Hands-on Tutorial on Server Implementation (draft)

Jackie Dinh

August 2017

Contents

1 Introduction

This tutorial provides a well-designed approach to develop a good performance server in C/C++. Here 'good performance' means that a server can handle at least 10.000 requests per second or 10.000 concurrent connections or c10k issue [1]. Throughout this tutorial, readers will learn practices and techniques of designing and building a robust application server. These techniques includes event-driven approach, multi-processes, sharing data and performance optimization. These practices are not the best but have been proven good enough in practice. More importantly, it accelerates software development with more focus on business logic of applications.

This tutorial covers some key aspects for implementing a high performance server. It by no means is full coverage of the topic of high performance servers, but hopes that it gives readers a first glimpse into high performance server implementation. Curious readers may take a look at [2] for recent development on the topic. Good example are WhatsApp who has claimed to have more than 2.000.000 of concurrent connections per server [3] [1]. Finally, it does not cover topics of tuning operation systems and hardware-related performance.

Some historical events for curious readers:

- In 1999, a scalable event mechanism was proposed for Unix by Banga [4].
- In 1999, Kegel [1] reached 10.000 concurrent connections for the first time.
- In 2000, Welsh proposed event-driven approach for highly concurrent servers [5, 6].
- In 2000, a framework for network IO called libevent was written by Niels Provos.
- In 2001, kqueue was introduced for FreeBSD to address scalability issue of select/poll mechanism by Lemon [7].
- In 2002, epoll - a variant of kqueue - was first introduced in kernel version 2.5.44.
- In 2006, Drepper proposed a new high-speed asynchronous networking API [8].
- In 2010, netmap - a framework for high speed packet I/O - paved a road to solve the c10m issue [9].
- In 2012, WhatsApp confirmed to have more than 2.000.000 of concurrent connections per server [3].
- In 2014, mTcp - a user-space networking stack - showed 25x higher performance against Linux [10].
- In 2015, MigratoryData confirmed to have over 10.000.000 of concurrent connections per server [11].

1.1 Prerequisites and source code

Ubuntu 16.04 is used in this tutorial. The only package needed is libevent:

```
# apt-get install libevent
```

Get source code:

[1] Recently, MigratoryData claimed to have more than 10.000.000 of concurrent connections per server

```
# git clone https://github.com/jackiedinh8/svrtut
```

Each section has its own source code, which will be developed over the course of this tutorial.

1.2 Organization of tutorial

The tutorial is organized in the following ways. In the next section, we will discuss some design choices and their reasoning for building good performance server. In the section 3, we will cover basics of libevent and write simple echo server on it. In the section 4, message queues will be described in details. Next, we will discuss multi-processes with message queues in the section 5. Asynchronous requests and forwarding messages will be discussed in the section 5. Caching and sharing technique described in the final section. At the end of each section, there are reading and practical exercises which encourage readers to take. The exercises with (*) are optional.

2 General Design

2.1 Event-driven architecture

The term 'c10k' was coined by Kegel in 1999 when he first claimed to have ten thousands of concurrent connections per server. Traditional server architectures were based on select/poll mechanism which has limited capabilities to handle a large number of concurrent connections [12]. The reason is that using this mechanism has a cost at O(n) for IO operations, therefore very expensive operations. Around that time, kqueue was introduced by [7] to solve it. Later, kqueue was implemented in Linux and known as epoll. kqueue has linear complexity O(1) for IO operations.

A new event-driven architecture (or asynchronous) was also introduced by Welsh [5] to replace traditional architectures. Event-driven approach was proven successful in project nginx, a popular web server. Several IO frameworks were also introduced based on this idea. Notable examples are libevent, libev, Asio, etc. In this tutorial, libevent is chosen as network programming library.

2.2 Multi-threads vs multi-processes

Thread is a light weight process which intend to use more cpu cycles as much as possible. However when come to programming, it creates complexity to develop stable and robust server. First, using multi-threads tends to use lock mechanisms to synchronize access to shared resources. Locks are possibly source of tensions between threads and therefore can cause huge impact on performance, in particular when using them improperly. Second, algorithms and data structures to guarantee thread-safety and synchronization leads to more complicated code, which results in cost of debugging, testing and maintenance.

Throughout this tutorial, multi-processes paradigm will be developed to tackle exactly two above issues. First, execution of a single process is much easier to be understood and debugged, therefore it eases development of single-process applications. Second, aside of avoiding lock mechanisms, using multi-processes instead of multi-threads could be useful if there is a way of exchanging data among processes, which is efficient and acceptable to errors[2]. This is what leads us to the next topic: lock-free message queues as an efficient way of process communications[3].

[2]We may sacrifice a very very small of possibility of error for performance

[3]For pro-thread viewpoint, Behren [13] presented a discussion on the topic

2.3 Lock-free message queues

Unix/Linux operating systems introduce several ways for interprocess communications (called IPC mechanism), namely tcp/ip stack, message queue, signals, pipes, shared memory etc. Most of time this results in wasting a lot of cpu cycles, except shared memory. This is because data must travels from one process to other process through all levels of operation systems back and forth. For example, using sockets or pipes to share data between processes A and B requires data to be copied from user space of A to kernel space of A, then to kernel space of B, then to user space of B. As a result, it requires at least copying data three times. In practice, methods based shared memory have lowest latency and largest throughput as it requires, not surprisingly, just two times of copying data. Curious reader can check benchmark tool called ipc-benchmark [14] for more details. The section 4 will elaborate this idea and introduce lock-free message queues, based on shared memory, to minimize to invoke system calls and copy data.

2.4 Isolation of IO layers

IO component is a main source of bottlenecks of performance. Asynchronous approach partly mitigates this issue by using non-block operations. In practice, the same approach is used by separating IO layer and logic layer so that logic layer can perform non-blocking IO operations by sending request to IO layer and receiving result when the request is processed. For heavy network application, network io layer could be executed on a separate process too[4]. So, typical server design is described in the listing 1. Other IO layer could be a component that sends requests and receives responses from external services[5].

```
[network io] <----> [main logic] <----> [other io]
```

Listing 1: A typical server design

2.5 Performance optimization

One of most important principles of performance optimization is: write simple code first, optimize later. By using tools to measure performance like gperftools [15] or pert tools [16], one can analyze and determine which parts of a server are used most or causes performance issues, and then focus on optimizing those parts. That is why performance optimization is often done at the end of software development cycle.

It is also worthy to mention that major parts of applications are executed in terms of system calls and data copies. As system calls can switch context from user-space to kernel space and vice versa, they can overhead in terms of wasting cpu cycles. Zero-copy is to reduce data copy times. Based on these observations, below are some guidelines on decreasing system calls and data copies.

- Avoid malloc/dealloc: use pre-allocated memory as much as possible to avoid system calls on requesting memory.
- Using shared memory for storing and caching data to reduce data access time. Instead of fetching data when handling every request, it makes sense to store data in local cache to reduce waiting time for getting data.
- If possible, all data needed in handling a request is designed to be fit into cache level L2 or L3, it dramatically boost performance at runtime. One way of doing this is to have all data defined in plain old structure due to data locality leading to effectively cache data.

[4]One may consider using multi-queue network cards to improve further performance of io network.
[5]For example, it could be fetching data from other sources.

- If exchanging data among servers is needed, then use UDP whenever packet loss is acceptable or extremely small[6].

Exercises:

1. Read the paper 'Kqueue - A Generic and Scalable Event Notification Facility' [7].
2. Read the article 'Scalable Event Multiplexing: epoll vs. kqueue' [12].
3. (*)Read the article 'An Architecture for Well-Conditioned, Scalable Internet Services' [5].
4. (*)Read the article 'Netmap: A Novel Framework for Fast Packet I/O' [9].
5. (*)Read the article 'mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems' [10].

3 Basics of event-driven approach

This section describes usage of framework provided by libevent, then we will write a simple echo server, based on libevent. In the end of the section, performance measurement using gperftools will be introduced. Before digging into it, let review some basics of the select() mechanisms.

3.1 The select() mechanism

The select() mechanism was introduced in Unix in 1983. It implements the readiness model, which provides a mechanism for handling asynchronous IO operations in Linux/Unix operating systems. The idea is that the select() mechanism, instead of infinite loop to check readiness of data on IO operations, provides a channel between kernel and user-space applications, which will notify applications when data becomes ready.

```
/*
 * This function return a positive integer if data is ready.
 */
int check_readiness(int _fd, int _wait_sec, int _wait_usec) {
   fd_set readfd;
   struct timeval tv;
   int ret = 0;

   tv.tv_sec = _wait_sec;
   tv.tv_usec = _wait_usec;

   FD_ZERO(&readfd);
   FD_SET(_fd, &readfd);
   errno = 0;

   ret = select(_fd+1, &readfd, NULL, NULL, &tv);
   if (ret > 0) {
      if(FD_ISSET(_fd, &readfd))
         return ret;
      else
         return -1;
   } else if (ret == 0) {
      return 0;
   } else {
      if (errno != EINTR)
          close(_fd);
```

[6]For example, all servers are in the same LAN network

```
        return -1;
    }
    return 0;
}

int main(int argc, char **argv) {
    int fd;
    // create fd
    if (check_readiness(fd, 3, 0) > 0) {
        //data is ready
    } else {
        // no data available
    }
}
```

Listing 2: The select() mechanism

The listing 2 presents basic usage of the select() mechanism. To check availability of data on IO operations, a set of file descriptors, which is define by fd_set, is passed to select(). The same set carries the result when select() is finished. The issue of this approach is that kernel and user-space applications both need to scan a set of file descriptors and this results in O(n) cost and does not scale well when the size of the set is increased. Reader can check [12] for more details on this as kqueue and epoll have O(1) cost.

3.2 Using libevent

Libevent introduces a way to implement event-driven approach by providing mechanisms to execute a callback function responding to a specific event, in particular readiness of IO operations[7]. Basically, it is built on top of underlying mechanisms like kqueue, epoll and poll/select. It also provides buffer management for network IO to ease developers' life.

The main concept of event-driven mechanism in libevent is 'event'. Events could be timeouts, errors or even readiness of IO operations on file, socket, so on. An event can have a callback function associated with it. Basic idea is that whenever an event occurs, the corresponding callback function is executed. To manage events and it callback functions, an event base is introduced. More specific, an event base holds a set of events and determine which events are active. Libevent also introduce the concept of 'bufferevent'. The idea of bufferevent is to add buffer management to event so that an application verify data in buffer before responding to events. In this tutorial, we will use socket-based bufferevents. A socket-based bufferevent is based on underlying network socket to detect readiness for read and/or write operations, and uses underlying network calls to transmit and receive data.

First, the function event_base_new() help us to create an event base structure event_base. Next, to create a socket-based bufferevent, we will use bufferevent_socket_new(). An bufferevent has two data-related callbacks (read and write) and one 'error' callbacks. Basically, the read callback is called whenever any data is available to read from the underlying socket, and the write callback is called whenever the output buffer of the underlying socket is empty or below certain amount of data. The error callback is called whenever errors or events occur on the underlying socket. These callbacks can be registered by bufferevent_setcb(). Next, you need to enable events on a bufferevent with bufferevent_enable(). Finally, calling event_base_dispatch() to start monitoring all events hold by the event base.

[7] Alternative choice is libev - another event-driven IO framework

```
#include <event2/event.h>

void readcb(struct bufferevent *bev, void *ptr) {
   // read something
}

void eventcb(struct bufferevent *bev, short events, void *ptr){
    // some events occur
    return;
}

int main(int argc, char** argv) {
    struct event *ev;
    struct event_base *base = event_base_new();

    /* socket fd is created somehow */
    bev = bufferevent_socket_new(base, fd, BEV_OPT_CLOSE_ON_FREE);
    bufferevent_setcb(bev, readcb, NULL, eventcb, NULL);
    bufferevent_enable(bev, EV_READ|EV_WRITE);

    event_base_dispatch(base);
    return 0;
}
```

3.3 First simple echo server

In this section, we will build a simple echo server, which accepts connections and sends back whatever it receives from client. A listen socket has a special event when a connection ready to be accepted. To responding to that event, Libevent provides function evconnlistener_new_bind(), which takes 'accept' callback and an ip address as arguments. One can also set error callback on listen socket by calling evconnlistener_set_error_cb().

```
static void
accept_error_cb(struct evconnlistener *listener, void *ctx) {
    struct event_base *base = evconnlistener_get_base(listener);
    int err = EVUTIL_SOCKET_ERROR();
    fprintf(stderr, "Got␣an␣error␣%d␣(%s)␣on␣the␣listener.␣"
                "Shutting␣down.\n", err, evutil_socket_error_to_string(err));
    event_base_loopexit(base, NULL);
}

int main(int argc, char **argv) {
    struct event_base *base;
    struct evconnlistener *listener;
    struct sockaddr_in sin;

    base = event_base_new();
    if (!base) {
         puts("Couldn't␣open␣event␣base");
         return 1;
    }

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(0); //listen on all interfaces
    sin.sin_port = htons(6849);

    // accept_conn_cb is defined later.
```

```
    listener = evconnlistener_new_bind(base, accept_conn_cb, NULL,
            LEV_OPT_CLOSE_ON_FREE|LEV_OPT_REUSEABLE, -1,
            (struct sockaddr*)&sin, sizeof(sin));
    if (!listener) {
         perror("Couldn't create listener");
         return 1;
    }
    evconnlistener_set_error_cb(listener, accept_error_cb);

    event_base_dispatch(base);
    return 0;
}
```

Now we need to tell libevent how to respond to event when a connection ready to be accepted. When accept callback is called, it can use socket fd of connecting socket, which is passed to it as an argument, to create a socket-based bufferevent with read, write and error callbacks. The read callback of echo server is simply copy data from underlying input to to output.

```
static void echo_read_cb(struct bufferevent *bev, void *ctx) {
        /* This callback is invoked when there is data to read on bev. */
        struct evbuffer *input = bufferevent_get_input(bev);
        struct evbuffer *output = bufferevent_get_output(bev);

        /* Copy all the data from the input buffer to the output buffer. */
        evbuffer_add_buffer(output, input);
}

static void echo_event_cb(struct bufferevent *bev, short events, void *ctx) {
        if (events & BEV_EVENT_ERROR)
                perror("Error from bufferevent");
        if (events & (BEV_EVENT_EOF | BEV_EVENT_ERROR)) {
                bufferevent_free(bev);
        }
}

static void accept_conn_cb(struct evconnlistener *listener,
    evutil_socket_t fd, struct sockaddr *address, int socklen, void *ctx) {
    /* We got a new connection! Set up a bufferevent for it. */
    struct event_base *base = evconnlistener_get_base(listener);
    struct bufferevent *bev = bufferevent_socket_new(
                base, fd, BEV_OPT_CLOSE_ON_FREE);

    bufferevent_setcb(bev, echo_read_cb, NULL, echo_event_cb, NULL);
    bufferevent_enable(bev, EV_READ|EV_WRITE);
}
```

That is how to implement the echo server. Reader take a look at [17] for more details and all source code.

```
Practical section: running echo server version 1.

On terminal 1
# ./echosvr_v1

On terminal 2
# ./clientsvr_v1
```

3.4 Timeout events

In libevent, all events can be timeout if time is provided. Based on this, we can create timeout events, which will trigger at certain time in the future, for example after a certain amount of time from the current moment. To create timeout events, simply pass time to add_new() function along with an event, which is associate with non-exist file descriptor (i.e. -1) and the flag EV_TIMEOUT as described in the listing 3. The flag EV_PERSIST causes timeouts to repeat.

```
#define TIMEOUT_SEC 5

void timeout_cb(evutil_socket_t fd, short what, void *ctx) {
   struct stats *stats = (struct stats*)ctx;
   time_t now;

   time(&now);
   stats->last_time = now;

   printf("statictis:␣\n");
   printf("␣␣␣␣␣msg_cnt:␣%u\n", stats->msg_cnt);
   printf("␣␣␣␣conn_cnt:␣%u\n", stats->conn_cnt);
   printf("␣␣start_time:␣%lu\n", stats->start_time);
   printf("␣␣␣last_time:␣%lu\n", stats->last_time);
   return;
}

int main(int argc, char **argv) {
  struct event *ev = NULL;
  struct timeval timeout_interval = {TIMEOUT_SEC,0};

  .
  .
  .

  ev = event_new(base, -1, EV_TIMEOUT|EV_PERSIST, timeout_cb, stats);
  if (!ev) {
     perror("Couldn't␣create␣timeout␣event");
     return 1;
  }
  event_add(ev, &timeout_interval);
  event_base_dispatch(base);
  return 0;
}
```

Listing 3: The select() mechanism

```
Practical section: running echo server version 2 with timeout to print statistics.

On terminal 1
# ./echosvr_v2

On terminal 2
# ./clientsvr_v2
```

3.5 Performance measurement

In this section, we introduce Google performance tool, which allows us to do cpu profiling, heap profile and memory check of an applications.

```
apt-get install google-perftools libgoogle-perftools4 libgoogle-perftools-dev
```

Note: gperftools seems not working on Ubuntu 16.04. Reader are encourage to explore it[8].

Exercises:

1. Read chapter R1,R2, R3, R4 and R6 from book 'Programming with Libevent' [18].
2. Implement echo server using select/poll mechanism.
3. Compile echo server in this chapter with the following function echo_read_cb(), then using gperftools to measure its performance. Analyze output from gperftools and explain it.

```
static void echo_read_cb(struct bufferevent *bev, void *ctx) {
        char buffer[4*1024] = {0}.
        /* This callback is invoked when there is data to read on bev. */
        struct evbuffer *input = bufferevent_get_input(bev);
        struct evbuffer *output = bufferevent_get_output(bev);

        /* Copy all the data from the input buffer to the output buffer. */
        evbuffer_add_buffer(output, input);
}
```

4. (*)Implement udp-based echo server using libevent. Hint: using concept 'event' in Libevent, read chapter R4 of the book 'Programming with Libevent'.

4 Multi-processes with message queues

In this section, we will build an effective way of communications between processes, based on data structure called lock-free message queue. Basically, lock-free message queue is one-consumer and one produce queue and use notification mechanism from select/poll as a way to notify consumer whenever producer puts data on the queue. Messages are stored in a circular buffer in a way that avoids using locks to access messages.

4.1 Basics of shared memory

The listing 4 describes echo server version 4 with storing statistics in a shared memory and a view tool to view statistics from the shared memory. To create a shared memory, we use shmget() by providing a key (an integer), size of the shared memory and flags. The flag defines permissions on the shared memory and other things like IPC_CREAT which instructs shmget() to create a new shared memory if it does not exist. A new created shared memory can be access by shmat() which return a pointer to the shared memory.

```
#define SHM_DEFAULT_OPEN_FLAG 0666
char* create_shm(key_t key, int size, int flags) {
  int shmid;
  char *data;
```

[8]This section will be updated.

```
  /* create the shared memory segment */
  shmid = shmget(key, size, flags);
  if (shmid < 0) {
     perror("shmget");
     return 0;
  }

  data = (char*)shmat(shmid, NULL, 0);
  if (data == (char*)-1) {
    return 0;
  }

  return data;
}


int
main(int argc, char **argv) {
  struct stats *stats = NULL;
  time_t now;

  .
  .
  .

  stats = (struct stats*)create_shm(STATS_SHM_KEY,
      sizeof(struct stats),
      SHM_DEFAULT_OPEN_FLAG | IPC_CREAT | IPC_EXCL);
  if (!stats) {
    printf("Couldn't create statistics");
    return 1;
  }

  memset(stats,0,sizeof(struct stats));
  time(&now);
  stats->start_time = now;
  listener = evconnlistener_new_bind(base, accept_conn_cb, stats,
      LEV_OPT_CLOSE_ON_FREE|LEV_OPT_REUSEABLE, -1,
      (struct sockaddr*)&sin, sizeof(sin));
  if (!listener) {
     perror("Couldn't create listener");
     return 1;
  }

  .
  .
  .

  return 0;
}
```

Listing 4: Shared Memory

It worthy to note that the point to the shared memory is passed as the last argument in functions evconnlistener_new_bind() and bufferevent_setcb(). In this way, when a new connection is accepted or a new incoming message from a connection, their counters are increase by one accordingly as described in the listing 5.

```
static void
echo_read_cb(struct bufferevent *bev, void *ctx) {
   static char data[BUFFER_SIZE];
   size_t recv_input_len = 0;
   struct evbuffer *input = bufferevent_get_input(bev);
   struct stats *stats = (struct stats*)ctx;

   recv_input_len = evbuffer_get_length(input);
   if (recv_input_len <= 0) return;

   //printf("Receive message from client, len=%lu\n",recv_input_len);
   stats->msg_cnt++;

   /* Copy all the data from the input buffer to the output buffer. */
   evbuffer_remove(input, data, recv_input_len);
   bufferevent_write(bev,data,recv_input_len);
   return;
}

static void
accept_conn_cb(struct evconnlistener *listener,
    evutil_socket_t fd, struct sockaddr *address, int socklen,
    void *ctx) {
   struct stats *stats = (struct stats*)ctx;
   struct event_base *base = evconnlistener_get_base(listener);
   struct bufferevent *bev = bufferevent_socket_new(
           base, fd, BEV_OPT_CLOSE_ON_FREE);

   /* We got a new connection! Set up a bufferevent for it. */
   printf("Setup␣new␣connection,␣fd=%d\n",fd);
   bufferevent_setcb(bev, echo_read_cb, NULL, echo_event_cb, ctx);
   bufferevent_enable(bev, EV_READ|EV_WRITE);
   stats->conn_cnt++;
}

int
main(int argc, char **argv) {
  struct stats *stats = NULL;

  .
  .
  .

  listener = evconnlistener_new_bind(base, accept_conn_cb, stats,
      LEV_OPT_CLOSE_ON_FREE|LEV_OPT_REUSEABLE, -1,
      (struct sockaddr*)&sin, sizeof(sin));
  if (!listener) {
     perror("Couldn't␣create␣listener");
     return 1;
  }

  .
  .
  .

  return 0;
}
```

Listing 5: Passing argument in libevent callbacks

```
Practical section: running echo server version 4 with storing statistics in shared
memory and a tool to view statistics from shared memory.

On terminal 1
# ./echosvr_v4

On terminal 2
# ./clientsvr_v4 4680 64 1 600

On terminal 3
#./viewtstat
```

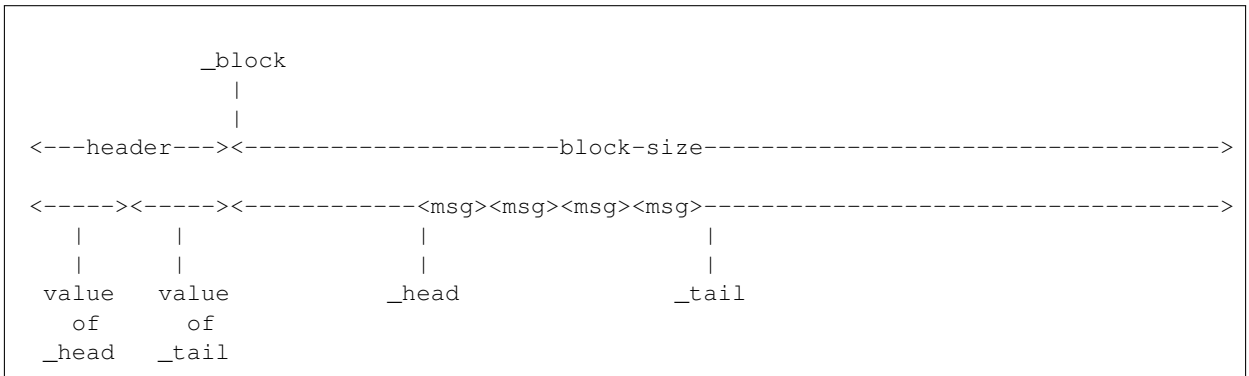
4.2 Data structure

Lock-free message queue consists of three components: notification mechanism, circular buffer and enqueue/dequeue operations. Its data structure are described as the following:

- _shm: a pointer to shared memory which is used to store messages. The circular buffer is implemented on this memory. It contains a pointer _addr to where shared memory starts.
- _fd: a fifo file whose notification mechanism is used to tell us when data is available by writing a byte into it. More specifically, whenever a message is put into the circular buffer, a byte is written into the fifo file so that message is available in the queue. When removing a message from the queue, a byte must be taken out of the fifo file.
- _wait_sec and _wait_usec: timeout parameters for enqueue/dequeue operations.
- _head: a pointer to the head of message queue.
- _tail: a pointer to the tail of message queue.
- _block: a pointer to first byte of memory region pointed by _shm.
- _block_size: the size of memory region pointed by _shm.

```
struct shm_data
{
    key_t key;
    size_t size;
    int id;
    char* addr;
};

struct shm_mq
{
   shm_data*    _shm;
   uint32_t     _fd;
   uint32_t     _wait_sec;
   uint32_t     _wait_usec;
   uint32_t*    _head;
   uint32_t*    _tail;
   char*        _block;
   uint32_t     _block_size;
}__attribute__((packed));
```



Listing 6: Layout of message queue

The layout of message queue is described in the listing 6. As in the listing, _head points to message which will be first dequeue while _tail points to the end of the last message which is put into the queue. When a message pointed by _head is popped out, _head moves to the next message. Likewise, when a message is put into the queue, it is copied at address pointed by at _tail and _tail moves forward to the end of the message. That is exactly what circular buffer does. It also note that the message queue implements first-in first-out policy.

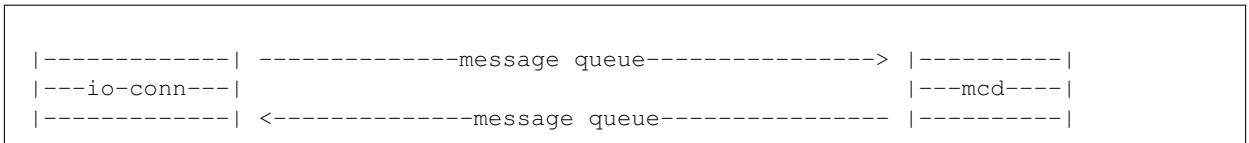
4.3 Format of message

A message, which is enqueued into the queue, is a self-describe data structure, consisting of its length, identification number, and data. The layout of a message is depicted in the listing 7. The field 'length' is total length of a message, including legnth, identification number, and data. It is used to determine how message is read out. The field 'id' provides a way to identify sources of message across processes. For example, 'id' can be socket id of a connection.

Listing 7: Layout of message queue

4.4 Isolation of IO layers

In this section, we implement the idea of isolating io layers. The echo server which is implemented in the previous section is split into two independent processes: one process handles network connections (name it 'io_conn' - io connector) and other handles logic of server (name it 'md' - main deamon). The listing 8 depicts design of new echo server. Note that there are two message queues in this design: one carries messages from io_conn to md and other - from md to io_conn.



Listing 8: Layout of echo server

The io_conn process is basically same as echo server, except that it uses message queue exchange messages with the md process. To do this, it first initializes two message queues: one to send messages to md and one to receive messages from md. The function initialize_mq in the listing 9 just does that.

```
int
initialize_mq(struct global_data *g_data) {
  int ret = 0;

  g_data->mq_io_2_md = (shm_mq_t *)
         malloc(sizeof(*g_data->mq_io_2_md));
  if (g_data->mq_io_2_md == 0) {
    return -1;
  }
  ret = shmmq_init(g_data->mq_io_2_md,
    "/tmp/echosvr_io_2_mq.fifo", 0, 0,
    IO_2_MD_KEY, MQ_SIZE);
  if (ret < 0) {
    return -2;
  }

  g_data->mq_md_2_io = (shm_mq_t *)
         malloc(sizeof(*g_data->mq_md_2_io));
  if (g_data->mq_md_2_io == 0) {
    return -1;
  }
  ret = shmmq_init(g_data->mq_md_2_io,
    "/tmp/echosvr_mq_2_io.fifo", 0, 0,
    MD_2_IO_KEY, MQ_SIZE);
  if (ret < 0) {
    return -3;
  }

  return ret;
}
```

Listing 9: Message queue initialization

The new function echo_read_cb simply forwards message to md by invoking the enqueue operation on message queue as in the listing 10.

```
static void
echo_read_cb(struct bufferevent *bev, void *ctx) {
   static char data[BUFFER_SIZE];
   size_t recv_input_len = 0;
   struct evbuffer *input = bufferevent_get_input(bev);
   struct connect_ctx *conn = (struct connect_ctx*)ctx;

   recv_input_len = evbuffer_get_length(input);
   if (recv_input_len <= 0) return;

   /* Copy all the data from the input buffer to the output buffer. */
   evbuffer_remove(input, data, recv_input_len);

   shmmq_enqueue(conn->g_data->mq_io_2_md,0,data,recv_input_len,conn->flowid);

   return;
}
```

Listing 10: The callback function.

Finally, io_conn needs to handle messages from md by listening on file descriptor of the message queue which transfers message from md to it. More specific, the callback function handle_msg_from_md reads message from the queue and forwards it to client.

```
void
handle_msg_from_md(int fd, short int event,void* ctx) {
   static char buf[BUFFER_SIZE];
   struct global_data *g_data = (struct global_data*)ctx;
   struct connect_ctx *conn = NULL;
   uint32_t len = 0;
   uint32_t flowid = 0;
   uint32_t cnt = 0;
   int ret = 0;

   while(1){
     len = 0;
     flowid = 0;
     cnt++;
     if ( cnt >= 100) break;

     ret = shmmq_dequeue(g_data->mq_md_2_io, buf, BUFFER_SIZE, &len, &flowid);
     if ( (len == 0 && ret == 0) || (ret < 0) )
        return;

     buf[len] = 0;
     conn = (struct connect_ctx*)flowset_getobj(g_data->flowset, flowid);
     if (!conn || !conn->bev) continue;
     bufferevent_write(conn->bev,buf,len);
   }

   return;
}

int
main(int argc, char **argv) {
  struct event_base *base;
  struct event *ev = NULL;
  .
  .
  .
  ev = event_new(base,ctx->mq_io_2_md->_fd,
       EV_TIMEOUT|EV_READ|EV_PERSIST, handle_msg_from_io, ctx);
  event_add(ev, NULL);
  .
  .
  .
  event_base_dispatch(base);
  return 0;
}
```

Listing 11: Handling message from md.

To get md implementation, similar modification is applied. This is left as an exercise.

4.5 Scalability with multi-processes

Remember that the server design, which is shown in the listing 1, is simple as each component has only one process. In practice, each component can be scaled out by adding more processes to it. For example in heavy load application such as real time video streaming service, one may need several io_conn and md forwards jobs to its 'sub-md' as depicted in the listing 12. Also note that message queues can be used to build any pipeline server architecture.

```
|-io-conn-|                              |-sub-md-|
                                         |-sub-md-|
|-io-conn-| <----->  |---md---| <----->  |-sub-md-|
                                         |-sub-md-|
|-io-conn-|                              |-sub-md-|
```

Listing 12: Multi-processes architecture in practice.

Exercises:

1. Figure out how the shared memory pointed by _shm is initialized by using function shm-data_init() (this function defined in shm.c).
2. Figure out the format of messages which is put into the queue).
3. Figure out how enqueue and dequeue operations works (see source code in mq.c).
4. (*) Modify data structure of message to count how many messages go through the queue. Write a small program to read that count from shared memory of the queue.
5. Figure out how the view tool works (see source code in viewtstats.c).
6. Figure out how md is implemented.
7. Write a client which send 10.000 messages to echo server and measure how much time to get 10.000 responses from the server. Use this client to measure time for the echo server in the previous section.
8. (*) Write a client that open 10.000 connections to echo server and for each connection send a message every second. Use simple client to measure response time of echo server in such condition.
9. (*) Implement a web socket server with two independent processes as the echo server.

5 Caching techniques

Caching is extremely important data structure in computer science. Its basics principle is to store data once and use many times later. In context of server implementation, it boosts performance by reducing cpu cycles for getting external data. In this section, we look at two important caches: memory pool and multi-hash.

5.1 Memory pool

Memory pool is designed to store fix-sized data and is initialized with pre-allocated memory. In that way, every time an application needs memory, it can get from memory pool and this avoids using malloc/dealloc, therefore increasing performance of the application. The heart of memory pool lies at its self-contain data structure [ref]. Basic idea is that it divides memory in all fix-sized memory chunks and keep track of free memory chunks. When allocating memory chunk, it returns a chunk available from the list of free chunks. When deallocating a memory chunk, it simply put that chunk into the list of free chunks. By nature of this algorithm , memory pool can not be used to share data among processes. This is solved be multi-hash in the next subsection.

```
typedef struct tag_mem_chunk
{
   int mc_free_chunks;
   struct tag_mem_chunk *mc_next;
} mem_chunk;
typedef mem_chunk* mem_chunk_t;

struct mempool {
   char *mp_startptr;        /* start pointer */
   mem_chunk_t mp_freeptr;   /* pointer to the start memory chunk */
   int mp_free_chunks;       /* number of total free chunks */
   int mp_total_chunks;      /* number of total free chunks */
   int mp_chunk_size;        /* chunk size in bytes */
   int mp_type;
};
```

Listing 13: data structure of memory pool

5.2 Multi-hash

Multi-hash is used to store fix-sized data chunks, which can be shared among processes. The principle for sharing data among processes is to avoid using locks to boost performance. Multi-hash accepts possible concurrency issues if their occurrence can be tolerate. 'tolerate' means that if it happens at very low probability such as one per millions. In that case it just ignores errors and let end users handle them.

Muti-hash is implemented by dividing the whole memory into memory segments and each memory segment has its own hashing algorithm. For example, when looking up a key, each memory segment uses remainder of key value divided by a primary numbers as a index to look into the memory segment. This operation continues until it succeeds to find a key or iterates all memory segments with success. Note that each memory segment can use different primary number.

```
      < -             length of one segment                    ->

  |   <-chunk-><-chunk-><-chunk-><-chunk->    <-chunk-><-chunk->
  |   <-chunk-><-chunk-><-chunk-><-chunk->    <-chunk-><-chunk->
  n
times
  |   <-chunk-><-chunk-><-chunk-><-chunk->    <-chunk-><-chunk->
  |   <-chunk-><-chunk-><-chunk-><-chunk->    <-chunk-><-chunk->
```

Listing 14: data structure of multi-hash

Exercises:

1. Figure out how memory pool works (see source code mempool.c).
2. Figure out how multi-hash works (see source code cache.c).
3. Write two programs that share the same multi-hash.

References

[1] D. Kegel, *The c10k issue*. http://www.kegel.com/c10k.html, 2014.

[2] R. D. Graham, "C10m: Defending the internet at scale," 2013.

[3] R. Risk, "Scaling to millions of simultaneous connections," 2012.

[4] G. Banga, J. C. Mogul, and P. Druschel, "A scalable and explicit event delivery mechanism for unix," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, (Berkeley, CA, USA), pp. 19–19, USENIX Association, 1999.

[5] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 230–243, Oct. 2001.

[6] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, (New York, NY, USA), pp. 230–243, ACM, 2001.

[7] J. Lemon, "Kqueue - a generic and scalable event notification facility," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 141–153, USENIX Association, 2001.

[8] U. Drepper, "The need for asynchronous, zero-copy network i/o," *Ottawa Linux Symposium*, pp. 247–260, 01 2006.

[9] L. Rizzo, "Netmap: A novel framework for fast packet i/o," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 2012.

[10] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: A highly scalable user-level tcp stack for multicore systems," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, (Berkeley, CA, USA), pp. 489–502, USENIX Association, 2014.

[11] M. D. Systems, "Scaling to millions of simultaneous connections," 2015.

[12] S. Han, "Scalable event multiplexing: epoll vs. kqueue," 12 2012.

[13] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 2003.

[14] NetOS, *Latency benchmarks of Unix IPC mechanisms.* https://www.cl.cam.ac.uk/research/srg/netos/projects/ipc-bench/, 2012.

[15] Google, *Google Performance Tools.* https://github.com/gperftools/gperftools, 2005.

[16] Linux, *Linux Performance Tools.* https://perf.wiki.kernel.org, 2009.

[17] J. Dinh, *Hands-on Tutorial on C/C++ Server.* https://github.com/jackiedinh8/svrtut, 2017.

[18] N. Mathewson, "Programming with libevent," 2010.