University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Message Passing and Network Programming

Advanced Operating Systems

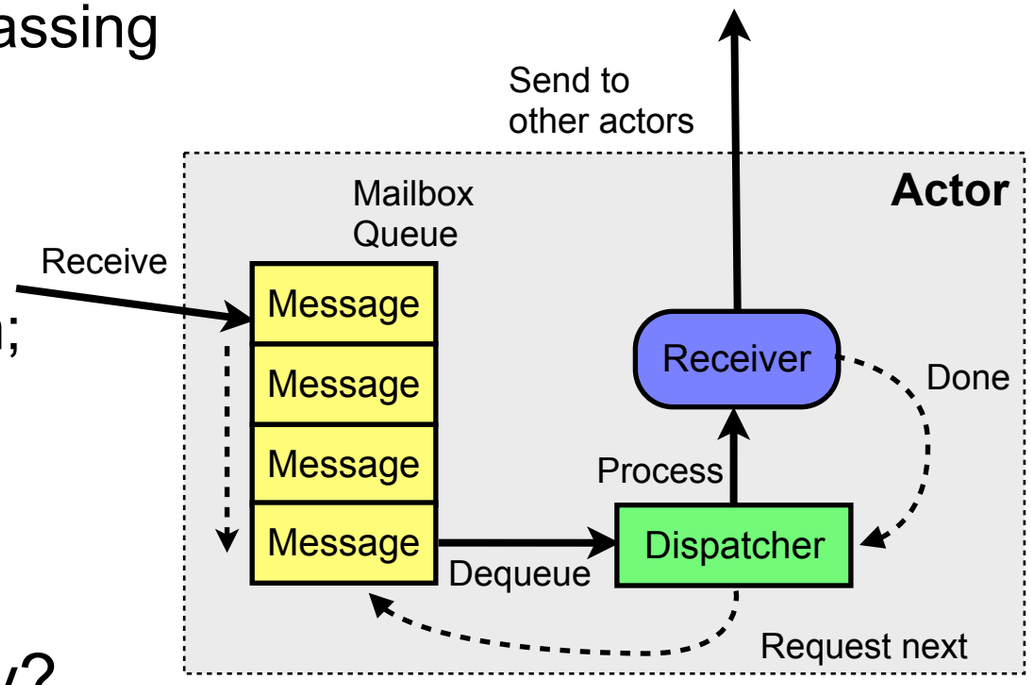
Lecture 13

Lecture Outline

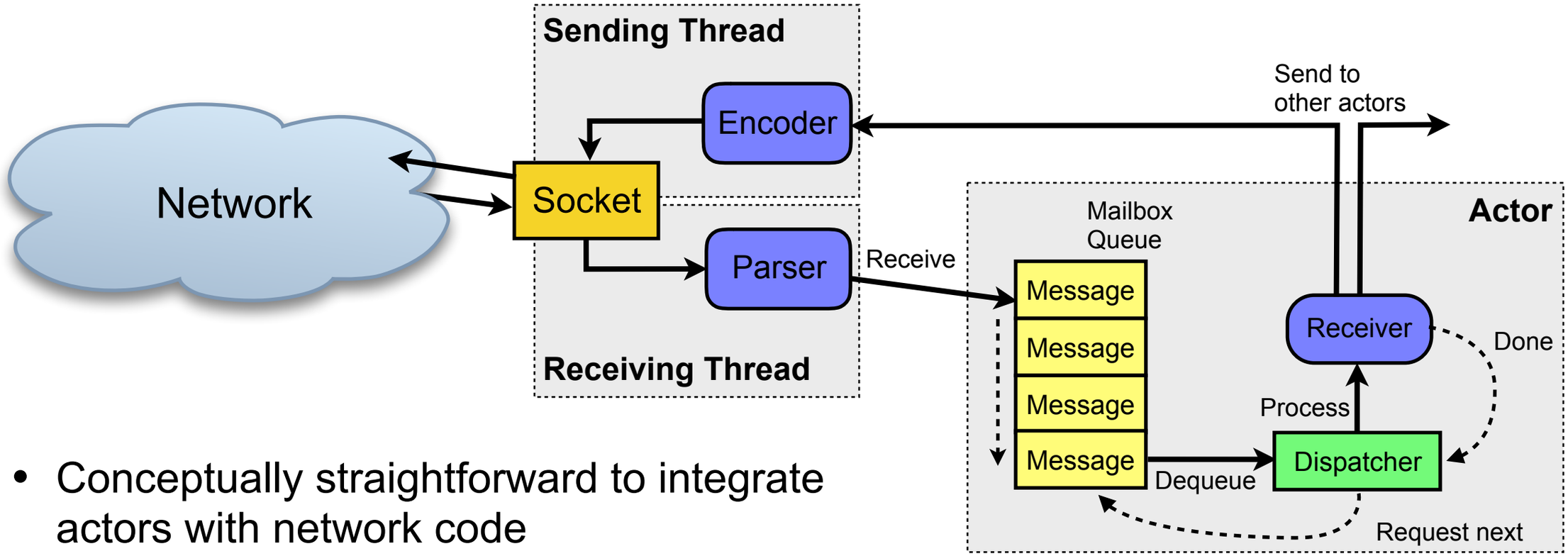
- Actors, sockets, and network protocols
- Asynchronous I/O frameworks
- Higher level abstractions

Message Passing and Network Protocols

- Recap:
  - Actor-based framework for message passing
  - Each actor has a receive loop
  - Calls to one function per state
  - Messages delivered by runtime system; processed sequentially
  - Actor can send messages in reply; return identity of next state
- Can we write network code this way?
  - Send data by sending a message to an actor representing a socket
  - Receive messages representing data received on a socket



Integrating Actors and Sockets



- Conceptually straightforward to integrate actors with network code
  - Runtime system maintains sending and receiving threads that manage the sockets
  - Receiving thread reads network protocol data from sockets, parses into messages, dispatches into the actor runtime
  - Sending thread receives messages from the actor runtime, encodes them as formatted packets of the network protocol, and sends via appropriate socket

- Network communication integrated with the actor system as message passing
- Adapters format and parse messages into the wire format

Parsing and Serialisation

- Parsing and serialisation requires socket awareness of the protocol:
  - E.g., make `Socket` an abstract class with `encode` and `parse` methods that are implemented by concrete subclasses and understand particular protocol
  - The `encode` method takes an object representing a protocol message, and returns a byte array that can be sent via the socket
  - The `parse` method takes a byte array received from the socket, and returns an object representing the protocol message
- Encapsulates parsing and serialisation – everything outside the socket uses strongly typed objects, not raw protocol data

Actors for Network Programming

- Conceptually clean – the simple message passing model abstracts away details of network transport
  - Easy to program and debug
  - Encapsulates parsing and serialisation
  - Helps correctness – on-the-wire format hidden, applications operate on strongly typed objects:

```
def receive = {
  case HttpRequest(GET, Uri.Path("/ping"), _, _, _) => {
    sender ! HttpResponse(entity = “PONG”)
  }
  case HttpRequest(PUT, …) => {
    …
  }
  case _: Tcp.ConnectionClosed => {
    …
  }
}
```

(Fragment of Scala+Akka+Spray code)

Actors for Network Programming

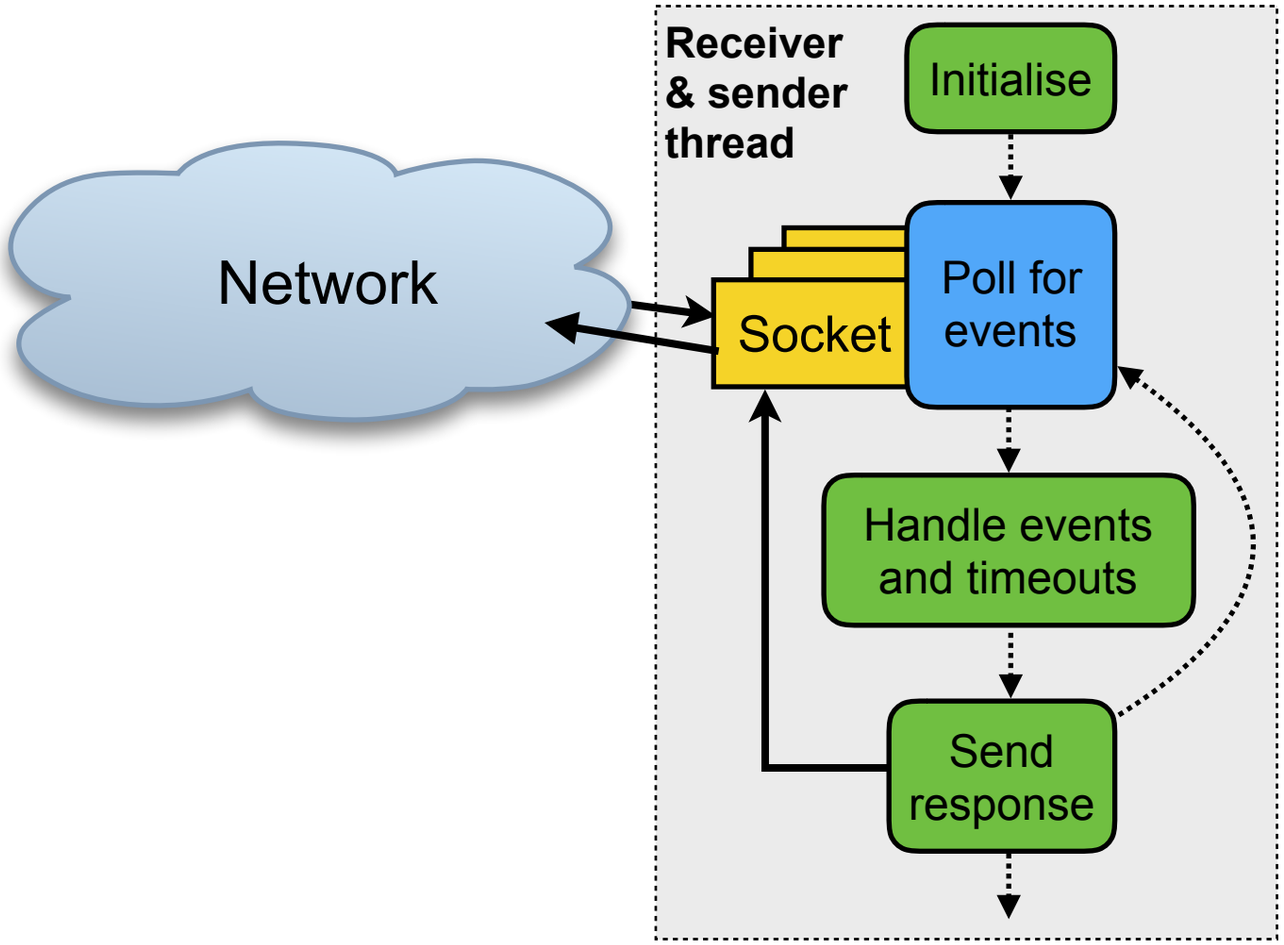
- Performance reasonable, but likely not outstanding
- Conceptually well suited to implementing protocols
  - Asynchronous reception matches network behaviour
  - Operates at high level of abstraction
  - Examples: EJabberD, WhatsApp → Erlang
- Potentially ill-suited to applications that track details of network behaviour
  - The “send message and forget it” style isn’t well suited to monitoring the progress or timing of delivery (e.g., for an adaptive video conference app)

Asynchronous I/O Frameworks

- Key insight from actor-based I/O: packet reception should be asynchronous to match the network
- However…
  - Sending API should be synchronous
    - The back pressure of a blocking send is useful to avoid congestion/sending queue build-up
    - Applications that query the state of a network interface need to know what is happening now, not what happened some time in the past, after a message exchange
    - Much simpler to implement in a synchronous manner
  - Actor-based implementations have relatively high overhead

- Implement as a blocking event loop, with timeout – an asynchronous (i.e., event driven) I/O framework

Event Loops



- Many implementations with similar functionality
  - `select()` and `poll()`: portable, but slow with large numbers of sockets
  - `epool()` (Linux), `kqueue` (FreeBSD/ macOS, iOS): efficient, less portable
  - `libevent`/`libev`/`libuv` wrap the above in a unified API (`libuv` currently seems most popular)
- Basic operation:
  - Pass a set of sockets, file descriptors, etc., to the kernel
  - Poll those for events – will block until something happens or timeout occurs
  - Process events, send any responses, loop back for next event
- Single threaded event loop
  - Long-running operations are passed to separate worker threads

Example: libev

…

```
static void udp_cb(EV_P_ ev_io *w, int revents) {
  // Called with data is readable on the UDP socket
}
```

Callback to handle event and send response if needed

```
int main(void) {
    int port = DEFAULT_PORT;
    puts("udp_echo server started...");

    // Setup a udp listening socket.
    sd = socket(PF_INET, SOCK_DGRAM, 0);
    bzero(&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sd, (struct sockaddr*) &addr, sizeof(addr)) != 0)
        perror("bind");

    // Do the libev stuff.
    struct ev_loop *loop = ev_default_loop(0);
    ev_io udp_watcher;
    ev_io_init(&udp_watcher, udp_cb, sd, EV_READ);
    ev_io_start(loop, &udp_watcher);
    ev_loop(loop, 0);

    close(sd);
    return EXIT_SUCCESS;
}
```

Initialise

Loop, processing events

Example: Rust MIO

```
fn main() {
    let ip       = IpAddr::V4(Ipv4Addr::new(0,0,0,0));
    let port     = 2223;
    let sockaddr = SocketAddr::new(ip, port);
    let socket   = UdpSocket::bind(&sockaddr).unwrap();

    let poll = Poll::new().unwrap();
    let mut events = Events::with_capacity(1024);

    poll.register(&socket, TOKEN, Ready::readable(), PollOpt::edge()).unwrap();

    loop {
        poll.poll(&mut events, None).unwrap();

        for event in &events {
            match event.token() {
                TOKEN => {
                    println!("got event");
                }
                _ => panic!("event with unexpected token")
            }
        }
    }
}
```

Initialise

Loop, processing events

Handle events

Event Loops

- Asynchronous I/O via event loops – efficient and highly scalable
  - Some implementations callback based (e.g., `libuv`), others expose the event set and allow applications to dispatch (e.g., Rust MIO)
    - I tend to prefer the latter, pattern matching. style – code is more obvious
  - Provided it does no long-running calculations, a single asynchronous I/O thread can usually saturate a single network interface
    - Use one thread per network interface
    - Use a thread pool to handle long-running calculations, passing requests to it for processing
- Model is the same as actor-based systems:
  - Sequence of events (i.e., messages) delivered to application; process each in turn, sending replies as needed
  - However, have direct access to underlying socket for sending operations, monitoring performance

Higher-level Abstractions

- Event-driven asynchronous I/O efficient and flexible → but can to obfuscated code
  - Implementation split across numerous callbacks; hard to follow the logic
  - Flexible, to implement complex protocols
- Can we provide an abstraction to simplify common protocol styles?
  - e.g., a server that responds to a single request with a single response

Your Server as a Function (1)

- Conceptually, a server represents a function:

```
fn service_request(&self, req : Request) -> Future<Response, Error>;
```

- Takes a request and returns a *future* – an eventual response or error:

```
enum Future<Response, Error> {
  InProgress,
  Success<Response>,
  Failed<Error>
}
```

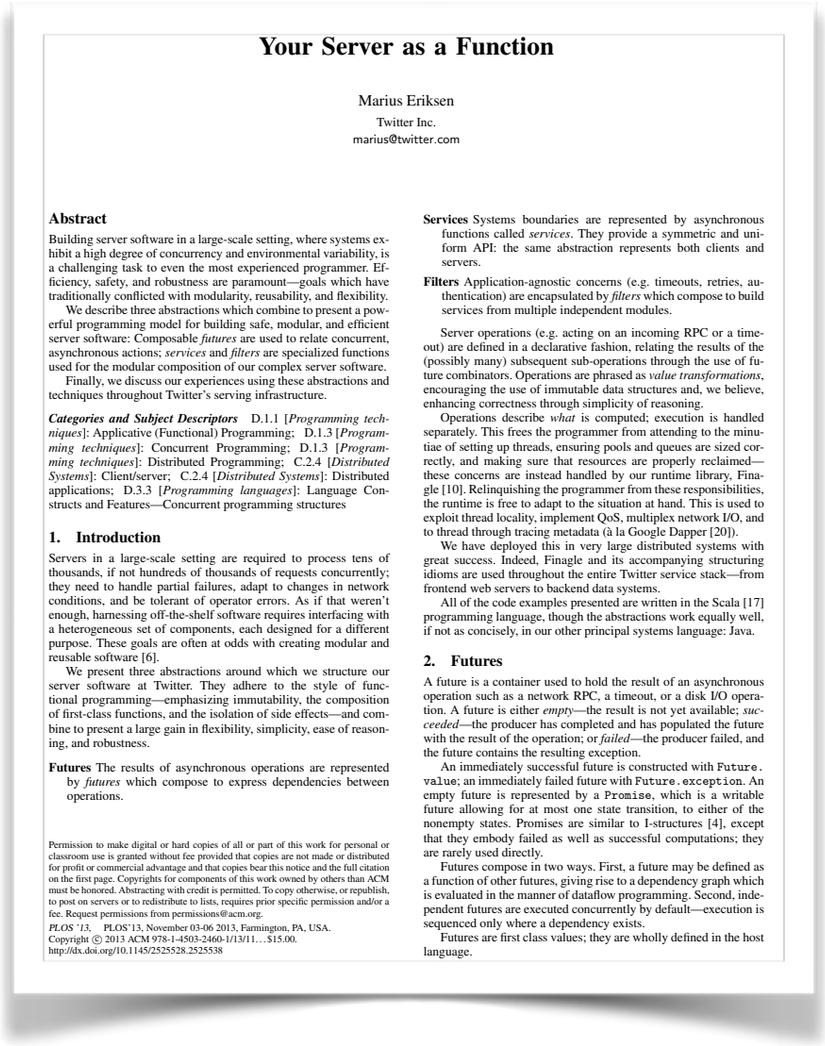
- Combine with parsing and encoding functions:

```
fn parse(&self, data : &[u8]) -> Result<Request, Error>;
fn encode(&self, res : Response) -> [u8];
```

Convert raw protocol data to strongly-typed objects

Your Server as a Function (2)

- These three functions, plus definitions of `Request`, `Response`, and `Error` types can specify a server
- Plug into a library handling asynchronous I/O operations/scheduling calls to the functions
- Potentially allows writing services at a very high level of abstraction



Your Server as a Function

Marius Eriksen
Twitter Inc.
marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a powerful programming model for building safe, modular, and efficient server software: Composable *futures* are used to relate concurrent, asynchronous actions; *services* and *filters* are specialized functions used for the modular composition of our complex server software.

Finally, we discuss our experiences using these abstractions and techniques throughout Twitter's serving infrastructure.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming techniques*]: Concurrent Programming; D.1.3 [*Programming techniques*]: Distributed Programming; C.2.4 [*Distributed Systems*]: Client/server; C.2.4 [*Distributed Systems*]: Distributed applications; D.3.3 [*Programming languages*]: Language Constructs and Features—Concurrent programming structures

1. Introduction

Servers in a large-scale setting are required to process tens of thousands, if not hundreds of thousands of requests concurrently; they need to handle partial failures, adapt to changes in network conditions, and be tolerant of operator errors. As if that weren't enough, harnessing off-the-shelf software requires interfacing with a heterogeneous set of components, each designed for a different purpose. These goals are often at odds with creating modular and reusable software [6].

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

Futures The results of asynchronous operations are represented by *futures* which compose to express dependencies between operations.

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

Server operations (e.g. acting on an incoming RPC or a timeout) are defined in a declarative fashion, relating the results of the (possibly many) subsequent sub-operations through the use of future combinators. Operations are phrased as *value transformations*, encouraging the use of immutable data structures and, we believe, enhancing correctness through simplicity of reasoning.

Operations describe *what* is computed; execution is handled separately. This frees the programmer from attending to the minutiae of setting up threads, ensuring pools and queues are sized correctly, and making sure that resources are properly reclaimed—these concerns are instead handled by our runtime library, Finagle [10]. Relinquishing the programmer from these responsibilities, the runtime is free to adapt to the situation at hand. This is used to exploit thread locality, implement QoS, multiplex network I/O, and to thread through tracing metadata (à la Google Dapper [20]).

We have deployed this in very large distributed systems with great success. Indeed, Finagle and its accompanying structuring idioms are used throughout the entire Twitter service stack—from frontend web servers to backend data systems.

All of the code examples presented are written in the Scala [17] programming language, though the abstractions work equally well, if not as concisely, in our other principal systems language: Java.

2. Futures

A future is a container used to hold the result of an asynchronous operation such as a network RPC, a timeout, or a disk I/O operation. A future is either *empty*—the result is not yet available; *succeeded*—the producer has completed and has populated the future with the result of the operation; or *failed*—the producer failed, and the future contains the resulting exception.

An immediately successful future is constructed with `Future.value`; an immediately failed future with `Future.exception`. An empty future is represented by a `Promise`, which is a writable future allowing for at most one state transition, to either of the nonempty states. Promises are similar to I-structures [4], except that they embody failed as well as successful computations; they are rarely used directly.

Futures compose in two ways. First, a future may be defined as a function of other futures, giving rise to a dependency graph which is evaluated in the manner of dataflow programming. Second, independent futures are executed concurrently by default—execution is sequenced only where a dependency exists.

Futures are first class values; they are wholly defined in the host language.

PLOS '13, PLOS'13, November 03-06 2013, Farmington, PA, USA.

http://dx.doi.org/10.1145/2525528.2525538

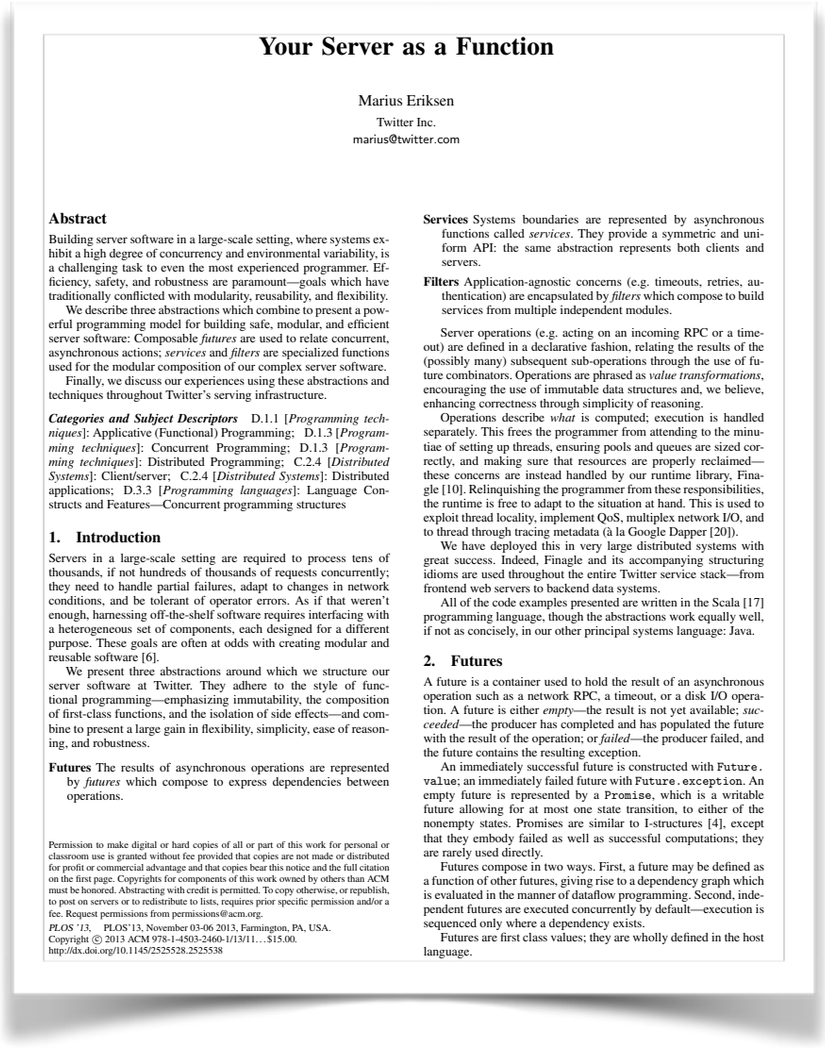
- Examples:
  - Twitter's `Finagle` library in Scala
  - The `Tokio` framework for Rust
- Further reading:
  - M. Eriksen, "Your server as a function", Proceedings of the Workshop on Programming Languages & Operating Systems, Farmington, PA, USA, November 2013. ACM. DOI: 10.1145/2525528.2525538
  - Describes the Finagle library

Summary

- Many types of network code are a good fit for actor frameworks
  - Protocol messages parsed/encoded to/from Actor messages
  - But – relatively high overheads; poorly suited to low-level applications
- Event driven asynchronous I/O framework solve many of these issues – but can obfuscate code
- Event driven approaches based on pattern matching lead to more obvious code than callback based systems
- The server-as-a-function approach can abstract asynchronous I/O into simple, high-level, frameworks for some applications

Further Reading

- M. Eriksen, “Your server as a function”, Proc. Workshop on Programming Languages and Operating Systems, Farmington, PA, USA, November 2013. ACM. DOI: 10.1145/2525528.2525538



Your Server as a Function

Marius Eriksen
Twitter Inc.
marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a powerful programming model for building safe, modular, and efficient server software: Composable *futures* are used to relate concurrent, asynchronous actions; *services* and *filters* are specialized functions used for the modular composition of our complex server software.

Finally, we discuss our experiences using these abstractions and techniques throughout Twitter's serving infrastructure.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming techniques*]: Concurrent Programming; D.1.3 [*Programming techniques*]: Distributed Programming; C.2.4 [*Distributed Systems*]: Client/server; C.2.4 [*Distributed Systems*]: Distributed applications; D.3.3 [*Programming languages*]: Language Constructs and Features—Concurrent programming structures

1. Introduction

Servers in a large-scale setting are required to process tens of thousands, if not hundreds of thousands of requests concurrently; they need to handle partial failures, adapt to changes in network conditions, and be tolerant of operator errors. As if that weren't enough, harnessing off-the-shelf software requires interfacing with a heterogeneous set of components, each designed for a different purpose. These goals are often at odds with creating modular and reusable software [6].

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

Futures The results of asynchronous operations are represented by *futures* which compose to express dependencies between operations.

PLOS '13, PLOS'13, November 03-06 2013, Farmington, PA, USA.

http://dx.doi.org/10.1145/2525528.2525538

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

Server operations (e.g. acting on an incoming RPC or a timeout) are defined in a declarative fashion, relating the results of the (possibly many) subsequent sub-operations through the use of future combinators. Operations are phrased as *value transformations*, encouraging the use of immutable data structures and, we believe, enhancing correctness through simplicity of reasoning.

Operations describe *what* is computed; execution is handled separately. This frees the programmer from attending to the minutiae of setting up threads, ensuring pools and queues are sized correctly, and making sure that resources are properly reclaimed—these concerns are instead handled by our runtime library, Finagle [10]. Relinquishing the programmer from these responsibilities, the runtime is free to adapt to the situation at hand. This is used to exploit thread locality, implement QoS, multiplex network I/O, and to thread through tracing metadata (à la Google Dapper [20]).

We have deployed this in very large distributed systems with great success. Indeed, Finagle and its accompanying structuring idioms are used throughout the entire Twitter service stack—from frontend web servers to backend data systems.

All of the code examples presented are written in the Scala [17] programming language, though the abstractions work equally well, if not as concisely, in our other principal systems language: Java.

2. Futures

A future is a container used to hold the result of an asynchronous operation such as a network RPC, a timeout, or a disk I/O operation. A future is either *empty*—the result is not yet available; *succeeded*—the producer has completed and has populated the future with the result of the operation; or *failed*—the producer failed, and the future contains the resulting exception.

An immediately successful future is constructed with `Future.value`; an immediately failed future with `Future.exception`. An empty future is represented by a `Promise`, which is a writable future allowing for at most one state transition, to either of the nonempty states. Promises are similar to I-structures [4], except that they embody failed as well as successful computations; they are rarely used directly.

Futures compose in two ways. First, a future may be defined as a function of other futures, giving rise to a dependency graph which is evaluated in the manner of dataflow programming. Second, independent futures are executed concurrently by default—execution is sequenced only where a dependency exists.

Futures are first class values; they are wholly defined in the host language.