accu 2023

WHAT I LEARNED FROM SOCKETS

FILIPP GELMAN

What I Learned From Sockets

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

April 21, 2023 Filipp Gelman, P.E.
fgelman1@bloomberg.net

TechAtBloomberg.com

What I Learned From Sockets

- Select is a powerful tool.

What I Learned From Sockets

- Select is a powerful tool.
  - Start several operations.
  - Wait for one or more operation to indicate activity.
  - React based on what happened.

What I Learned From Sockets

- Select is a powerful tool.
  - Start several operations.
  - Wait for one or more operation to indicate activity.
  - React based on what happened.
- Select is useful beyond just sockets.

What I Learned From Sockets

- Select is a powerful tool.
  - Start several operations.
  - Wait for one or more operation to indicate activity.
  - React based on what happened.
- Select is useful beyond just sockets.
- Select can be implemented in C++.

What I Learned From Sockets

- Concurrent or asynchronous operations involve waiting.
- There are different ways to wait.
- Select allows us to isolate how to wait from the operations.

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`

Agenda

Introduction To Sockets

0. Files and file descriptors
1. `read` and `write`
2. Sockets
3. Blocking vs. Non-Blocking

There will be code!

Files and File Descriptors

```
int fd = open("somefile", O_CREAT | O_TRUNC | O_RDWR, 0664);

// use fd

close(fd);
```

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

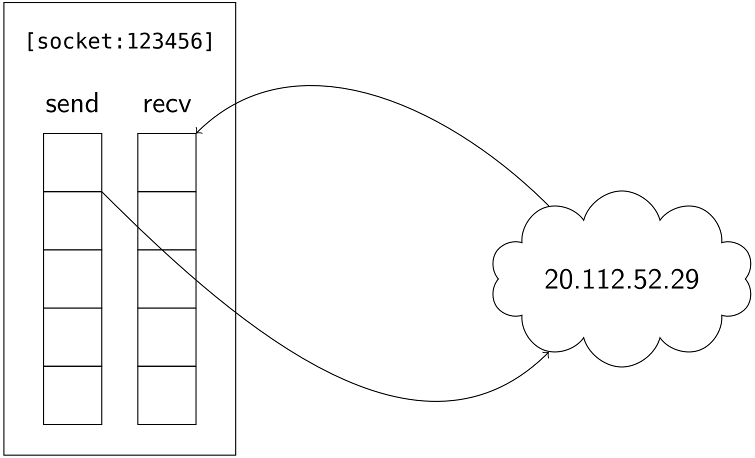
```
write(sock, "GET / HTTP/1.1\r\nHost: www.bloomberg.com\r\n\r\n", 43);

char buffer[1024];

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

close(sock);
```

Blocking vs. Non-Blocking

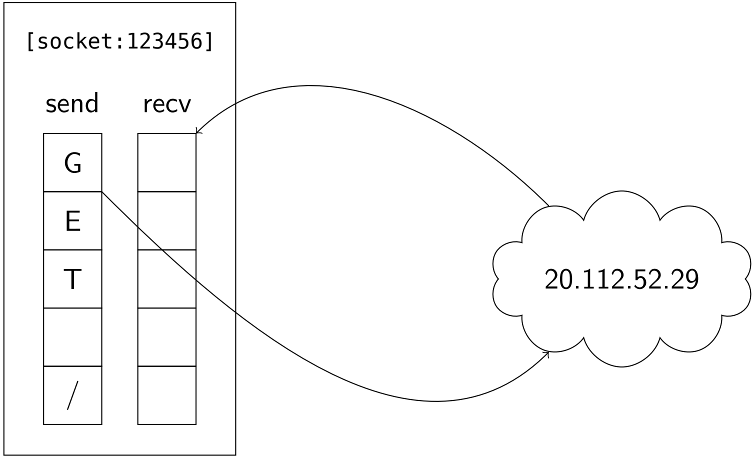
Blocking:

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


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

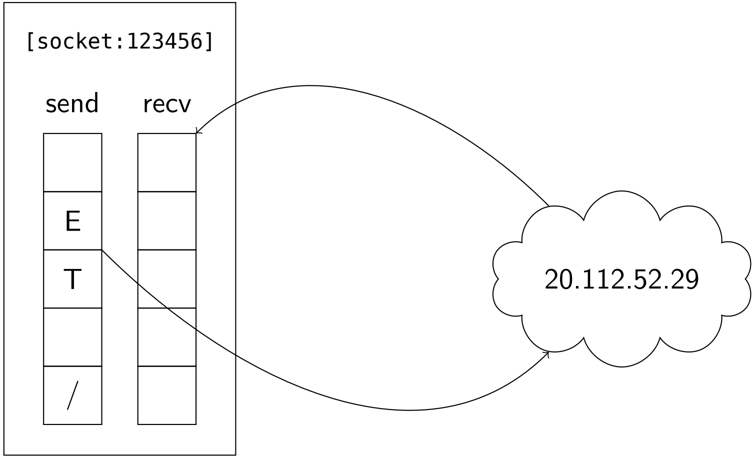
Blocking vs. Non-Blocking



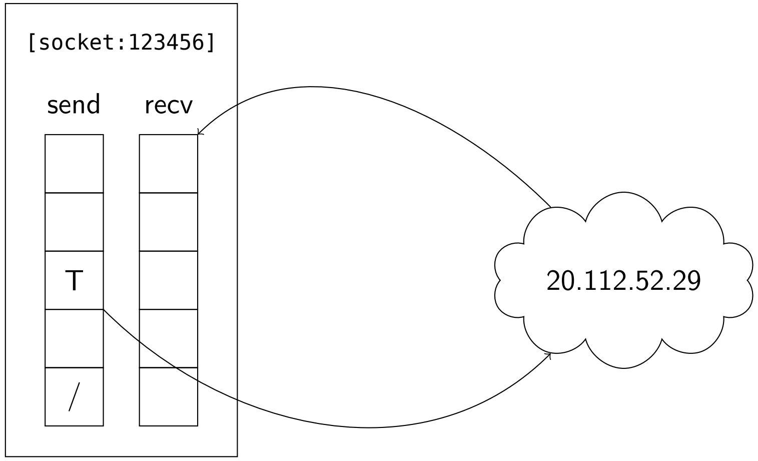
Blocking vs. Non-Blocking



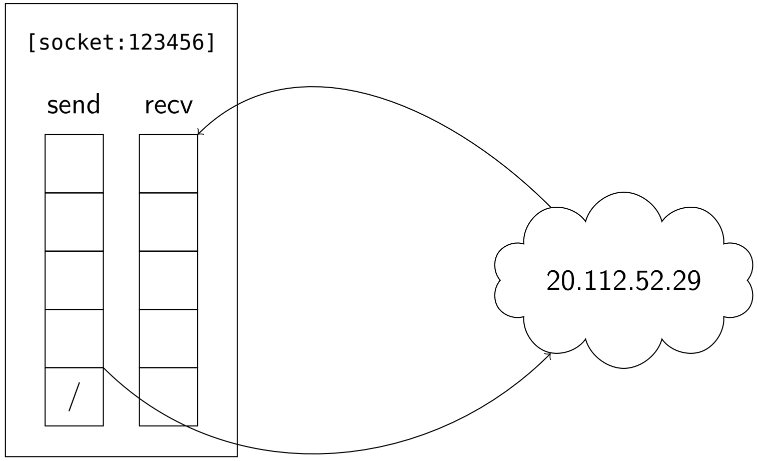
Blocking vs. Non-Blocking



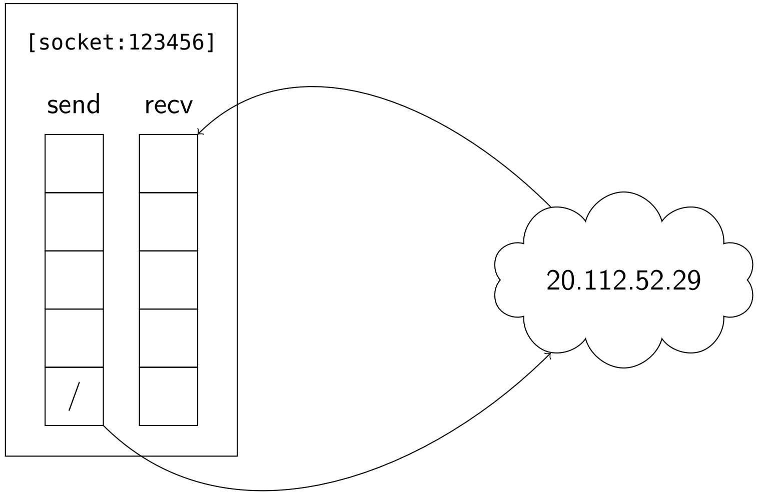
Blocking vs. Non-Blocking



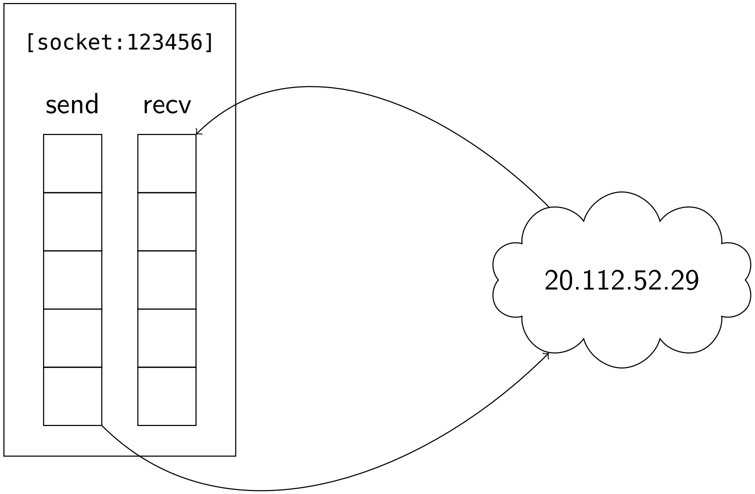
Blocking vs. Non-Blocking



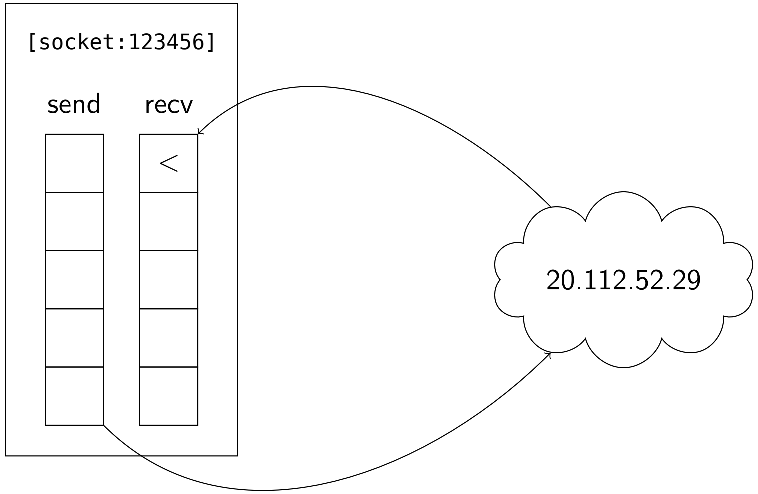
Blocking vs. Non-Blocking



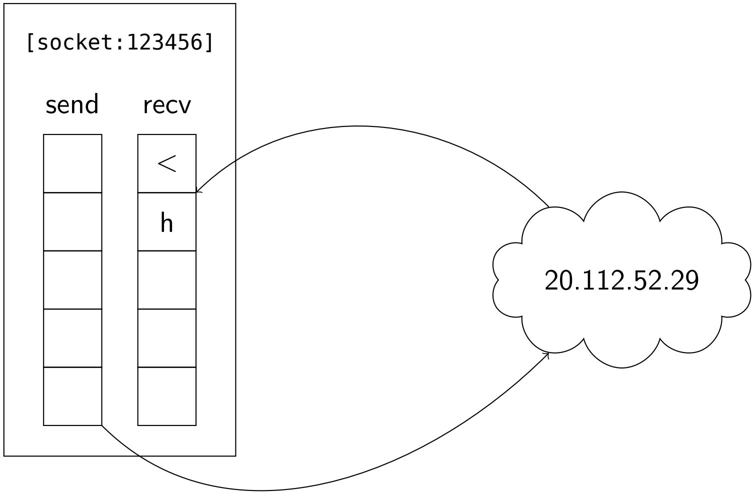
Blocking vs. Non-Blocking



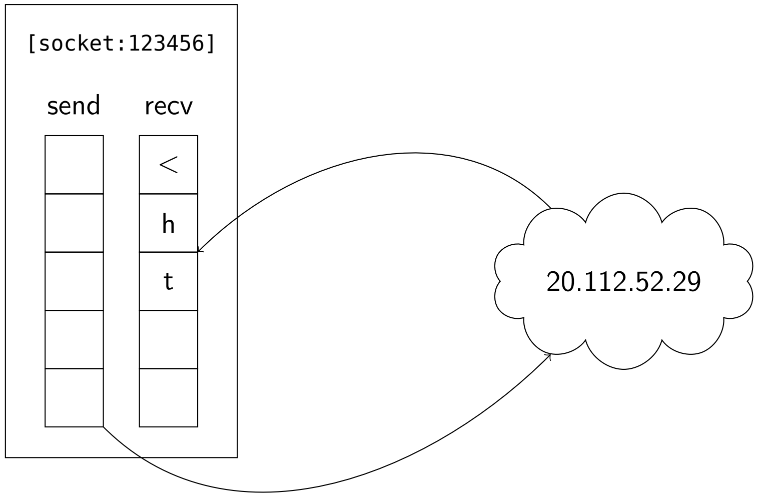
Blocking vs. Non-Blocking



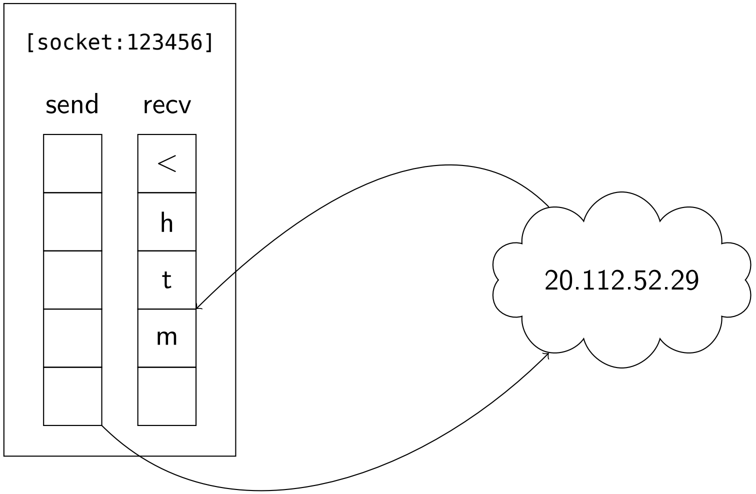
Blocking vs. Non-Blocking



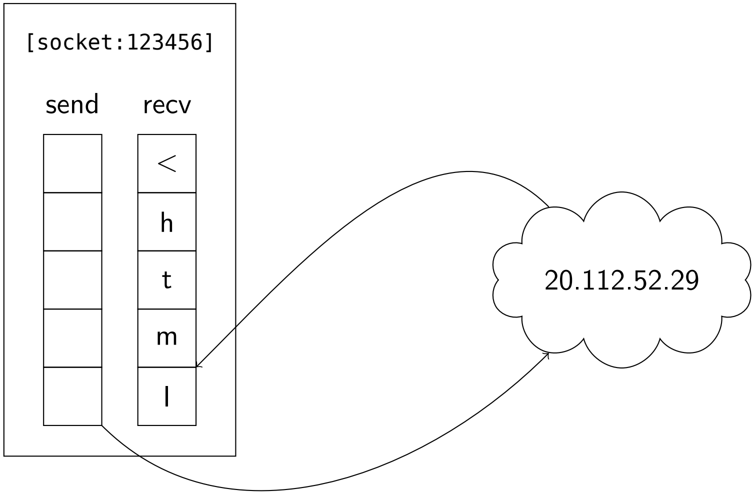
Blocking vs. Non-Blocking



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
}
```

Poll

```
while (true) {
    pollfd pfds[2] = {
        pollfd{.fd = sock, .events = POLLIN, .revents = 0},
        pollfd{.fd = sock2, .events = POLLIN, .revents = 0}};

    // wait
    poll(pfds, 2, -1);

    // react
}
```

Poll

React:

```
if (pfds[0].revents & POLLIN) {
    int result = read(sock, buffer, 1024);
    // handle result
}

if (pfds[1].revents & POLLLIN) {
    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

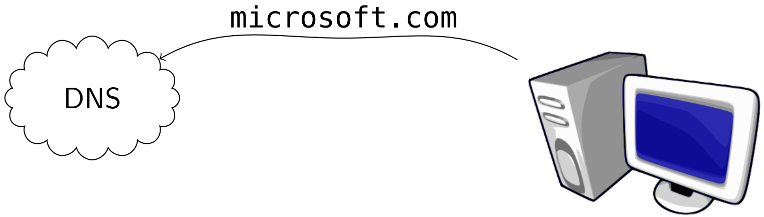
React:

```
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
}
```

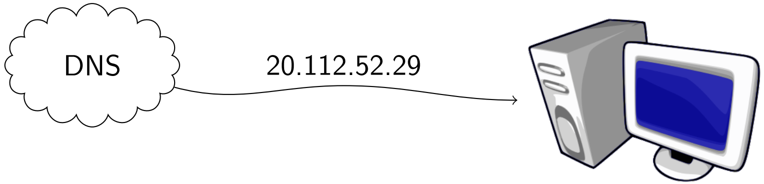
Unix Readiness Model

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

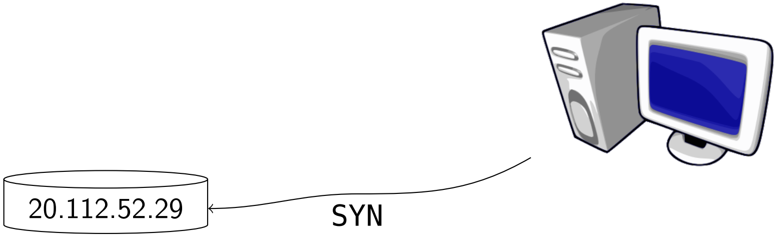
Establishing Connections



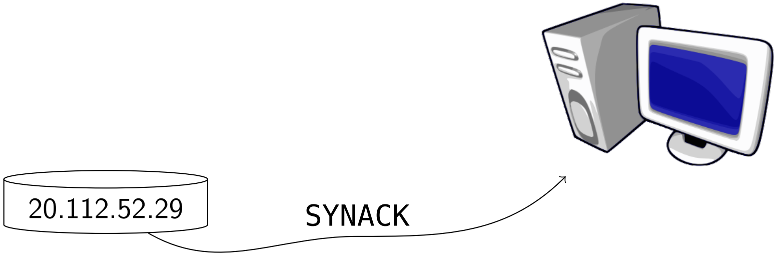
Establishing Connections



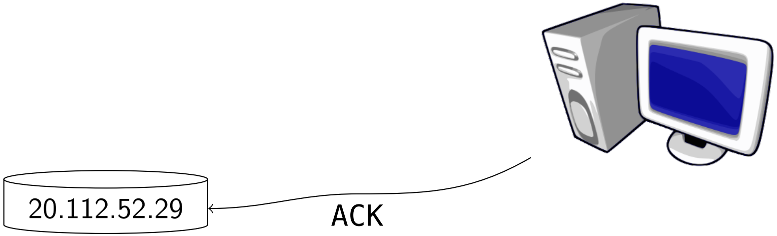
Establishing Connections



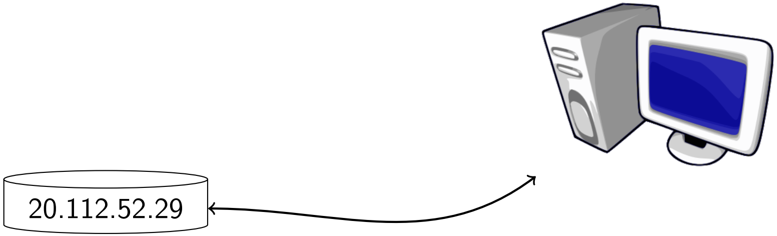
Establishing Connections



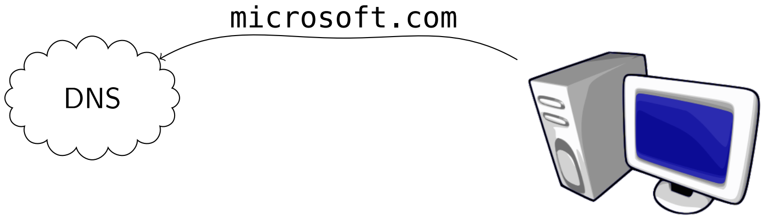
Establishing Connections



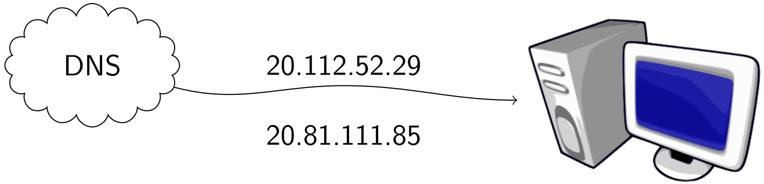
Establishing Connections



Establishing Connections



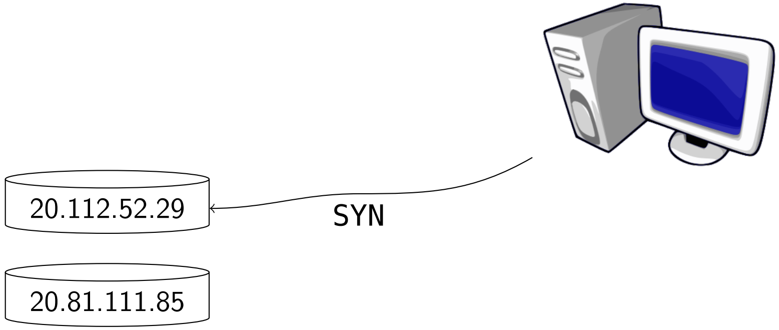
Establishing Connections



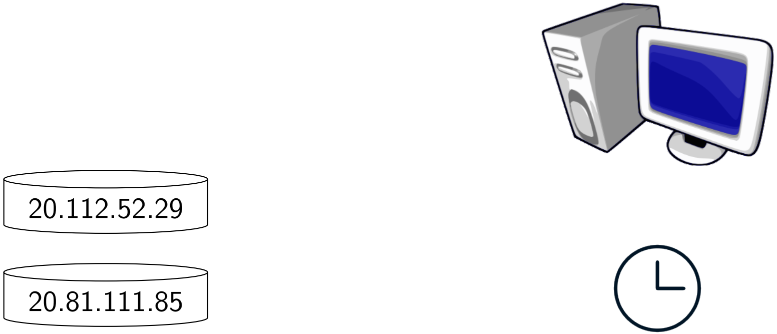
Happy Eyeballs

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

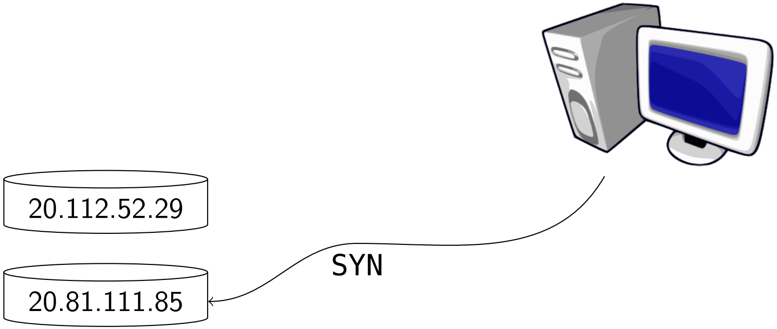
Happy Eyeballs



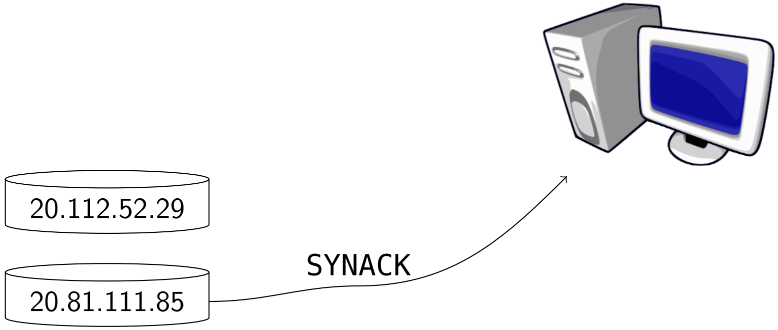
Happy Eyeballs



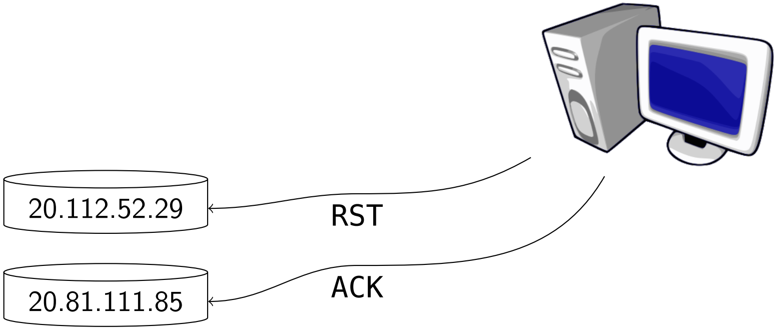
Happy Eyeballs



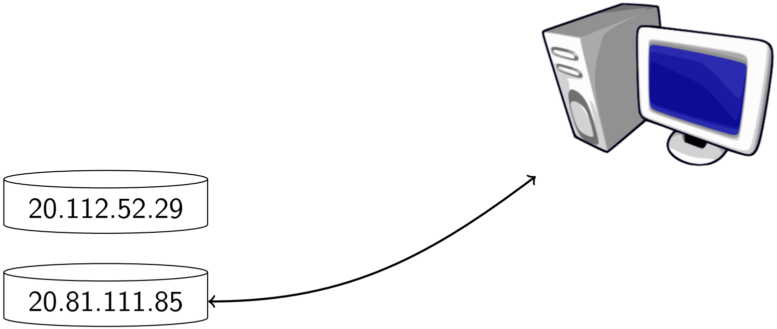
Happy Eyeballs



Happy Eyeballs



Happy Eyeballs



Happy Eyeballs

```cpp
UniqueFd connect(span<sockaddr_in const> addrs) {
    // establish connection to an address in addrs

    // ...
}
```

Happy Eyeballs

```
UniqueFd epfd(epoll_create1(0));
set<UniqueFd, less<void>> connections;

for (sockaddr_in addr : addrs) {
    // establish connections with delay
}

while (!connections.empty()) {
    // wait for connection attempts to complete
}

// all connections failed
return UniqueFd();
```

Happy Eyeballs

```
// establish connections with delay
for (sockaddr_in addr : addrs) {
    // set up
    UniqueFd conn(socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, 0));
    connect(conn, &addr, sizeof(addr));

    epoll_event evt{
        .events = EPOLLOU | EPOLLHUP,
        .data = epoll_data{.fd = *conn}};

    epoll_ctl(*epfd, EPOLL_CTL_ADD, *conn, &evt);

    // wait

    // react
}
```

Happy Eyeballs

```
// establish connections with delay
for (sockaddr_in addr : adrs) {
    // set up

    // wait
    int count = epoll_wait(*epfd, &evt, 1, 250);

    // react
}
```

Happy Eyeballs

```
for (sockaddr_in addr : addrs)
    // set up
    // wait: int count = epoll_wait()
    // react
    if (!count) continue;

    auto it = connections.find(event.data.fd);

    if (even.revents == EPOLLOUT) {
        // connection established
        return move(connections.extract(it).value());
    } else {
        // connection failed
        epoll_ctl(*epfd, EPOLL_CTL_DEL, **it, nullptr);
        connections.erase(it);
    }
}
```

Happy Eyeballs

```
while (!connections.empty()) {
    // wait
    epoll_event evt;
    int count = epoll_wait(*epfd, &evt, 1, -1);

    // react
}

return nullopt;
```

Happy Eyeballs

```
auto wait = [&](int timeout) -> optional<epoll_event> {
    optional<epoll_event> ret(in_place);
    int count = epoll_wait(*epfd, &*ret, 1, timout);
    if (!count) ret.reset();
    return ret;
};
```

Happy Eyeballs

```
auto react = [&](epoll_event event) -> optional<UniqueFd> {
    auto it = connections.find(event.data.fd);
    if (event.events == EPOLLOUT)
        return make_optional(move(connections.extract(it).value()));
    epoll_ctl(*epfd, EPOLL_CTL_DEL, **it, nullptr);
    connections.erase(it);
    return nullopt;
};
```

Happy Eyeballs

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

    if (auto event = wait(250))
        if (auto fd = react(*event))
            return fd;
}

while (!connections.empty())
    if (auto fd = react(*wait(-1))) return fd;

return nullopt;
```

Happy Eyeballs

Demo!

Happy Eyeballs

```cpp
void connect(span<sockaddr_in const> addrs, UniqueFd out) {
    // establish connection to an address in addrs
    // send the connected socket to out_socket

    // ...
}
```

Happy Eyeballs

```
UniqueFd epfd(epoll_create1(0));
set<UniqueFd, less<void>> connections;

epoll_event evt{
    .events = EPOLLHUP,
    .data = epoll_data{.fd = *out}};

epoll_ctl(epfd, EPOLL_CTL_ADD, *out, &evt);
```

Happy Eyeballs

```
auto react = [&](epoll_event event) -> bool {
    if (event.data.fd == *out) return true;

    auto it = connections.find(event.data.fd);
    if (event.events == EPOLLOUT) {
        send_fd(*out, **it);
        return true;
    }
    epoll_ctl(*epfd, EPOLL_CTL_DEL, **it, nullptr);
    connections.erase(it);
    return false;
};
```

Happy Eyeballs

```
// set up
for (sockaddr_in addr : addrs) {
    // connect

    if (auto event = wait(250))
        if (react(*event)) return;
}

while (!connections.empty())
    if (react(*wait(-1))) return;
```

Happy Eyeballs

Demo!

Happy Eyeballs++

```
void connect(span<sockaddr_in const> addrs, WriteHandle<FdHandle> out) {
    Select select;
    select.insert(out);
    select.modify(out, Events::hup);

    FdHandle timer(/* ... */);
    select.insert(timer);
    select.modify(timer, Events::in);

    set<FdHandle, less<void>> connections;

    // ...
}
```

Happy Eyeballs++

```cpp
auto react = [&](Select::Result result) -> bool {
    if (result.handle == timer) return false;
    if (result.handle == out) return true;
    auto it = connections.find(result.handle);
    if (result.events == Events::out) {
        out.write(move(connections.extract(it).value()));
        return true;
    } else {
        connections.erase(it);
        return false;
    }
};
```

Happy Eyeballs++

```
void connect(span<sockaddr_in const> addrs, WriteHandle<FdHandle> out) {
    // set up
    for (sockaddr_in addr : addrs) {
        // connect
        // set timer

        if (react(select.wait())) return;
    }

    // disable timer

    while (!connections.empty()) if (react(select.wait())) return;
}
```

Happy Eyeballs++

`select.wait()` is not a system call!

```
co_await select;
```

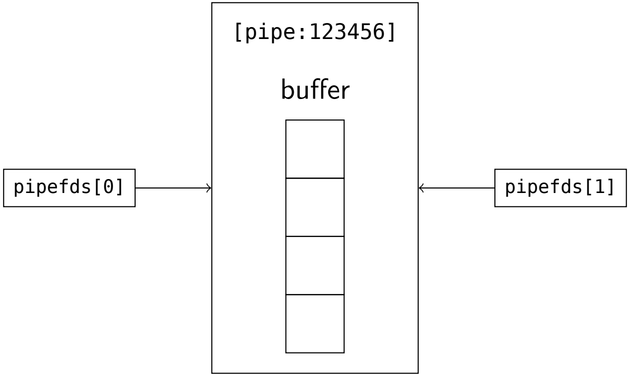
```
select.sender() | execution::then(react);
```

Pipe

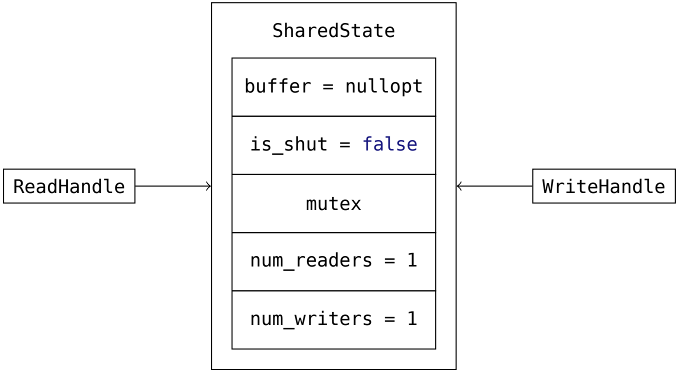
Pipe

```
int pipefds[2];

pipe(pipefds);
```

Pipe



Conceptual Pipe



Conceptual Pipe

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 Pipe

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 Pipe

`Success`:

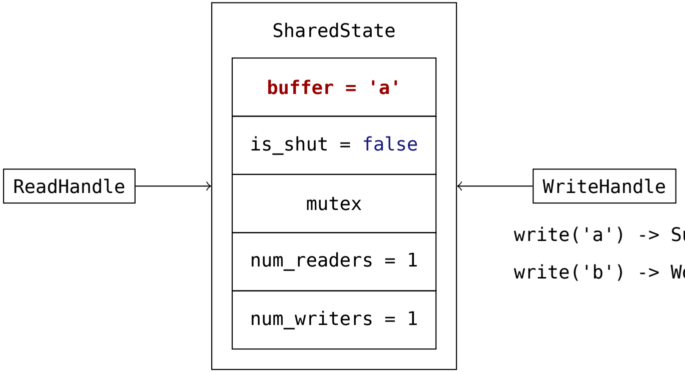
```
template <typename T>
struct Success { T value; };

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

Conceptual Pipe

```
struct EndOfFile {};

struct WouldBlock {};
```

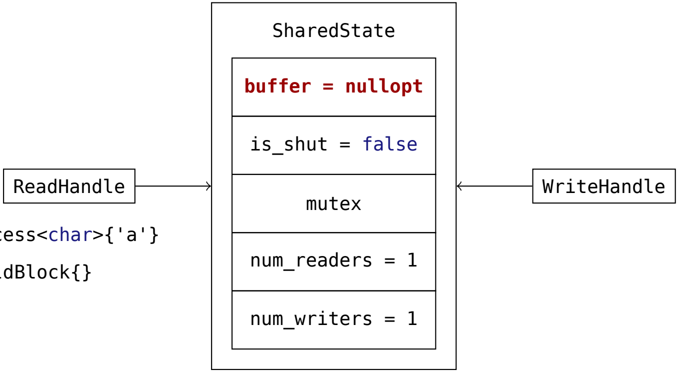
```
template <typename T>
using Result = variant<Succes<T>, EndOfFile, WouldBlock>;
```

Conceptual Pipe



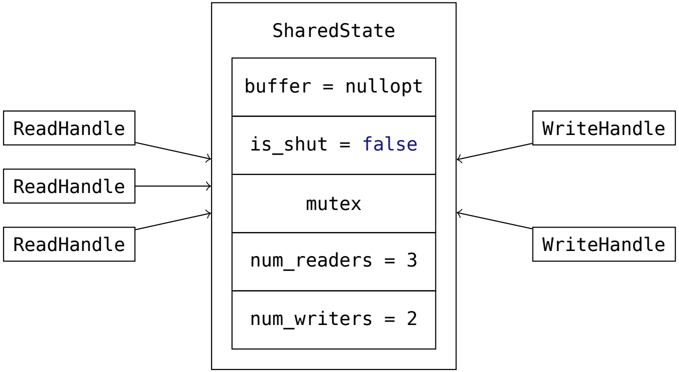
```
write('a') -> Success<void>{}
write('b') -> WouldBlock{}
```

Conceptual Pipe

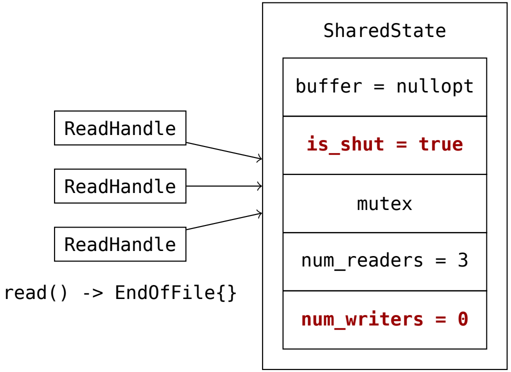


```
read() -> Success<char>{'a'}
read() -> WouldBlock{}
```

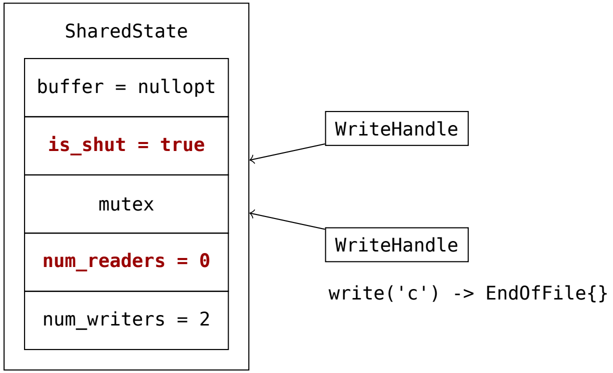
Conceptual Pipe



Conceptual Pipe



Conceptual Pipe



Conceptual Pipe

```
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 Pipe

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

    Result<char> read() const;
};

class WriteHandle {
    shared_ptr<SharedState> state;

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

Conceptual Pipe

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

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

Conceptual Pipe

```
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 Pipe

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

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

    if (is_shut) return EndOfFile{};

    return WouldBlock{};
}
```

Conceptual Pipe

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

    if (is_shut) return EndOfFile{};

    if (buffer) return WouldBlock{};

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

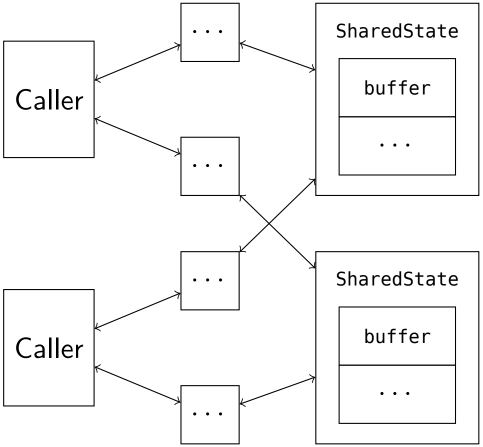
Conceptual Pipe

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

    is_shut = true;
}
```

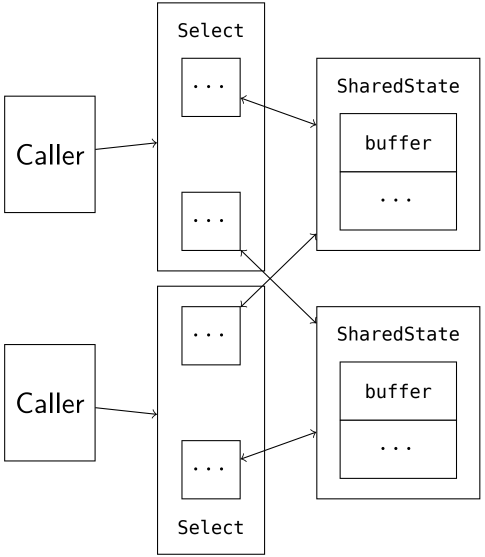
Implementing Select

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

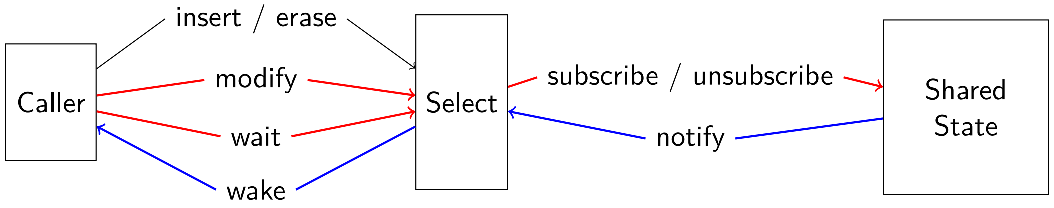
Implementing Select



Implementing Select



Implementing Select



Implementing Select

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

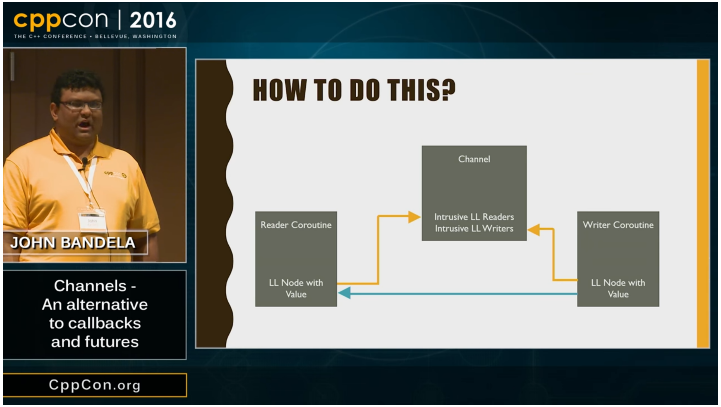
```
struct SharedState {
    multimap<Events, Select*>
    // ...
};
```

Implementing Select

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

```cpp
struct SharedState {
    // ...
    Links links;  // non-owning
};
```

Implementing Select



https://youtu.be/N3CkQu39j5I

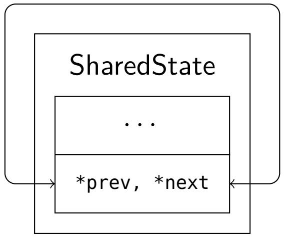
Implementing Select

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

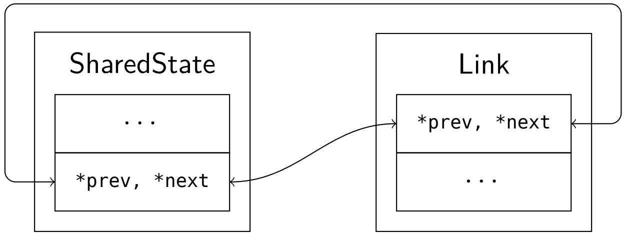
Implementing Select

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

Implementing Select

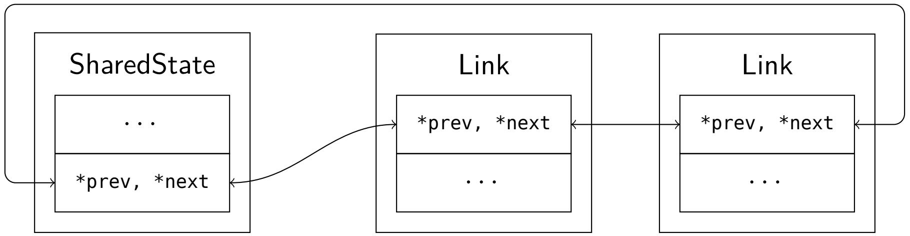


Implementing Select



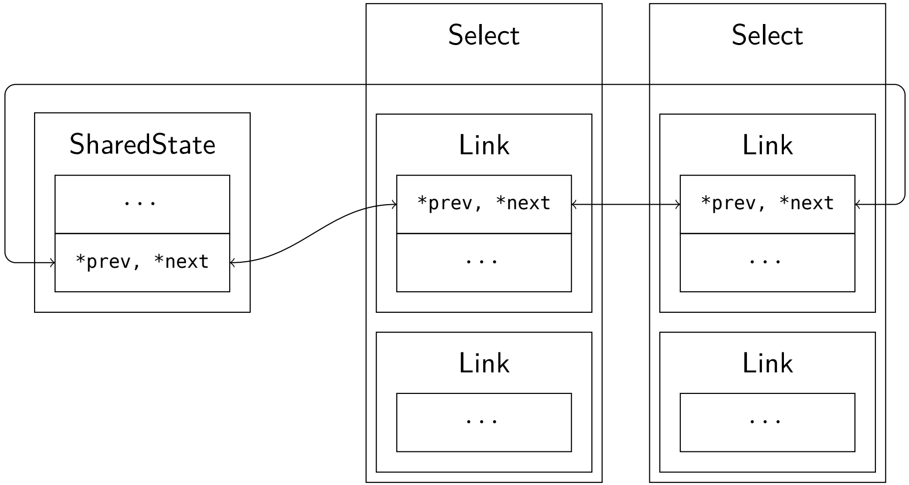
Subscribe

Implementing Select



Subscribe

Implementing Select



Implementing Select

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

    if (buffer) {
        if (!is_shut) links.extract(Events::out, notify);

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

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

Implementing Select

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

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

    links.extract(Events::in, notify);

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

Implementing Select

```
void SharedState::shut() {
    Notify notify;
    scoped_lock lock(mutex);

    links.extract(Events::hup, notify);

    is_shut = true;
}
```

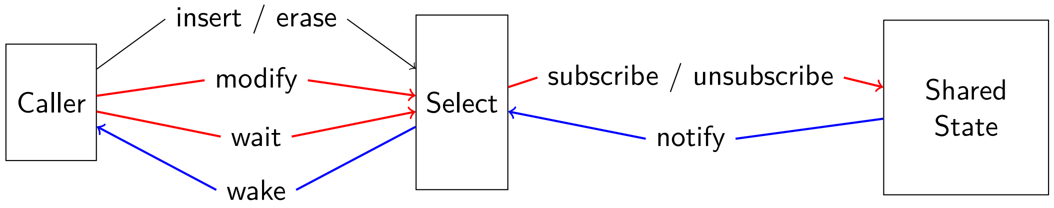
Implementing Select

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

    if (is_shut) {
        link.revents |= Events::hup;
        if ((link.events & Events::in) && buffer)
            link.revents |= Events::in;
    } else {
        if (buffer) {
            if (link.events & Events::in) link.revents |= Events::in;
        } else {
            if (link.events & Events::out) link.revents |= Events::out;
        }
    }

    if (link.revents) return false;

    link.subscribed = true;
    links.push(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.ListHead::unlink();
    return true;
}
```

Implementing Select



Implementing Select

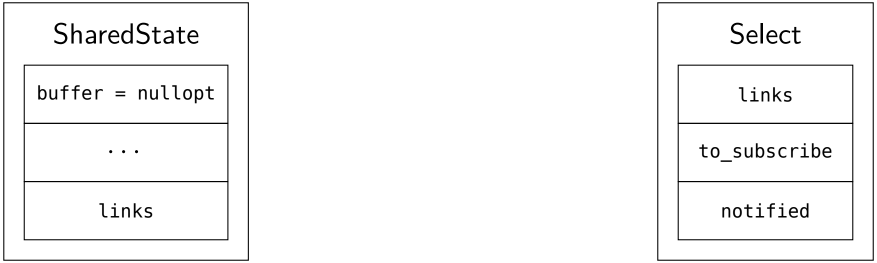
```cpp
int epfd = epoll_create1(0);

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);
```

```cpp
Select s;


s.insert(handle);


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

auto result = s.wait();

s.erase(handle);
```

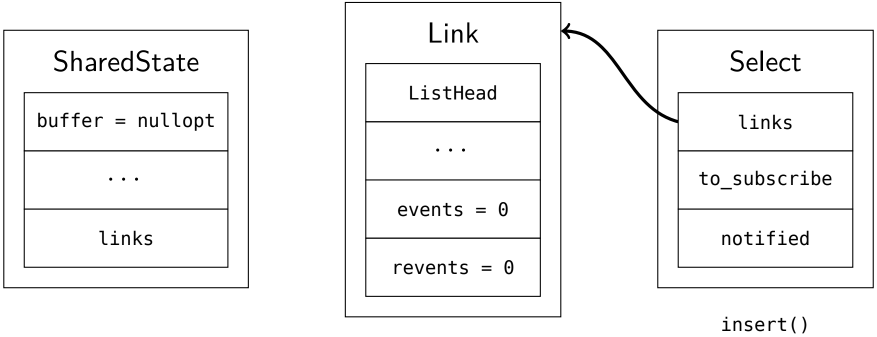
Implementing Select

```
class Select {
    set<Link> links;  // owning
    List<Link> to_subscribe;  // non-owning
    List<Link> notified;  // non-owning
    mutex mutex;
    condition_variable cond;

public:
    void insert(Handle);

    void erase(Handle) noexcept;

    void modify(Handle) noexcept;

    Result wait() noexcept;
};
```

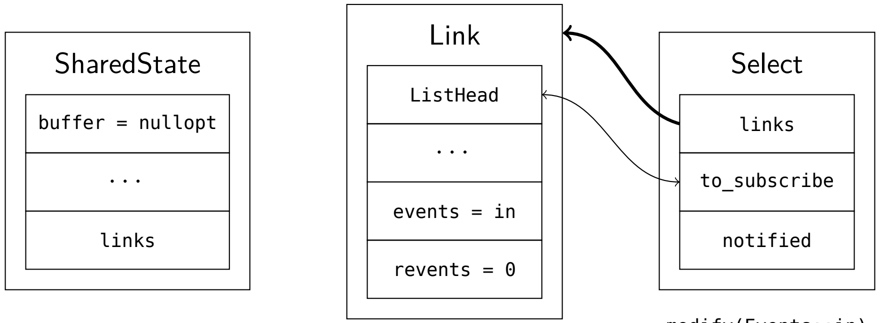
Implementing Select



Implementing Select

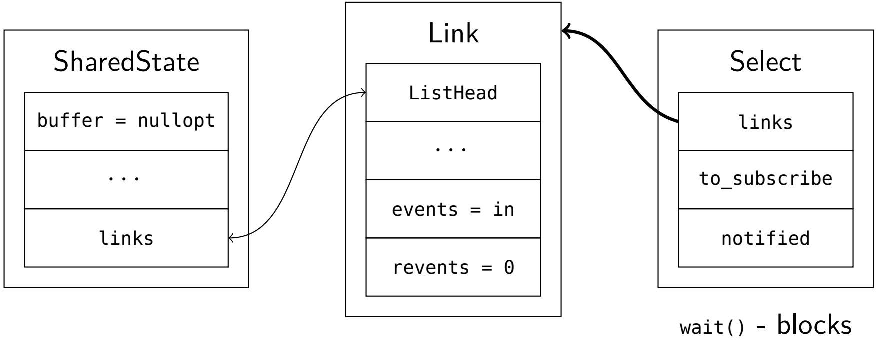


Implementing Select

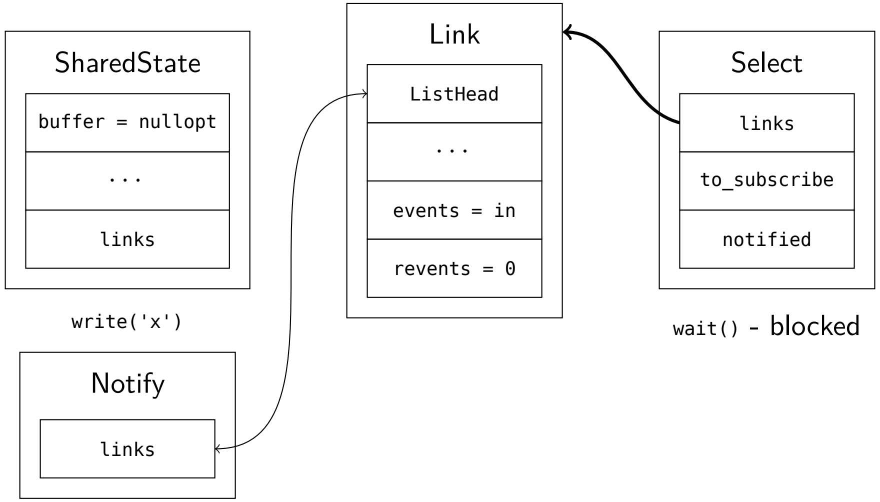


`modify(Events::in)`

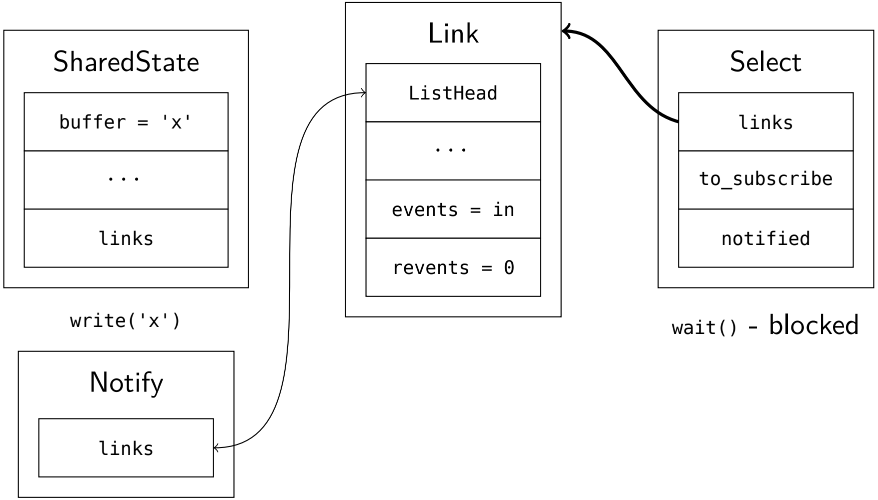
Implementing Select



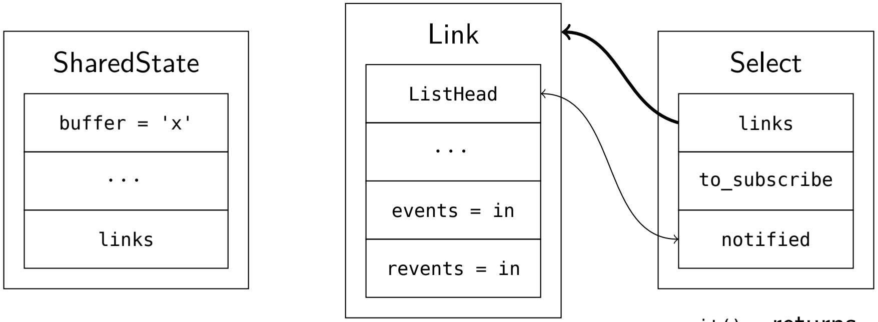
Implementing Select



Implementing Select

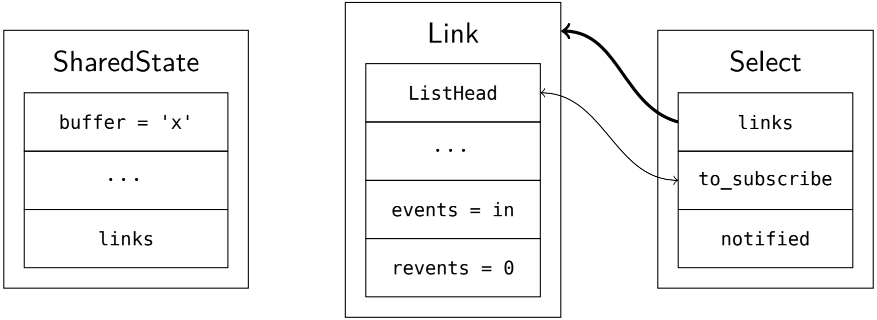


Implementing Select

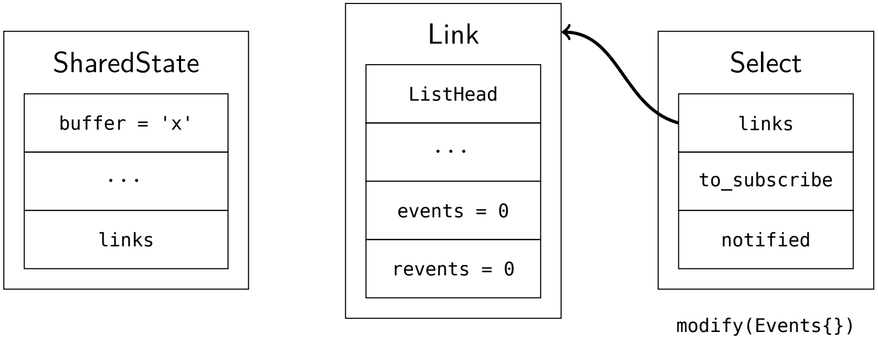


wait() - returns

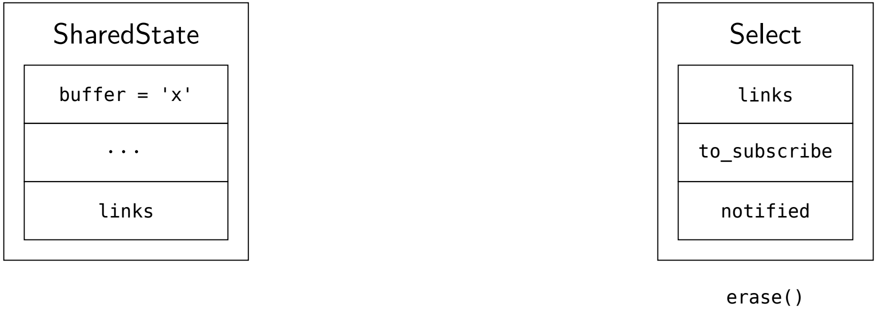
Implementing Select



Implementing Select



Implementing Select



Implementing Select

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

    links.emmplace(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()) {
    to_subscribe.append(*link);
    return {link->handle, exchange(link->revents, Events{})};
}

while (Link* link = to_subscribe.pop()) {
    if (!link->handle->subscribe(*link)) {
        to_subscribe.append(*link);
        return Result{
            .handle = link->handle,
            .events = 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);

        unlink(link, lock);
    }

    links.erase(iter);
}
```

Implementing Select

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

    {
        unique_lock lock(mutex);

        unlink(link, lock);

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

Implementing Select

```
void Select::unlink(Link& link, unique_lock& lock) {
    if (link.unsubscribe()) return;
    while (link.notifying) cond.wait(lock);
    link.ListHead::unlink();
}
```

Implementing Select

```cpp
void Links::extract(Events events, Notify& notify) {
    for (Link& link : this->links) {
        if (link.events & events) {
            link.revents = events;
            notify.links.push(link);
        }
    }
}
```

Implementing Select

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

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

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

        link->select.cond.notify_one();  // wake up select
    }
}
```

Implementing Select

```
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;
        }
    }
}
```

Implementing Select

```
void capitalize_select(ReadHandle in, WriteHandle out) {
    Select select;
    select.insert(in);
    select.insert(out);

    while (true) {
        // wait

        // react
    }
}
```

Implementing Select

```
// set up read
select.modify(in, Events::in | Events::hup);
select.modify(out, Events::hup);

{
    // wait
    Select::Result result = select.wait();

    // react
    if (result.handle == out) return;
    if (!(result.events & Events::in)) return;
}

char capital = toupper(get<Success<char>(in.read()).value);
```

Implementing Select

```
// set up write
select.modify(in, Events{});
select.modify(out, Events::out | Events::hup);

{
    // wait
    Select::Result result = select.wait();

    // react
    if (!(result.events & Events::out)) return;
}

if (out.write(capital) == EndOfFile{}) return;
```

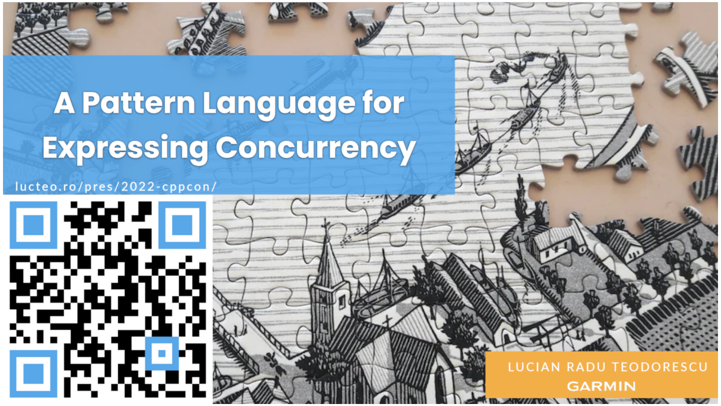
Enhancements

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

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

Selecting Senders

`https://youtu.be/xiaqNvqRB2E`

Selecting Senders

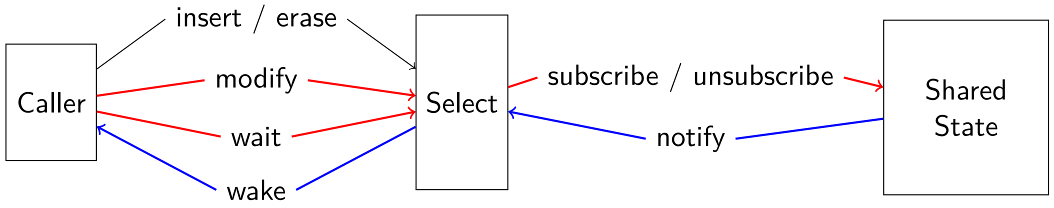

A Pattern Language for Expressing Concurrency

lucteo.ro/pres/2022-cppcon/

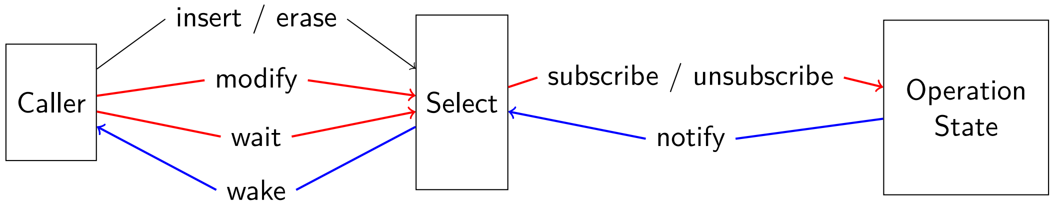
LUCIAN RADU TEODORESCU

GARMIN

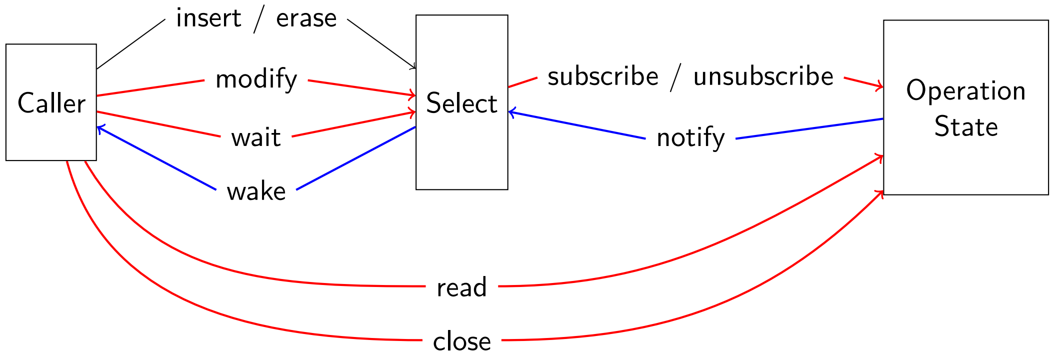
Selecting Senders



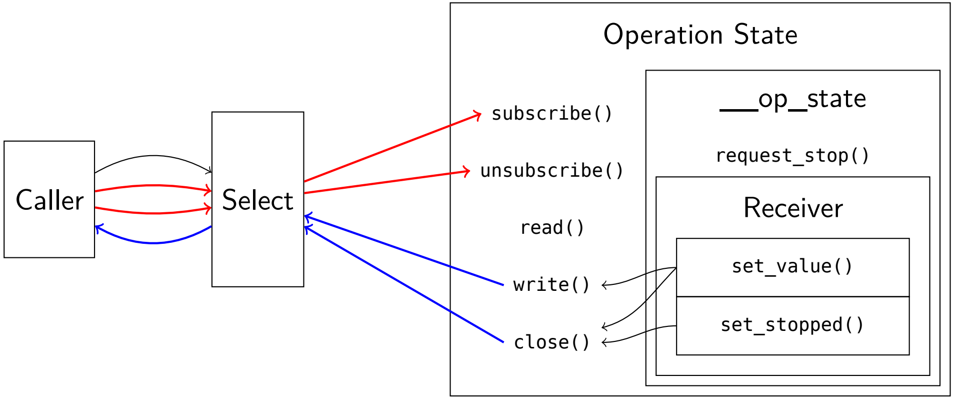
Selecting Senders



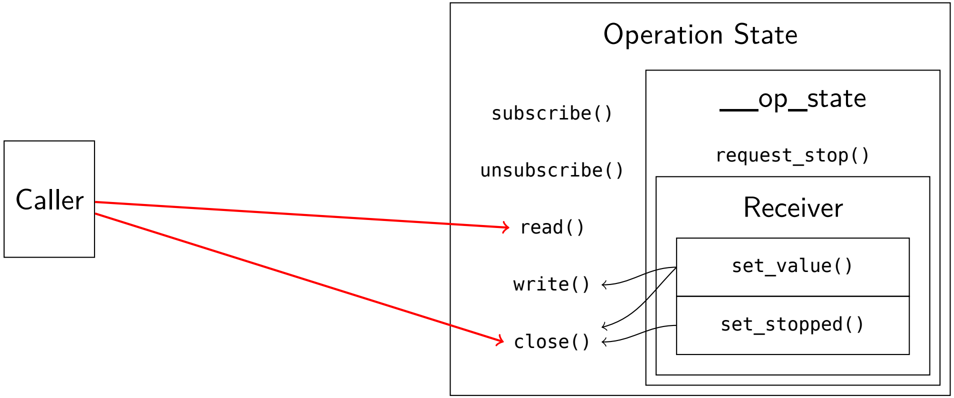
Selecting Senders



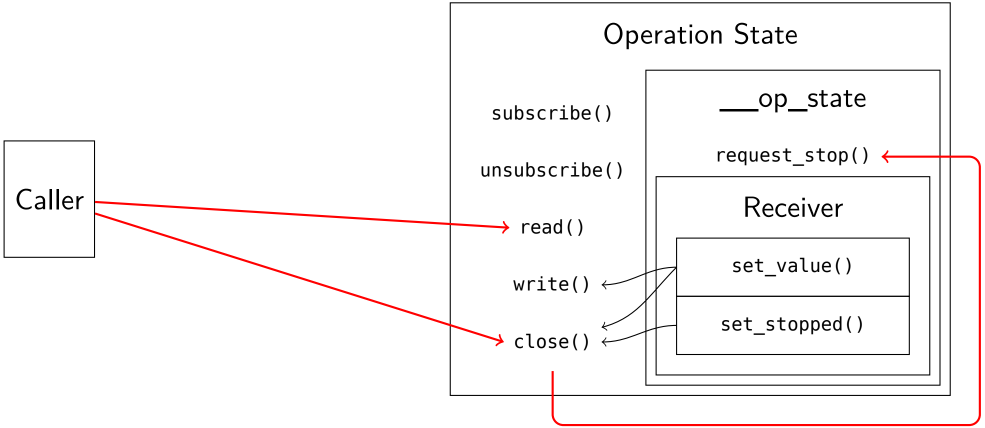
Selecting Senders



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;

    Sender 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()) 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

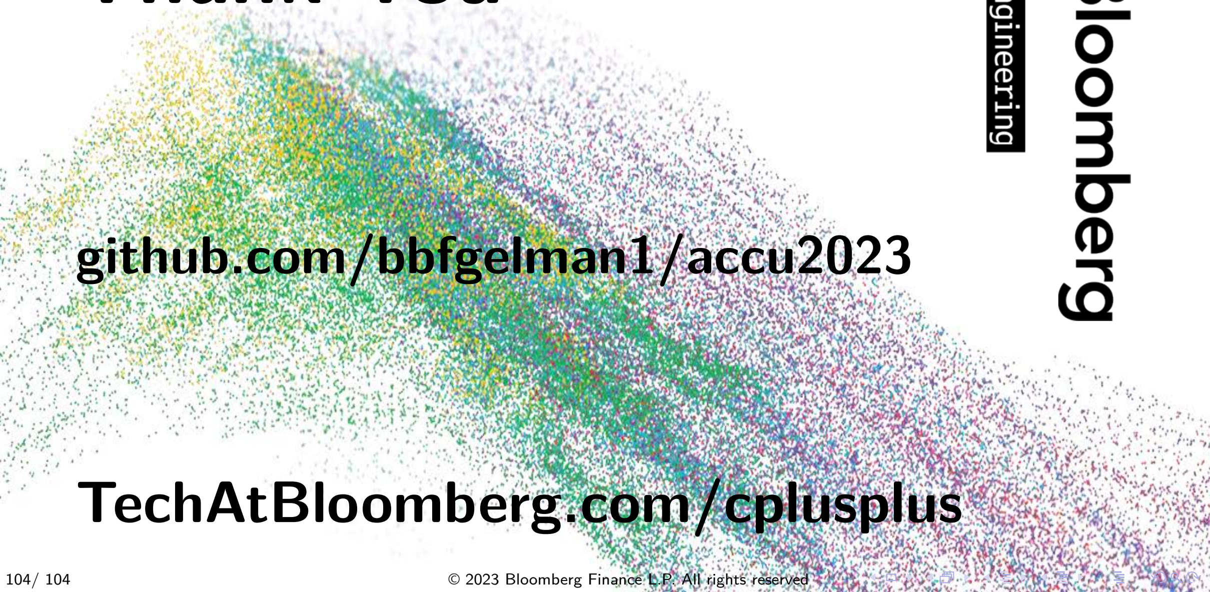
```
result = sync_wait(select.sender());
```

```
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



github.com/bbfgelman1/accu2023

TechAtBloomberg.com/cplusplus