What I Learned From Sockets

Applying the Unix Readiness Model When Composing Concurrent Operations in C++

FILIPP GELMAN

What I Learned From Sockets

Applying the Unix Readiness Model When Composing Concurrent Operations in C++

Bloomberg

Engineering

September 15, 2022

Filipp Gelman, P.E.
fgelman1@bloomberg.net

TechAtBloomberg.com

Agenda

1. Introduction to sockets
2. Select
3. Implementation in C++
4. Senders, Receivers, and Coroutines

Agenda

0. What I learned from sockets
1. Introduction to sockets
2. Select
3. Implementation in C++
4. Senders, Receivers, and Coroutines

What I Learned From Sockets

- Concurrent operations involve waiting.
  1. Setup
  2. **Wait**
  3. React

What I Learned From Sockets

- Concurrent operations involve waiting.
  1. Setup - regular code
  2. Wait
  3. React - regular code

What I Learned From Sockets

- Concurrent operations involve waiting.
  1. Setup
  2. Wait
  3. React
- What to do while waiting?

What I Learned From Sockets

What to do while waiting?

- Suspend the calling thread.
- Yield control to executor.
- Busy wait.

What I Learned From Sockets

What to do while waiting **for what?**

What I Learned From Sockets

What to do while waiting **for what?**

- For one operation to complete.

What I Learned From Sockets

What to do while waiting **for what?**

- For one operation to complete.
- For all operations to complete.

What I Learned From Sockets

What to do while waiting **for what?**

- For one operation to complete.
- For all operations to complete.
- For **any** operation to complete.

What I Learned From Sockets

- Break code up.

What I Learned From Sockets

- Break code up.
  - Regular code (math)

What I Learned From Sockets

- Break code up.
  - Regular code (math) - most code, easy to test.

What I Learned From Sockets

- Break code up.
  - Regular code (math) - most code, easy to test.
  - Code that waits - glues together regular code.

What I Learned From Sockets

- Break code up.
  - Regular code (math) - most code, easy to test.
  - Code that waits - glues together regular code.
- Wait for any of several things.

What I Learned From Sockets

- Break code up.
  - Regular code (math) - most code, easy to test.
  - Code that waits - glues together regular code.
- Wait for any of several things.
- React based on what happened.

What I Learned From Sockets

- Break code up.
  - Regular code (math) - most code, easy to test.
  - Code that waits - glues together regular code.
- Wait for any of several things.
- React based on what happened.

Don't communicate by sharing memory. Share memory by communicating.

Conclusion

How can I `.get()` the first of several futures?

How can I `co_await` the first of several awaitables?

How can I `select` several senders?

Conclusion

How can I `.get()` the first of several futures?
Stop using `std::future`.

How can I `co_await` the first of several awaitables?

How can I `select` several senders?

Conclusion

How can I `.get()` the first of several futures?
Stop using `std::future`.

How can I `co_await` the first of several awaitables?
Make them look like senders.*

How can I `select` several senders?

Conclusion

How can I `.get()` the first of several futures?
Stop using `std::future`.

How can I `co_await` the first of several awaitables?
Make them look like senders.*

How can I `select` several senders?
Make them look like sockets.*

Conclusion

How can I `.get()` the first of several futures?
Stop using `std::future`.

How can I `co_await` the first of several awaitables?
Make them look like senders.*

How can I `select` several senders?
Make them look like sockets.*

Select requires cooperation.

Select can itself be a sender/awaitable.

`wg21.link/p2300`

Introduction To Sockets

1. `read` and `write`
2. Sockets
3. Blocking vs. Non-Blocking

There will be code!

read and write

```
char buffer[1024];
int result = read(fd, buffer, 1024);
```

```
if (result > 0) {
    // read this many bytes
} else if (result == 0) {
    // end of file
} else {
    // error, check errno
}
```

read and write

```
int result = write(fd, "hello\n", 6);
```

```
if (result > 0) {
    // wrote this many bytes
} else if (result == 0) {
    // end of file (file system out of space)
} else {
    // error, check errno
}
```

Sockets

```
int sock = socket(AF_INET, SOCK_STREAM, 0);

sockaddr_in addr{
    .sin_family = AF_INET,
    .sin_port = htons(80),
    .sin_addr = {.s_addr = /* 69.187.24.15 */},
    .sin_zero = {}};

connect(sock, &addr, sizeof(addr));
```

Sockets

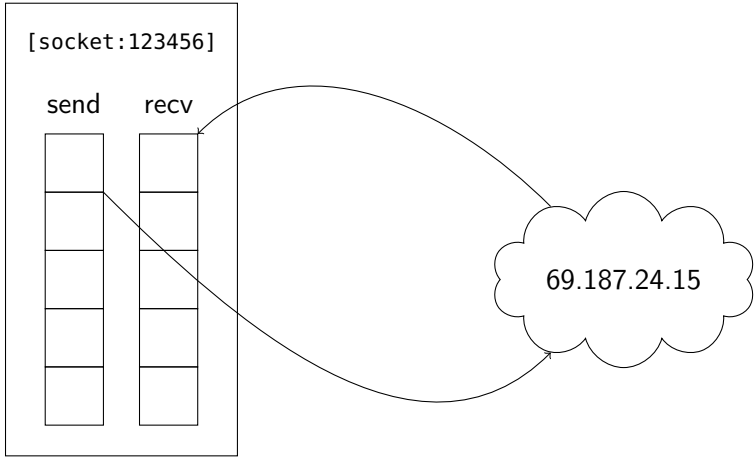
```
write(sock, "GET / HTTP/1.1\r\nHost: www.bloomberg.com\r\n\r\n", 43);

char buffer[1024];

while (int result = read(sock, buffer, 1024); result > 0) {
    render(buffer, result);
}

close(sock);
```

Blocking vs. Non-Blocking

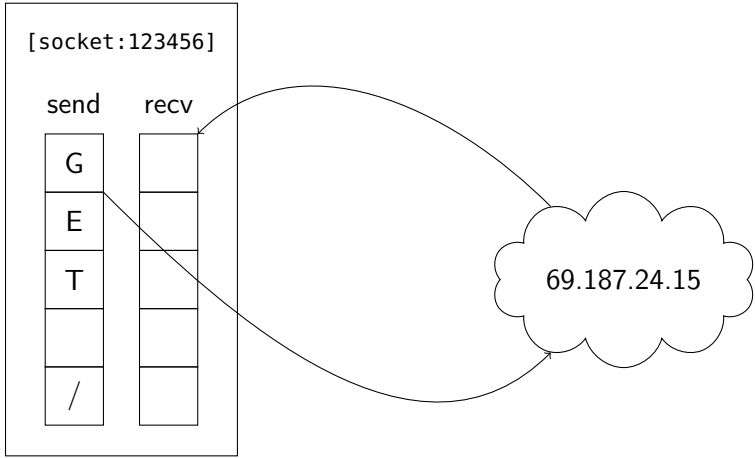
Blocking:

```
int result = read(sock, buffer, 1024);


int result = write(sock, "hello\r\n\r\n", 7);
```

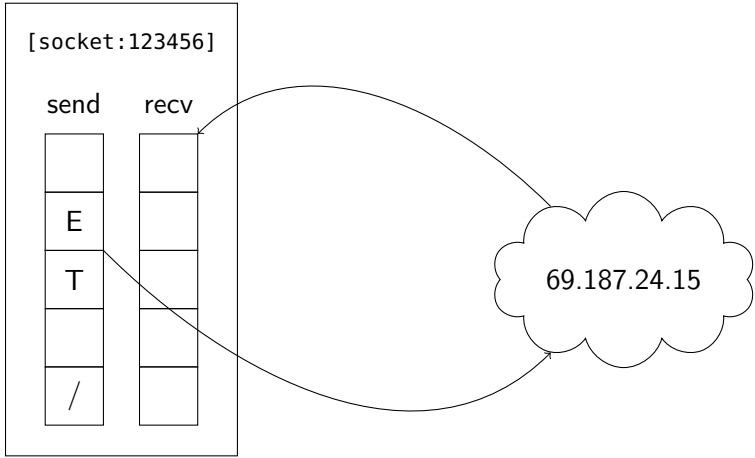
Blocking vs. Non-Blocking



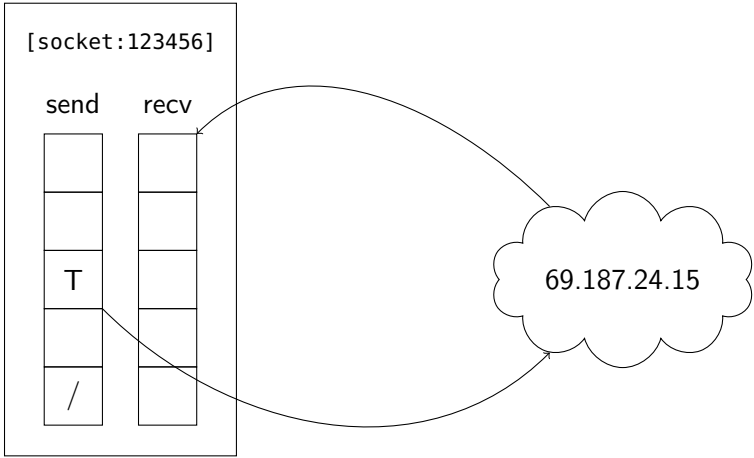
Blocking vs. Non-Blocking



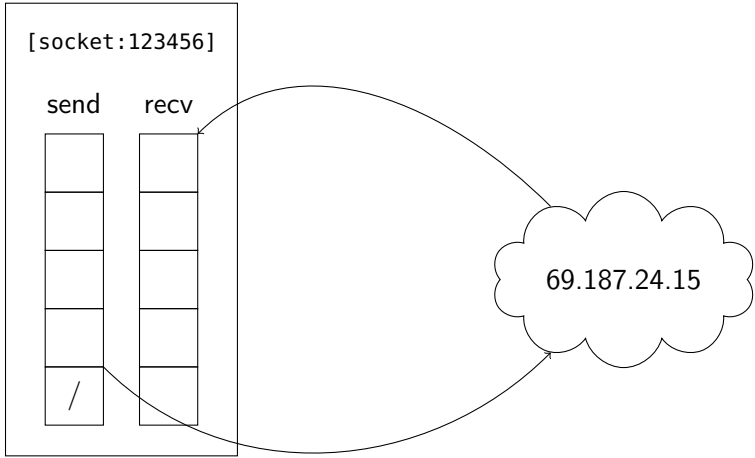
Blocking vs. Non-Blocking



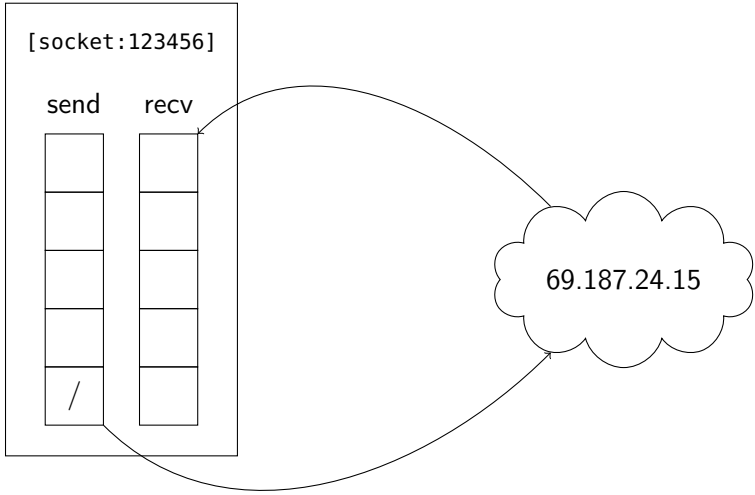
Blocking vs. Non-Blocking



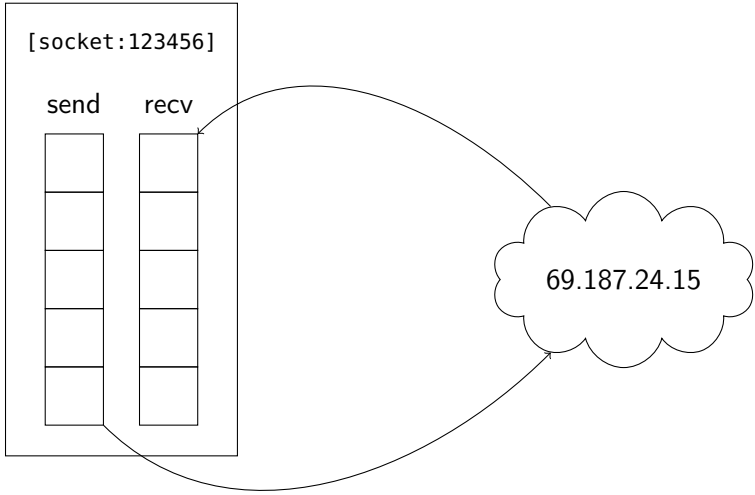
Blocking vs. Non-Blocking



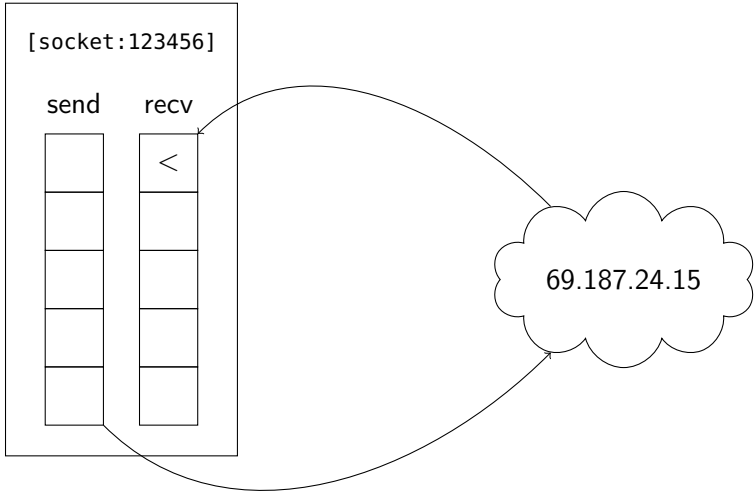
Blocking vs. Non-Blocking



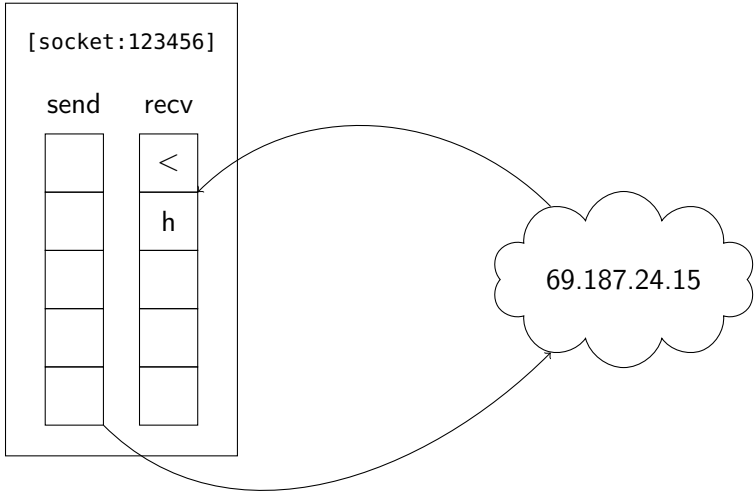
Blocking vs. Non-Blocking



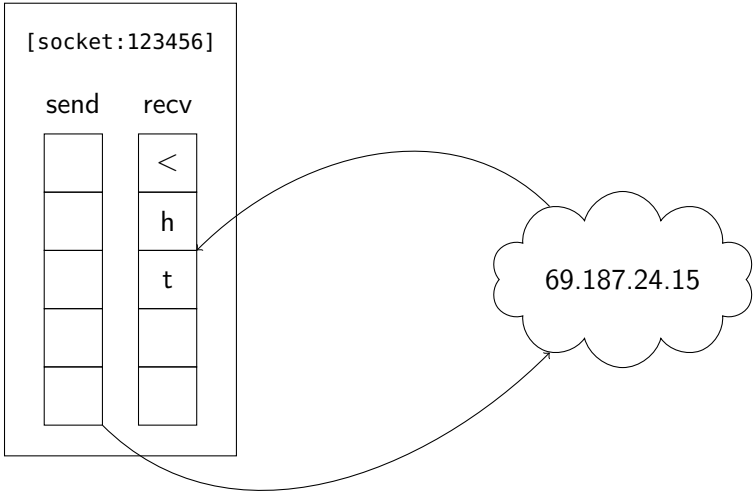
Blocking vs. Non-Blocking



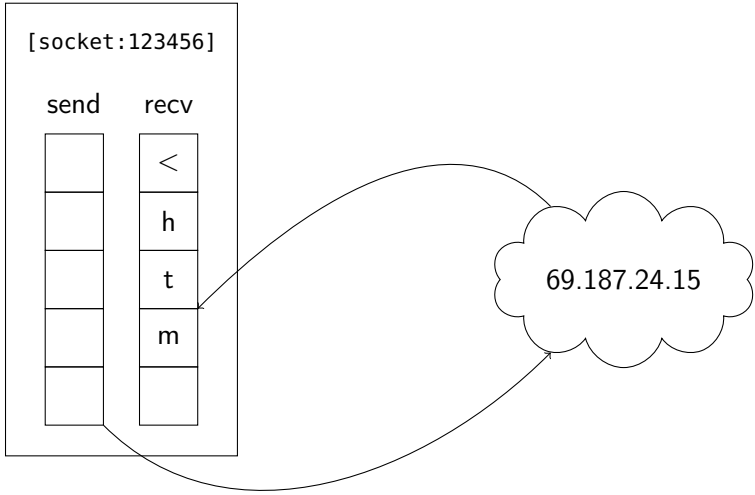
Blocking vs. Non-Blocking



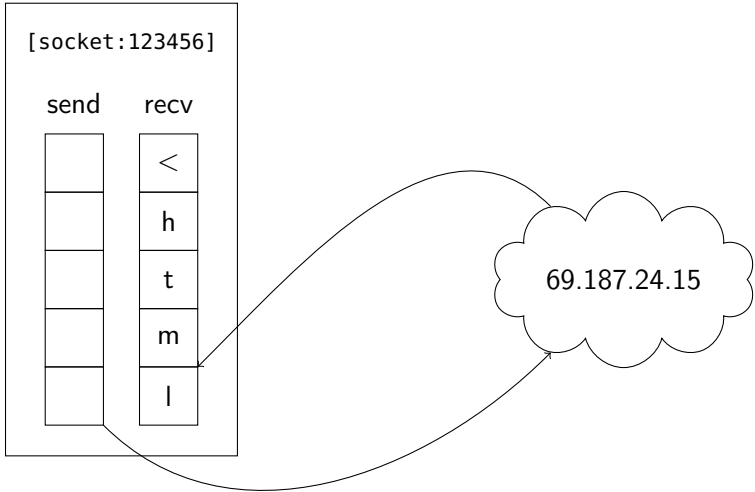
Blocking vs. Non-Blocking



Blocking vs. Non-Blocking



Blocking vs. Non-Blocking



Blocking vs. Non-Blocking

```
int sock = socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, 0);
```

Blocking vs. Non-Blocking

```
int result = read(sock, buffer, 1024);


if (result > 0) {
    // read this many bytes
} else if (result == 0) {
    // end of "file" - other side done writing
} else if (errno == EAGAIN) {
    // no data yet
} else {
    // error
}
```

Concurrency?

```
int sock2 = socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, 0);


connect(sock2, /* 20.81.111.85 */);


while (true) {
    result = read(sock, buffer, 1024);
    // handle result


    result = read(sock2, buffer, 1024);
    // handle result
}
```

Select

```
while (true) {
    fd_set fds;
    FD_ZERO(&fds);
    FD_SET(sock, &fds);
    FD_SET(sock2, &fds);

    // wait
    select(FD_SETSIZE, &fds, nullptr, nullptr, nullptr);

    // react
}
```

`fd_set == bitset<FD_SETSIZE>`

Select

React:

```
if (FD_ISSET(sock, &fds)) {
    int result = read(sock, buffer, 1024);
    // handle result
}

if (FD_ISSET(sock2, &fds)) {
    int result = read(sock2, buffer, 1024);
    // handle result
}
```

Epoll

```
int epfd = epoll_create1(0);

epoll_event evts[2] = {
    epoll_event{
        .events = EPOLLIN,
        .data = epoll_data_t{.fd = sock}},
    epoll_event{
        .events = EPOLLIN,
        .data = epoll_data_t{.fd = sock2}}};

epoll_ctl(epfd, EPOLL_CTL_ADD, sock, evts + 0);
epoll_ctl(epfd, EPOLL_CTL_ADD, sock2, evts + 1);
```

Epoll

```
while (true) {
    epoll_event evt;

    // wait
    epoll_wait(epfd, &evt, 1, -1);

    // react
}
```

Epoll

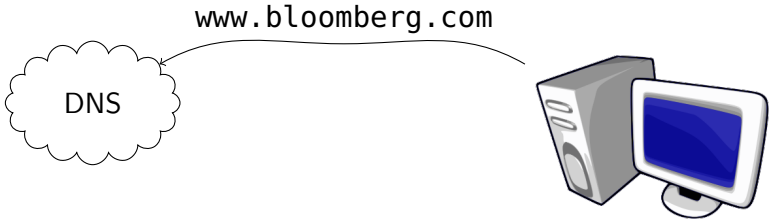
React:

```
if (evt.data.fd == sock) {
    int result = read(sock, buffer, 1024);
    // handle result
} else if (evt.data.fd == sock2) {
    int result = read(sock2, buffer, 1024);
    // handle result
}
```

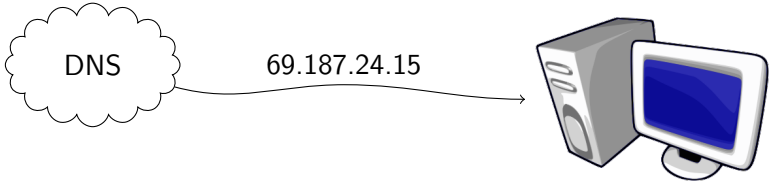
Unix Readiness Model

- Perform initial setup
- `while (true)`
  - **Wait** for events (blocking).
  - **React** to events (non-blocking).
  - On completion or error: `break`;
- `close()`

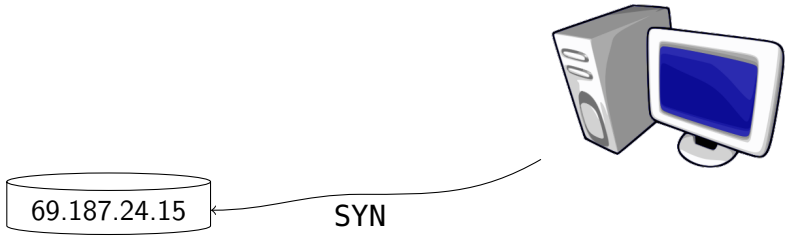
Establishing Connections



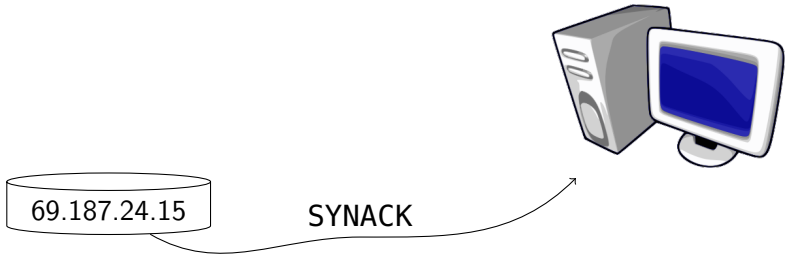
Establishing Connections



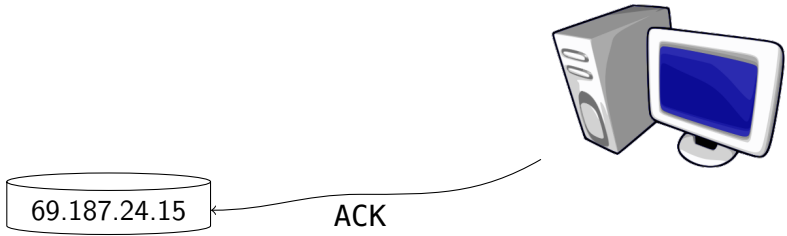
Establishing Connections



Establishing Connections

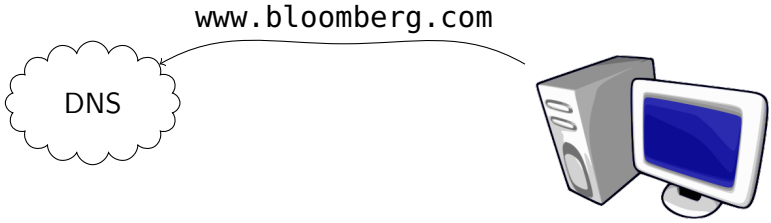


Establishing Connections

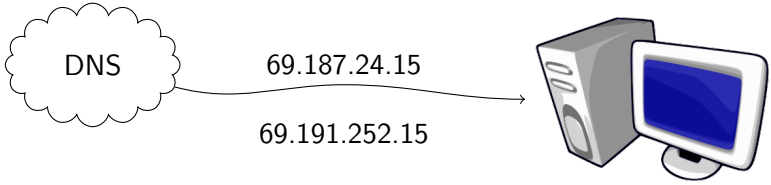


Establishing Connections

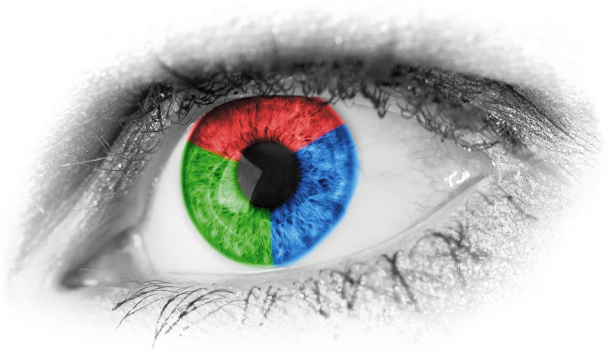
Establishing Connections



Establishing Connections

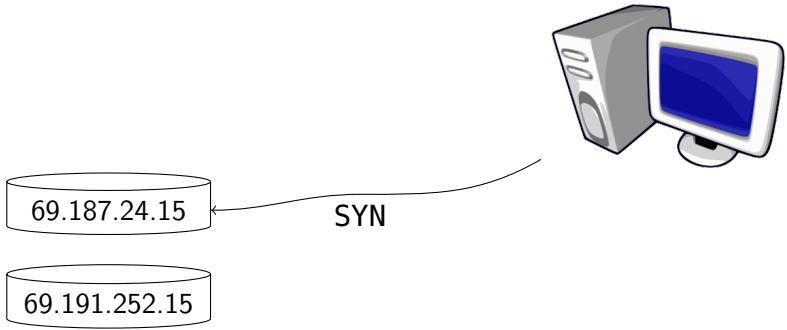


Happy Eyeballs

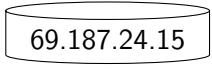


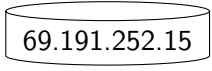
https://datatracker.ietf.org/doc/html/rfc8305

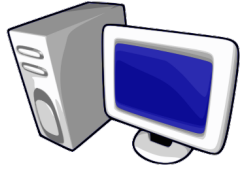
Happy Eyeballs



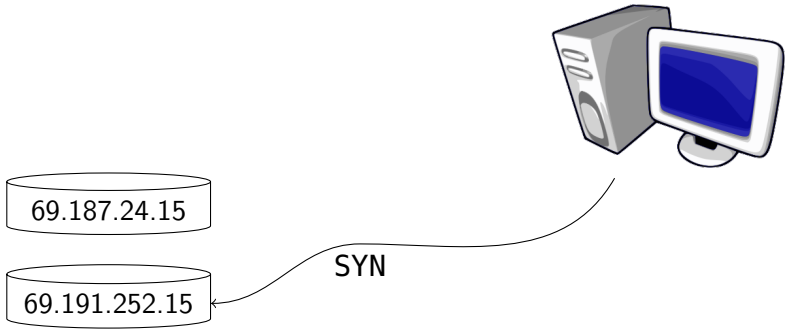
Happy Eyeballs



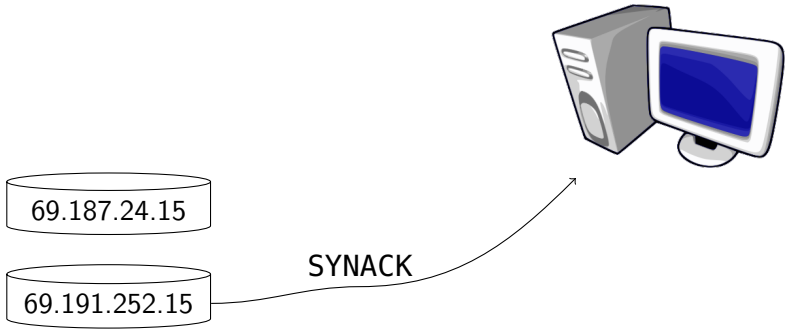




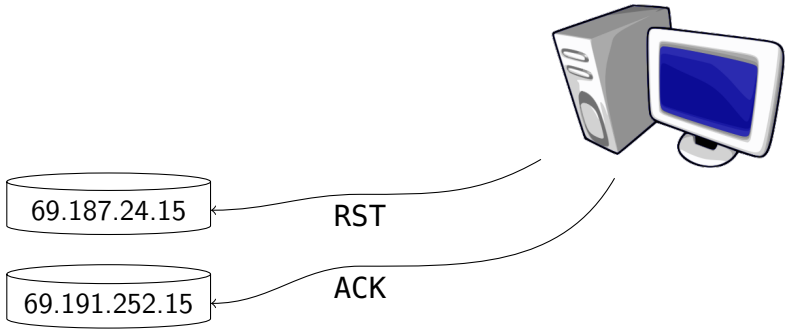
Happy Eyeballs



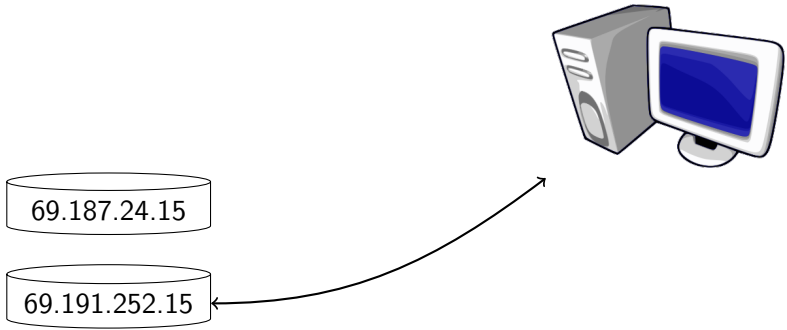
Happy Eyeballs



Happy Eyeballs



Happy Eyeballs



Happy Eyeballs

```
int connect(vector<sockaddr_in> addrs) {
    // establish connection to an address in addrs

    // ...
}
```

Happy Eyeballs

```
int epfd = epoll_create1(0);
vector<epoll_event> events(addrs.size());

for (sockaddr_in addr : addrs) {
    int sock = socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, 0);
    connect(sock, &addr, sizeof(addr));

    epoll_event evt{
        .events = EPOLLOUT | EPOLLHUP,
        .data = epoll_data_t{.fd = sock}};

    epoll_ctl(epfd, EPOLL_CTL_ADD, sock, &evt);

    // wait

    // react
}

return -1;
```

Happy Eyeballs

```
// wait
int result = epoll_wait(epfd, events.data(), events.size(), 250);

// react
for (int i = 0; i < result; ++i) {
    if (events[i].revents == EPOLLOUT) {
        // connection established
        return events[i].data.fd;
    } else {
        // connection failed
        epoll_ctl(epfd, EPOLL_CTL_DEL, events[i].data.fd, nullptr);
    }
}
```

Happy Eyeballs

```
void connect(vector<sockaddr_in> addrs, int out_socket) {
    // establish connection to an address in addrs
    // send the connected socket to out_socket

    // ...
}
```

Happy Eyeballs

```
int epfd = epoll_create1(0);
vector<epoll_event> events(addrs.size() + 1);

epoll_event evt{
    .events = EPOLLHUP,
    .data = epoll_data_t{.fd = out_socket}};

epoll_ctl(epfd, EPOLL_CTL_ADD, out_socke,t &evt);
```

Happy Eyeballs

```
for (sockaddr_in addr : addrs) {
    // connect()
    // epoll_ctl(EPOLL_CTL_ADD)

    // wait
    int result = epoll_wait(/* ... */);

    // react
    for (int i = 0; i < result; ++i) {
        // handle event i
    }
}
```

Happy Eyeballs

```
// handle event i
if (events[i].data.fd == out_socket) return;

int error;
getsockopt(/* ... */);
if (events[i].revents == EPOLLOUT) {
    // sendmsg(out_socket, events[i].data.fd);
    return;
} else {
    // epoll_ctl(EPOLL_CTL_DEL);
}
```

Happy Eyeballs++

```
void connect(queue<sockaddr_in> addrs, WriteHandle<Socket> out) {
    Select select;
    select.insert(out, Events::hup);

    Timer next_connection;
    next_connection.set(clock::now());
    select.insert(next_connection, Events::io);

    set<Socket> connections;

    while (true) {
        // ...
    }
}
```

Happy Eyeballs++

```
void connect(queue<sockaddr_in> addrs, WriteHandle<Socket> out) {
    Select select;
    select.insert(out, Events::hup);

    Timer delay;
    delay.set(clock::now());
    select.insert(next_connection, Events::io);

    set<Socket> connections;

    while (true) {
        // ...
    }
}
```

Happy Eyeballs++

```
//  while (true)

auto result = select.wait();

if (result.handle == out) return;

else if (auto iter = connections.find(result.handle);
         iter != connections.end()) {
    // check connection status
} else {
    // start next connection
}

// end while (true)
```

Happy Eyeballs++

```
// check connection status
if (result.event & Events::io) {
    out.write(*iter);
    return;
} else {
    select.erase(result.handle);
    connections.erase(iter);
    goto next_connection;
}
```

Happy Eyeballs++

```
// start next connection
next_connection:
if (!addrs.empty()) {
    Socket sock = connect_to(addrs.front());
    addrs.pop_front();
    select.insert(sock, Events::io | Events::hup);
    delay.set(clock::now() + 250ms);
} else if (connections.empty()) {
    return;
}
```

Happy Eyeballs++

```
void connect(queue<sockaddr_in> addrs, WriteHandle<Socket> out) {
    // ...

    while (true) {
        auto result = select.wait();

        // ...
    }
}
```

Happy Eyeballs++

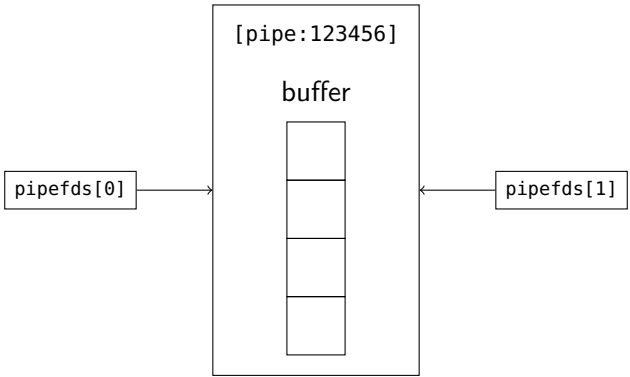
```
DetachedCoroutine connect(queue<sockaddr_in> addrs, WriteHandle<Socket> out) {
    // ...

    while (true) {
        auto result = co_await select;

        // ...
    }
}
```

Pipe

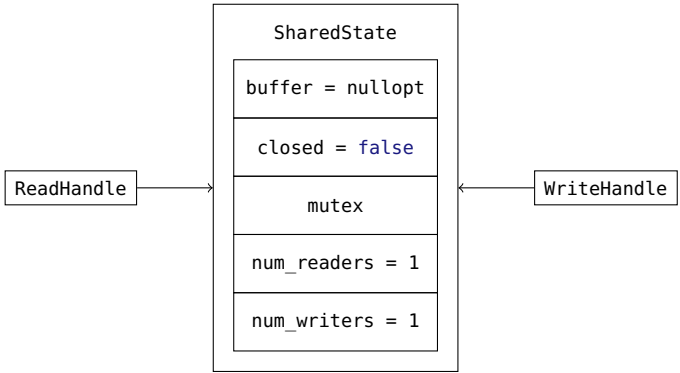
Pipe

```
int pipefds[2];

pipe(pipefds);
```

Pipe



Conceptual Socket



Conceptual Socket

Read	`int`	Write	`int`
read one byte	1	write one byte	1
end of file	0	end of file	0
operation would block	-1	operation would block	-1

Conceptual Socket

Read	`variant`	Write	`variant`
read one byte	`Success<char>`	write one byte	`Success<void>`
end of file	`EndOfFile`	end of file	`EndOfFile`
operation would block	`WouldBlock`	operation would block	`WouldBlock`

Conceptual Socket

Success:

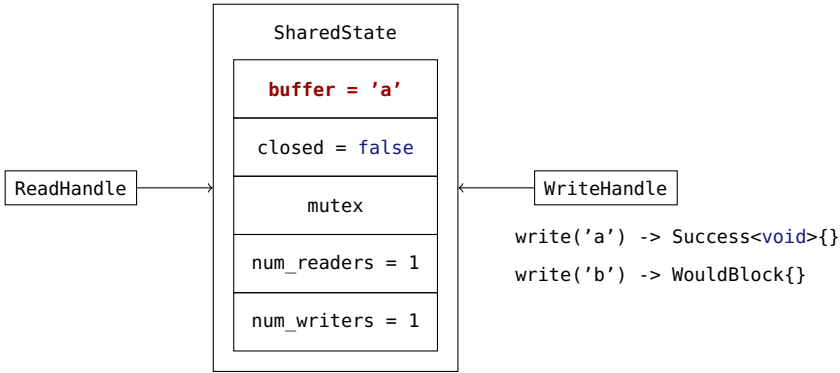
```
template <typename T>
struct Success { T value; };

template <>
struct Success<void> {};
```

Conceptual Socket

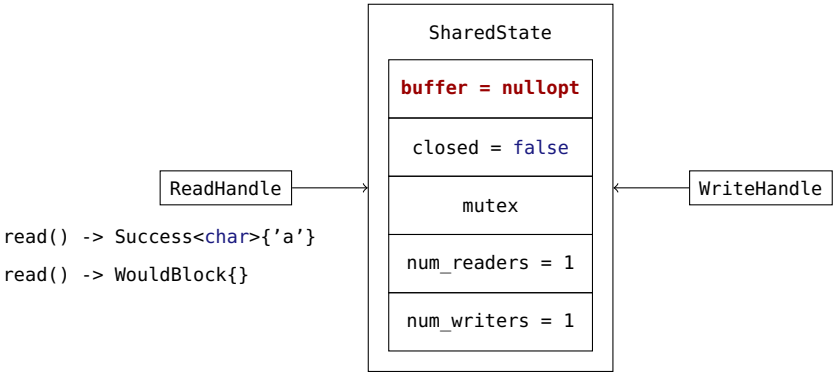
```
struct EndOfFile {};

struct WouldBlock {};
```

```
template <typename T>
using Result= variant<Succes<T>, EndOfFile, WouldBlock>;
```

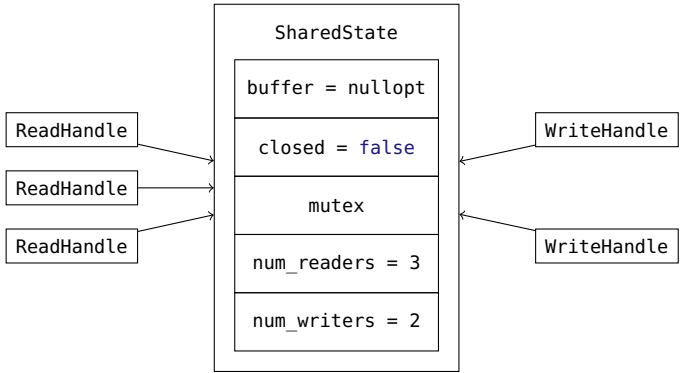
Conceptual Socket



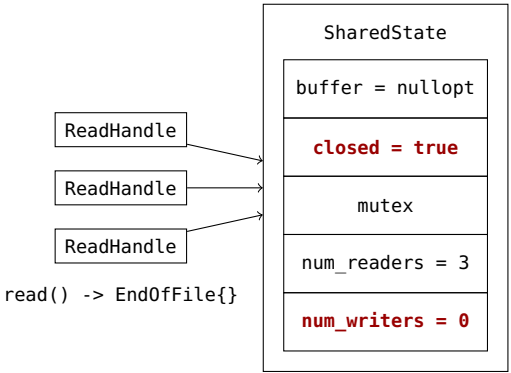
Conceptual Socket



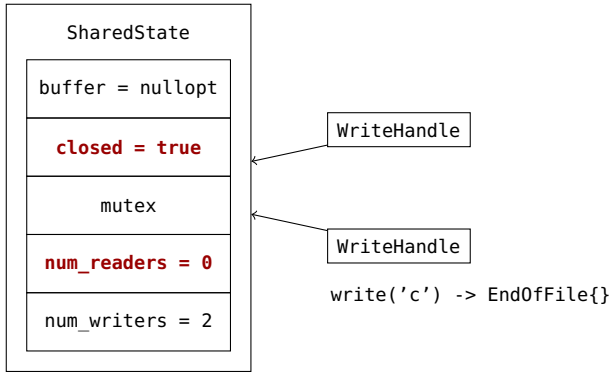
Conceptual Socket



Conceptual Socket



Conceptual Socket



Conceptual Socket

```
void capitalize_busy_wait(ReadHandle in, WriteHandle out) {
    while (true) {
        Result<char> input = in.read();

        if (input == EndOfFile{}) return;
        if (input == WouldBlock{}) continue;

        char capital = toupper(get<Success<char>>(input).value);

        while (true) {
            Result<void> output = out.write(capital);

            if (output == EndOfFile{}) return;
            if (output == WouldBlock{}) continue;
            break;
        }
    }
}
```

Conceptual Socket

```
class ReadHandle {
    shared_ptr<SharedState> state;

    Result<char> read() const;
};

class WriteHandle {
    shared_ptr<SharedState> state;

    Result<void> write(char value) const;
};
```

Conceptual Socket

```
struct SharedState {
    optional<char> buffer;
    bool closed{false};
    mutex mutex;
    atomic<unsigned> num_readers{0}, num_writers{0};

    Result<char> read();
    Result<void> write(char value);
    void close();
};
```

Conceptual Socket

```
class WriteHandle {
    shared_ptr<SharedState> state;

public:
    friend auto operator<=>(WriteHandle const&, WriteHandle const&) = default;

    WriteHandle(shared_ptr<SharedState>);

    // rule of 6 - similar to ReadHandle
    // write
}
```

Conceptual Socket

```
Result<char> SharedState::read() {
    scoped_lock lock(mutex);

    if (buffer) return Success{*exchange(buffer, nullopt)};

    if (closed) return EndOfFile{};

    return WouldBlock{};
}
```

Conceptual Socket

```
Result<char> SharedState::write(char value) {
    scoped_lock lock(mutex);

    if (closed) return EndOfFile{};

    if (buffer) return WouldBlock{};

    buffer.emplace(value);
    return Success<void>{};
}
```

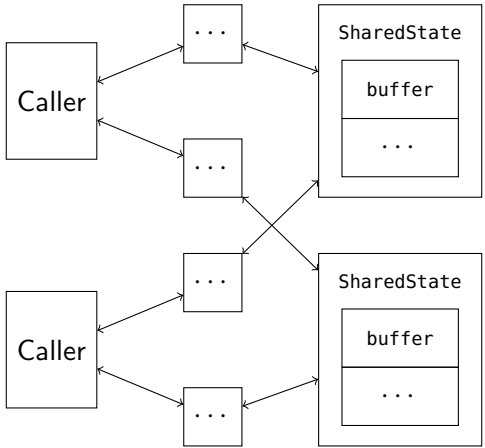
Conceptual Socket

```
void SharedState::close() {
    scoped_lock lock(mutex);

    closed = true;
}
```

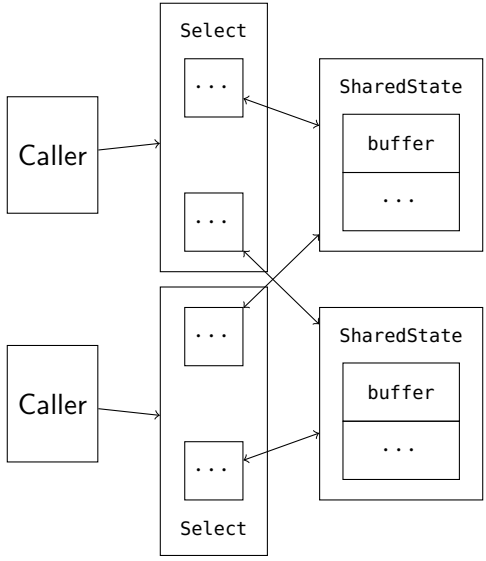
Implementing Select

- **One** caller waits for events from **many** handles.
- **One** shared state notifies **many** callers when events occur.
- Many-to-many relationship.

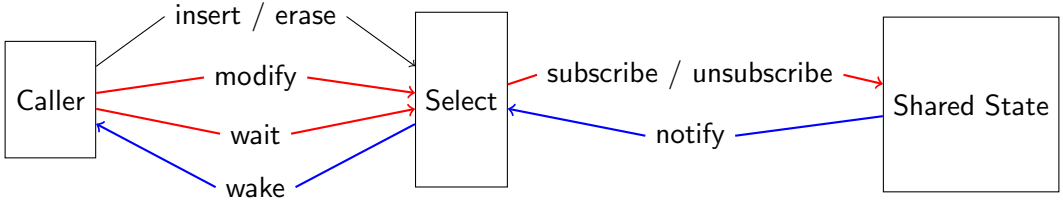
Implementing Select



Implementing Select



Implementing Select



Implementing Select

```
class Select {
    map<Handle, Events>
    // ...
};
```

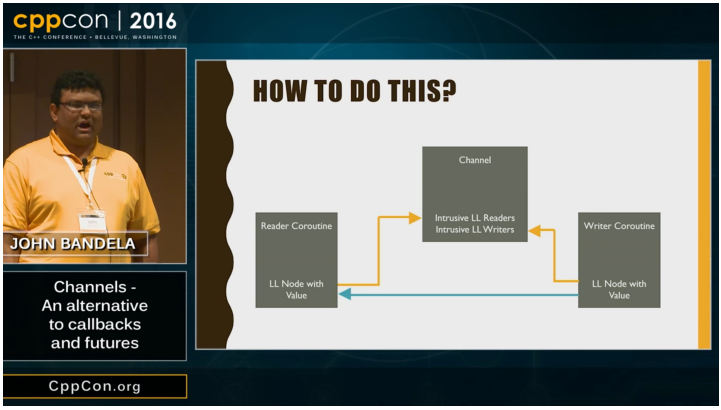
```
struct SharedState {
    multimap<Events, Select*>
    // ...
};
```

Implementing Select

```
class Select {
    set<Link, less<void>>  // owning
    // ...
};
```

```
struct SharedState {
    // ...
    intrusive_list<Link>  // non-owning
};
```

Implementing Select



https://youtu.be/N3CkQu39j5I

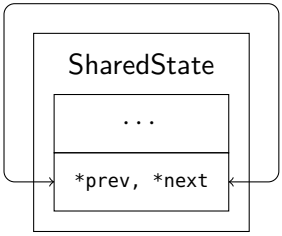
Implementing Select

- The job of `Select` is to subscribe all `Link` to their corresponding `SharedState`.
- The job of `SharedState` is to notify all subscribed `Link` of their corresponding events.

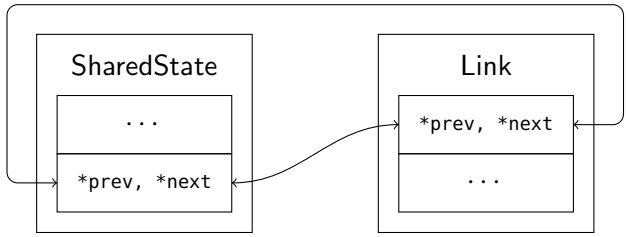
Implementing Select

- The job of `Select` is to **subscribe** all `Link` to their corresponding `SharedState`.
- The job of `SharedState` is to **notify** all subscribed `Link` of their corresponding events.

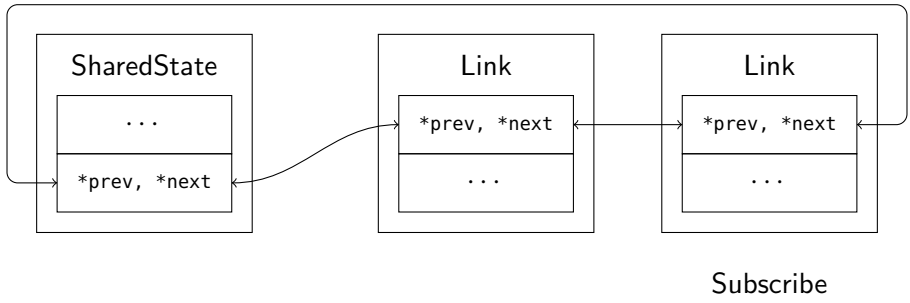
Implementing Select



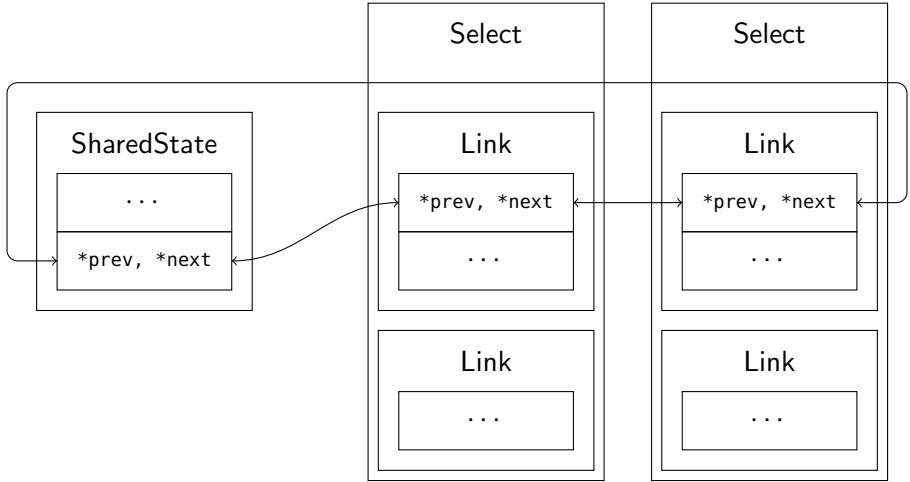
Implementing Select



Implementing Select



Implementing Select



Implementing Select

```
Result<char> SharedState::read() {
    Notify notify{Events::io};
    scoped_lock lock(mutex);

    if (buffer) {
        for (Link& link : links)
            if (get_if<WriteHandle>(&link.handle) && (link.events & Events::io))
                notify.append(link);

        return Success{*exchange(buffer, nullopt)};
    }

    // return closed ? EndOfFile{} : WouldBlock{};
}
```

Implementing Select

```
Result<char> SharedState::write(char value) {
    Notify notify{Events::io};
    scoped_lock lock(mutex);

    // if (closed || buffer) return EndOfFile{} or WouldBlock{};

    for (Link& link : links)
        if (get_if<ReadHandle>(&link.handle) && (link.events & Events::io))
            notify.append(link);

    buffer.emplace(value)
    return Success<void>{};
}
```

Implementing Select

```
void SharedState::close() {
    Notify notify{Events::hup};
    scoped_lock lock(mutex);

    for (Link& link : links)
        notify.append(link);

    closed = true;
}
```

Implementing Select

```
bool SharedState::subscribe(Link& link) {
    scoped_lock lock(mutex);

    if (link.events & Events::io) {
        if (get_if<ReadHandle>(&link.handle)) {
            if (buffer) link.revents = Events::io;
        } else {
            if (!buffer) link.revents = Events::io;
        }
    }
    if (closed) link.revents |= Events::hup;

    if (link.revents) return false;

    link.subscribed = true;
    links.append(link);
    return true;
}
```

Implementing Select

```
bool SharedState::unsubscribe(Link& link) {
    scoped_lock lock(mutex);

    if (!link.subscribed) return false;
    if (link.notifying)  return false;

    link.subscribed = false;
    link.unlink();
    return true;
}
```

Implementing Select

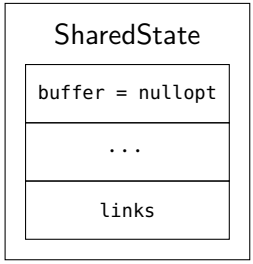
```
int epfd = epoll_create1(0);

epoll_event evt{.events = 0, .data = {}};
epoll_ctl(epfd, EPOLL_CTL_ADD, socket, &evt);

evt.events = EPOLLIN | EPOLLHUP;
epoll_ctl(epfd, EPOLL_CTL_MOD, socket, &evt);

epoll_wait(epfd, &evt, 1, -1);

epoll_ctl(epfd, EPOLL_CTL_DEL, socket, &evt);
```

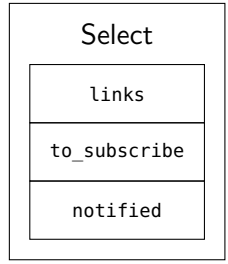
```
Select s;


s.insert(handle);


s.modify(handle, Events::io | Events::hup);

auto result = s.wait();

s.erase(handle);
```

Implementing Select

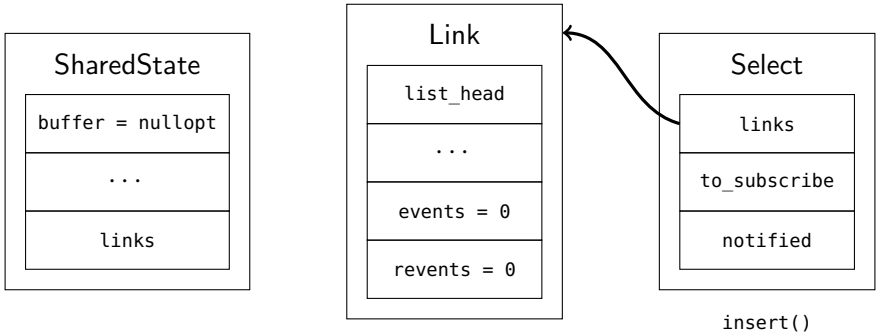
```
class Select {
    set<Link> links;
    intrusive_list<Link> to_subscribe;
    intrusive_list<Link> notified;

public:
    void insert(Handle, Events = Events{});

    void erase(Handle);

    void modify(Handle);

    Result wait();
};
```

Implementing Select

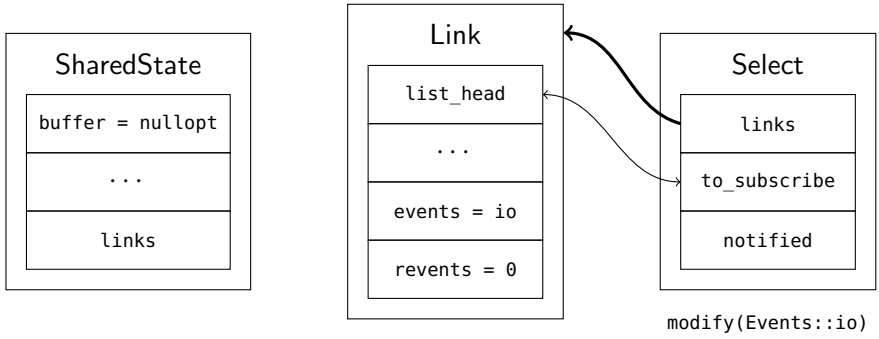




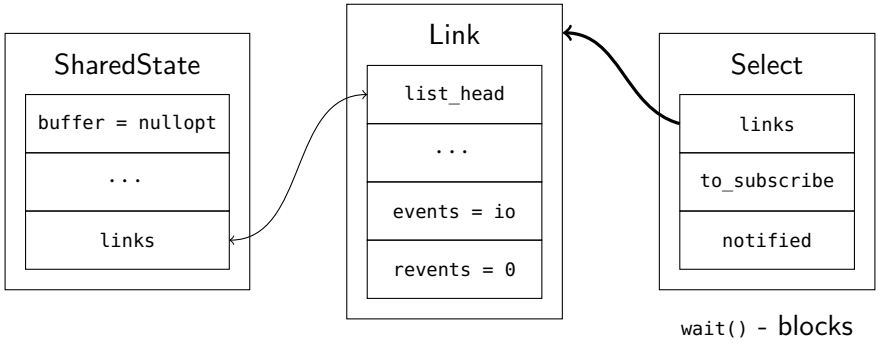
Implementing Select



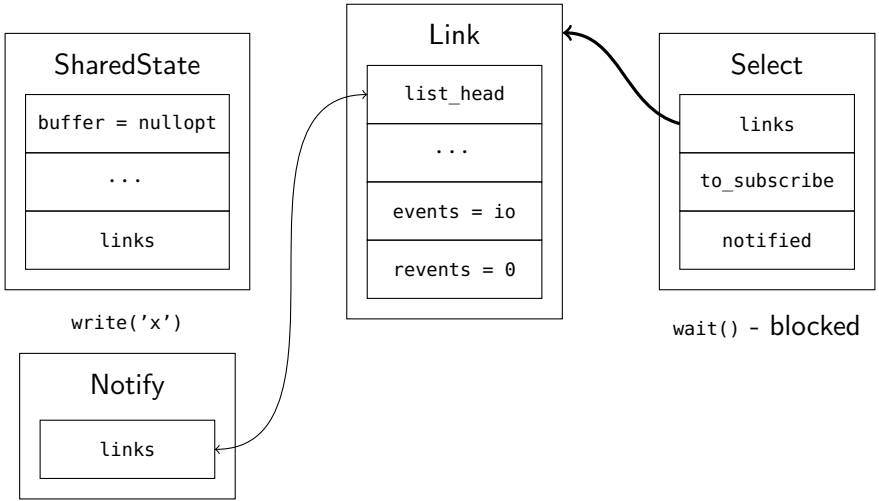
Implementing Select



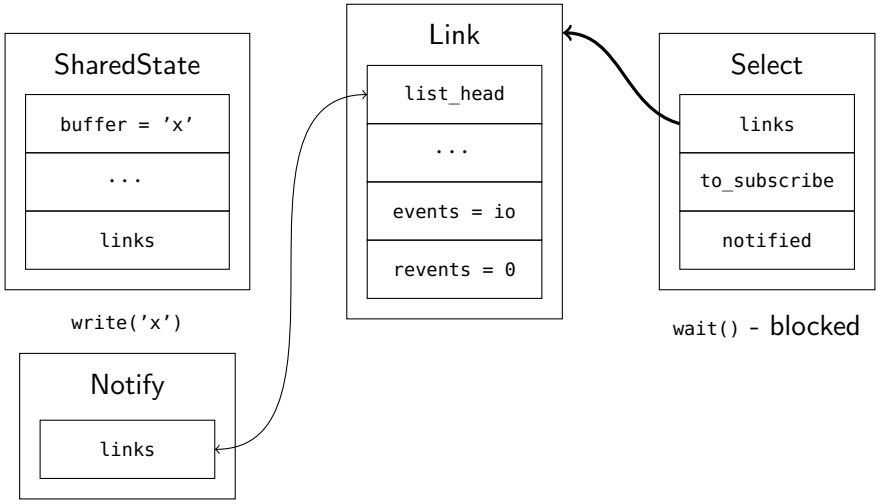
Implementing Select



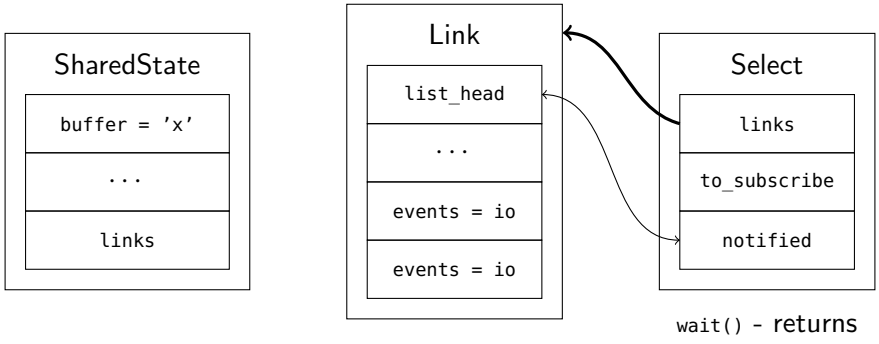
Implementing Select



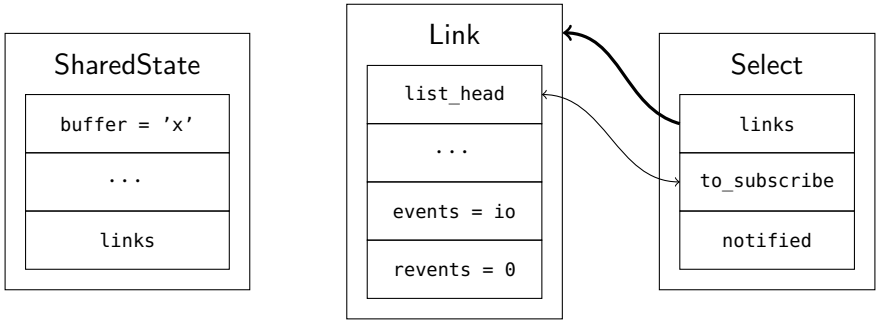
Implementing Select



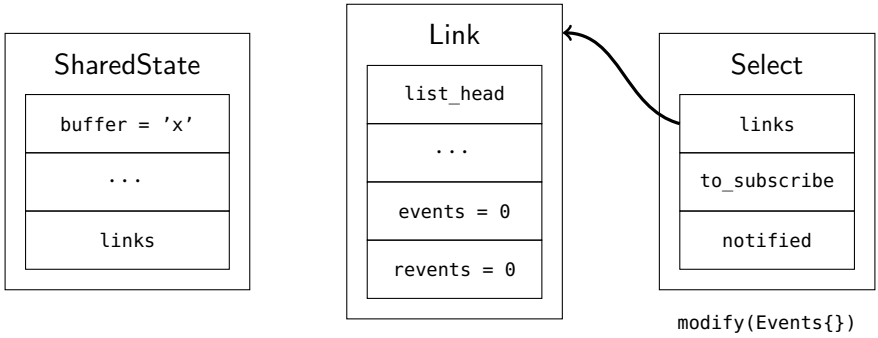
Implementing Select



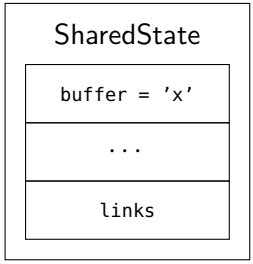
Implementing Select

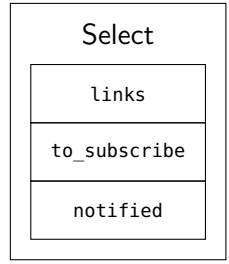


Implementing Select



Implementing Select





`erase()`

Implementing Select

```
void Select::insert(Handle handle) {

    links.try_emplace(move(handle), *this);

}
```

Implementing Select

```
Select::Result Select::wait() {
    unique_lock lock(mutex);

    while (true) {
        //
    }
}
```

Implementing Select

```
// begin while (true)

if (Link* link = notified.pop_front()) {
    to_subscribe.append(*link);
    return {link->handle, exchange(link->revents, Events{})};
}

while (Link* link = to_subscribe.pop_left()) {
    if (!link->handle->subscribe(*link)) {
        to_subscribe.append(*link);
        return {link->handle, exchange(link->revents, Events{})};
    }
}

cond.wait(lock);

// end while (true)
```

Implementing Select

```
void Select::erase(Handle handle) {
    auto iter = links.find(handle);

    {
        unique_lock lock(mutex);

        iter->ensureUnlinked(lock);
    }

    links.erase(iter);
}
```

Implementing Select

```
void Select::modify(Handle handle, Events events) {
    Link& link = *links.find(handle);

    {
        unique_lock lock(mutex);

        link.ensureUnlinked(lock);

        if ((link.events = events))
            to_subscribe.append(link);
    }
}
```

Implementing Select

```
void Link::ensureUnlinked(unique_lock& lock) {
    if (!subscribed) return;

    if (handle->unsubscribe(*this)) return;

    while (subscribed) this->cond.wait(lock);

    this->unlink();
}
```

Implementing Select

```
void Notify::append(Link& link) {
    link.notifying = true;
    links.append(link);
}
```

Implementing Select

```
Notify::~Notify() {
    while (Link* link = links.pop_front()) {
        scoped_lock lock(link->select.mutex);

        link->notifying = link->subscribed = false;  // clear flags

        link->select.notified.append(*link);  // add to notified list

        this->cond.notify_one();  // wake up ensureUnlinked callers
        select.cond.notify_one();  // wake up wait callers
    }
}
```

Implementing Select

```
Select select;
select.insert(a);
select.insert(b);
select.modify(a, Events::io);
select.modify(b, Events::hup);

while (true) {
    auto result = select.wait();

    if (result.handle == a) {
        // read some data from a
    } else {
        // b was shut
    }
}
```

Enhancements

```
class ReadHandle {
    Result<char> read();
};

class WriteHandle {
    Result<void> write(char);
};
```

Enhancements

```
class ReadHandle {
    Result<size_t> read(span<char>);
};

class WriteHandle {
    Result<size_t> write(span<char const>);
};
```

Enhancements

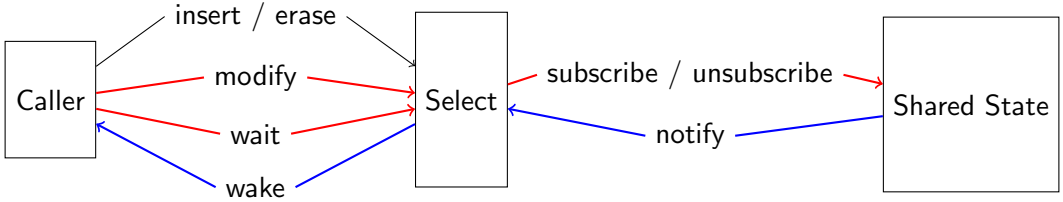
```
template <typename T>
class ReadHandle {
    Result<T> read();
};

template <typename T>
class WriteHandle {
    Result<void> write(T);
};
```

Selecting Senders

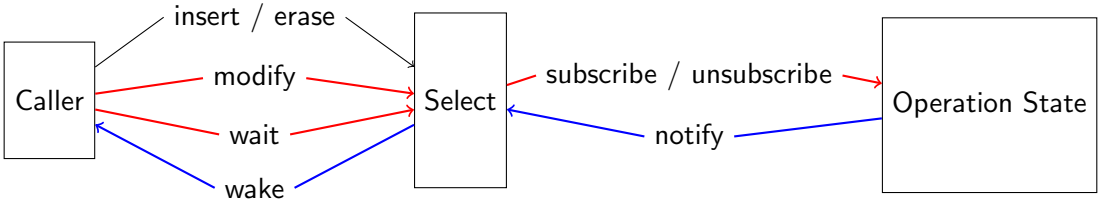
https://youtu.be/xiaqNvqRB2E

Selecting Senders

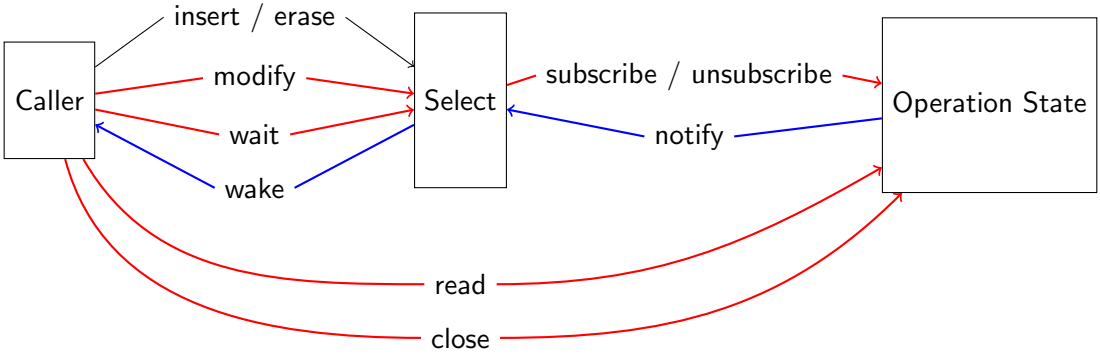
Selecting Senders



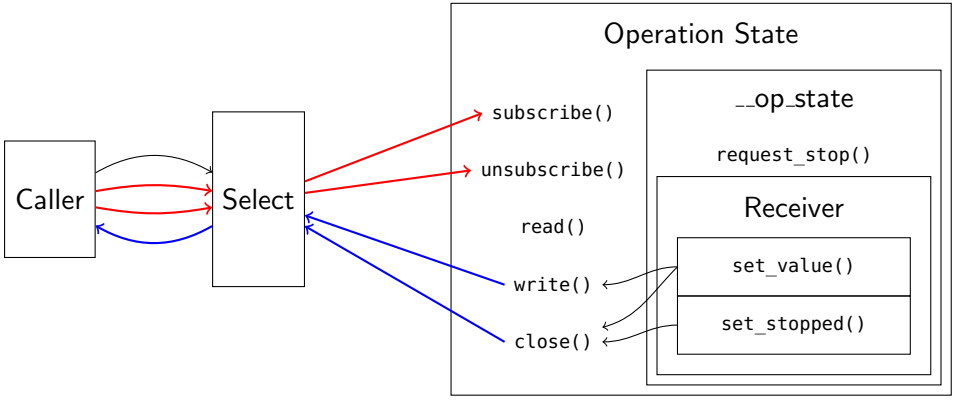
Selecting Senders



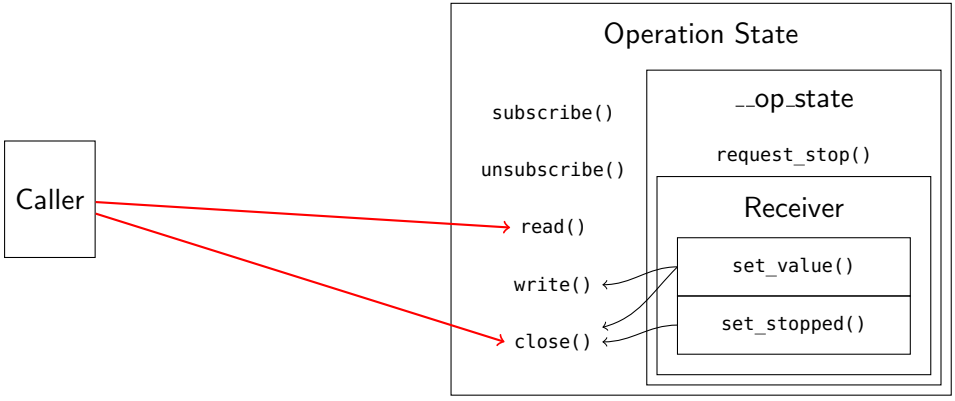
Selecting Senders



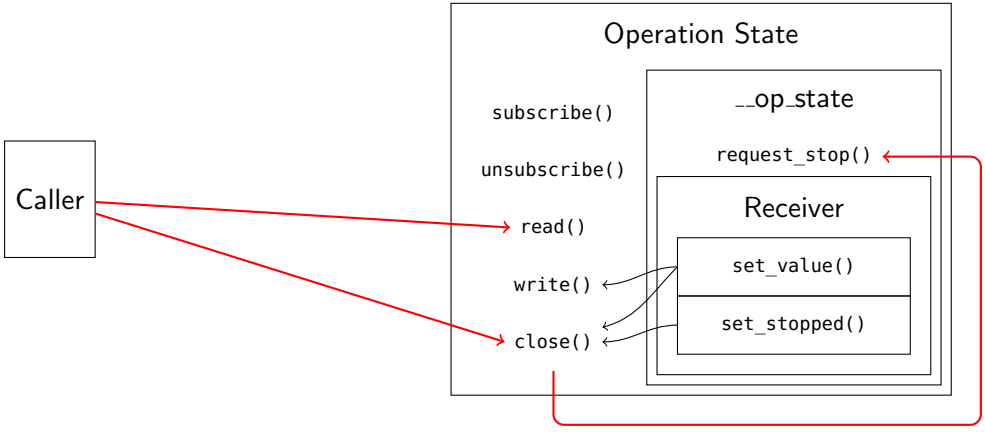
Selecting Senders



Selecting Senders



Selecting Senders



Select is a Sender

```
class Select {
    // ...
    condition_variable cond;

public:
    Result wait();  // blocks calling thread
};
```

Select is a Sender

```
class Select {
    struct OperationState {
        virtual void set_value(Result) = 0;
    };
    OperationState* op_state;

public:
    struct Sender;

    operator Sender();
};
```

Select is a Sender

```
template <typename RECEIVER>
struct SelectOperationState : OperationState {
    Select& select;
    RECEIVER receiver;

    void set_value(Result r) override {
        set_value(move(receiver), move(r));
    }

    void run() {
        // check notified list for previous events
        // check to_subscribe list for immediate events

        select.op_state = this;
    }
}
```

Select is a Sender

```
struct Sender {
    Select& select;

    template <typename RECEIVER>
    friend auto connect(Sender s, RECEIVER r) -> SelectOperationState<RECEIVER> {
        return {{}, s.select, move(r)};
    }
};
```

Select is a Sender

```
void Notify::~Notify() {
    while (Link* link = links.pop_front()) {
        unique_lock lock(link->select.mutex);
        // ...
        if (auto* op_state = exchange(link->select.op_state, nullptr)) {
            // select has a waiting receiver
            lock.unlock();
            op_state->set_value({link->handle, exchange(link->revents, Events{})});
        } else {
            link->select.notified.append(link);
        }
    }
}
```

Select is a Sender

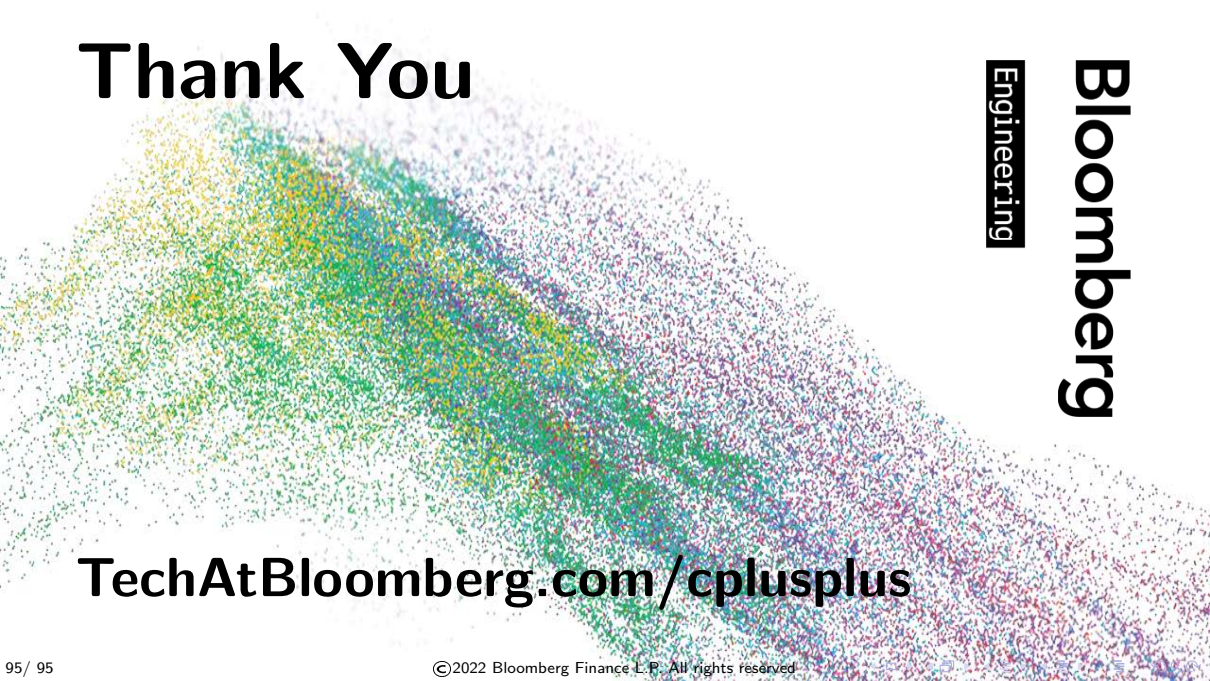
```
result = sync_wait(select);
```

```
result = co_await select;
```

What I Learned From Sockets

1. **Act** without blocking.
2. **Wait** by blocking.
3. **React** without blocking.
4. Repeat.

Thank You



Bloomberg

Engineering

TechAtBloomberg.com/cplusplus