_Handler` is an abstract class that partially implements the `Event_Handler` interface. Specifically, `Buffered_Output_Handler` uses the `handle_output` method to handle unsent data, while leaving the two methods `handle_input` and `handle_close` unimplemented. The classes of concrete event handlers extend the class `Buffered_Output_Handler`, rather than `Event_Handler`, and only need to implement the above two unimplemented methods.
  In order to hide output handling from application developers, the `Buffered_Output_Handler` class hides the method `handle_output` and provides developers with the `write_data` method, rather than the `write` system call. The `write_data` method normally writes data out as the `write` system call, but stores the unsent data into a buffer on output blocking and thus enables output event handling.
- `Memory_Buffer` is a class of dynamic sized buffers. Each concrete event handler allocates a single `Memory_Buffer` object to store unsent data when the send buffer of the corresponding socket of the handler is full. Since send buffers in the kernel rarely become full, the physical buffer spaces of the dynamic output buffers are created only when required and are immediately freed when not required. Normally, the maximum buffer size is set to a large number, for example 1 MB. Since each message in the inter-user communication applications is generally small, overflowing of `Memory_Buffer` usually implies that the network has been jammed for

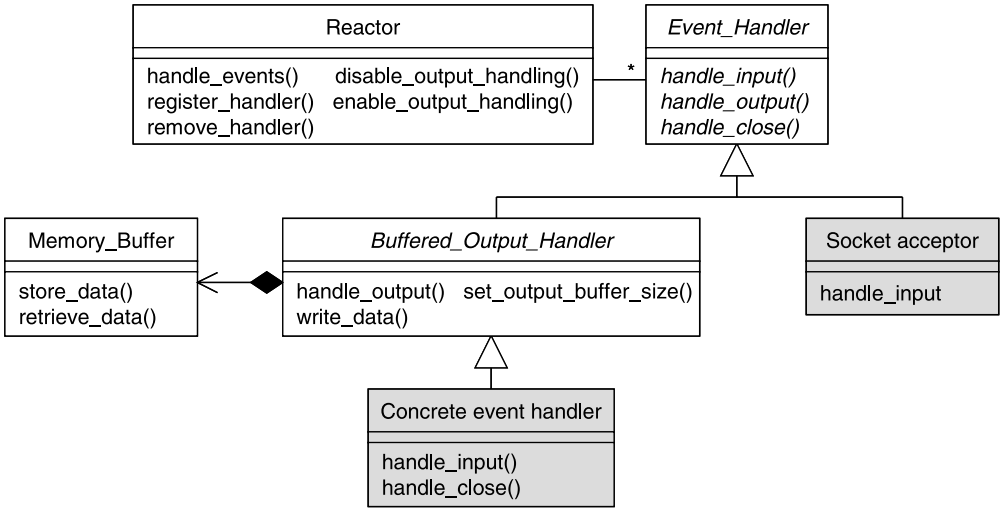

Fig. 2. Class diagram of the event-driven framework with the output buffering mechanism.

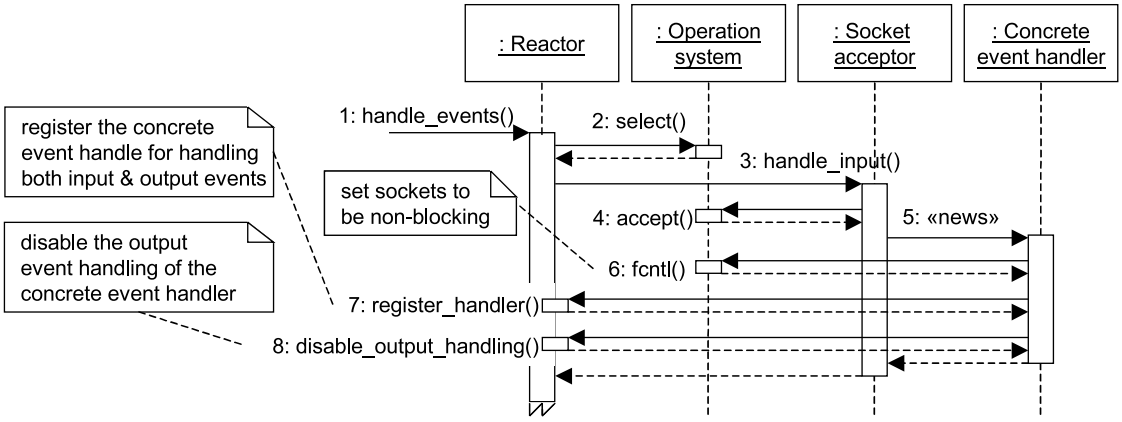

Fig. 3. Sequence diagram for accepting a new client.

a while. Thus, it is reasonable to claim connection failure in such situations.

This paper now illustrates the following interactions in more detail, including: (1) how to accept a new client, and (2) how to handle output buffering. Note that the UML sequence diagram [6] is used to demonstrate these interactions.

First, the sequence diagram in Fig. 3 illustrates how the socket acceptor accepts a new client. When invoked to handle an input message (in Step 3), the socket acceptor accepts a connection request (in Step 4) and creates a concrete event handler (in Step 5). The concrete event handler then sets the corresponding socket to the non-blocking mode (in Step 6), registers itself for event handling with the `Rector` object (in Step 7) and initially disables output handling (in Step 8).

Second, the sequence diagram in Fig. 4 illustrates how a concrete event handler, `hs`, writes messages to another handler, `hd`, with a full send buffer while handling input messages. In this case, the messages (from `hs`) cannot be sent out due to the send buffer of the corresponding socket in `hd` being full (in Step 5). Subsequently, in `hd`, the unsent data is stored into its own `Memory_Buffer` (in Step 6) and output handling is enabled (in Step 7) to output the unsent data later. When the socket has available space for output, the `handle_output` method of `hd` (in Step 9) is invoked to send the data in `Memory_Buffer` out (in Steps 10 and 11). Finally, output handling for `hd` (in Step 12) is disabled if all the data are sent out successfully.

4. Request blocking problem and solution

This section investigates the request blocking problem for event-driven programming in inter-user communication applications. Section 4.1 introduces the request blocking problem. Section 4.2 then presents a solution to solve

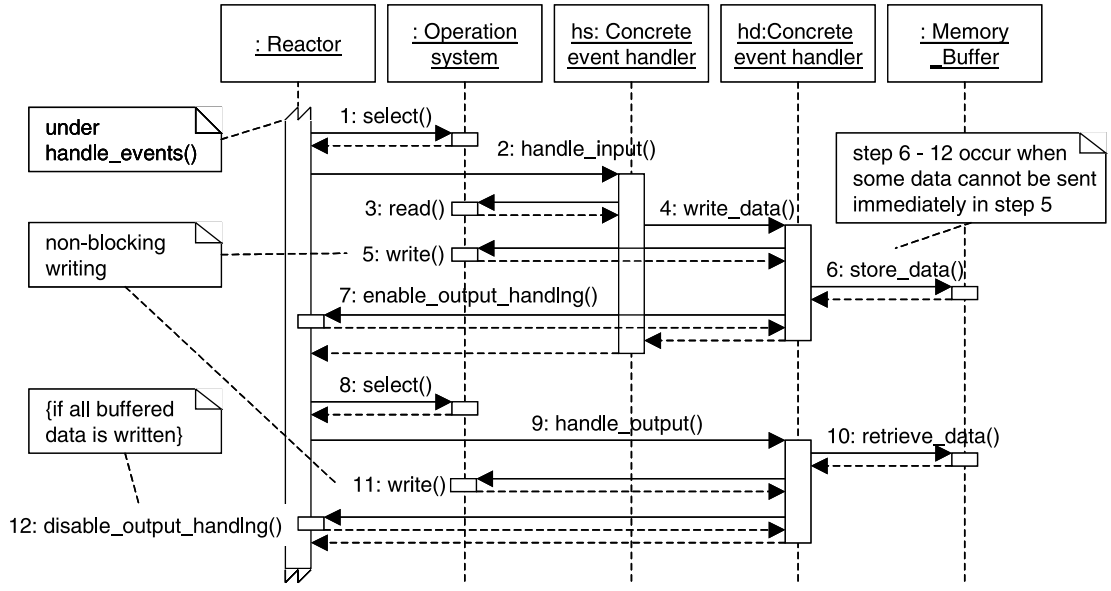

Fig. 4. Sequence diagram for handling output buffering.

the request blocking problem for HTTP requests only. Next, Section 4.3 leverages the solution in Section 4.2 to solve the request blocking problem for all other requests.

4.1. Request blocking problem

For event-driven programming, application developers must also be careful about using possible blocking operations in event handling. Besides the output blocking operations in Section 3, blocking operations are classified into two types: namely local blocking operations and request blocking operations.

The former include explicit system or function calls that may block execution locally, such as `wait`, `sleep`, `flock`, and `semop` [34]. Developers should either prevent from using these functions or use alternatives, instead.

Request blocking operations are involved in service requests over network. For example, when a game server S_G needs to retrieve game records from (or store records into) a database server S_{DB}, S_G generally performs the following three steps: (1) create a connection to S_{DB}; (2) send request messages to S_{DB}; (3) receive response messages from S_{DB}. However, the operation in Step (3) is clearly a blocking one.

In event-driven based servers, if a straightforward design is used that directly bundles the three operations together within a single input event handler, called a *source event handler* here, this service request obviously becomes blocked. Consequently, the performance of the game server S_G degrades.

Many developers usually notice the above example for database requests before coding. However, unfortunately many services are requested implicitly. For example, the Harvest and Squid projects [7,43] noticed that the DNS-related function call, `gethostbyname`, may issue a request to a DNS server and wait for the response from that server. Thus, for event-driven programming, it becomes crucial for application developers to identify more operations with remote service requests, as listed below.

- *DNS-related functions.* For example, accessing DNS servers via some library calls such as `gethostbyname`, `gethostbyaddr`, `getaddrinfo`, and `getnameinfo` [10,36].
- *Database-related functions.* For example, accessing database servers via JDBC [14,38] or ODBC [21] drivers.
- *LDAP-related functions.* For example, accessing the servers of OpenLDAP via its client library [26].
- *HTTP access.* For example, making HTTP requests via `libwww` [25].
- *Remote file access.* For example, accessing a file mounted on a remote host via the NFS service [9,10].

Note that the last operation involving remote file access may also block event-driven based servers for inter-user communication applications, because traffic to the remote file server may also be jammed.

Regarding local file access, the research in [28] indicates that most local file operations generally cannot be integrated with the `select` system call. Namely, `select` cannot be used to detect the completion of these operations. Furthermore, some of these operations, such as `open` and `stat`, may still be blocking. The above blocking problem is critical for the HTTP server applications [28] because HTTP servers generally require frequent accessing of local files. However, this paper is less concerned with local file access, since inter-user communication applications generally process messages on the fly without frequently accessing local files. For example, a game server generally does not need to save player chat messages into local files. If a game server does need to access files frequently for some reason, the server can use database, instead, and the solution presented in the remainder of this section remains useful.

In order to solve the request blocking problem in event-driven applications, the Harvest and Squid projects [7,43] used helper processes to resolve DNS queries without incurring blocking. However, they did not design a reusable software architecture for this problem.

For solving this problem in a reusable way, this paper first presents a service requesting mechanism for dealing with HTTP requests in Section 4.2. Then, in Section 4.3, this mechanism is applied to all the other service requests with blocking operations.

4.2. Solutions for HTTP access requests

This section presents a mechanism, called the *service requesting mechanism*, for dealing with HTTP requests, and incorporates this mechanism into the event-driven framework in this paper. In this mechanism, an event handler, called the *source event handler*, creates another event handler, called the *service requestor* here, to send an HTTP request to a remote service provider and wait for the response. After receiving the response, the service requestor transfers the response back to its source event handler. These activities are performed without any blocking.

Fig. 5 modifies the class diagram of the framework in Fig. 2 by adding two classes, `Service_Requestor` and concrete service requestor, detailed below:

- `Service_Requestor` is an abstract class that extends `Buffered_Output_Handler` for service requesting.
  This class provides application developers with a `request` method that is used to establish a connection to a server, such as an HTTP server, and forward an HTTP request to the server. A concrete event handler (or the source event handler as defined in Section 4.1) requires the following parameters to invoke this method: (1) server IP address and port; (2) the pointer back to the source event handler; and (3) the HTTP request message.

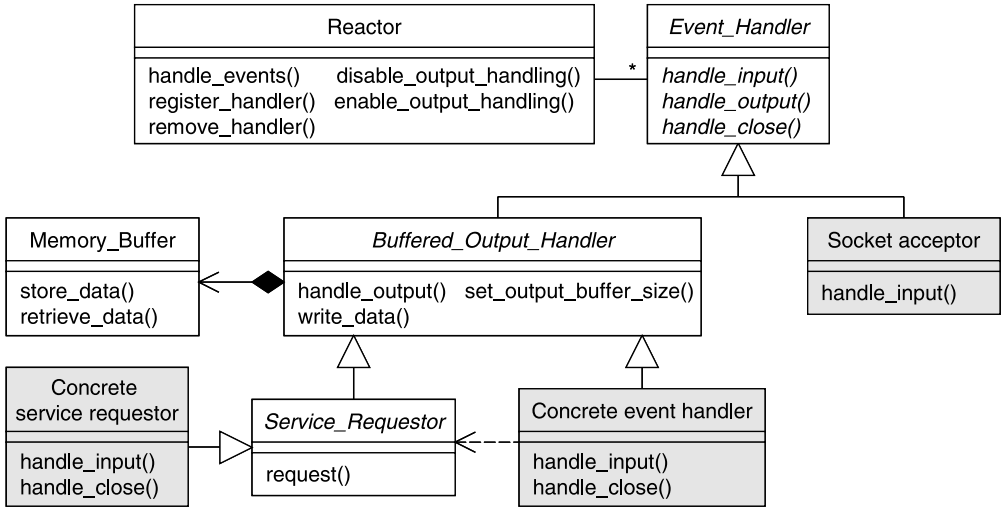

Fig. 5. Class diagram of the event-driven framework with service requestors.

- Concrete service requestors are application-specific classes implementing the `Service_Requestor`. A concrete service requestor object is created by a source event handler to handle one and only one HTTP request. After requestor creation, the source event handler invokes the `request` method of the requestor to connect to the corresponding HTTP server and then registers the requestor with the `Reactor` object. When the server replies, the `Reactor` object invokes the `handle_input` of the requestor to process the response.

Next, the following interactions are illustrated in more detail: (1) how to establish a connection to a remote service provider and send an HTTP request to that provider, and (2) how to handle service provider responses.

First, the sequence diagram in Fig. 6 illustrates how a concrete event handler, `EH`, establishes a connection to a remote service provider, `SVCP`, and forwards an HTTP request to that provider. When making an HTTP request, `EH` creates a service requestor, `SR` (in Step 4) and then call the `request` method of `SR` (in Step 5). This method connects to `SVCP` in a non-blocking manner (in Step 6), registers `SR` itself with the `Reactor` object (in Step 7) and stores the request message in the `Memory_Buffer` (in Step 8). The stored request message is sent (in Step 10) immediately upon connection establishment.

Second, the sequence diagram in Fig. 7 illustrates how the service requestor, `SR`, handles the responses from the service provider, `SVCP`. On receiving responses from `SVCP` (in Steps 2 and 3), `SR` processes those responses and may invoke some callback functions of `EH` (in Step 4). Once

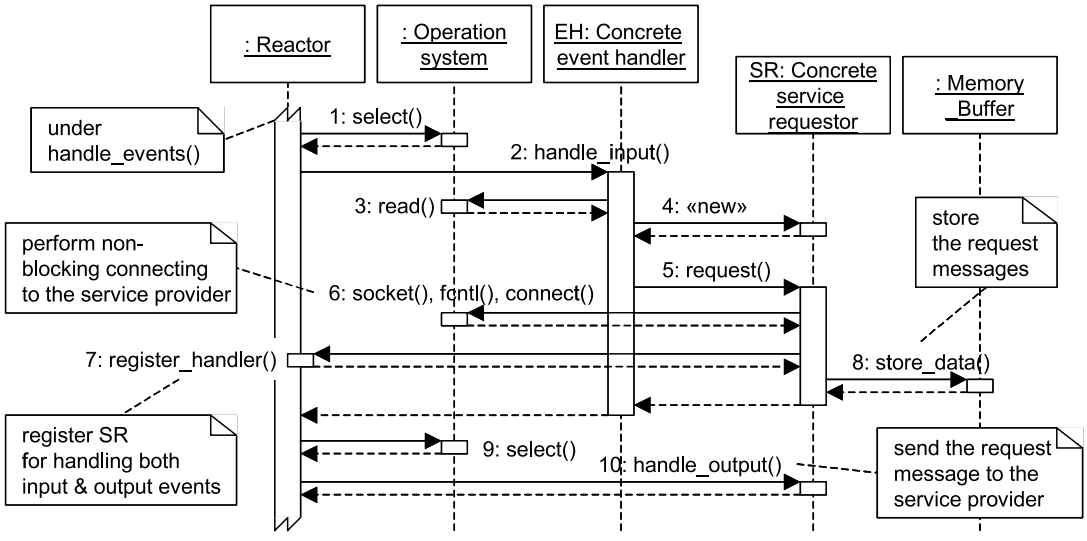

Fig. 6. Sequence diagram for establishing a connection and sending an HTTP request.

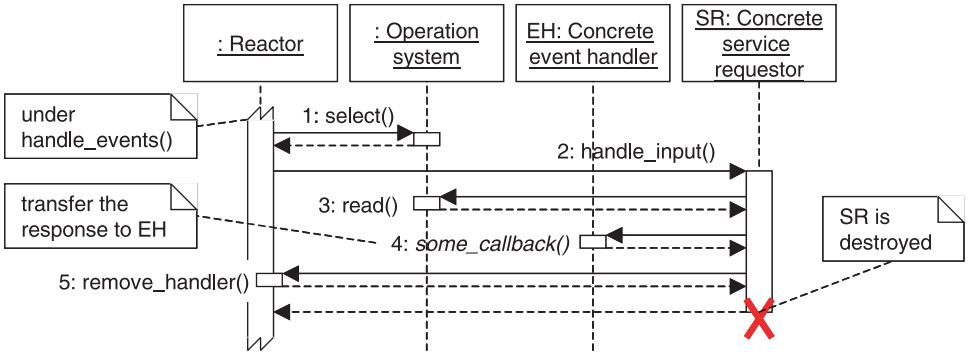

Fig. 7. Sequence diagram for handling responses.

the response has been completely received, SR destroys itself (in Step 5).

4.3. Solutions for other service requests

The previous subsection presents a service requesting mechanism for dealing with HTTP requests in the event-driven programming model. Since HTTP and its tools are pervasive, for example, web servers like Apache [2] and IIS [20], and web programming tools like PHP [29], JSP [37] and ASP [19], a straightforward solution for all the service requests would be to leverage the above solution based on the HTTP servers directly.

For example, if a source event handler (defined in Section 4.1) needs to access database servers or get the IP address of a given hostname, the handler makes an HTTP request to a web server, and the corresponding web page programs (say in PHP) then return database records or the IP address. Since it is easy to write the code of service requestors, as described in Section 4.2, and the corresponding web page programs (in PHP, JSP, or ASP), application developers can easily develop the above service request. Note that web page languages such as PHP, JSP, or ASP are usually sufficiently general and high-level to program service requests such as those listed in Section 4.1.

However, leveraging the HTTP technologies as above may incur significant overhead for web page processing (in PHP, JSP, or ASP). For example, accessing a database or getting the IP address of a given hostname in PHP generally may include process forking and page interpretation.

Since the incurred overhead may become significant, this paper designs additional helper processes for handling requests directly. The idea of helper processes has been used by the researchers in [7,28,43] for calling DNS-related functions and disk I/O access. However, they do not design a reusable software architecture for helper processes, as this paper does.

Consider the example of accessing database servers. A helper process contains an acceptor thread and a pool of worker threads, as illustrated in Fig. 8. The acceptor thread repeats to accept new connection requests from application servers (such as game servers) and then queues the sockets corresponding to these connection requests. Each worker thread then repeats the following steps:

1. Retrieve one socket from the queue.
2. Receive the request message (including the URL and the parameters) from the socket.
3. Identify the request and then create the corresponding service handler to process that request. For example, for a database request, the corresponding service handler sends the requests to the database servers, receives the response messages and then returns the results to the game server. Meanwhile, for a DNS request, the corresponding service handler simply calls the DNS library and returns the results to the application servers.

The class diagram in Fig. 9 shows the components of the helper processes. The class responsibilities are described as follows:

- Acceptor_Thread is the thread that waits to accept incoming connections. The sockets corresponding to accepted connections are placed in a Socket_Queue (described below).
- Socket_Queue is the queue that stores socket descriptors.
- Worker_Threads are threads that process requests in the Socket_Queue. Each Worker_Thread object

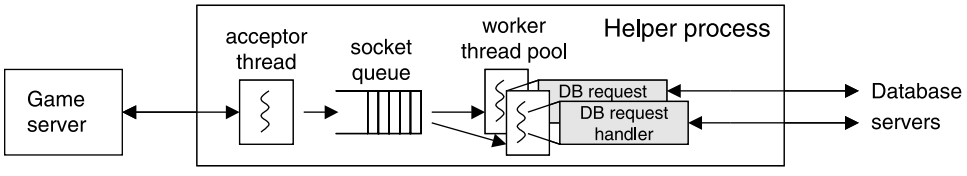

Fig. 8. Handling database requests using multi-threads.

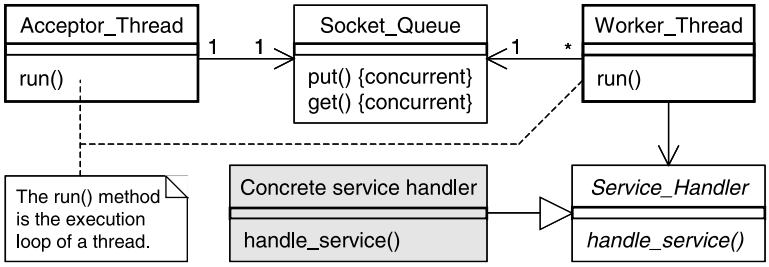

Fig. 9. Class diagram of helper processes.

creates a concrete service handler (described below) for application specific services.

- `Service_Handler` is a class of service handler interface that defines a hook method: `handle_service`. Classes of concrete service handlers that inherit `Service_Handler` implement this method for application specific services, such as database access services.

Fig. 10 shows the interaction of handling services in a helper process. The UML collaboration diagram is used to show this interaction:

- The acceptor thread `Ta` repeats the following two steps. In Step `a1`, it accepts a new socket, and in Step `a2`, it places accepted sockets in the `Socket_Queue`. If one or more worker threads are waiting in the queue, one of them is woken up.
- A worker thread, `Tw`, repeats the following steps. In Step `w1`, `Tw` attempts to obtain a socket from the `Socket_Queue`. `Tw` waits in this queue until the `Socket_Queue` becomes not empty. In Step `w2`, `Tw` reads the request message from the socket. In Step `w3`, `Tw` invokes the `handle_service` method of its own concrete service handler by passing the request message as an argument. In Step `w3.1`, the method invokes library calls, such as `gethostbyname`, for the request. Finally, in Step `w3.2`, the method returns the result to the application server.

Note that the above helper process has also an additional advantage, solving the following problem, called the *limited service problem* in this paper. Consider that a database server generally supports a limited number of service connections. The problem can be easily solved in the helper process designed here by simply limiting the number of worker threads to the number of connections to the database server and letting each thread hold a single connection. For example, if a database server supports only 20 connections, the helper process dedicated to all requests to the database server has a maximum of 20 worker threads.

In fact, the advantage of the helper process described above also applies to the processing of HTTP requests, for the following reason. A web server such as Apache generally limits the number of daemon processes for simultaneous requests [2]. When the number of concurrent HTTP requests exceeds the limited number, some of the additional HTTP requests may fail to establish connections or suffer from long latency [42]. Consequently, the helper process can be used simply to limit the number of worker threads to being the same as the number of HTTP daemon processes and thus let the service handler work like a HTTP proxy. For example, if an Apache server only allows 100 daemons, the helper process that is dedicated to all HTTP requests to the Apache server also has a maximum of 100 worker threads. The helper process thus can hold thousands of HTTP requests in the socket queue via the acceptor thread, while guaranteeing that 100 HTTP daemons in the Apache server are always available for the helper process.

The above thread pool can actually be implemented more efficiently, as described in [32,42]. Briefly, when worker threads are idle, some of these threads can be dynamically removed to reduce the overhead of context switching. The details can be read in [42] and are omitted here.

From the above, this paper suggests that helper processes be deployed as follows. For each server with limited service resources (e.g. an Apache server with a limited number of daemons or a database server with a limited number of connections), one helper process is dedicated to the server.

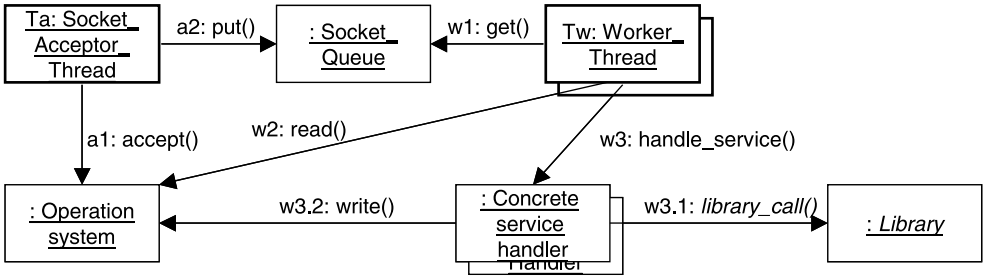

Fig. 10. Collaboration diagram of handling services in a helper process.

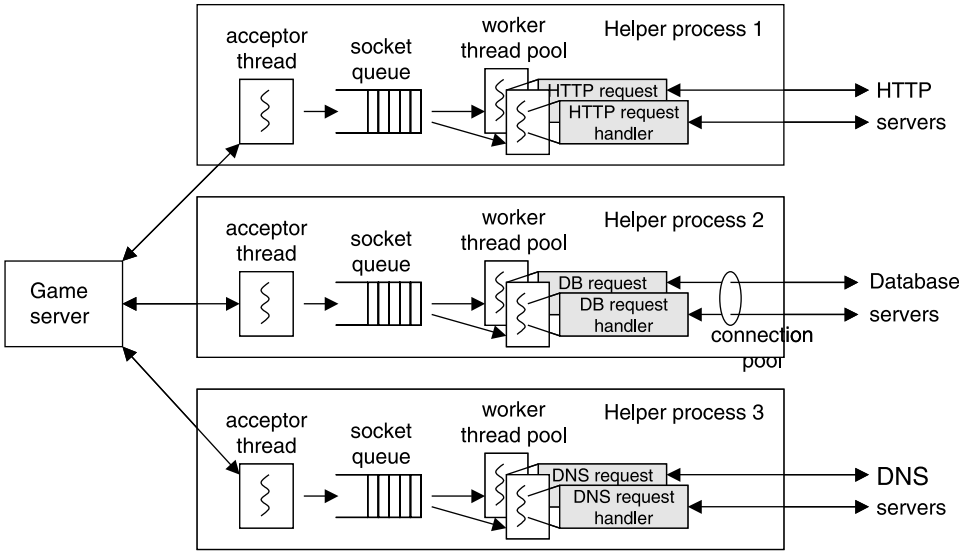

Fig. 11. Case of deploying helper processes.

Meanwhile, other services such as DNS services can be grouped into one or more helper processes, depending on the situation. Fig. 11 illustrates a case of deploying helper processes.

Finally, one may ask why the event-driven model is not used for the helper processes. Surely, the event-driven model can be used to implement the helper processes. However, the fact that these requests must wait for responses complicates helper process design. On the other hand, since threads in helper processes are almost independent, developers can easily maintain and debug the code in the thread model.

5. Experiments

Practically, we have implemented an event-driven framework with the output buffering mechanism and the service requesting mechanism, described in Sections 3 and 4, respectively. Section 5.1 briefly mentions the CYC game system [39] that was built on top of our event-driven framework. Section 5.2 presents the performance analysis for using the output buffering mechanism. The performance analysis for using the service requesting mechanism is similar and therefore is omitted in this paper.

5.1. Brief description of the CYC game system

The CYC game system [39] provides players with casual games, such as Chinese Chess, Bridge, Mahjong, etc. The system, popular in Taiwan and Hong Kong, has supported up to 10,000 concurrent players.

Fig. 12 shows the architecture of the CYC game system. This system includes a set of game servers (processes running on the FreeBSD operating system [15]), each of which can serve up to 1000 players. All game servers are connected to a coordinator (a process running on FreeBSD) that coordinates the communication among these servers and record the numbers of players handled by game servers (used to find the appropriate game severs for players). Game servers are also connected to database helpers and DNS helpers when accessing database and DNS servers. Note that

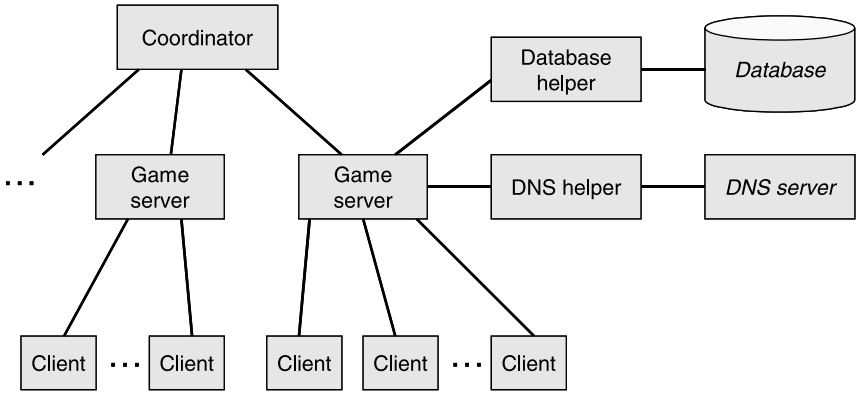

Fig. 12. The deployment of the CYC game system.

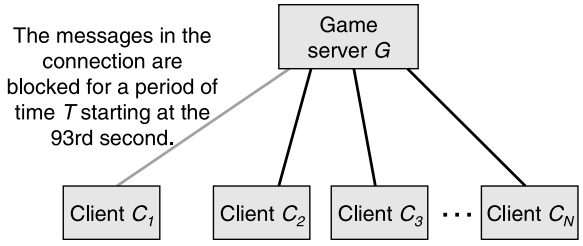

Fig. 13. The deployment of the evaluating the output buffering mechanism.

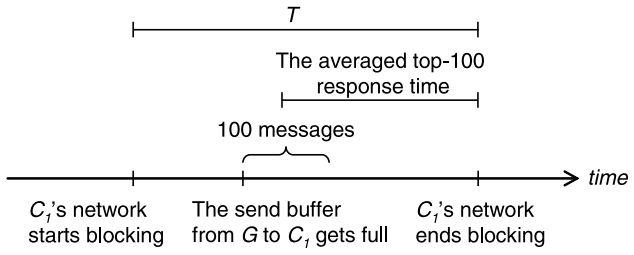

Fig. 14. The averaged top-100 response time.

game servers need to access database servers for players' game playing records and DNS servers for players' host domain names.

The coordinator, game servers, and helpers were developed based on our event-driven framework. The initial version of the system without our framework suffered from the blocking problems as described in Sections 3 and 4. The subsequent versions were revised based on our framework. The new versions elegantly solved these blocking problems by simply implementing event handlers and service handlers, as in the shadowed classes in Figs. 5 and 9. In addition, the extra helper processes also solved the limited service problem (as described in Section 4.3).

5.2. Performance analysis for the output blocking problem

This subsection presents the performance analysis on the experiments related to the output blocking problem. From the CYC game system, we logged all the events of some game server with about 300 players for 10 min. The log contains 28,573 received messages and 142,570 sent messages that represent the activity of the game server during that period.

For performance analysis, we simulated the activity of the log as follows. Let one host simulate the 300 clients (players) and the other simulate the game server following the messages indicated in the log. Both hosts ran on FreeBSD 5.3 and each of them was equipped with an AMD Athlon XP 2000+ CPU, 512 MB RAM, 80 GB hard disk, and a 100 Mb Ethernet card. Besides, they were connected to a 100 Mb switch hub directly.

In our experiment, we only consider the response times of the messages among clients (players), e.g. the messages for chatting or playing cards. Namely, for each of such messages among clients, add into the message M the time when sending M from the sender. When receiving M, the recipient measures the traveling time of M from the sender. Normally, the response times are short, unless the server is overloaded or the network traffic is jammed or blocked.

From the log, we chose one client, called C_1, as shown in Fig. 13, who entered the system at the 93rd second. Then, we blocked all messages to client C_1 for a period of time T starting at the 93rd second to simulate that the network between C_1 and the game server G was jammed or blocked, as shown in Fig. 13. We used the technique of divert sockets [3,8] to simulate the blocking network.

For each T, we measured the averaged top-100 response times in the two cases that (1) the output buffering mechanism was used and (2) the mechanism was not. Note that the averaged top-100 response time is the average of the highest 100 response times among all the messages. The top-100 response times reflect the worst response times among clients. In the case that T is sufficiently large, the averaged top-100 response time is the average of the response times of the first 100 messages after C_1's send buffer gets full, as shown in Fig. 14.

Fig. 15 shows our experiment results. Consider the second case that the mechanism is not used. When $T \leq 2$ s,

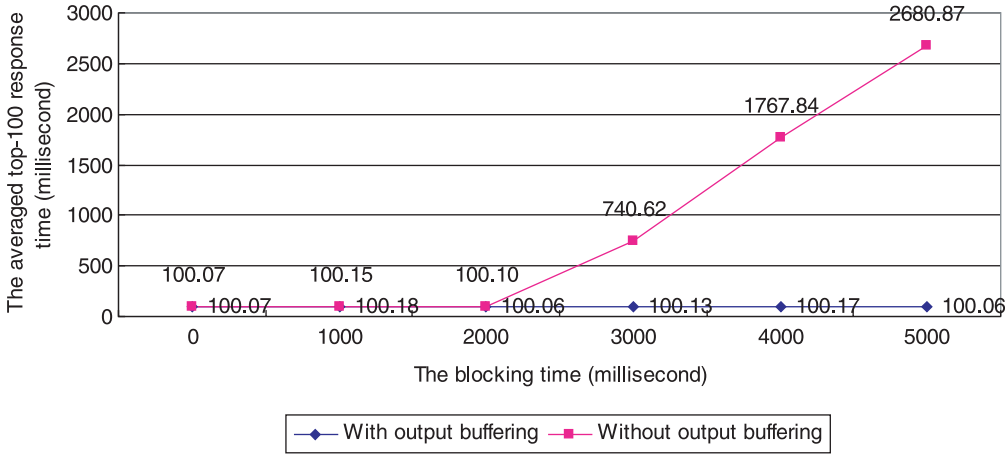

Fig. 15. The averaged top-100 response times vs. the blocking times in Fig. 13.

the averaged top-100 response times are still very low because in the FreeBSD kernel the send buffer (with about 32 kB) from the game server to client C_1 is not overflown and the whole game system therefore is not blocked, except for client C_1. However, when $T > 2$ s, the send buffer is overflown and the whole game system is blocked for the rest period of time T. Thus, the averaged top-100 response times grow nearly linear as T in this case. If the output buffering mechanism is used, the averaged top-100 response times become all very low as shown in Fig. 15. That is, the performance is greatly improved in this case.

6. Summary

Event-driven programming is a widely used technology for concurrent programming. However, the major drawback of event-driven programming is: the need to pay attention to the blocking problems in event handling. This paper addressed two major blocking problems in inter-user communication applications, namely output blocking and request blocking. For the former problem, this paper presents an output buffering mechanism to solve this problem. Meanwhile, for the latter problem this paper presents a service requesting mechanism with helper processes to solve this problem.

Based on the output buffering and service requesting mechanisms, this paper designs an event-driven framework for inter-user communication applications, as shown in Figs. 5 and 9. Application developers can apply this framework to develop their servers simply by implementing some event handlers and service handlers, as in the shadowed classes in Figs. 5 and 9. Therefore, based on this framework, they can easily avoid blocking problems.

Practically, the event-driven framework presented in this paper has been applied to the CYC game system in [39], which has been supported up to 10,000 concurrent players. The initial version of the game system was implemented based on a simple event-driven framework, but not on our framework. The blocking problems presented in this paper were not solved in this version. Since the authors served as consultants and incorporated our framework into the game system, the following problems have been gracefully solved:

- The output blocking problem.
- The request blocking problem. Since game records need to be maintained on remote servers sometimes, this problem becomes significant.
- The limited service problem (as described in Section 4.3). When the game system has thousands of concurrent players, HTTP requests for updating records may be missing sometimes due to the limited numbers of HTTP daemon processes on Apache servers. The extra helper processes described in Section 4.3 can be used to help solve this problem.

The above practical experience demonstrates that our event-driven framework can be easily used to develop and maintain a reliable and efficient game system. In fact, our framework can also be applied to other applications. For example, event-driven based HTTP servers, such as Zeus [46], mathopd [5], and thttpd [1], can be implemented on top of this framework by putting file access operations into helper processes.

Acknowledgements

The authors would like to thank ThinkNewIdea, Inc. [39] for offering required data for this research. The authors would also like to thank the anonymous referees for their valuable comments, which help to improve the presentation of this paper.

References

[1] ACME Laboratories, thttpd, http://www.acme.com/software/thttpd/ (last access: May 2005).
[2] The Apache Software Foundation, The Apache HTTP Server Project, http://httpd.apache.org/ (last access: May 2005).
[3] I. Baldine, Divert Sockets mini-HOWTO, http://www.faqs.org/docs/Linux-mini/Divert-Sockets-mini-HOWTO.html (last access: May 2005).
[4] Blizzard Entertainment, Inc., Blizzard Entertainment—Warcraft III, http://www.blizzard.com/war3/ (last access: May 2005).
[5] M. Boland, Mathopd, http://www.mathopd.org/ (last access: May 2005).
[6] G. Booch, I. Jacobson, J. Rumbaugh, The Unified Modeling Language User Guide, Addison-Wesley, Massachusetts, 1998.
[7] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, K.J. Worrell, A hierarchical internet object cache, Proceedings of the 1996 USENIX Technical Conference, San Diego, CA, USA, 1996, pp. 153–163.
[8] A. Cobbs, Divert, http://www.freebsd.org/cgi/man.cgi?query=divert (last access: May 2005).
[9] D.E. Comer, Internetworking with TCP/IP, fourth ed., Principles, Protocols, and Architecture, vol. 1, Prentice Hall, New Jersey, 2000.
[10] D.E. Comer, D.L. Stevens, Internetworking with TCP/IP, second ed., Client-Server Programming and Applications—BSD Socket Version, vol. 3, Prentice Hall, New Jersey, 1996.
[11] EFnet chat network, EFnet—The Original IRC Network, http://www.efnet.org/ (last access: May 2005).
[12] Electric Arts, Inc., ORIGIN—Ultima Online, http://www.uo.com/ (last access: May 2005).
[13] M.E. Fayad, D.C. Schmidt, R.E. Johnson, Build Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley, New York, 1999.
[14] M. Fisher, J. Ellis, J. Bruce, JDBC API Tutorial and Reference, third ed., Addison-Wesley, New Jersey, 2003.
[15] The FreeBSD core team, The FreeBSD project, http://www.freebsd.org/ (last access: May 2005).
[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Massachusetts, 1995.
[17] ICQ, Inc., *ICQ.com*, http://web.icq.com/ (last access: May 2005).

[18] M.K. McKusick, K. Bostic, M.J. Karels, J.S. Quarterman, The Design and Implementation of the 4.4BSD Operation System, Addison-Wesley, Massachusetts, 1996.
[19] Microsoft Corporation, Active Server Pages, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/active-servpages.asp (last access: May 2005).
[20] Microsoft Corporation, Internet Information Services, http://www.microsoft.com/WindowsServer2003/iis/default.mspx (last access: May 2005).
[21] Microsoft Corporation, MSDN: ODBC, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/HTML/_core_odbc.asp (last access: May 2005).
[22] Microsoft Corporation, MSDN: Thread Stack Size, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/thread_stack_size.asp (last access: May 2005).
[23] Microsoft Corporation, MSN Messenger, http://messenger.msn.com/ (last access: May 2005).
[24] B. Nichols, D. Buttlar, J.P. Farrel, Pthreads Programming: A POSIX Standard for Better Multiprocessing, Reilly, California, 1996.
[25] H. Nielsen, T. Berners-Lee, J. Groff, Libwww—the W3C Sample Code Library, http://www.w3.org/Library/ (last access: May 2005).
[26] OpenLDAP Foundation, OpenLDAP Software Man Pages: ldap, http://www.openldap.org/software/man.cgi?query=ldap (last access: May 2005).
[27] J. Ousterhout, Why threads are a bad idea (for most purposes), invited talk at the 1996 USENIX Technical Conference, San Diego, CA, USA, 1996, see also http://home.pacbell.net/ouster/threads.pdf (last access: May 2005).
[28] V. Pai, P. Druschel, W. Zwaenepoel, Flash: an efficient and portable web server, Proceedings of USENIX Annual Technical Conference, Monterey, CA, USA, 1999, pp. 199–212.
[29] The PHP Group, PHP: Hypertext Preprocessor, http://www.php.net/ (last access: May 2005).
[30] W. Pree, Design Patterns for Object-Oriented Software Development, Addison-Wesley, Massachusetts, 1995.
[31] D.C. Schmidt, Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching, in: Pattern Languages of Program Design, Addison-Wesley, Massachusetts, 1995. pp 529–545.
[32] D.C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Object, vol. 2, Wiley, New York, 2000.
[33] S. Srinvasan, Design patterns in object-oriented frameworks, IEEE Computer 32 (2) (1999) 24–32.
[34] W.R. Stevens, Advanced Programming in the UNIX Environment, Addison-Wesley, Massachusetts, 1992.
[35] W.R. Stevens, TCP/IP Illustrated, The Protocols, vol. 1, Addison-Wesley, Massachusetts, 1994.
[36] W.R. Stevens, UNIX Network Programming, second ed., Networking API: Sockets and XTI, vol. 1, Prentice Hall, New Jersey, 1998.
[37] Sun Microsystems, Inc., JavaServer Pages Technology, http://java.sun.com/products/jsp/ (last access: May 2005).
[38] Sun Microsystems, Inc., JDBC Technology, http://java.sun.com/products/jdbc/ (last access: May 2005).
[39] ThinkNewIdea, Inc., CYC Game League, http://cycgame.com/ (last access: May 2005).
[40] J. Tranter, Exploring the sendfile system call, Linux Gazette, Issue 91, June 2003.
[41] Valve Corporation, The official Counter-Strike web site, http://www.counter-strike.net/ (last access: May 2005).
[42] M. Welsh, D. Culler, E. Brewer, SEDA: an architecture for well-conditioned scalable internet services, Proceedings of the 18th ACM Symposium on Operating Systems Principles, Alberta, Canada, 2001, pp. 230–243.
[43] D. Wessels et al., Squid Web Proxy Cache, http://www.squid-cache.org/ (last access: May 2005).
[44] Yahoo!, Inc., Yahoo! Chat, http://chat.yahoo.com/ (last access: May 2005).
[45] Yahoo!, Inc., Yahoo! Games, http://games.yahoo.com/ (last access: May 2005).
[46] Zeus Technology Limited, Zeus Web Server, http://www.zeus.co.uk/products/zws/ (last access: May 2005).