1. It-Almost-Works Load Balancer. Let us suppose that for whatever reason you need to "distribute" TCP/IP requests coming on a port to several processes listening to different ports on the same host machine.

This is the reason why `IAWLB` is born.

`IAWLB` accepts a request and pushes the socket given by *accept*( ) in a FIFO list; workers pop from the list as soon as they can. Each worker is a thread forwarding to a specific destination address, according to what you've given in the command line.

The list is handled by **SyncedList**, a class using locks to allow several threads to pop a socket concurrently (a single thread is in charge of pushing data on the list).

2. The layout of `IAWLB` is very simple.

```
⟨ Include files 20 ⟩
⟨ Globals and more 10 ⟩
⟨ Functions 11 ⟩
⟨ Thread code 13 ⟩
⟨ Main 3 ⟩
```

3. ⟨ Main 3 ⟩ ≡

```
int main(int argc, char **argv)
{
  size_t i;
  int backlog = 1;     /* backlog for listen( ) */
  int thr_num = 0;     /* number of created threads */
  thr_info thr[MAX_NUM_OF_DEST_PORT];
    /* we have a limit to the number of destination addresses we can forward to */
  ⟨ Check arguments 4 ⟩
  ⟨ Server listening 5 ⟩
  ⟨ Create threads 7 ⟩
  ⟨ Start server listening 9 ⟩
  ⟨ Receiving loop 19 ⟩
  return 0;
}
```

This code is used in section 2.

4. You must call the program with at least two arguments: the first is the port `IAWLB` is accepting requests from, the next arguments say where to forward the requests. Each destination argument can be a port, or an IP address followed by a colon and then a port.

⟨ Check arguments 4 ⟩ ≡

```
if (argc < 3) {
  std::cerr << "LISTEN_ON␣[IP:]PORT␣[[IP:]PORT]*\n";
  return EXIT_FAILURE;
}
```

This code is used in section 3.

5. The first argument is the local port `IAWLB` is bound to.

⟨ Server listening 5 ⟩ ≡

```
struct addrinfo hints = {0};
struct addrinfo *sinfo = NULL;
struct addrinfo *sp = NULL;

hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

int rv = getaddrinfo(NULL, argv[1], &hints, &sinfo);

if (rv ≠ 0) {
   std::cerr ≪ "getaddrinfo␣err:␣" ≪ gai_strerror(rv) ≪ "\n";
   return EXIT_FAILURE;
}
```

See also section 6.

This code is cited in section 16.

This code is used in section 3.

6. The function *getaddrinfo*( ) gives a linked list of *addrinfo* structures; we traverse this list and stop to the first match we can bind to. If none does, it's an error and we exit. I think it could work also if we just try to pick the first one, avoiding the loop.

⟨ Server listening 5 ⟩ +≡

```
int serversock;     /* we shall listen with this */

for (sp = sinfo; sp ≠ NULL; sp = sp→ai_next) {
   serversock = socket(sp→ai_family, sp→ai_socktype, sp→ai_protocol);
   if (serversock ≡ −1) {
      perror("server␣socket");
      continue;
   }
   int yes = 1;

   if (setsockopt(serversock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof (yes)) ≡ −1) {
      perror("setsockopt");
      return EXIT_FAILURE;
   }
   if (bind(serversock, sp→ai_addr, sp→ai_addrlen) ≡ −1) {
      close(serversock);
      perror("bind");
      continue;
   }
   break;
}
if (sp ≡ NULL) {
   std::cerr ≪ "cannot␣listen␣on␣" ≪ argv[1] ≪ "\n";
   return EXIT_FAILURE;
}
freeaddrinfo(sinfo);
```

7. The remaining arguments are the addresses `IAWLB` must forward to. For each of them we create a "dispatcher", which is a POSIX thread. Each address can be made of an actual address part, optional, and a port part; *get_addr*( ) deals with the optional address part and *get_port*( ) with the port (implemented in ⟨ Functions 11 ⟩). These are informations each thread is interested to, hence they are put in a structure which the *new_dispatcher*( ) function will pass as last argument to *pthread_create*( ).

⟨ Create threads 7 ⟩ ≡

```
for (i = 2; i < argc ∧ (i − 2) < (sizeof (thr)/sizeof (thr[0])); ++i) {
    thr[thr_num].addr = get_addr(argv[i]);
    thr[thr_num].port = get_port(argv[i]);
    thr[thr_num].idx = i − 2;
    int r = new_dispatcher(&thr[thr_num]);
    if (r < 0) {
        std::cerr ≪ "cannot␣create␣dispatcher␣addr␣" ≪ thr[thr_num].addr ≪ ":" ≪
            thr[thr_num].port ≪ "\n";
        return EXIT_FAILURE;
    }
    ++thr_num;
}
```

See also section 8.

This code is used in section 3.

8. The number of thread, *thr_num*, determines the backlog argument to the *listen*( ) function.

⟨ Create threads 7 ⟩ +≡

```
backlog += thr_num;
```

9. Now that we know how many threads we have, we can listen using the right value for *backlog*.

⟨ Start server listening 9 ⟩ ≡

```
if (listen(serversock, backlog) ≡ −1) {
    perror("listen");
    return EXIT_FAILURE;
}
```

This code is used in section 3.

10. Workers. Each thread is a worker which "knows" something about itself, kept into a *thr_info_s* struct.

⟨ Globals and more 10 ⟩ ≡
```
typedef struct thr_info_s {
  pthread_t thr;
  size_t idx;
  std::string addr;
  std::string port;
  int status;
  std::string msg;
} thr_info;
```
See also sections 12, 15, 24, and 29.

This code is used in section 2.

11. Each thread is created by *new_dispatcher*( ), which gets a pointer to **thr_info** as argument and passes it to *pthread_create*( ) so that the thread, *process_request*, can use it.

⟨ Functions 11 ⟩ ≡
```
int new_dispatcher(thr_info *pt)
{
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  int r = pthread_create(&pt→thr, &attr, process_request, pt);
  pthread_attr_destroy(&attr);
  return r ≠ 0 ? −1 : 0;
}
```
See also sections 25, 26, 27, and 28.

This code is cited in section 7.

This code is used in section 2.

12. We need to make the *process_request*( ) function visible ahead to the *new_dispatcher*( ).

⟨ Globals and more 10 ⟩ +≡
```
static void *process_request(void *arg);
```

13. The *process_request*( ) function does the real work. It enters an infinite loop, from which it shouldn't exit, if everything's ok.

⟨ Thread code 13 ⟩ ≡

```
static void *process_request(void *arg)
{
  thr_info *ti = static_cast⟨thr_info *⟩(arg);
  int sockcaller;   /* where we store the popped socket */
  int sockfd;   /* to communicate with the destination */
  size_t id = ti→idx;
  FOREVER {
    ⟨ Wait for a pushed socket 14 ⟩
    Log("%zu: popped sock %d for (%s, %s)\n", id, sockcaller, ti→addr.c_str( ), ti→port.c_str( ));
    ⟨ Connect to destination 16 ⟩
    Log("%zu: resolved address; sock %d connected to (%s, %s)\n", id, sockfd, ti→addr.c_str( ),
        ti→port.c_str( ));
    ⟨ Pipe the data (bidirectionally) 17 ⟩
    struct timespec ts = {0, 50000000};   /* wait for 50 ms */
    if (nanosleep(&ts, NULL) < 0) {   /* We break on signal or other event. */
      break;
    }
  }
  return arg;
}
```

This code is used in section 2.

14. Our thread tries to pop a socket from *accepted_sock*, which is a synchronized FIFO list.

⟨ Wait for a pushed socket 14 ⟩ ≡

```
while (¬accepted_sock.pop(&sockcaller)) ;
```

This code is used in section 13.

15. The main loop of the server accepts incoming connection and pushes the accepted socket on the list *accepted_sock* so that each thread can pop from it.

⟨ Globals and more 10 ⟩ +≡

```
SyncedList⟨int⟩ accepted_sock;
```

16. Once the thread succeeds popping a socket from the synchronized FIFO list, it must connect to its destination server. The code is very similar to what we've already done in ⟨ Server listening 5 ⟩, except that here we must decide what it happens to the thread once something goes wrong. By design, the thread exits. That means, on a long run all workers could be gone, the list won't be emptied and clients should receive a refused connection error.

⟨ Connect to destination 16 ⟩ ≡

```
struct addrinfo hints = {0};
struct addrinfo *servinfo;
struct addrinfo *pa;

hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

int status = getaddrinfo(ti->addr.c_str(), ti->port.c_str(), &hints, &servinfo);

if (status != 0) {
  ti->status = status;
  ti->msg = gai_strerror(status);
  Log("%zu: error: %s (status %d)\n", id, ti->msg.c_str(), ti->status);
  pthread_exit(arg);   /* the thread that can't "resolve" its destination, must die. */
}
for (pa = servinfo; pa != NULL; pa = pa->ai_next) {
  sockfd = socket(pa->ai_family, pa->ai_socktype, pa->ai_protocol);
  if (sockfd == -1) {
    continue;
  }
  if (connect(sockfd, pa->ai_addr, pa->ai_addrlen) == -1) {
    Log("%zu: failed connect for socket %d\n", id, sockfd);
    close(sockfd);
    continue;
  }
  break;
}
if (pa == NULL) {
  ti->status = -1;
  ti->msg = "failed to connect";
  freeaddrinfo(servinfo);
  Log("%zu: error: %s (status %d)", id, ti->msg.c_str(), ti->status);
  pthread_exit(arg);   /* failed? then die */
}
freeaddrinfo(servinfo);
```

This code is used in section 13.

17. If the thread was able to connect to its destination, then it has both the ends of the connection: the accepted socket (from the client calling) *sockcalled*, and the socket for the destination IAWLB's worker acts as a client for: from the point of view of the destination, the client is IAWLB.

⟨ Pipe the data (bidirectionally) 17 ⟩ ≡

```
char readbuf[1024 * MAX_READ_BUFFER] = "";
ssize_t readstat = recv(sockcaller, readbuf, sizeof (readbuf), 0);
if (readstat > 0) {
  Log("%zu: read %zd from sock %d\n", id, readstat, sockcaller);
  ssize_t targetlen = 0;
  while (targetlen < readstat) {
    ssize_t sendstat = send(sockfd, readbuf, readstat, MSG_NOSIGNAL);
      /* EPIPE is ok, but no SIGPIPE, please */
    targetlen += sendstat;
  }
  Log("%zu: sent %zd to sock %d\n", id, targetlen, sockfd);
  struct pollfd pfd[1] = {{sockfd, POLLIN, 0}};   /* Let's read the answer from destination now. */
  int pret = poll(pfd, 1, ANS_TIMEOUT * 1000);
  if (pret ≡ 0) {
    Log("%zu: timed out on sock %d\n", id, sockfd);
  }
  else if (pret > 0) {
    ssize_t anstat = recv(sockfd, readbuf, sizeof (readbuf), 0);
    if (anstat > 0) {
      Log("%zu: received answer from sock %d (%zd bytes)\n", id, sockfd, anstat);
      pfd[0].fd = sockcaller;
      pfd[0].events = POLLOUT;
      pfd[0].revents = 0;
      int uret = poll(pfd, 1, 5);   /* 5 ms; this isn't a magic number, but just a random small one */
      if (uret > 0) {
        targetlen = 0;
        while (targetlen < anstat) {
          ssize_t sendstat = send(sockcaller, readbuf, anstat, MSG_NOSIGNAL);
            /* send the answer back to the caller */
          targetlen += sendstat;
        }
        Log("%zu: sent back answer to sock %d (bytes %zd)\n", id, sockcaller, targetlen);
      }
    }
    else {
      Log("%zu: error receiving from sock %d (%d)\n", id, sockfd, anstat);
    }
  }
}
```

See also section 18.

This code is used in section 13.

18. The thread is in charge of closing the socket accepted for the incoming connection (the one popped from the FIFO list) and the socket it opened to communicate with its destination.

⟨ Pipe the data (bidirectionally) 17 ⟩ +≡
  close(*sockfd*);
  close(*sockcaller*);

19. The server loop. Once IAWLB is listening to incoming connections, it needs to loop forever pushing the accepted sockets on the synchronized FIFO list *accepted_sock* again and again. This is not intended to be stopped, though of course adding signals handling before entering this loop would be a good thing. Likely we want also to end smoothly, letting the threads to finish their work before killing them all abruptly. These desiderable things aren't here because the purpose of this server isn't to replace a real production-ready load balancer.

```
⟨ Receiving loop 19 ⟩ ≡
  FOREVER {
    struct sockaddr_storage caller_addr = {0};
    socklen_t addr_size = sizeof (caller_addr);
    int ax_sock = accept(serversock, static_cast⟨struct sockaddr *⟩(&caller_addr), &addr_size);
    if (ax_sock ≡ −1) {
      perror("accept");
      continue;
    }
    char addr_buf[INET6_ADDRSTRLEN];   /* we can deal with IPv6... likely... get_in_addr( ) tries to deal
        with part of this when converting from binary to text form. */
    inet_ntop(caller_addr.ss_family,
        get_in_addr(static_cast⟨struct sockaddr *⟩(&caller_addr)),
        addr_buf,
        sizeof (addr_buf));
    Log("request␣from␣%s\n", addr_buf);
    accepted_sock.enqueue(ax_sock);
  }
```

This code is cited in section 38.

This code is used in section 3.

20. We need some include files; let's begin with C++ "common" stuffs.

```
⟨ Include files 20 ⟩ ≡
#include <iostream>
#include <string>
#include <cstdio>
#include <cstdarg>
#include <cstdlib>
#include <cerrno>
```

See also sections 21, 22, and 23.

This code is used in section 2.

21. We use POSIX thread library. This code isn't very much portable, after all!

```
⟨ Include files 20 ⟩ +≡
#include <pthread.h>
```

22. We need also several include files to deal with networking things.

⟨ Include files 20 ⟩ +≡

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>   /* for inet_ntop() */
#include <netdb.h>   /* getaddrinfo() etc. */
#include <unistd.h>   /* for close() */
#include <poll.h>   /* for poll() */
```

23. And at last we need our **SyncedList**, which is a template class implemented in the include file ⟨ `SyncedList.hpp` 30 ⟩ (because it's a template).

⟨ Include files 20 ⟩ +≡

```
#include "SyncedList.hpp"
```

24. Other functions and defines. We need other functions we've used in the code, and also few defines we have here and there.

⟨ Globals and more 10 ⟩ +≡
```
#define MAX_READ_BUFFER 256 /* in kBytes */
#define ANS_TIMEOUT 31 /* in seconds */
#define MAX_NUM_OF_DEST_PORT 32
#define FOREVER for ( ; ; )
    /* sugar, mainly because I don't like how CWEB formats this expression, and so keeping it in a macro
       means to see it only here; I will figure out how to typeset it better later (or maybe never) */
```

25. The destination address can be made by two parts: the first, optional, is the address. If the address is missing, it is assumed to be `localhost`.

⟨ Functions 11 ⟩ +≡
```
std::string get_addr(const char *s)
{
  std::string addr(s);
  size_t p = addr.find_first_of(':');
  if (p ≡ std::string::npos ∨ p ≡ 0) {
    return "localhost";
  }
  return addr.substr(0, p − 1);
}
```

26. The second part of a destination address is the port, and it's mandatory.

⟨ Functions 11 ⟩ +≡
```
std::string get_port(const char *s)
{
  std::string port(s);
  size_t p = port.find_first_of(':');
  if (p ≡ std::string::npos) {
    return port;
  }
  return port.substr(p + 1);
}
```

27. Likely we want to deal with IPv6 addresses, too; *get_in_addr*( ) helps in doing so.

⟨ Functions 11 ⟩ +≡
```
void *get_in_addr(struct sockaddr *sa)
{
  if (sa→sa_family ≡ AF_INET) {
    return &((static_cast⟨struct sockaddr_in *⟩(sa))→sin_addr);
  }
  return &((static_cast⟨struct sockaddr_in6 *⟩(sa))→sin6_addr);
}
```

28. Logging. All programs need some logging. This is very basic and it should work fine in this case where several threads try to output something. Of course it slows done everything, but it shouldn't be an issue as to have interspersed pieces of outputs from different threads.

⟨ Functions 11 ⟩ +≡

```
void Log(const char *s, ...)
{
  pthread_mutex_lock(&logmu);
  va_list ap;
  va_start(ap, s);
  (void) vprintf(s, ap);
  va_end(ap);
  fflush(stdout);    /* we must flush or the mutex is unuseful */
  pthread_mutex_unlock(&logmu);
}
```

29. The logging uses this mutex.

⟨ Globals and more 10 ⟩ +≡

```
pthread_mutex_t logmu = PTHREAD_MUTEX_INITIALIZER;
```

30. The synchronized FIFO list. This is a sort of wrapper for the STL *std* :: *list* template class, but with something added to access it concurrently and with only a way to put and get data. In fact elements are added "back" with the *enqueue*( ) method, while they are popped from the "head" (or "front") with the *pop*( ) method (which I should have called dequeue).

⟨ SyncedList.hpp 30 ⟩ ≡
```
⟨ SyncedList includes 36 ⟩
template⟨typename T⟩
class SyncedList {
private:
  pthread_mutex_t m_;
  pthread_cond_t c_;
  std::list⟨T⟩ list_;
public:
  ⟨ Constructor and destructor 31 ⟩
  ⟨ Pop and enqueue 32 ⟩
  ⟨ Concurrency management 34 ⟩
};
```
This code is cited in section 23.

31. We have to initialize the mutex and the condition, and destroy them when finished.

⟨ Constructor and destructor 31 ⟩ ≡
```
SyncedList()
{
  (void) pthread_mutex_init(&m_, NULL);
  (void) pthread_cond_init(&c_, NULL);
}
~SyncedList()
{
  (void) pthread_mutex_destroy(&m_);
  (void) pthread_cond_destroy(&c_);
}
```
This code is used in section 30.

32. The *pop*( ) method locks the mutex and waits for data if the list is empty; if there's at least one element on the list, it takes that element from the "front".

On the other side, the *enqueue*( ) method puts new values on the back of the list, and wakes up (*wakeup*( )) threads waiting for that condition, so that one will succeed to acquire the lock and will consume an element of the list, then releasing the lock.

⟨ Pop and enqueue 32 ⟩ ≡

```
bool pop(T * dst)
{
  bool taken = false;
  int r = pthread_mutex_lock(&m_);
  if (list_.size() ≡ 0) {
    waitdata();
  }
  if (list_.size() > 0) {
    *dst = list_.front();
    list_.pop_front();
    taken = true;
  }
  r = pthread_mutex_unlock(&m_);
  return taken;
}
```

See also sections 33 and 37.

This code is used in section 30.

33. When a new element is enqueued, we wake up all the threads waiting for that condition.

⟨ Pop and enqueue 32 ⟩ +≡

```
void enqueue(T & e)
{
  int r = pthread_mutex_lock(&m_);
  list_.push_back(e);
  wakeup();
  r = pthread_mutex_unlock(&m_);
}
```

34. The method *waitdata*( ) must be called when the lock $m\_$ is acquired. The return code of the function *pthread_cond_wait*( ) isn't checked: we don't expect it to be different from 0 (ok), but of course this isn't acceptable in every cases — just don't use this on production code!

⟨ Concurrency management 34 ⟩ ≡

```
void waitdata()
{     /* it must hold the lock m_ already */
  while (list_.size() ≡ 0) {
    (void) pthread_cond_wait(&c_, &m_);
  }
}
```

See also section 35.

This code is used in section 30.

35. The method *wakeup*( ) broadcast a signal to every thread waiting on the condition, so that they can start consuming the elements of the list.

⟨ Concurrency management 34 ⟩ +≡

```
void wakeup()
{
  (void) pthread_cond_broadcast(&c_);
}
```

36. ⟨ SyncedList includes 36 ⟩ ≡

```
#include <list>
#include <pthread.h>
```

This code is used in section 30.

37. Corners of slight improvements. As already said, IAWLB isn't production-ready. It works (more or less), but it isn't something you can trust. It doesn't cope very well with errors nor with signaling from the user, e.g. to exit gracefully; threads can exit/terminate and they won't be recreated, so that certain destination addresses won't receive requests anymore and in the worst condition, where no threads are left, requests will accumulate in the list, in theory indefinitely.

An interesting corner to polish could be the way the main thread feeds the FIFO list.

For each new enqueued element, the **SyncedList**'s method *enqueue*( ) locks the list and wakes up all the threads waiting for the list to be not-empty. It looks inefficient in both cases

- when requests are sparse,
- when requests are frequent and arrive in bunches in the same time.

The **SyncedList** must not be in charge of anything of this, except that it should provide a method to enqueue several elements altogether, locking only once and broadcasting the condition only once.

The *enqueue*( ) method can be overloaded to accept another list as input, and this list will be "poured" into the synchronized list in a single lock-wakeup step.

⟨ Pop and enqueue 32 ⟩ +≡

```
void enqueue(std::list<T> & e)
{
  int r = pthread_mutex_lock(&m_);
  list_.insert(list_.cbegin(),
    e.begin(), e.end());    /* change cbegin() into begin() before C++11 */
  wakeup();
  r = pthread_mutex_unlock(&m_);
}
```

38. The "hardest" change is in the ⟨ Receiving loop 19 ⟩ code. We need a local list where we can enqueue sockets without the need for locking, and two "flushing" mechanisms: one is based on the number N of threads (i.e. destination addresses), the other is based on the time passed since the last "flushing".

The local list (a buffer) must be flushed whichever condition is met first: the buffer contains enough sockets so that all the threads can work, or a certain amount of time has passed.

This is an important change I won't do.

39. Index.

IAWLB