vollmann engineering gmbh
>

IoT Minimized

Networking with C++OS on Raspberry Pi Pico W

emBO++

March 2024

Detlef Vollmann
vollmann engineering gmbh

vollmann engineering gmbh
>

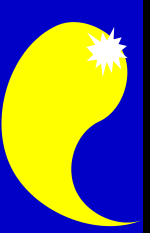
IoT Minimized

Networking with C++OS on Raspberry Pi Pico W

Detlef Vollmann
vollmann engineering gmbh
Luzern, Switzerland

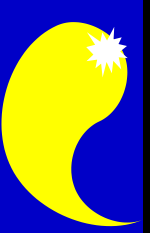
dv@vollmann.ch
http://www.vollmann.ch/

Currently looking for a new project



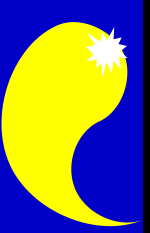
Overview

- Asynchronicity
- Sender/Receiver
- Networking API
- Example
- Implementation
- References



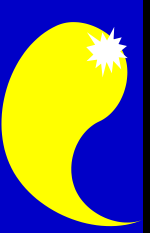
Networking and Concurrency

- Networking generally means concurrency
  - multiple network connections
  - network communication concurrent to other tasks



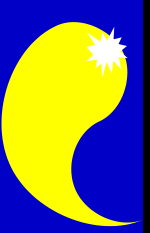
Synchronous Networking

- Multi-tasking OS allows for synchronous networking
- Own task/thread for each connection



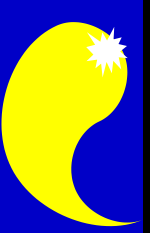
Asynchronous Networking

- Asynchronous networking allows multiple connections inside same task
- C++ committee discussed/discusses multiple asynchronous network models
- ASIO as base for Networking TS
  - callback based
  - may never be standardized



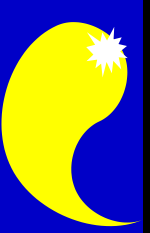
Sender/Receiver Networking

- Sender/receiver provide a generic asynchronous event model
- Dietmar Kühl proposes a networking API based on sender/receiver



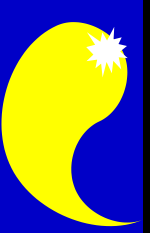
Coroutines

- Coroutines are a C++ mechanism for asynchronicity
- There's coroutine support in ASIO and sender/receiver



Overview

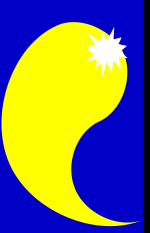
- Asynchronicity
- Sender/Receiver
- Networking API
- Example
- Implementation
- References



Sender / Receiver

```
sender auto s
  = just(3) |                    // produce '3' immediately
    on(scheduler1) |                 // transition context
    transform([](int a){return a+1;}) | // chain continuation
    transform([](int a){return a*2;}) | // another continuation
    on(scheduler2) |                 // transition context
    let_error([](auto e)
        {return just(3);}); // with default value on errors
int r = sync_wait(s);    // wait for the result
```

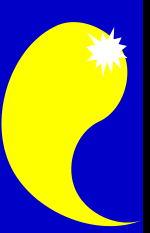
- Asynchronous STL
  - sender stores work to do
  - receiver collects the result
  - factories only provide sender
  - consumers only provide receiver
  - algorithms provide both



Sender / Receiver

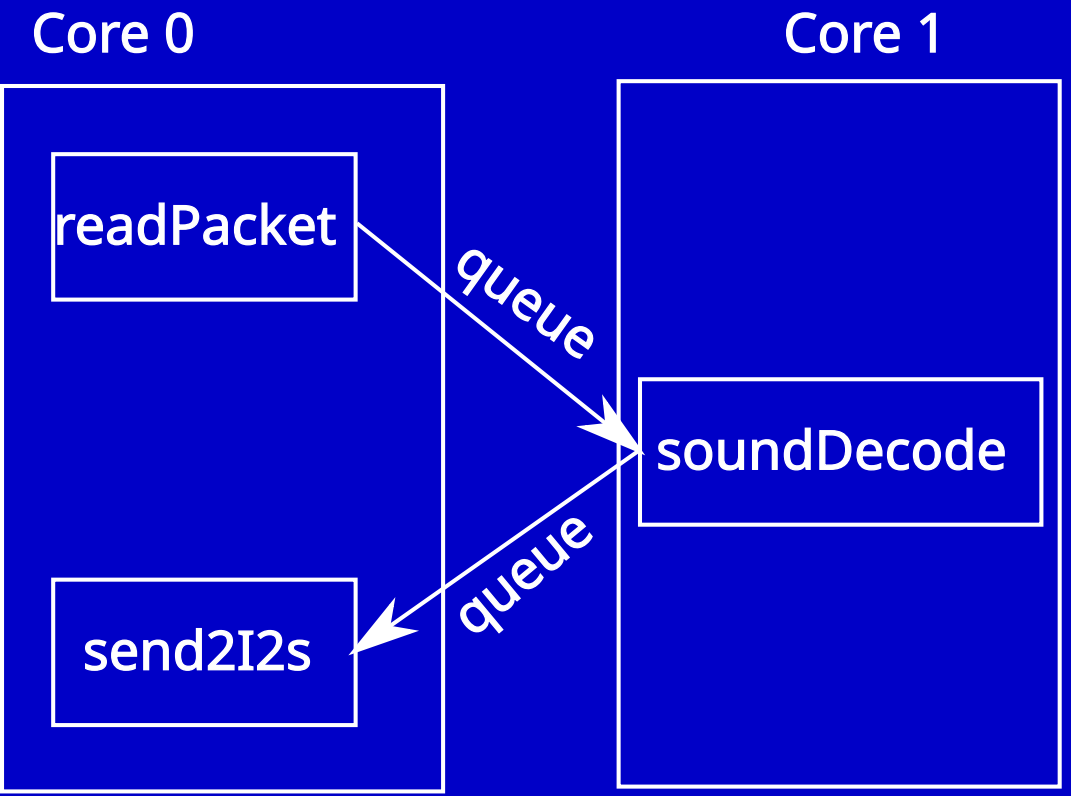
- Events are senders
- Handlers are receivers
- Schedulers run the work on execution contexts

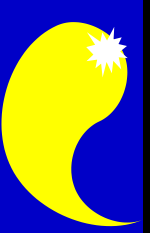
```
Timer<TIM1> t;
ExecContext<0> ctx;
sender auto work =
    on(ctx.scheduler<22, 300>(), t.async_wait(200ms))
        | then([] { sendHeartBeat(); });

sender auto loop = repeat_effect(work);

start_detached(loop);
```

Examples

- Wireless speaker
  - receive sound stream from WiFi
  - decode it
  - send to I2S
  - dual core

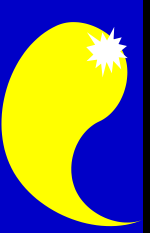




Sender / Receiver

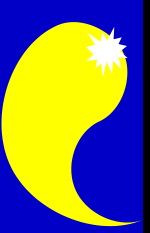
- Arbitrarily long event chains are possible

```
sender auto work =
    srv.readPacket()
        | wireQ.async_push()
        | transfer(sched1)
        | then(wireQ.async_pop())
        | then(decode)
        | pcmQ.async_push()
        | transfer(io)
        | then(pcmQ.async_pop)
        | then(sendI2sChunk);
```

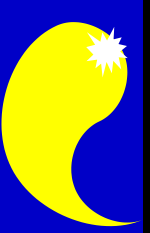
Overview

Asynchronicity
Sender/Receiver
Networking API
Example
Implementation
References



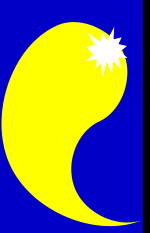
Network Addresses

- IPv4: `ip::address_v4`
- IPv6: `ip::address_v6`
- `ip::address` essentially `std::variant` of the above



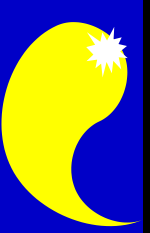
Sockets

- TCP: `ip::tcp::socket` ⇔ `basic_stream_socket<ip::tcp>`
- UDP: `ip::udp::socket` ⇔ `basic_datagram_socket<ip::udp>`
- TCP listening: `ip::tcp::acceptor` ⇔ `basic_socket_acceptor<ip::tcp>`



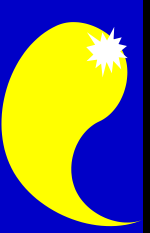
Synchronous Functions

- `send()`, `receive()`
- `read_some()`, `write_some()`
- `connect()`, `bind()`, `open()`, `close()`



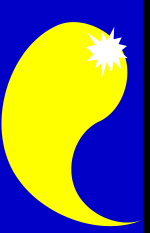
Synchronous Functions

```
socket_acceptor server(endpoint(address_v4::any(), 12345));

while (true)
{
    stream_socket client{server.accept()};
    serialOut("Client connected\n");
    while (true)
    {
        size_t n = client.read_some(buffer(buf));
        serialOut("Got bytes\n");
        if (n == 0) break;

        client.write_some(buffer(buf, n));
    }
    serialOut("Client done\n");
}
```

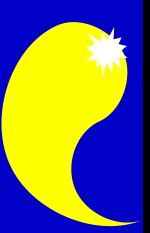
Asynchronous Functions

- `async_send()`, `async_receive()`
- `async_read_some()`, `async_write_some()`
- `async_connect()`, `async_accept()`



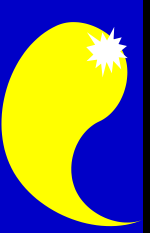
Asynchronous Functions

```
socket_acceptor server(endpoint(NI::address_v4::any(), 12345));

auto readWrite =
    NN::async_read_some(client, NN::buffer(netBuf))
    | EX::then([](size_t n)
               {
                   serialOut("Bytes got:");
                   LINT(n);

                   stop = n == 0;
                   return n;
               })
    | EX::let_value([] (size_t n)
                    {
                        return EX::on(io.scheduler(),
                                      NN::async_write(client, NN::buffer(netBuf, n)))
                    });
```



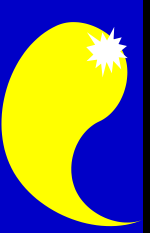
Asynchronous Functions

```
auto echoLoop =
    EX::repeat_effect_until(
        EX::on(io.scheduler(), readWrite),
        [] { return stop; });
auto acceptLoop =
    EX::on(io.scheduler(), NN::async_accept(server))
        | EX::then([&](stream_socket&& s, auto&& endpoint) {
                    serialOut("Client connected");
                    client = std::move(s);
                    EX::start_detached(echoLoop);
                    serialOut("Done");
                });
while (true) {
    EX::run(io, acceptLoop);
    io.run();
    myprintf("Client done");
}
```

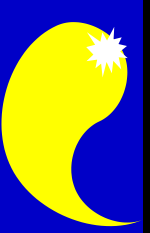
Asynchronous Functions w/ Coroutines

```
socket_acceptor server(endpoint(NI::address_v4::any(), 12345));

auto clientLoop =
  [&] () -> cppos::CoSender
  {
    myprintf("Starting CoSender");
    auto [client, ep] = co_await EX::on(io.scheduler(),
                                        NN::async_accept(server));
    myprintf("Client connected");
```

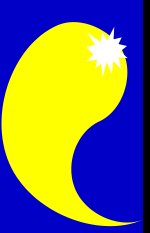
Asynchronous Functions w/ Coroutines

```
while (true)
{
  auto [n] = co_await EX::on(io.scheduler(),
                             NN::async_read_some(client,
                                                 NN::buffer(netBuf)));
  myprintf("Bytes got:");
  LINT(n);

  if (n == 0) break;

  co_await EX::on(io.scheduler(),
                  NN::async_write_some(client, NN::buffer(netBuf, n)));
}

};
```

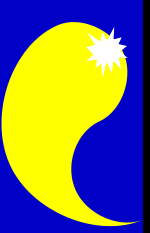



Asynchronous Functions w/ Coroutines

```
while (true)
{
  EX::run(io, clientLoop());
  io.run();
  myprintf("Client done");
}
```

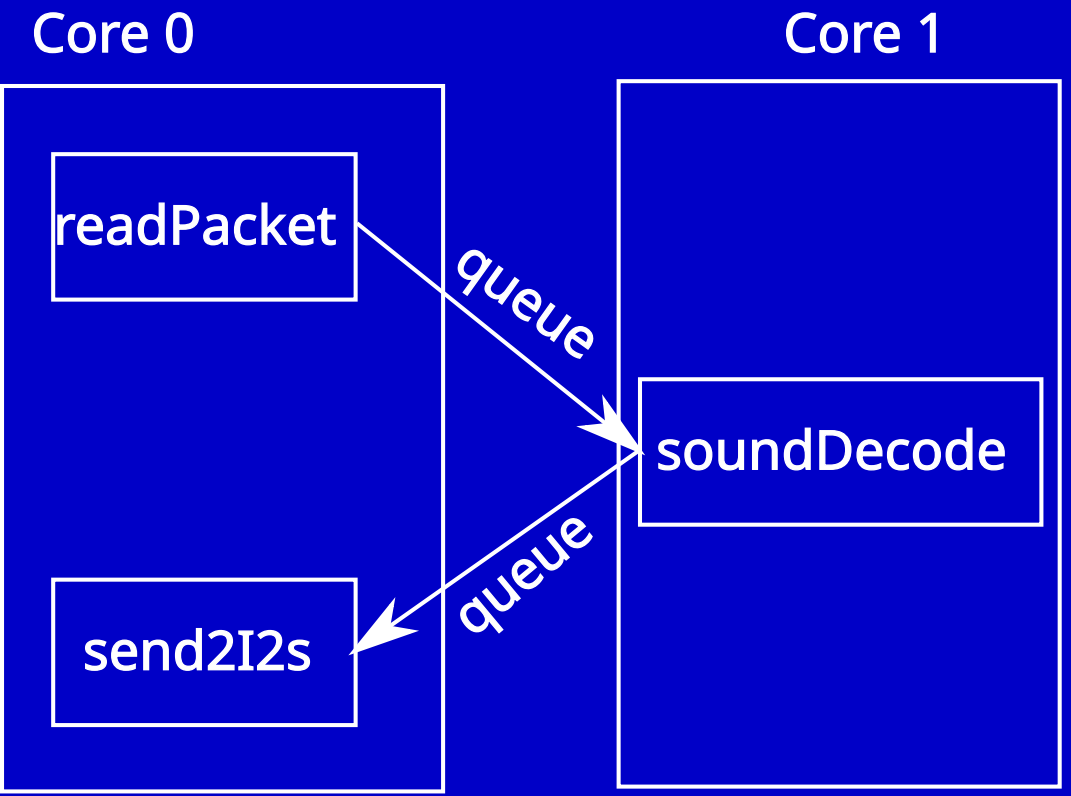



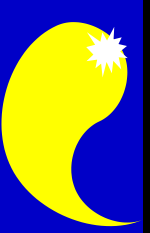
Overview



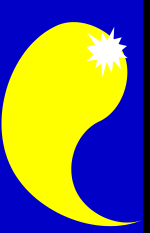
Examples

- Wireless speaker
  - receive sound stream from WiFi
  - decode it
  - send to I2S
  - dual core



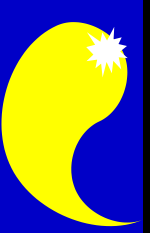


Overview

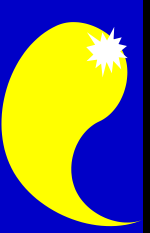


Implementation

- Raspberry Pi Pico W connects a CYW43439 for WiFi via SPI
- picowi provides driver
- lwIP provides network stack
- kuhllib provides network API and sender/receiver
- C++OS provides executors and queue

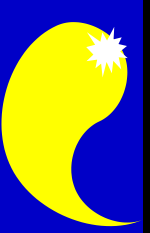


Overview



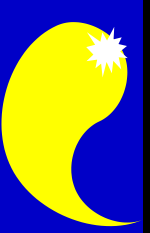
Code

- Any code shown here can be found at
  `https://gitlab.com/cppos1/demos`
  branch emBO24



References

- These slides:
  `http://www.vollmann.ch/`
- Standard proposals, TSes:
  `http://wg21.link/`
- Executors (Sender/Receiver):
  `P2300`
- Queues:
  `P0260R8`
- Networking TS:
  `N4771`
- Networking Sender/Receiver:
  `P2762`



Questions

- ??

Detlef Vollmann

Code: https://gitlab.com/cppos1/demos

dv@vollmann.ch
http://www.vollmann.ch/
Currently looking for a new project

