Building a modern server application requires a method of accepting hundreds, thousands, and even tens of thousands of events simultaneously, whether they are internal requests or network connections effectively handling their operation. There are many solutions available, but the libevent library and libev libraries have both revolutionized the performance and event handling capability. In this article, we will examine the basic structure and methods available for using and deploying these solutions within your UNIX® applications. Both libev and libevent can be used in your high performance applications, including those deployed within the IBM Cloud or Amazon EC2 environment, where you need to support large numbers of simultaneous clients or operations.

Introduction

One of the biggest problems facing many server deployments, particularly web server deployments, is the ability to handle a large number of connections. Whether you are building cloud-based services to handle network traffic, distributing your application over IBM Amazon EC instances, or providing a high-performance component for your web site, you need to be able to handle a large number of simultaneous connections.

A good example is the recent move to more dynamic web applications, especially those using AJAX techniques. If you are deploying a system that allows many thousands of clients to update information directly within a web page, such as a system providing live monitoring of an event or issue, then the speed at which you
can effectively serve the information is vital. In a grid or cloud situation, you might have permanent open connections from thousands of clients simultaneously, and you need to be able to serve the requests and responses to each client.

Before looking at how libevent and libev are able to handle multiple network connections, let's take a brief look at some of the traditional solutions for handling this type of connectivity.

Handling multiple clients

There are a number of different traditional methods that handle multiple connections, but usually they result in an issue handling large quantities of connections, either because they use too much memory, too much CPU, or they reach an operating system limit of some kind.

The main solutions used are:

• **Round-robin**: The early systems use a simple solution of round-robin selection, simply iterating over a list of open network connections and determining whether there is any data to read. This is both slow (especially as the number of connections increases) and inefficient (since other connections may be sending requests and expecting responses while you are servicing the current one). The other connections have to wait while you iterate through each one. If you have 100 connections and only one has data, you still have to work through the other 99 to get to the one that needs servicing.

• **poll, epoll, and variations**: This uses a modification of the round-robin approach, using a structure to hold an array of each of the connections to be monitored, with a callback mechanism so that when data is identified on a network socket, the handling function is called. The problem with poll is that the size of the structure can be quite large, and modifying the structure as you add new network connections to the list can increase the load and affect performance.

• **select**: The `select()` function call uses a static structure, which had previously been hard-coded to a relatively small number (1024 connections), which makes it impractical for very large deployments.

There are other implementations on individual platforms (such as `/dev/poll on Solaris, or kqueue on FreeBSD/NetBSD) that may perform better on their chosen OS, but they are not portable and don't necessarily resolve the upper level problems of handling requests.

All of the above solutions use a simple loop to wait and handle requests, before dispatching the request to a separate function to handle the actual network
interaction. The key is that the loop and network sockets need a lot of management code to ensure that you are listening, updating, and controlling the different connections and interfaces.

An alternative method of handling many different connections is to make use of the multi-threading support in most modern kernels to listen and handle connections, opening a new thread for each connection. This shifts the responsibility back to the operating system directly but implies a relatively large overhead in terms of RAM and CPU, as each thread will need its own execution space. And if each thread (ergo network connection) is busy, then the context switching to each thread can be significant. Finally, many kernels are not designed to handle such a large number of active threads.

The libevent approach

The libevent library doesn't actually replace the fundamentals of the `select()`, `poll()` or other mechanisms. Instead, it provides a wrapper around the implementations using the most efficient and high-performance solution on each platform.

To actually handle each request, the libevent library provides an event mechanism that acts as a wrapper around the underlying network backend. The event system makes it very easy and straightforward to add handlers for the connections while simplifying the underlying I/O complexities. This is the core of the libevent system.

Additional components of the libevent library add further functionality, including a buffered event system (for buffer data to/from clients) and core implementations for HTTP, DNS and RPC systems.

The basic method for creating a libevent server is to register functions to be executed when a particular operation occurs, such as accepting a connection from a client, and then call the main event loop `event_dispatch()`. Control of the execution process is now handled by the libevent system. The event system is autonomous once you have registered the events and the functions that will call them, and you can add (register) or remove (deregister) events from the event queue while the application is running. It is this freedom with the event registration that enables you to build flexible network handling systems, as you can add new events to handle newly opened connections.

For example, you could create a network server by opening a listening socket, and then registering a callback function each time the `accept()` function needs to be called to open a new connection. A fragment outlining the basics of this are shown in Listing 1:

Listing 1. Creating a network server by opening a listening socket and
registering a callback function each time the accept() function needs to be called to open a new connect

```c
int main(int argc, char **argv)
{
  ...
  ev_init();
  /* Setup listening socket */
  event_set(&ev_accept, listen_fd, EV_READ|EV_PERSIST, on_accept, NULL);
  event_add(&ev_accept, NULL);
  /* Start the event loop. */
  event_dispatch();
}
```

The `event_set()` function creates the new event structure, while `event_add()` adds the event to the event queue mechanism. The `event_dispatch()` then starts the event queue system and starts listening (and accepting) requests.

A more complete example is provided in Listing 2, which builds a very simple echo server:

**Listing 2. Building a simple echo server**

```c
#include <event.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define SERVER_PORT 8080
int debug = 0;

struct client {
  int fd;
  struct bufferevent *buf_ev;
};

int setnonblock(int fd)
{
  int flags;
  flags = fcntl(fd, F_GETFL);
  flags |= O_NONBLOCK;
  fcntl(fd, F_SETFL, flags);
}

void buf_read_callback(struct bufferevent *incoming, 
                         void *arg)
{
  struct evbuffer *evreturn;
  char *req;
```
req = evbuffer_readline(incoming->input);
if (req == NULL)
    return;

evreturn = evbuffer_new();
evbuffer_add_printf(evreturn,"You said %s\n", req);
bufferevent_write_buffer(incoming, evreturn);
evbuffer_free(evreturn);
free(req);
}

void buf_write_callback(struct bufferevent *bev,  
                        void *arg)
{
}

void buf_error_callback(struct bufferevent *bev,  
                        short what,  
                        void *arg)
{
    struct client *client = (struct client *)arg;
bufferevent_free(client->buf_ev);
close(client->fd);
free(client);
}

void accept_callback(int fd,  
                      short ev,  
                      void *arg)
{
    int client_fd;
    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof(client_addr);
    struct client *client;
    client_fd = accept(fd,  
                       (struct sockaddr *)&client_addr,  
                       &client_len);
    if (client_fd < 0)
    {
        warn("Client: accept() failed");
        return;
    }
    setnonblock(client_fd);
    client = calloc(1, sizeof(*client));
    if (client == NULL)
        err(1, "malloc failed");
    client->fd = client_fd;
    client->buf_ev = bufferevent_new(client_fd,  
                                   buf_read_callback,  
                                   buf_write_callback,  
                                   buf_error_callback,  
                                   client);
    bufferevent_enable(client->buf_ev, EV_READ);
}

int main(int argc,  
          char **argv)
{
    int socketlisten;
    struct sockaddr_in addresslisten;
    struct event accept_event;
    int reuse = 1;
    event_init();
socketlisten = socket(AF_INET, SOCK_STREAM, 0);
if (socketlisten < 0)
{
    fprintf(stderr,"Failed to create listen socket");
    return 1;
}
memset(&addresslisten, 0, sizeof(addresslisten));
addresslisten.sin_family = AF_INET;
addresslisten.sin_addr.s_addr = INADDR_ANY;
addresslisten.sin_port = htons(SERVER_PORT);
if (bind(socketlisten,
    (struct sockaddr *)&addresslisten,
    sizeof(addresslisten)) < 0)
{
    fprintf(stderr,"Failed to bind");
    return 1;
}
if (listen(socketlisten, 5) < 0)
{
    fprintf(stderr,"Failed to listen to socket");
    return 1;
}
setsockopt(socketlisten,
    SOL_SOCKET,
    SO_REUSEADDR,
    &reuse,
    sizeof(reuse));
setnonblock(socketlisten);
event_set(&accept_event,
    socketlisten,
    EV_READ|EV_PERSIST,
    accept_callback,
    NULL);
event_add(&accept_event,
    NULL);
event_dispatch();
close(socketlisten);
return 0;
}

The different functions and their operation are discussed below:

- **main()**: The main function creates the socket to be used for listening to connections, and then creates the callback for accept() to handle each connection through the event handler.

- **accept_callback()**: The function called by the event system when a connection is accepted. The function accepts the connection to the client; adds the client socket information and a bufferevent structure; adds callbacks for read/write/error events on the client socket to the event
structure; and passes the client structure (with the embedded eventbuffer and client socket) as an argument. Each time a corresponding client socket contains any read, write or error operations, the corresponding callback function is called.

- **buf_read_callback()**: Called when the client socket has data to be read. As an echo service, the function writes "you said..." back to the client. The socket remains open to accept new requests.

- **buf_write_callback()**: Called when there is data to be written. In this simple service you don't need this service, and so the definition is blank.

- **buf_error_callback()**: Called when an error condition exists. This includes when the client disconnects. In all situations the client socket is closed, and the event entry for the client socket is removed from the event list. The memory for the client structures is freed.

- **setnonblock()**: Sets the network socket to non-blocking I/O.

As the number of clients connect, new events to handle the client connection are added to the event queue and removed when the client disconnects. Behind the scenes, libevent is handling the network sockets, identifying which clients need to be serviced, and calling the corresponding functions in each case.

To build the application, compile the C source code adding the libevent library:

```
gcc -o basic basic.c -levent
```

From a client perspective, the server just echoes back any text sent to it (see Listing 3 below).

**Listing 3. Server echoes back text sent to it**

```
$ telnet localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello!
You said Hello!
```

Network applications like this can be useful with large scale distributed deployments, such as an IBM Cloud system, where you need to service multiple connections.

It is difficult with such a simple solution to see the performance benefits and the large number of simultaneous connections. Using the embedded HTTP implementation can help to understand the mass scalability.

**Using the built-in HTTP server**
The plain network-based libevent interface is useful if you want to build native applications, but it is increasingly common to develop an application based around the HTTP protocol and a web page that loads, or more commonly dynamically reloads, information. If you are using any of the AJAX libraries, it expects an HTTP at the other end, even if the information you are returning is XML or JSON.

The HTTP implementation within libevent is not going to replace Apache's HTTP server, but it can be a practical solution for the sort of large-scale dynamic content associated with both cloud and web environments. For example, you could deploy a libevent based interface to your IBM Cloud management or other solution. Since you can communicate using HTTP, the server can integrate with the other components.

To use the libevent service, you use the same basic structure as already described for the main network event model, but instead of having to handle the network interfacing, the HTTP wrapper handles that for you. This turns the entire process into the four function calls (initialize, start HTTP server, set HTTP callback function, and enter event loop), plus the contents of the callback function that will send data back. A very simple example is provided in Listing 4:

Listing 4. Simple example of using the libevent service

```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <event.h>
#include <evhttp.h>

void generic_request_handler(struct evhttp_request *req, void *arg)
{
    struct evbuffer *returnbuffer = evbuffer_new();

    evbuffer_add_printf(returnbuffer, "Thanks for the request!");
    evhttp_send_reply(req, HTTP_OK, "Client", returnbuffer);
    evbuffer_free(returnbuffer);
    return;
}

int main(int argc, char **argv)
{
    short http_port = 8081;
    char *http_addr = "192.168.0.22";
    struct evhttp *http_server = NULL;

    event_init();
    http_server = evhttp_start(http_addr, http_port);
    evhttp_set_gencb(http_server, generic_request_handler, NULL);

    fprintf(stderr, "Server started on port %d\n", http_port);
    event_dispatch();
    return(0);
}
```

Boost network performance with libevent and libev

Trademarks

© Copyright IBM Corporation 2010. All rights reserved.
Given the previous example, the basics of the code here should be relatively self-explanatory. The main elements are the `evhttp_set_gencb()` function, which sets the callback function to be used when an HTTP request is received, and the `generic_request_handler()` callback function itself, which populates the response buffer with a simple message to show success.

The HTTP wrapper provides a wealth of different functionality. For example, there is a request parser that will extract the query arguments from a typical request (as you would use in a CGI request), and you can also set different handlers to be triggered within different requested paths. An interface to your database could be provided using the path '/db', or an interface through to memcached as '/memc', with different callbacks and handling accordingly.

One other element of the libevent toolkit is support for generic timers. These allow you to schedule events after a specific period. You can combine this with the HTTP implementation to provide a lightweight service to serve up the contents of a file, updating the data returned as the file content is modified. For example, if you were providing a live update service during a busy news event where the front-end web application kept periodically reloading the news item, you could easily serve up the content. The entire application (and web service) would be in memory making the response times very quick.

This is the main purpose behind the example in Listing 5:

**Listing 5. Using a generic timer to provide a live update service during a busy news event**

```c
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <event.h>
#include <evhttp.h>

#define RELOAD_TIMEOUT 5
#define DEFAULT_FILE "sample.html"

char *filedata;
time_t lasttime = 0;
char filename[80];
int counter = 0;

void read_file()
{
    int size = 0;
    char *data;
    struct stat buf;
    stat(filename,&buf);
    if (buf.st_mtime > lasttime)
    {
        if (counter++
```
```c
fprintf(stderr,"Reloading file: %s", filename);
else
    fprintf(stderr,"Loading file: %s", filename);

FILE *f = fopen(filename, "rb");
if (f == NULL) {
    fprintf(stderr,"Couldn't open file\n");
    exit(1);
}

fseek(f, 0, SEEK_END);
size = ftell(f);
fseek(f, 0, SEEK_SET);
data = (char *)malloc(size+1);
filedata = (char *)malloc(size+1);
strcpy(filedata, data);
fclose(f);

fprintf(stderr," (%d bytes)\n", size);
lasttime = buf.st_mtime;
}

void load_file()
{
    struct event *loadfile_event;
    struct timeval tv;

    read_file();
    tv.tv_sec = RELOAD_TIMEOUT;
    tv.tv_usec = 0;
    loadfile_event = malloc(sizeof(struct event));
    evtimer_set(loadfile_event,
        load_file,
        loadfile_event);
    evtimer_add(loadfile_event,
        &tv);
}

void generic_request_handler(struct evhttp_request *req, void *arg)
{
    struct evbuffer *evb = evbuffer_new();
    evbuffer_add_printf(evb, "%s", filedata);
    evhttp_send_reply(req, HTTP_OK, "Client", evb);
    evbuffer_free(evb);
}

int main(int argc, char *argv[])
{
    short http_port = 8081;
    char *http_addr = "192.168.0.22";
    struct evhttp *http_server = NULL;

    if (argc > 1) {
        strcpy(filename, argv[1]);
        printf("Using %s\n", filename);
    }
    else {
        strcpy(filename, DEFAULT_FILE);
    }
```
event_init();
load_file();
http_server = evhttp_start(http_addr, http_port);
evhttp_set_gencb(http_server, generic_request_handler, NULL);
fprintf(stderr, "Server started on port %d\n", http_port);
event_dispatch();
}

The basic mechanics of the server are the same as the previous example. First, the
script sets up an HTTP server which will just respond to requests on the basic URL
host/port combination (no processing of the request URI). The first step is to load the
file (read_file()). The same function is used to load the original and will be used
during the callback by the timer event.

The read_file() function uses the stat() function call to check the modification
time of the file, only re-reading the contents of the file if it changed since the last time
the file was loaded. The function loads the file data using a single call to fread(),
copying the data into a separate structure, before using a strcpy() to move the
data from the loaded string to the global string.

The load_file() function is the one that will act as the function when the timer is
triggered. It calls read_file() to load the content, and then sets the timer using
the RELOAD_TIMEOUT value as the number of seconds before the file load is
attempted. The libevent timer uses the timeval structure, which allows timers to be
specified in both seconds and microseconds. The timer is not periodic; you set it
when the timer event is triggered, and then the event is removed from the event
queue.

To compile, use the same format as the previous examples: $ gcc -o
basichttpfile basichttpfile.c -levent

Now create a static file to be used as the data; the default is sample.html, but you
can specify any file as the first argument on the command-line (see Listing 6 below).

Listing 6. Create a static file to be used as the data

$ ./basichttpfile
Loading file: sample.html (8046 bytes)
Server started on port 8081

At this point, the program is ready to accept requests, but the reload timer is also in
operation. If you change the content of sample.html, the file should automatically be
reloaded with a message logged. For example, the output in Listing 7 shows the
initial load and two reloads:
Listing 7. Output showing the initial load and two reloads

```
$ ./basichttpfile
Loading file: sample.html (8046 bytes)
Server started on port 8081
Reloading file: sample.html (8047 bytes)
Reloading file: sample.html (8048 bytes)
```

Note that to get the full benefit, you must ensure that your environment does not have an ulimit on the number of open file descriptors. You can change this (with suitable permissions or root access) using the ulimit command. The exact setting will depend on your OS, but on Linux® you can set the number of open file descriptors (and therefore network sockets) with the \(-n\) option:

Listing 8. Using the \(-n\) option to set the number of open file descriptors

```
$ ulimit -n
1024
```

To increase your limit, specify a figure: $ ulimit \(-n\) 20000.

To check the performance of the server, you can use a benchmarking application, such as Apache Bench 2 (ab2). You can specify the number of simultaneous queries, as well as total number of requests. For example, to run a benchmark using 100,000 requests, 1000 of them simultaneously: $ ab2 \(-n\) 100000 \(-c\) 1000 http://192.168.0.22:8081/.

Running this sample system, using the 8K file shown in the server sample, I achieved almost 11,000 requests/s. Keep in mind that the libevent server is running in a single thread, and a single client is unlikely to stress the server, since it will also be limited by the method of opening requests. Even so, that rate is impressive for a single threaded application given the comparative size of the document being exchanged.

Alternative language implementations

Although C is a practical language for many system applications, it is not often used in modern environments where a scripting language can be more flexible and practical. Fortunately, most scripting languages, such as Perl and PHP, are written natively in C and so can make use of a C library like libevent through their extension modules.

For example, Listing 9 shows the basic structure of a script for a Perl network server. The `accept_callback()` function would be the same as the accept function in the core libevent example in Listing 1.
### Listing 9. Basic structure of a script for a Perl network server

```perl
my $server = IO::Socket::INET->new(  
    LocalAddr => 'localhost',  
    LocalPort => 8081,  
    Proto => 'tcp',  
    ReuseAddr => SO_REUSEADDR,  
    Listen => 1,  
    Blocking => 0,  
) or die $@;

my $accept = event_new($server, EV_READ|EV_PERSIST, \&accept_callback);

$main->add;

event_mainloop();
```

The library implementations in these languages tend to support the core of the libevent system and not always the HTTP wrapper. Using these solutions with a scripted application is therefore made more complex. There are two routes: either embed the language into your C-based libevent application or use one of the many HTTP implementations built on top of the scripted language environment. For example, Python includes the very capable HTTP server class (httplib/httplib2).

Despite this functionality, it should be pointed out that there is nothing in a scripting language that cannot be re-implemented in C. However, there is a time consideration and integrating with your existing codebase may be more critical.

### The libev library

The libev system is, like libevent, an event loop based system that builds on top of the native implementations of `poll()`, `select()`, and so on to provide an event based loop. At the time I wrote this article, the libev implementation had a lower overhead, leading to higher benchmarks. The libev API is more raw, and there is no HTTP wrapper, but libev does provide support for more types of events built-in to the implementation. For example, there is an `evstat` implementation that can be used to monitor attribute changes on multiple files, which could have been used in the HTTP file solution in Listing 4.

The fundamentals, however, remain the same. You create the necessary network listening sockets, register the events to be called during execution, and then start the main event loop with the libev handling the rest of the process.

For example, using the Ruby interface, you can provide an echo server similar to that provided in the first code listing, shown here in Listing 10.

### Listing 10. Using Ruby interface to provide an echo server

```ruby
ibm.com/developerWorks developerWorks®
Boost network performance with libevent and libev Trademarks
© Copyright IBM Corporation 2010. All rights reserved.
require 'rubygems'
require 'rev'

PORT = 8081

class EchoServerConnection < Rev::TCPSocket
  def on_read(data)
    write 'You said: ' + data
  end
end

server = Rev::TCPServer.new('192.168.0.22', PORT, EchoServerConnection)
server.attach(Rev::Loop.default)

puts "Listening on localhost:#{PORT}"

Rev::Loop.default.run

The Ruby implementation is particularly nice, since wrappers have been provided for many common network solutions, including HTTP client, OpenSSL and DNS. Other scripted languages include a comprehensive Perl and Python implementation that you might want to try.

Summary

Both libevent and libev provide a flexible and powerful environment for supporting high-volume network (and other I/O) interfaces for servicing either server-side or client-side requests. The aim is to support thousands, and even tens of thousands, of connections in an efficient (low CPU/RAM) format. In this article, you have seen examples of this, including the built-in HTTP service in libevent, that can be used to support IBM Cloud, EC2 or AJAX-based web applications.
Resources

Learn

- The C10K problem provides an excellent overview of the issues of handling 10,000 connections.

- The IBM Cloud Computing web site provides information on different cloud implementations.

- Read System Administration Toolkit: Standardizing your UNIX command-line tools (Martin Brown, developerWorks, May 2006): Learn how to use the same command across multiple machines.

- For an article series that will teach you how to program in bash, see Bash by example, Part 1: Fundamental programming in the Bourne again shell (bash) (Daniel Robbins, developerWorks, March 2000), Bash by example, Part 2: More bash programming fundamentals (Daniel Robbins, developerWorks, April 2000), and Bash by example, Part 3: Exploring the ebuild system (Daniel Robbins, developerWorks, May 2000).


- Navigate the cloud computing labyrinth (Brett McLaughlin, developerWorks, March 2009): Helps you make an educated decision about the best cloud computing platform for your particular application requirements.


- IBM products are available for the Amazon EC2 platform from the developerWorks Cloud Computing Resource Center.

- New to AIX and UNIX: Visit the New to AIX and UNIX page to learn more about AIX and UNIX.

- The developerWorks AIX and UNIX zone hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials.

- Listen to interesting interviews and discussions for software developers on developerWorks podcasts.

- developerWorks technical events and webcasts: Stay current with developerWorks technical events and webcasts.

Get products and technologies

- Get the libev library, including download and documentation.
• Get the libevent library.
• The ruby libev (rev) library and documentation.
• Memcached is an RAM cache for storing and handling data (which uses libevent at it's core, as well as being used with other libevent servers).
• Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

• Participate in developerWorks blogs and get involved in the developerWorks community.

• Follow developerWorks on Twitter.
• Get involved in the My developerWorks community.
• Participate in the AIX and UNIX® forums:
  • AIX Forum
  • AIX Forum for developers
  • Cluster Systems Management
  • IBM Support Assistant Forum
  • Performance Tools Forum
  • Virtualization Forum
  • More AIX and UNIX Forums

About the author

Martin Brown

Martin Brown has been a professional writer for over eight years. He is the author of numerous books and articles across a range of topics. His expertise spans myriad development languages and platforms - Perl, Python, Java, JavaScript, Basic, Pascal, Modula-2, C, C++, Rebol, Gawk, Shellscript, Windows, Solaris, Linux, BeOS, Mac OS/X and more - as well as web programming, systems management and integration. Martin is a regular contributor to ServerWatch.com, LinuxToday.com, IBM developerWorks and a regular blogger at Computerworld, The Apple Blog, and other sites. He is also a Subject Matter Expert (SME) for Microsoft. He can be contacted through his website at http://www.mcslp.com.