JavaCro'15

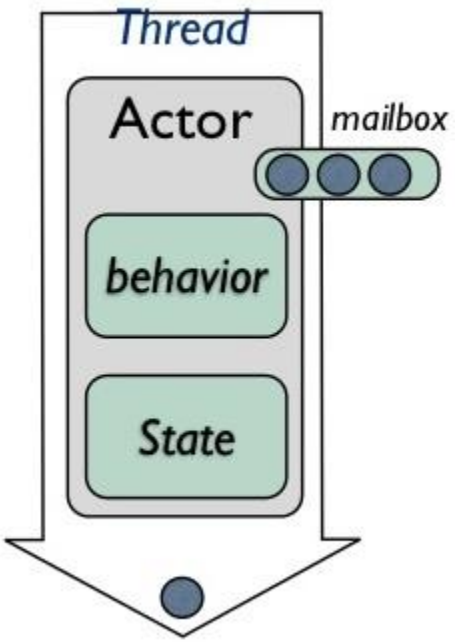
Reactive I/0

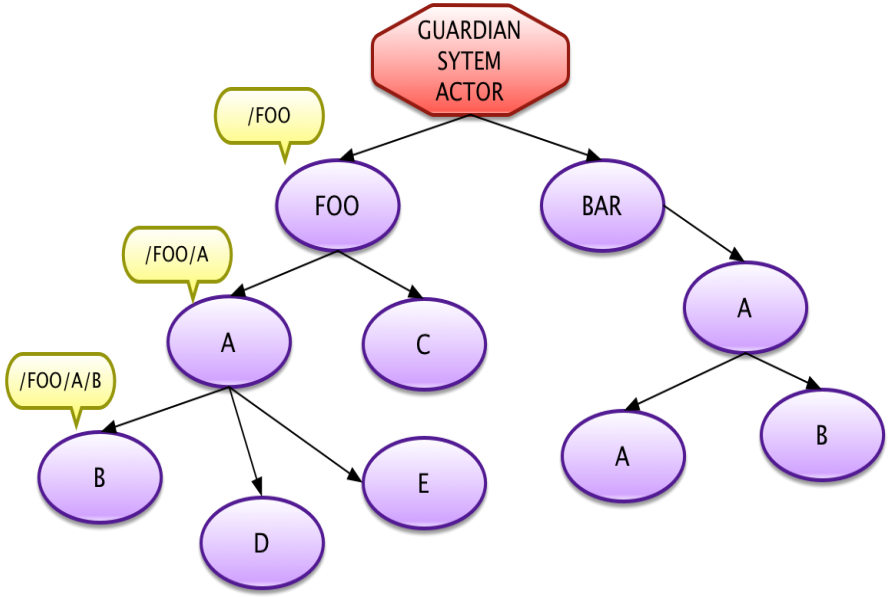
Ivan Turčinović
ivan.turcinovic@inovatrend.com

- Toolset for building ***concurrent*** applications
  fault-tolerant
  scalable

- Core part of Typesafe Platfom

Actors



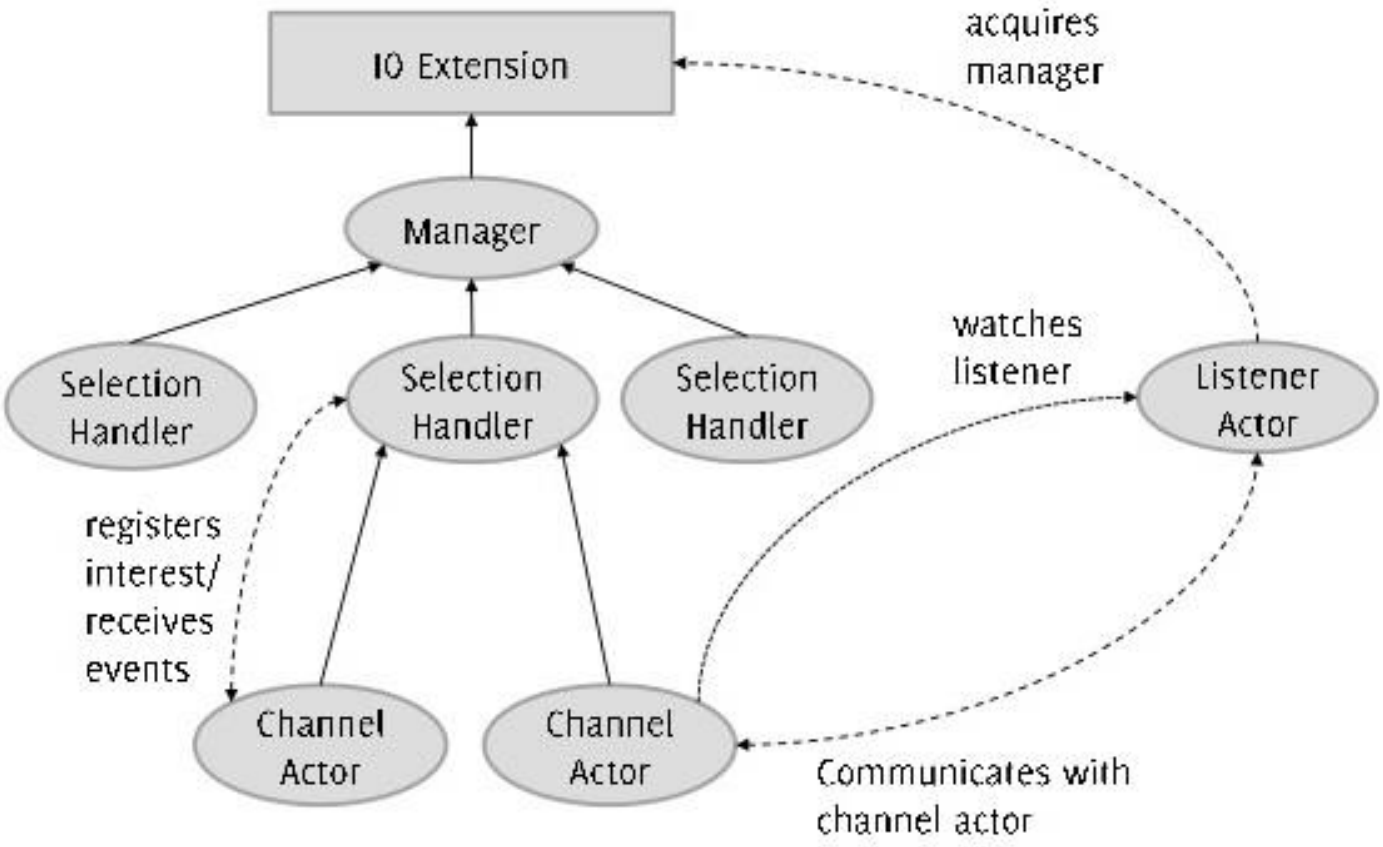



```
public class MyActor extends UntypedActor {
    @Override
    public void onReceive(Object msg) throws Exception {
        //...
    }
}
```

Akka I/O

- Akka I/O package follows same principles
  - message passing
  - reactive
  - actor based API
  - with immutable data representation
- Built with collaboration with spray.io team
  - toolkit for building REST/HTTP-based integration layers
- TCP and UDP implementations available

Akka I/O Architecture



Basic entities

- ByteString
  - actors communicate with immutable objects only
  - ByteString - immutable container for bytes
  - Rope-like data structure
- TCP messages
  - TCP.connect/TCP.connected/TCP.register
  - TCP.bind/TCP.bound
  - TCP.commandFailed/TCP.connectionClosed/TCP.close
  - TCP write - write, writeFile, compoundWrite
- UDP messages
  - Unconnected UDP
  - Connected UDP

TCP Client – creating connection

```
public class ClientListener extends UntypedActor {
    private final ActorRef msgProcessor;

    public ClientListener(InetSocketAddress remote, ActorRef msgProcessor) {
        this.msgProcessor = msgProcessor;

        ActorRef tcp = Tcp.get(getContext().system()).manager();                    1.

        tcp.tell(TcpMessage.connect(remote), getSelf());                            2.
    }

    @Override
    public void onReceive(Object msg) throws Exception {
        if (msg instanceof Tcp.Connected) {                                         3.
            msgProcessor.tell(msg, getSelf());

            getSender().tell(TcpMessage.register(getSelf()), getSelf());           4.

            getContext().become(connected(getSender()));                            5.
        }
    }
```

TCP Client – send/receive messages, closing connection

```
private Procedure<Object> connected(final ActorRef connection) {
    return msg -> {
        if (msg instanceof Tcp.Received) {                                    1.
            final ByteString data = ((Tcp.Received) msg).data();
            msgProcessor.tell(data, getSelf());

        } else if (msg instanceof ClientMessage) {
            connection.tell(TcpMessage.write(((ClientMessage) msg).toByteString()), getSelf()); 2.

        } else if (msg.equals("close")) {
            connection.tell(TcpMessage.close(), getSelf());                   3.

        } else if (msg instanceof Tcp.ConnectionClosed) {                     4.
            msgProcessor.tell(msg, getSelf());
            getContext().stop(getSelf());
        }

    };
}
```

TCP Server – accepting connections

```
public class ServerActor extends UntypedActor {
    private final ActorRef tcpManager;

    public ServerActor(int port) {
        this.tcpManager = Tcp.get(getContext().system()).manager();
        tcpManager.tell(TcpMessage.bind(getSelf(), new InetSocketAddress(port), 100), getSelf());   1.
    }

    @Override
    public void onReceive(Object msg) throws Exception {
        if (msg instanceof Tcp.Bound) {                                                              2.
            tcpManager.tell(msg, getSelf());

        } else if (msg instanceof Tcp.Connected) {                                                   3.
            final Tcp.Connected conn = (Tcp.Connected) msg;
            tcpManager.tell(conn, getSelf());

            final ActorRef listener = getContext().actorOf(Props.create(ConnectionListener.class));
            getSender().tell(TcpMessage.register(listener), getSelf());                              4.

        } else if (msg instanceof Tcp.CommandFailed) {
            getContext().stop(getSelf());

        }
    }

}
```

TCP Server – send/receive messages

```
public class ConnectionListener extends UntypedActor {

    @Override
    public void onReceive(Object msg) throws Exception {
        if (msg instanceof Tcp.Received) {
            final ByteString incoming = ((Tcp.Received) msg).data();
            ByteString response = process(incoming);

            getSender().tell(TcpMessage.write(response), getSelf());

        } else if (msg instanceof Tcp.ConnectionClosed) {
            getContext().stop(getSelf());

        }
    }
```

1.

2.

3.

TCP - Throttling Reads and Writes

- Data congestion needs to be handled at the user level, for both writes and reads.
- Throttling writes
  - ACK-based
  - NACK-based
  - NACK-based with write suspending
- Throttling reads
  - Push-reading
  - Pull-reading

Lessons learned

- Message sending/receiving is NOT one-by-one
  - onReceive ByteString can have more then one message sent
  - Application is responsible for encoding/decoding messages
- Client – one connection actor per client is more than enough
- Server – each connection has one dedicated actor for receiving/sending messages
  - Gives really nice control of clients on server side
  - Authentication, blocking, stats per connection easy to implement
- Broadcasting message from server to client done really easy
  - Send message to all ServerActor children

Lessons learned cont'd

- DON'T rely only on TCP. ConnectionClosed messages for „disconnects"
  - It works OK in „lab" conditions
  - Problems occur in low qualilty network connections + VPN !!
- Solution:
  - Rely on own heartbeat protocol
  - Simple PingPongMessage
  - Sent frequently – in our case each 5 sec. Gives very little or no overhead
  - App keeps track of heartbeat messages
    - no response message within specified time – something is wrong with connection – kill it!

Akka – Spring integration

- Not so much related to akka I/O ☺
- https://github.com/typesafehub/activator-akka-java-spring
- SpringExtension - Akka Extension to provide access to Spring managed Actor Beans

```
@Bean(name = "mySpringBasedActor")
@Scope("prototype")
public MyActor myActor() {
    return new MyActor(terminalManager, userManager);
}
```

```
ActorSystem system = getContext().system();
Props myActorProps = SpringExtension.SpringExtProvider.get(system).props("mySpringBasedActor");
ActorRef myActorRef = getContext().actorOf(myActorProps, "mySpringBasedActor");
```

Thank You!