accu 2022

STRUCTURED NETWORKING

DIETMAR KÜHL

Structured Networking

ACCU 2022
April 7, 2022

Dietmar Kühl
Senior Software Developer
dkuhl@bloomberg.net

Bloomberg Engineering

TechAtBloomberg.com

Objective

Provide a basis for networking:

Portable classes and functions for networking

Integrate networking with the concurrency approach

Note: the approach described is so far experimental!

- programs don’t control all progress
    - connections getting ready
    - data/requests being received
    - transfers completing

⇒ there is a lot of waiting

Networking TS vs. P2300

- The Networking TS has an approach dealing with concurrency
- P2300 (sender/receiver) addresses concurrency differently
  - P2300 is pursued as the general concurrency approach
  - Here networking integrated with P2300 is described

Structured Concurrency

1. decompose work into senders each representing work
2. combine the work representation with a receiver
3. start the resulting entity

TechAtBloomberg.com

Bloomberg

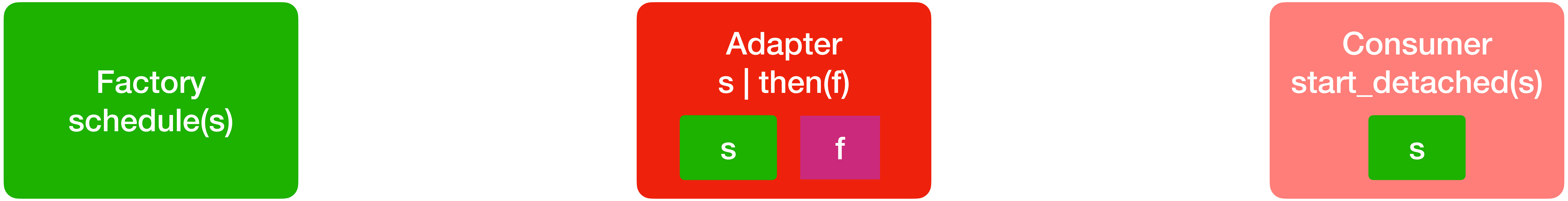
Engineering

```
thread_pool      p(...);
scheduler auto shed = p.scheduler();

sender auto s = when_all(
        schedule(shed) | then([]{ return frob(); }) | then([](auto x){ return borf(x); }),
        schedule(shed) | then([]{ return compute(); }));

auto[x, y] = sync_wait(s);
```

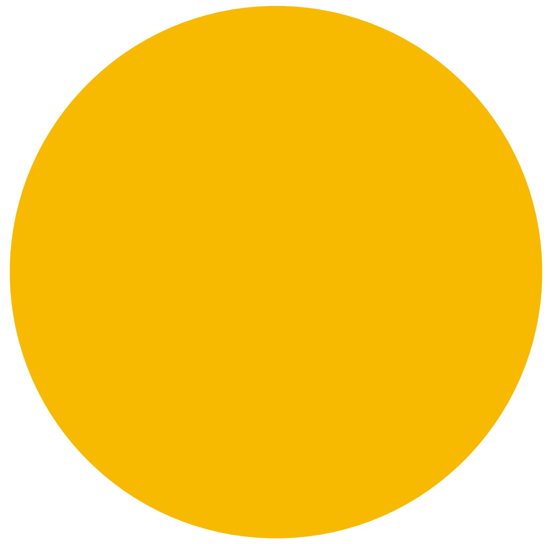
Sender: Description of Work



connect(sender, receiver) -> operation_state

get_completion_scheduler(sender) -> scheduler

Receiver: Destination for Results



get_env(receiver) -> env

get_stop_token(env)

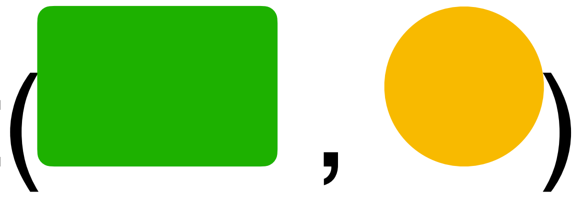
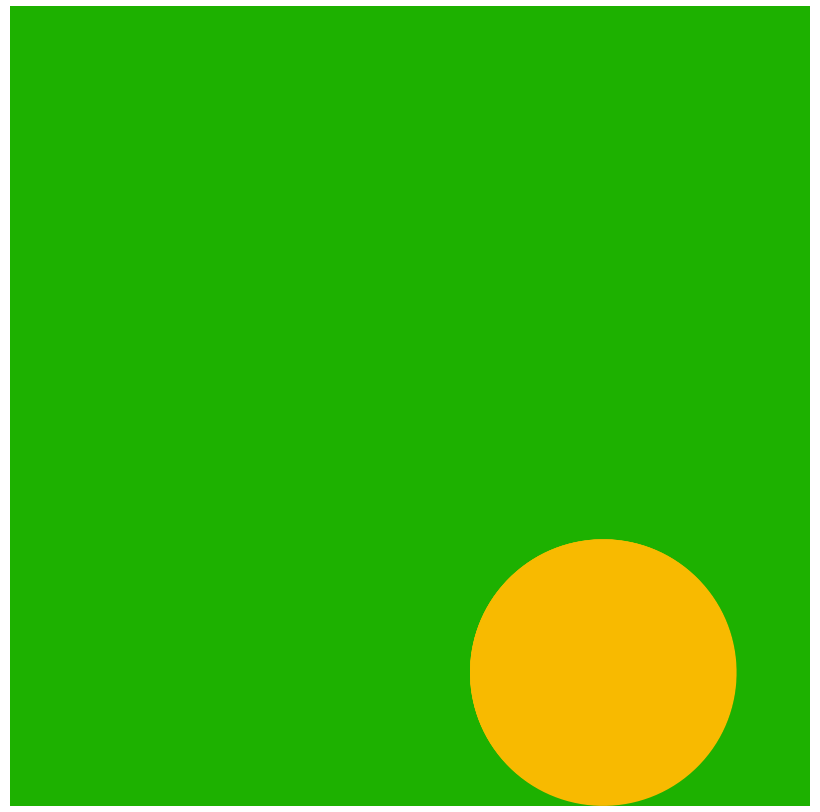
get_allocator(env)

set_value(receiver, results…)

set_error(receiver, error)

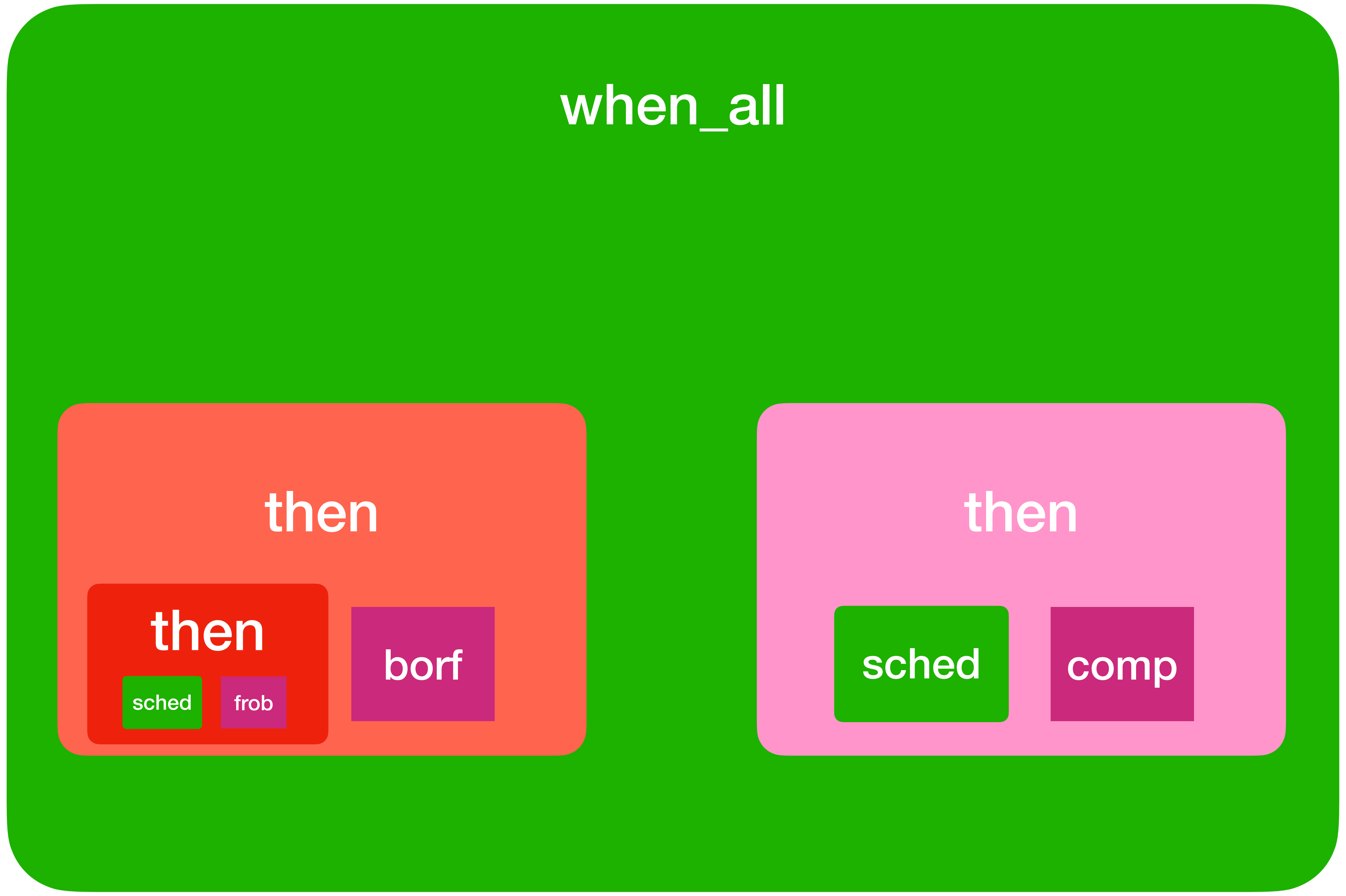
set_stopped(receiver)

Operation State: Ready to Execute Task

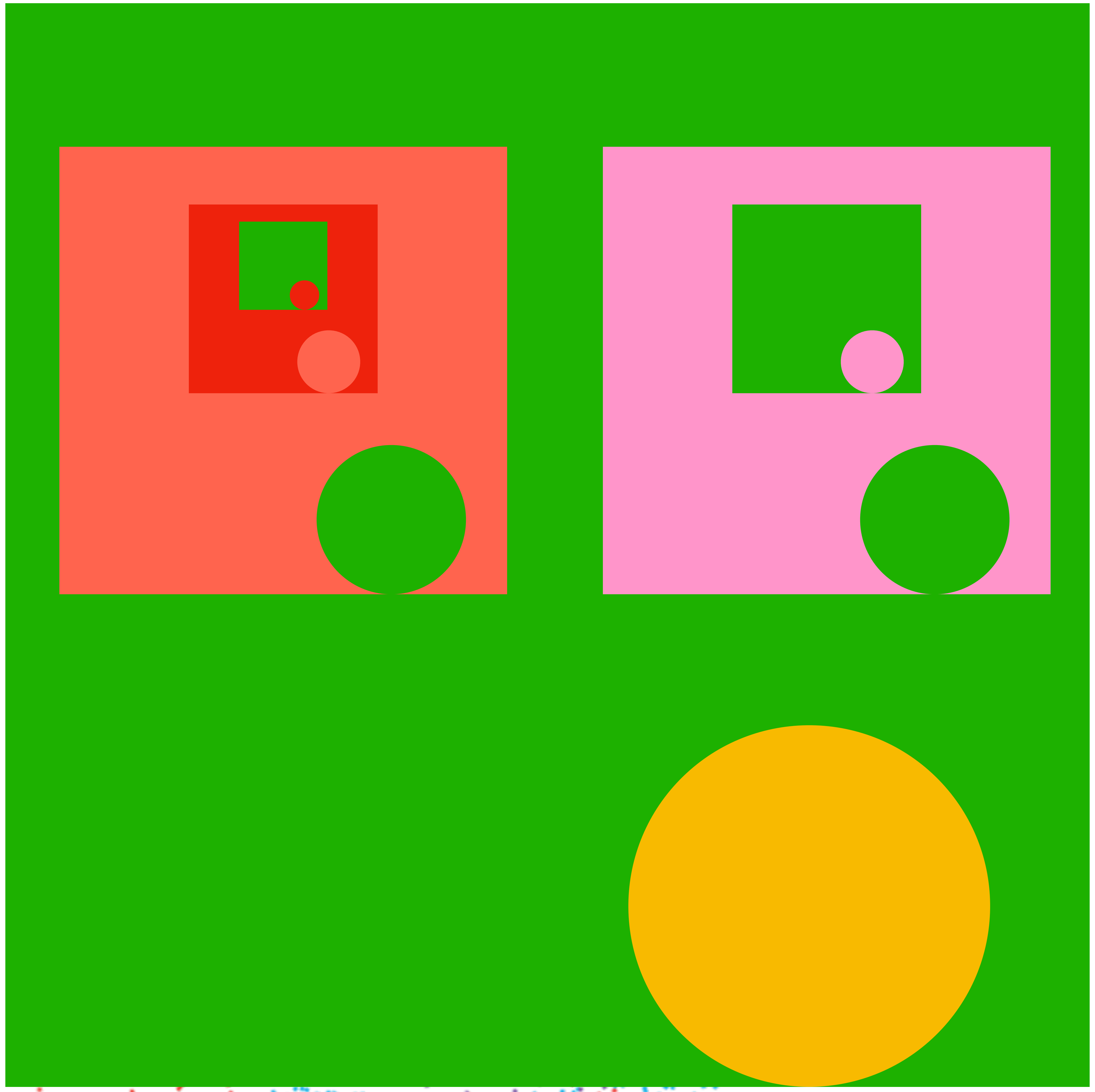
connect() ⇒ 

start(operation_state) ⇒ call one of the completions

Sender



Operation State



Various Sender Algorithms

- Continuation: then(s, fun), let_value(s, fun)
- Fork/join: split(s), when_all(s…)
- Parallel execution: bulk(s, size, fun)
- Control scheduler: transfer(s, scheduler), on(scheduler, s)
- Rewrite Result: upon_error(s, fun), upon_stopped(s, fun)

- Transformation to operation states allows specialization
  - The nested algorithms are statically known
  - Operations could get fused
  - Use more effective approach for known setups

Senders Can Be Awaitables

- set_value(arg…) becomes co_await result
- set_error(e) becomes an exception thrown from co_await
- set_stopped() terminates the entire coroutine

Coroutines Can Be Senders

- co_yield/co_return become set_value(a…)
- An exception becomes set_error(e) or set_stopped()
- Coroutines can be cancelled by destroying them

TechAtBloomberg.com

Bloomberg Engineering

Senders For I/O

- async_connect, async_accept, async_open
- async_readsome, async_writesome
- async_send, async_receive
- async_wait

Example: Echo Server

- Accept clients on a fixed port
- Any text is just sent back to the client
- Using just one thread
- … and few allocations
- An echo server is akin to a request/response server

```
int main() {
    io_context          context;
    socket_acceptor  server(endpoint(address_v4::any(), 12345));

    sender auto sndr =
            schedule(context.scheduler())
          | async_accept(server)
          | then([&](auto, auto){ cout << “accept completed\n”; })
        ;

    run(context, move(sndr));
}
```

TechAtBloomberg.com

Bloomberg

Engineering

```
int main() {
    io_context          context;
    socket_acceptor  server(endpoint(address_v4::any(), 12345));

    run(context,
            schedule(context.scheduler())
          | async_accept(server)
          | then([&](auto, auto){ cout << “accept completed\n”; })
        );


}
```

TechAtBloomberg.com

Bloomberg

Engineering

```
run(context,
        schedule(context.scheduler())
      | async_accept(server)
      | then([&](auto, auto){ cout << "accept completed\n"; })
   );
```


Bloomberg
Engineering

```
run(context,
    repeat_effect(
        schedule(context.scheduler())
      | async_accept(server)
      | then([&](auto, auto){ cout << “accept completed\n”; })
    ));
```

TechAtBloomberg.com

Bloomberg

Engineering

```
run(context,
    repeat_effect(
        schedule(context.scheduler())
        | async_accept(server)
        | then([&](error_code ec, stream_socket){
            if (!ec) cout << “client connected\n”;
        })
    ));
```

TechAtBloomberg.com

Bloomberg

Engineering

```
run(context,
    repeat_effect(
        schedule(context.scheduler())
        | async_accept(server)
        | then([&](error_code ec, stream_socket stream){
            if (!ec) run_client(context, move(stream));
        })
    ));
```

```
struct client {
    stream_socket stream;
    char          buffer[1024];
    bool          done = false;

    client(stream_socket&& stream): stream(move(stream)) {}
    client(client&& other): stream(move(other.stream)) {}
};
```

TechAtBloomberg.com

```
struct client {
    stream_socket stream;
    char                 buffer[1024];
};
void run_client(io_context& context, stream_socket&& stream) {
    client c(move(stream));

        schedule(context.scheduler())
        | async_read_some(c.stream, buffer(c.buffer))
        | then([](int n){ cout << “read=“ << n << “\n”; }
        ;
}
```

TechAtBloomberg.com

Bloomberg Engineering

```
struct client {
    stream_socket stream;
    char                    buffer[1024];
};
void run_client(io_context& context, stream_socket&& stream) {
    client c(move(stream));
    start_detached(
        schedule(context.scheduler())
        | async_read_some(c.stream, buffer(c.buffer))
        | then([](int n){ cout << “read=“ << n << “\n”; }
        );
}
```

```
struct client {
    stream_socket stream;
    char              buffer[1024];
};
void run_client(io_context& context, stream_socket&& stream) {
    client c(move(stream));
    start_detached(
        schedule(context.scheduler())
        | async_read_some(c.stream, buffer(c.buffer))
        | then([](int n){ cout << “read=“ << n << “\n”; }
        );
}
```

```
struct client {
    stream_socket stream;
    char          buffer[1024];
};
void run_client(io_context& context, stream_socket&& stream) {
    client c(move(stream));
    start_detached(
        schedule(context.scheduler())
        | async_read_some(c.stream, buffer(c.buffer))
        | then([](int n){ cout << “read=“ << n << “\n”; }
    );
}
```

TechAtBloomberg.com

Bloomberg
Engineering

```
struct client {
    stream_socket stream;
    char                    buffer[1024];
};
void run_client(io_context& context, stream_socket&& stream) {
    client c(move(stream));
    start_detached(
        schedule(context.scheduler())
        | async_read_some(c.stream, buffer(c.buffer))
        | then([](int n){ cout << “read=“ << n << “\n”; }
        );
}
```

```
void run_client(io_context& context, stream_socket&& stream) {




    client c(move(stream));
    start_detached(
        schedule(context.scheduler())
        | async_read_some(c.stream, buffer(c.buffer))
        | then([](int n){ cout << “read=“ << n << “\n”; }
        );
}
```

TechAtBloomberg.com

Bloomberg

Engineering

```
void run_client(io_context& context, stream_socket&& stream) {
    start_detached(



                  schedule(context.scheduler())
                | async_read_some(c.stream, buffer(c.buffer))
                | then([&c](int n){
                      cout << “read=“ << n << “\n”;
                })

          });
}
```

TechAtBloomberg.com

Bloomberg
Engineering

```
void run_client(io_context& context, stream_socket&& stream) {
    start_detached(
        just() | let_value([&, c=client(move(stream))]() mutable {
            return
                schedule(context.scheduler())
                | async_read_some(c.stream, buffer(c.buffer))
                | then([&c](int n){
                    cout << "read=" << n << "\n";
                })
                ;
        });
}
```

TechAtBloomberg.com

Bloomberg Engineering

```
void run_client(io_context& context, stream_socket&& stream) {
    start_detached(
        just() | let_value([&, c=client(move(stream))]() mutable {
            return repeat_effect(
                    schedule(context.scheduler())
                    | async_read_some(c.stream, buffer(c.buffer))
                    | then([&c](int n){
                        cout << “read=“ << n << “\n”;
                    })
                    );
        });
}
```

TechAtBloomberg.com

Bloomberg Engineering

```
void run_client(io_context& context, stream_socket&& stream) {
    start_detached(
            just() | let_value([&, c=client(move(stream))]() mutable {
                return repeat_effect_until(
                        schedule(context.scheduler())
                        | async_read_some(c.stream, buffer(c.buffer))
                        | then([&c](int n){
                            cout << "read=" << n << "\n"; c.done = n <= 0;
                        }),
                        [&c]{ return c.done; });
            });
}
```

TechAtBloomberg.com

```
void run_client(io_context& context, stream_socket&& stream) {
    start_detached([&]()->task {
        client c(move(stream));
        while (true) {
            int n = co_await schedule(context.scheduler())
                          | async_read_some(c.stream, buffer(c.buffer));
            cout << "read=" << n << "\n";
            if (n <= 0)
                break;
        };
}
```

TechAtBloomberg.com

Bloomberg

Engineering

Reading and Writing

- Reading and writing are independent:
  - Alternating reading and writing would be bad
  - Either can make progress while the other does not
- Reader pushes messages into a queue for the writer
- when_all(sender…) waits until multiple senders are done

```
void run_client(io_context& context, stream_socket&& stream) {
    start_detached(
            just() | let_value([&, c=client(move(stream))]() mutable {
                return when_all(
                    make_reader(context, c),
                    make_writer(context, c)
            });
}
```

```
auto make_reader(io_context& context, client& stream) {
    return repeat_effect_until(

                        schedule(context.scheduler())
                      | async_read_some(c.stream, c.next_read_buffer())
                      | then([&c](int n){
                            if (0 < n) c.add_read(n);
                            else c.done = true;
                        })

            ,
        [&c]{ return c.done; })
        | then([&c](auto&&...){ c.stop(); });
}
```

TechAtBloomberg.com

Bloomberg

Engineering

```
auto make_reader(io_context& context, client& stream) {
    return repeat_effect_until(

                          schedule(context.scheduler())
                      | async_read_some(c.stream, c.next_read_buffer())
                      | then([&c](int n){
                            if (0 < n) c.add_read(n);
                            else c.done = true;
                         })
          ,
        [&c]{ return c.done; })
        | then([&c](auto&&...){ c.stop(); });
}
```

TechAtBloomberg.com

Bloomberg Engineering

```
auto make_reader(io_context& context, client& stream) {
    return repeat_effect_until(
        just() | let_value([&]{
            return schedule(context.scheduler())
                   | async_read_some(c.stream, c.next_read_buffer())
                   | then([&c](int n){
                          if (0 < n) c.add_read(n);
                          else c.done = true;
                      });
        }),
        [&c]{ return c.done; })
        | then([&c](auto&&...){ c.stop(); });
}
```

```
struct client {
    static constexpr std::uint64_t mask = 0x3ff;
    char buffer[mask + 1];
    uint64_t readpos = 0; uint64_t writepos = 0;

    mutable_buffer next_read_buffer() {
        auto begin = readpos & mask;
        return mutable_buffer(begin, sizeof(buffer) - begin);
    }
    void add_read(int n) {
        readpos += n;
    }
};
```

- The read buffer is ready waiting for something to write into
- The write buffer needs to wait for something to be read
- Getting the write buffer is just a sender!

```
auto make_writer(io_context& context, client& c) {
    return repeat_effect_until(
        c.next_write_buffer()
        | let_value([&](const_buffer b) {
            return schedule(context.scheduler())
                | async_write(c.stream, b)
                ;
          },
          [&c]{ return c.done; }
    );
}
```

TechAtBloomberg.com

Bloomberg

Engineering

```
struct client {
    template <receiver Receiver> struct state;
    struct sender {
        using completion_signatures = EX::completion_signatures<
            set_value_t(const_buffer), set_stopped_t()>;
        client* c;
        template <receiver R>
        friend auto tag_invoke(connect_t, sender const& self, R&& r) {
            return state<R>(self.c, std::forward<R>(r));
        }
    };
    sender next_write_buffer() { return sender{this}; }
};
```

```
struct state_base { virtual void complete() = 0; };
template <receiver Receiver>
struct state: state_base {
    client*                          c;
    remove_cvref_t<Receiver> r;
    template <receiver R>
    state(client* c, R&& r): c(c), r(forward<R>(r)) {}
    friend void tag_invoke(start_t, state& self) {
        if (self.c->readpos != self.c->writepos) self.complete();
        else self.c->completion = &self;
    }
    void complete() override { …. }
};
```

TechAtBloomberg.com

Bloomberg

Engineering

```
struct state_base { virtual void complete() = 0; };
template <receiver Receiver>
struct state: state_base {
  remove_cvref_t<Receiver> r;
  client*                  c;
  void complete() override {
    size_t begin = c->writepos & mask;
    size_t size = min(c->readpos - c->writepos,
                      sizeof(c->buffer) - begin);
    writepos += size;
    set_value(move(r), const_buffer(c->buffer + begin, size));
  }
};
```

```
struct client {
    struct sender_base;
    static constexpr std::uint64_t mask = 0x3ff;
    char buffer[mask + 1];
    uint64_t readpos = 0; uint64_t writepos = 0;
    sender_base* completion = nullptr;
    void add_read(int n) {
        readpos += n;
        If (completion)
            exchange(completion, nullptr)->complete();
    }
};
```

Echo Server Summary

- There is one allocation when a new client is added
- There is a tiny bit of lifetime management
- As is, there is a bit of trivial code missing to shutdown the client
- The read buffer side should also be a sender

Thank you!

We are hiring: http://bloomberg.com/engineering

TechAtBloomberg.com

Resources

- http://wg21.link/p2300: sender/receiver propsal
- http://wg21.link/n4734: Networking TS
- https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/: Structured Concurreny
- https://github.com/dietmarkuehl/kuhllib: sender/receiver + networking implementation (examples on branch accu-2022)

TechAtBloomberg.com

Bloomberg

Engineering