Erlang Solutions

gen_statem Unveiled
A Theoretical Exploration of State Machines

Nelson Vides
Senior Erlang Consultant and Core MongooseIM developer

nelson.vides@erlang-solutions.com

Erlang Solutions - About Us

Who we are

We build massively scalable, distributed systems for high volumes of concurrent users, using resilient and fault-tolerant technologies

Our team proudly gives back to the community through open source technologies and conferences

Our work

We work on some of the most famous systems in the world

Our team

- Globally based
- Diverse & multi-cultural
- United by our passion for technology

WhatsApp aws Klarna VOCALINK mastercard cisco EE PEPSICO ERICSSON Bloomberg Roche adidas RUNTASTIC pando NAYAPAY

Erlang Solutions

Protocols

A system of rules that allows two or more entities of to transmit information.

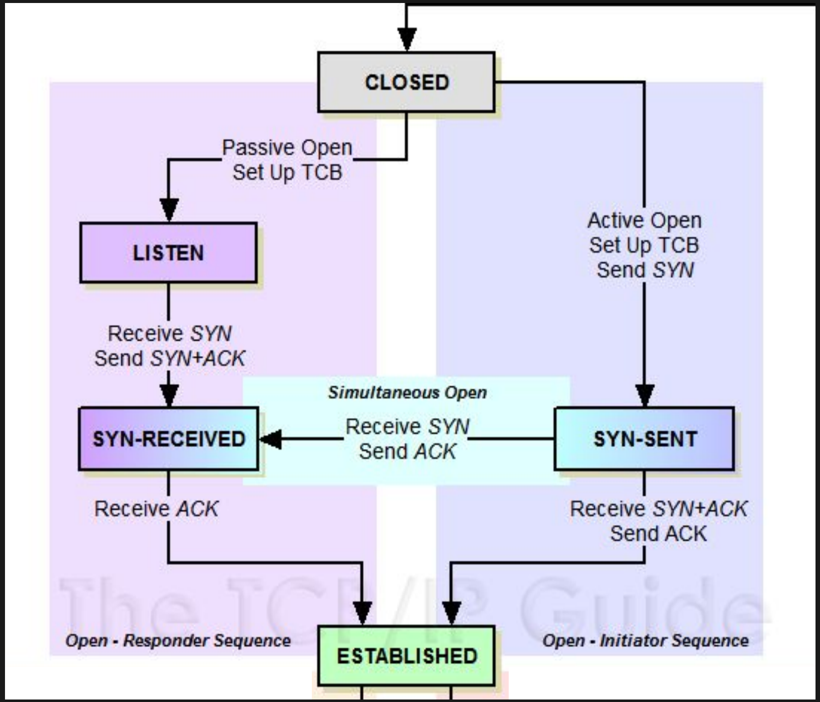
“The protocol defines the rules, syntax, semantics, and synchronisation of communication and possible error recovery methods.”

Wikipedia: Communication protocol

Erlang Solutions

TCP

- Connection-oriented
- Stream-oriented
- Ordered
- Acknowledged

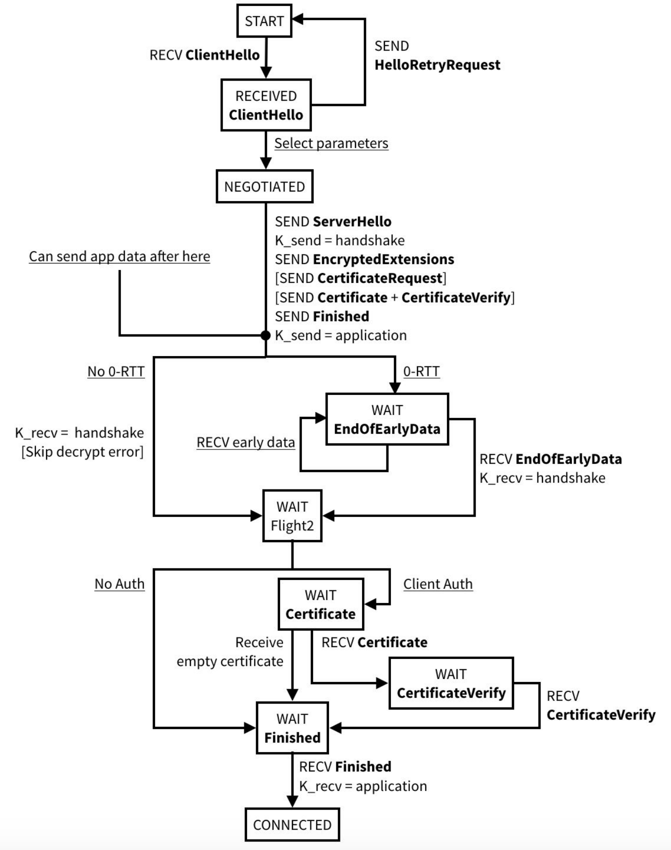


[The TCP/IP guide](#)

Erlang Solutions

TLS

- Privacy
- Integrity
- Authenticity

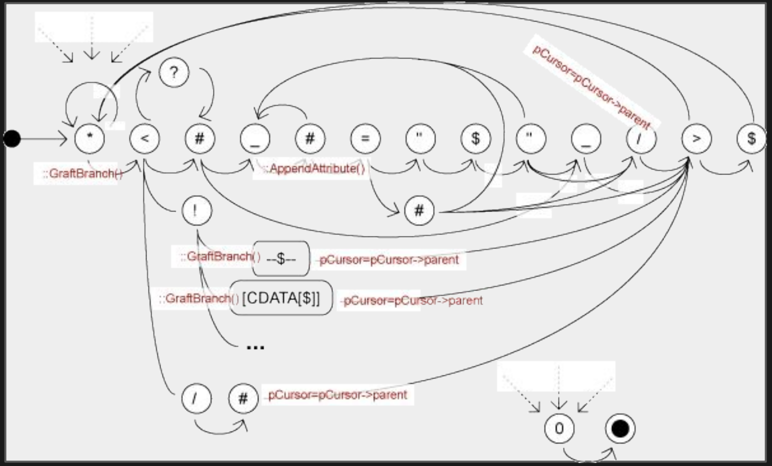


[A Readable TLSv1.3 Specification — cryptologie.net]

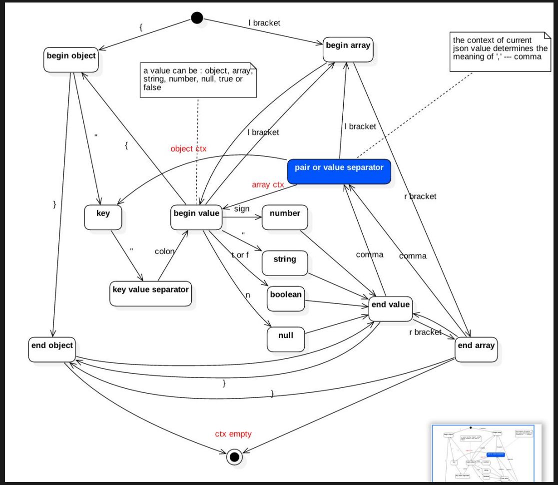
Erlang Solutions

XML or JSON

- File formats
- Structured



PugXML

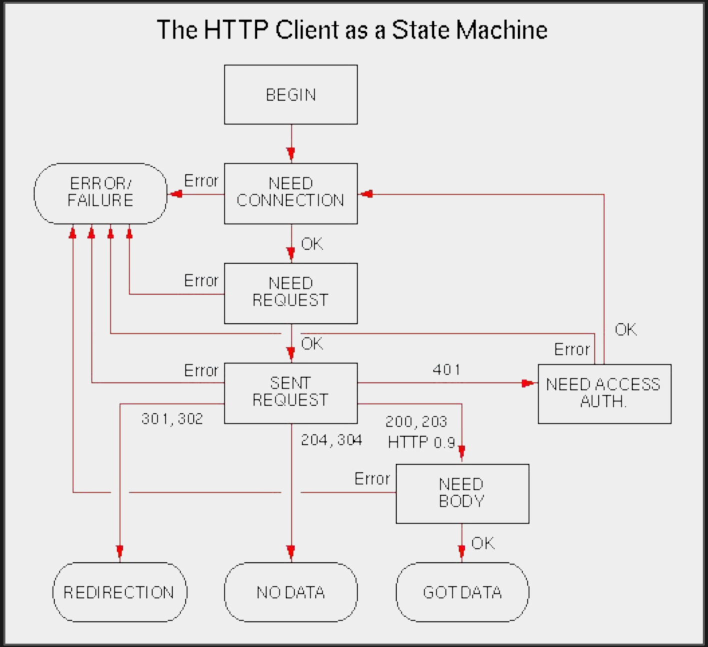


JSON Parser

Erlang Solutions

HTTP

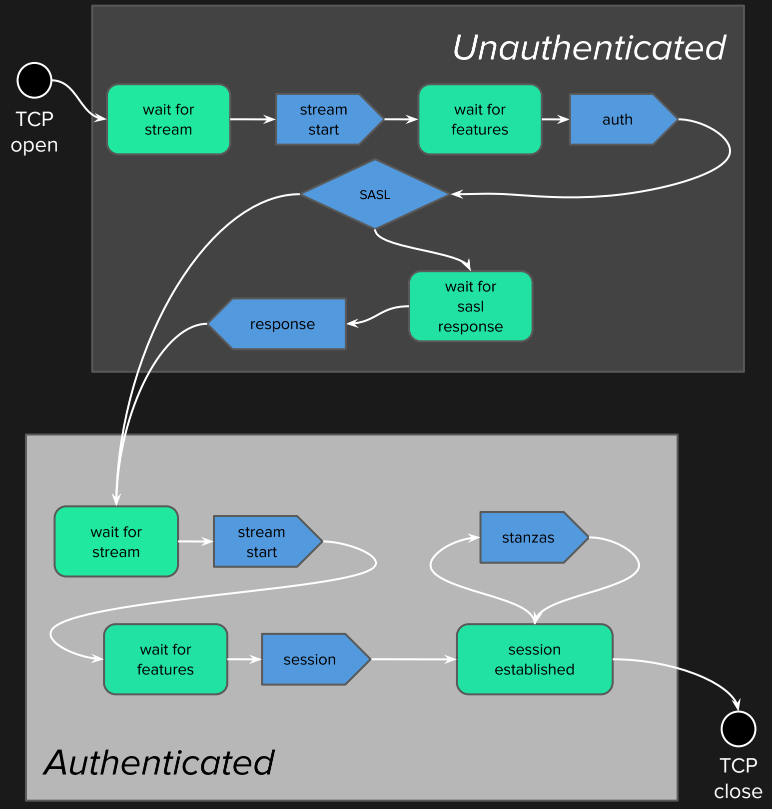
- Stateless
- Request-response



[Protocol Modules as State machines](#)

XMPP

- Extensible
- Instant messaging
- Presence information
- Contact list maintenance



Erlang Solutions

Much like a flowchart on steroids, finite state machines represent an easy way to visualise complex computation flows through a system.

Erlang Solutions

Models

State machines can be seen as models that represent system behaviour

Formalities

- *An Alphabet* $\boldsymbol{\Sigma}$: a finite set of input symbols $\boldsymbol{\alpha}$
    - Σ = {0,1}; Σ = [ASCII]; Σ=[bounded valid XML]
- *A string* $\boldsymbol{\omega}$ *over* $\boldsymbol{\Sigma}$: a finite concatenation of symbols of the alphabet Σ
    - “0110101” over {0, 1}
- *The Power of an Alphabet*: $\boldsymbol{\Sigma}^*$, the (infinite) set of all possible strings over $\boldsymbol{\Sigma}$, including the empty set (an empty string).
    - {{“”}, {“0”}, {“1”}, {“”, “0”}, {“”, “1”}, {“”, “0”, “1”}, ...}

Finite State Machines

- an alphabet Σ, a finite set of states Q, an a subset of final states F
- and nothing else
- a function δ such that given states $p \in Q$, an input symbol $\alpha \in \Sigma$:
  - $\delta(p, \alpha) \rightarrow q$, such that $q \in Q$

An FSM is said to consume a string ω over Q when:

- $\delta(p, \omega) = q$ with $q \in F$

FSMs are equivalent to ***regular grammars*** (regexes)

Pushdown Automata

- an alphabet Σ, a finite set of states Q, an a subset of final states F
- a stack Γ of symbols of Σ
- a function δ such that given states $p \in Q$, an input symbol $\alpha \in \Sigma$, and the stack Γ:
    - $\delta(p, \alpha, \Gamma) \rightarrow (q, \Gamma 2)$ where $q \in Q$ and $\Gamma 2$ is Γ after popping the last symbol, pushing a new symbol, or both.

A PDA is said to consume a string ω over Q when:

- $\delta(p, \omega, \Gamma) = (q, \varnothing)$ with $q \in F$ and $\varnothing$ the empty stack

PDAs are equivalent to ***Context-Free Grammars*** (parsers)

Erlang Solutions

Turing Machines

- an alphabet Σ, a finite set of states Q, an a subset of final states F
- an infinite tape T of cells with symbols in Σ, and a tape head H
- a function δ such that given states $p \in Q$, an input symbol $\alpha \in \Sigma$, and the tape T with its head H:
  - $\delta(p, \alpha, H) \rightarrow (q, \beta, H')$ where $q \in Q$, $\beta \in \Sigma$ to write in H', and H' is the new head of T after moving one step to left or right and writing a new symbol to it

A Turing Machine is said to consume a string ω over Q when:

- $\delta(p, \omega, \Gamma)$ is undefined, that is, the machine *halts*

TMs are equivalent to ***unrestricted grammars*** (compilers)

How do they compare?

Unrestricted Grammars

TM

Context-free Grammars

PDA

Regular Grammars

FSM

How do they compare?

Unrestricted Grammars

TM

multi-tape TM, TM with tape bounded on one side

Context-free Grammars

PDA

PDA with a finite buffer

Regular Grammars

FSM

2xFSM running together

How do they compare?

Unrestricted Grammars

TM

multi-tape TM, TM with tape bounded on one side

PDA with 2 stacks

Context-free Grammars

PDA

PDA with a finite buffer

PDA + FSM

Regular Grammars

FSM

2xFSM running together

NxFSM running together

Conceptually, Finite State Machines can keep track of *one* thing.
While Pushdown Automata can keep track of up to *two* things.
While Turing Machines can keep track of a countably *infinite* number of things.

Erlang Solutions

Output?

Do FSMs produce output?

Finite State Transducers

- an alphabet Σ, a finite set of states Q, an a subset of final states F
- and an output alphabet Λ,
- a function δ such that given states $p \in Q$, an input symbol $\alpha \in \Sigma$:
    - $\delta(p, \alpha) \rightarrow (q, \lambda)$ such that $q \in Q$, $\lambda \in \Lambda$

An FST is said to consume a string ω over Q when:

- $\delta(p, \omega) = q$ with $q \in F$

FSTs are (also) equivalent to ***regular grammars*** (regexes)

$\delta(p, \alpha) \to (q, \lambda)$

- Mealy:
  - Transition: $\delta_1(p, \alpha) \to q \in Q$
  - Output: $\delta_2(p, \alpha) \to \lambda \in \Lambda$
- Moore:
  - Transition: $\delta_1(p, \alpha) \to q \in Q$
  - Output: $\delta_2(p) \to \lambda \in \Lambda$

How do they compare?

Turing machines and Pushdown automata

FST

Mealy > Moore

FSM

Erlang Solutions

How do they compare?

Turing machines and Pushdown automata

FST

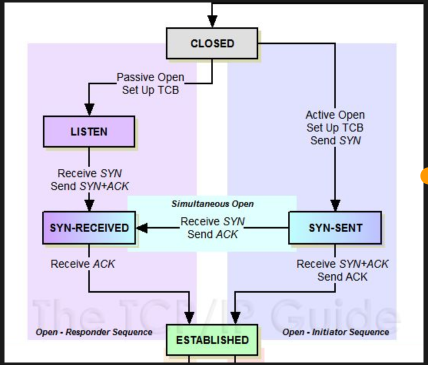
Mealy > Moore

These can be composed!

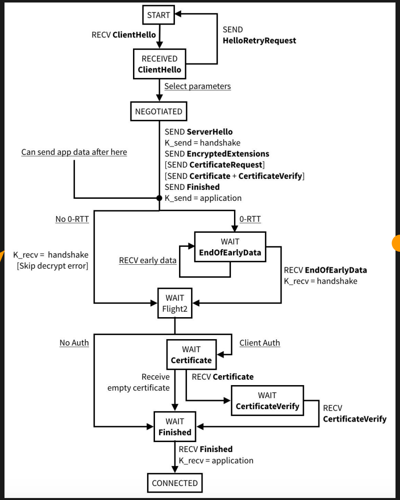
FSM

Composition

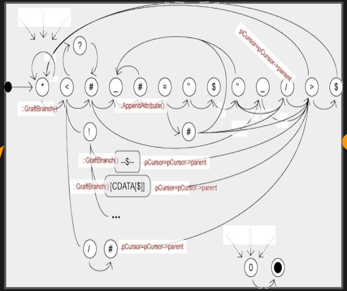
- Given sets of states P, Q, R
- And alphabets Σ, Λ, Γ
  - Mealy 1: $\delta(P, \Sigma) \rightarrow (Q, \Lambda)$
  - Mealy 2: $\delta(Q, \Lambda) \rightarrow (R, \Gamma)$
  - Mealy Composition: $\delta(P, \Sigma) \rightarrow (R, \Gamma)$



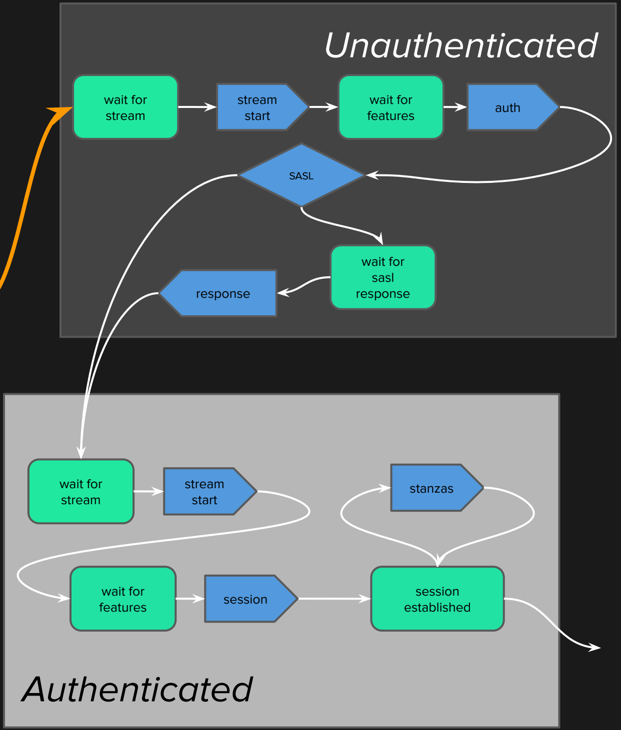
The TCP/IP guide



A Readable TLSv1.3 Specification — cryptologie.net



PugXML



Erlang Solutions

An algebra

Given FSMs M_1 and M_2, the following are true:

- $(M_1 \cup M_2)$ is an FSM
- $(M_1 \cap M_2)$ is an FSM
- $(M_1 . M_2)$ is an FSM
- The reverse, and the inverse, are also FSMs
- $\varnothing$ is the neutral element under union and concatenation
- Homomorphisms also preserve FSMs

FSMs form a semiring under union and concatenation

Erlang Solutions

Why is such algebra useful?

Proving theorems!

gen_statem: $\delta(P, \Sigma) \rightarrow (Q, \Lambda)$

- P and Q are the states as implemented by the programmer
- Σ are messages in the mailbox
- Λ are side-effects

gen_statem

An extended mailbox

Erlang Solutions

A mental picture

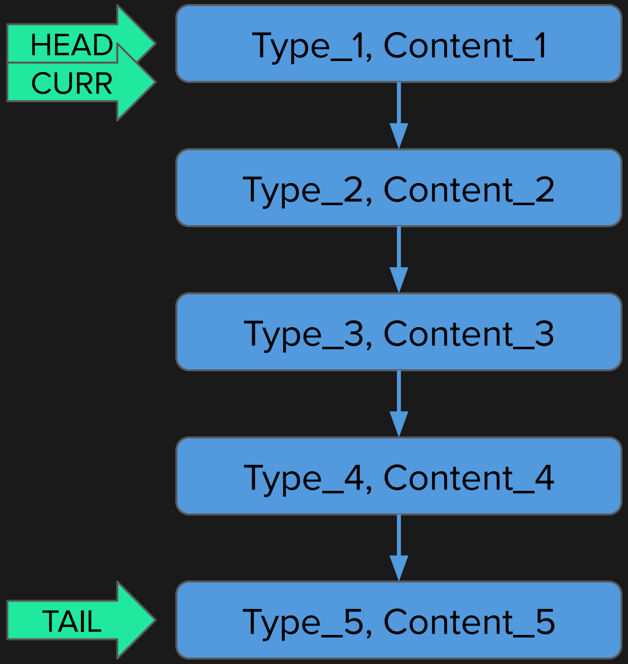
- Only event one queue that is an extension of the process mailbox.
- This queue has got:
  - A *head* pointing at the oldest event;
  - A *current* pointing at the next event to be processed.
  - A *tail* pointing at the youngest event;

A mental picture

- ***Postpone*** causes the current position to move to its next younger event so the previous current position is still in the queue reachable from head.
- ***Not postponing*** an event i.e consuming it causes the event to be removed from the queue and current position to move to its next younger event.
- ***NewState =/= State*** causes the current position to be set to head i.e the oldest event.
- ***next_event*** inserts event(s) at the current position i.e as just older than the previous current position.
- ***{timeout, 0, Msg}*** inserts a {timeout, Msg} event after tail i.e as the new youngest received event.

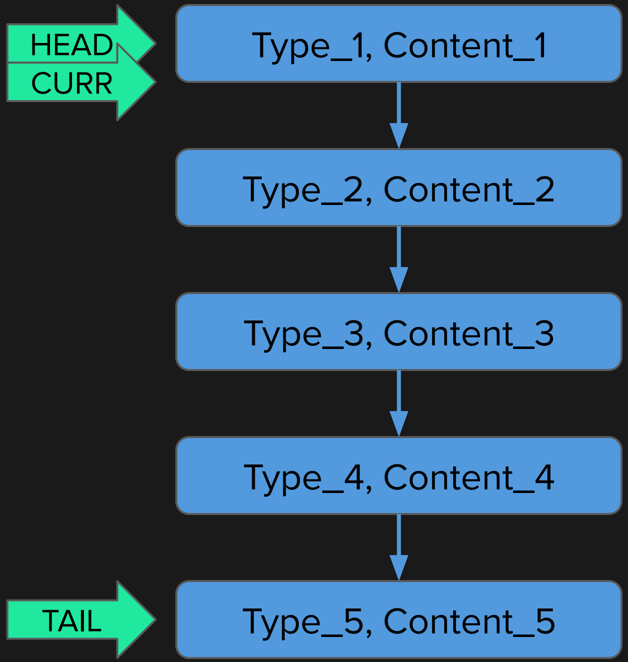
The event queue

```
handle_event(Type1, Content1, State1, Data) ->
```



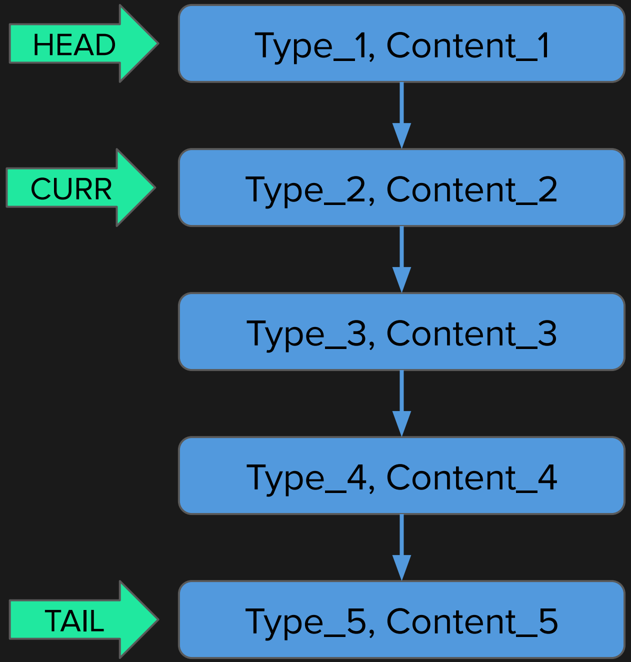
The event queue

```erlang
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
```



The event queue

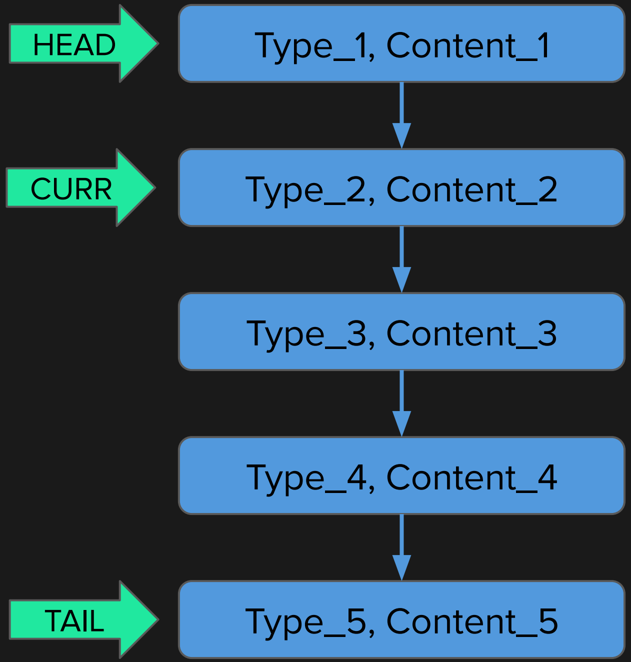
```erlang
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
```



Erlang Solutions

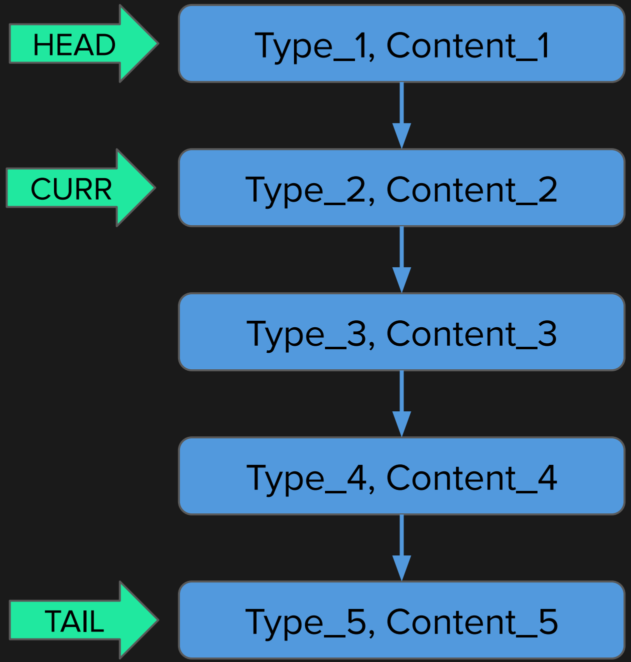
The event queue

```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
```



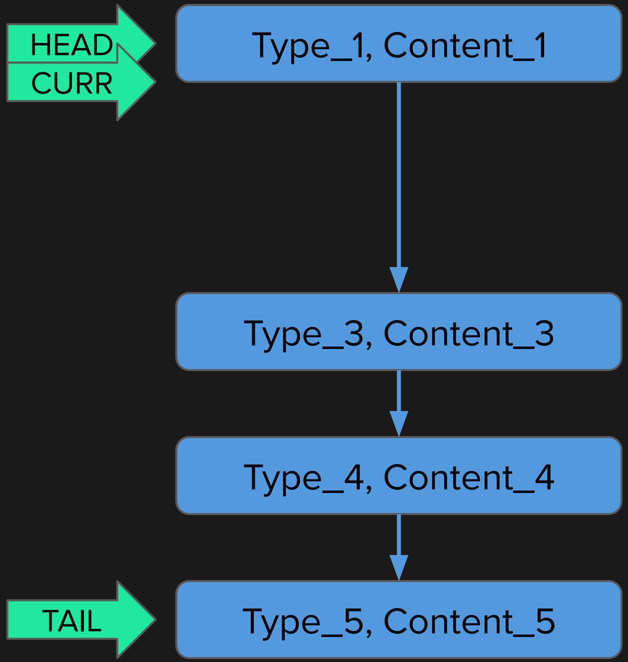
The event queue

```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
```



The event queue

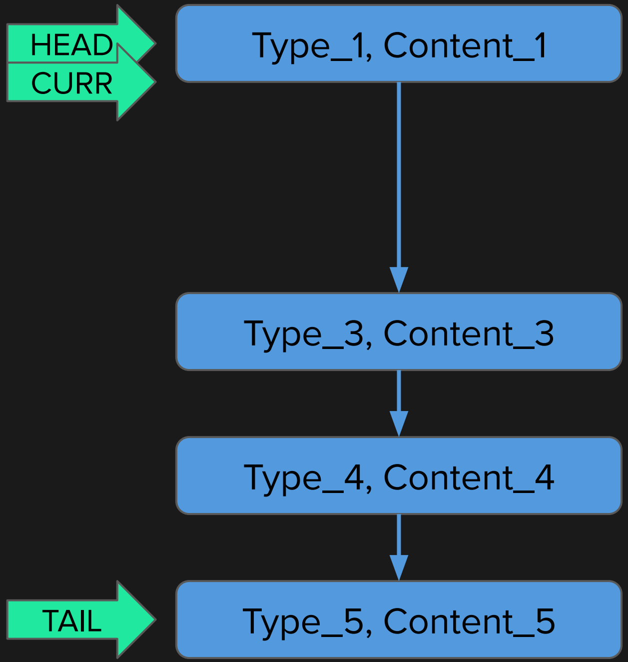
```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
```



Erlang Solutions

The event queue

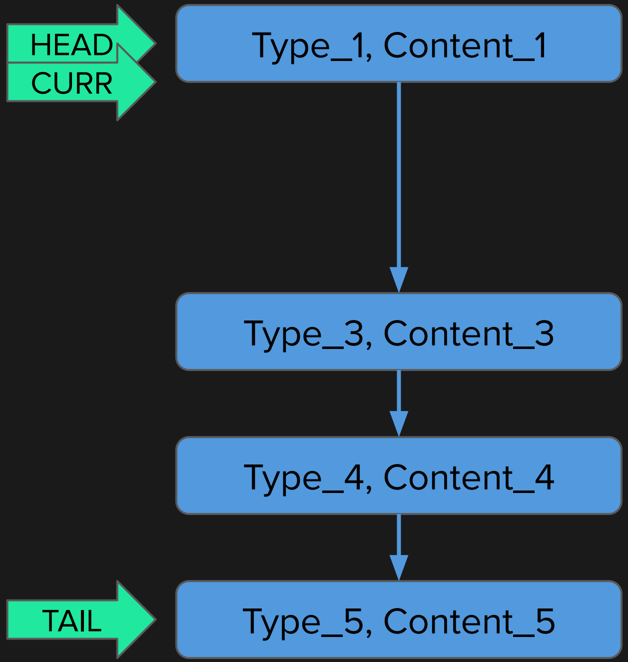
```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
```



Erlang Solutions

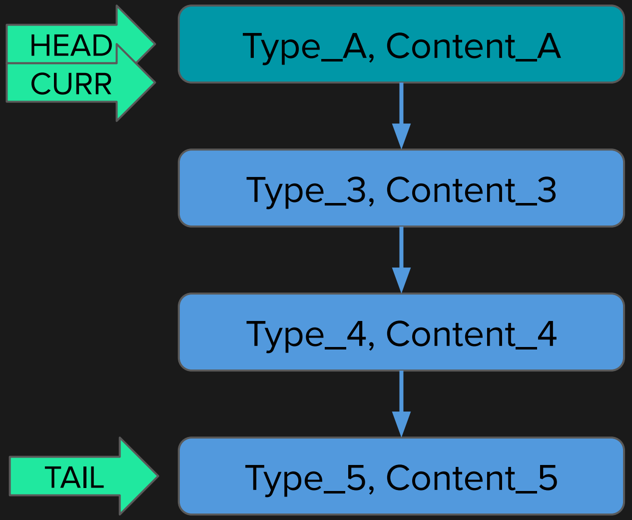
The event queue

```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
    {keep_state_and_data, [{next_event, TypeA, ContentA}]};
```



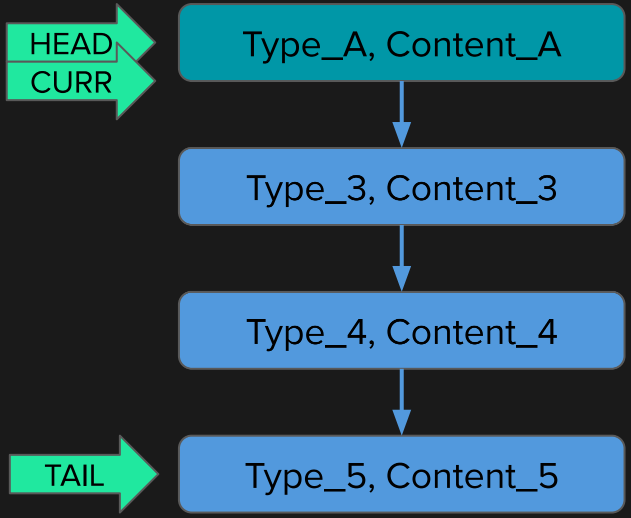
The event queue

```erlang
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
    {keep_state_and_data, [{next_event, TypeA, ContentA}]};
```



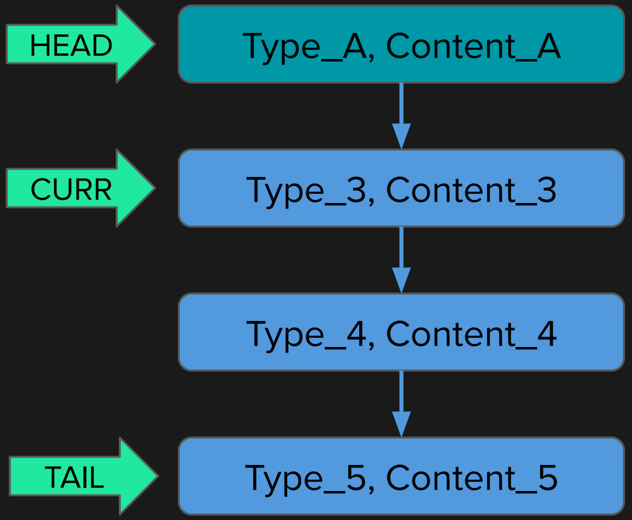
The event queue

```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
    {keep_state_and_data, [{next_event, TypeA, ContentA}]};
...
handle_event(TypeA, ContentA, State2, Data) ->
    {keep_state_and_data, [postpone]};
```



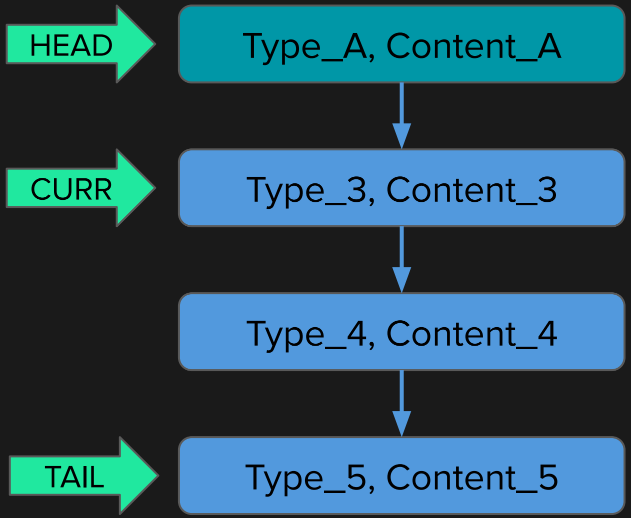
The event queue

```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
    {keep_state_and_data, [{next_event, TypeA, ContentA}]};
...
handle_event(TypeA, ContentA, State2, Data) ->
    {keep_state_and_data, [postpone]};
```



The event queue

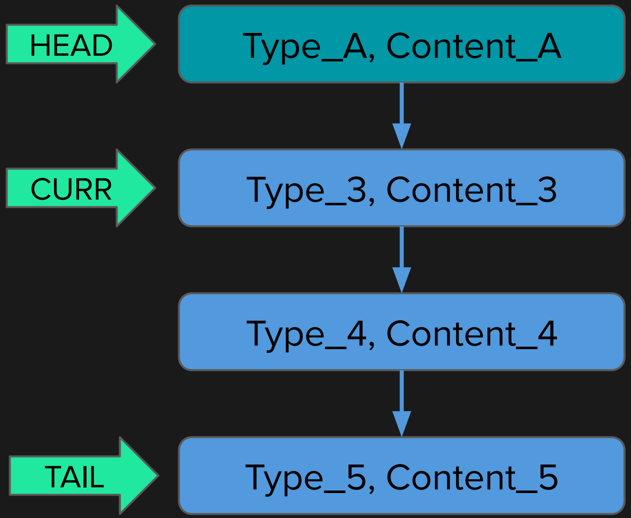
```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
    {keep_state_and_data, [{next_event, TypeA, ContentA}]};
...
handle_event(TypeA, ContentA, State2, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type3, Content3, State2, Data) ->
```



Erlang Solutions

The event queue

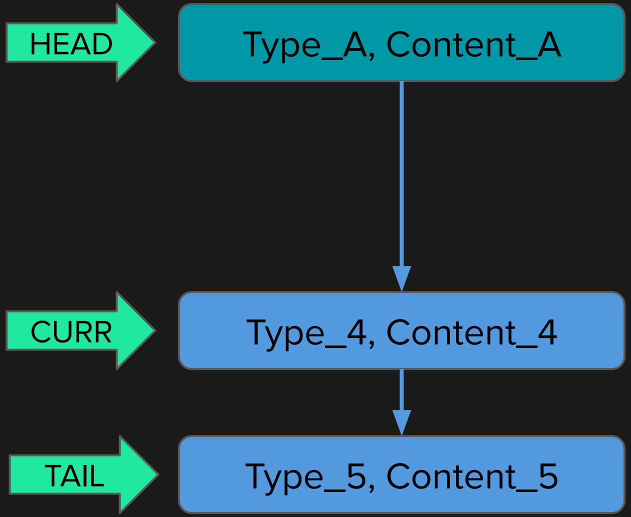
```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
    {keep_state_and_data, [{next_event, TypeA, ContentA}]};
...
handle_event(TypeA, ContentA, State2, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type3, Content3, State2, Data) ->
    keep_state_and_data;
```



Erlang Solutions

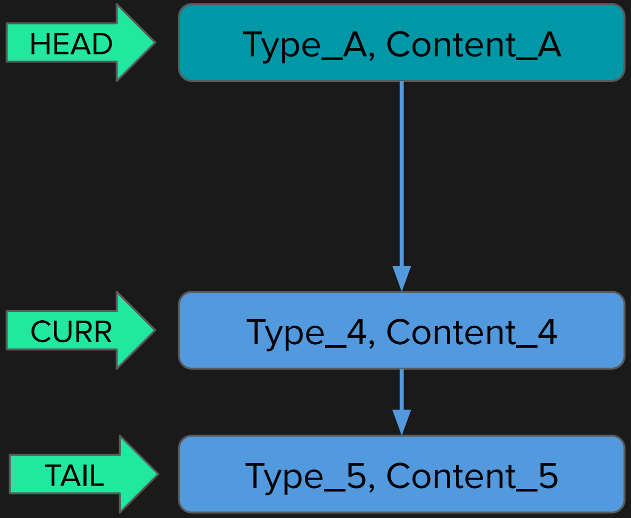
The event queue

```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
    {keep_state_and_data, [{next_event, TypeA, ContentA}]};
...
handle_event(TypeA, ContentA, State2, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type3, Content3, State2, Data) ->
    keep_state_and_data;
```



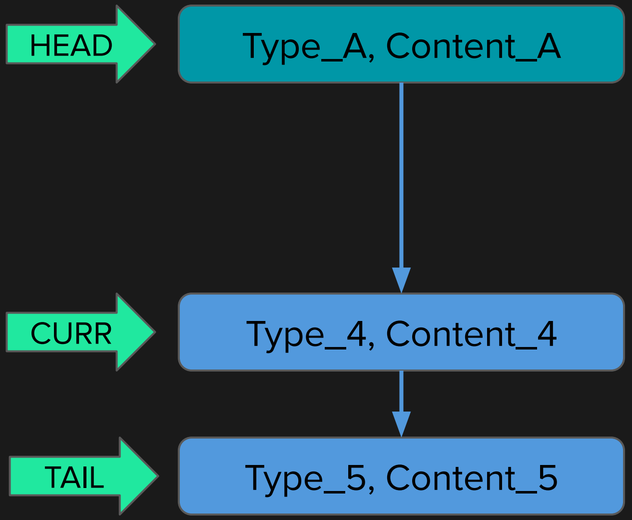
The event queue

```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
    {keep_state_and_data, [{next_event, TypeA, ContentA}]};
...
handle_event(TypeA, ContentA, State2, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type3, Content3, State2, Data) ->
    keep_state_and_data;
...
handle_event(Type4, Content4, State2, Data) ->
```



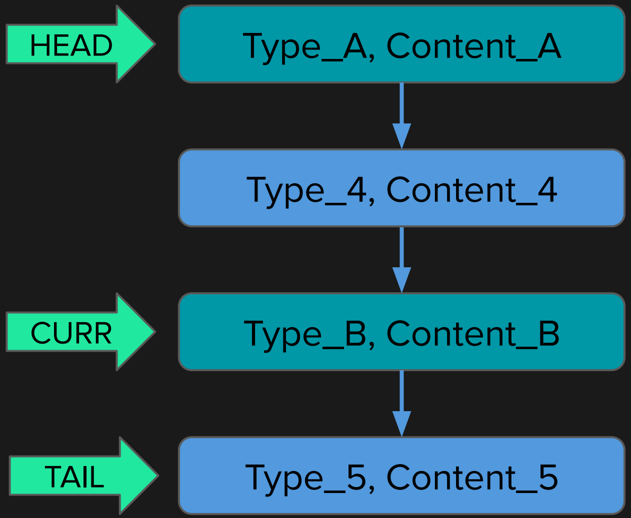
The event queue

```
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
    {keep_state_and_data, [{next_event, TypeA, ContentA}]};
...
handle_event(TypeA, ContentA, State2, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type3, Content3, State2, Data) ->
    keep_state_and_data;
...
handle_event(Type4, Content4, State2, Data) ->
    {keep_state_and_data,
        [postpone, {next_event, TypeB, ContentB}]};
```



The event queue

```erlang
handle_event(Type1, Content1, State1, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type2, Content2, State1, Data) ->
    {next_state, State2};
...
handle_event(Type1, Content1, State2, Data) ->
    {keep_state_and_data, [{next_event, TypeA, ContentA}]};
...
handle_event(TypeA, ContentA, State2, Data) ->
    {keep_state_and_data, [postpone]};
...
handle_event(Type3, Content3, State2, Data) ->
    keep_state_and_data;
...
handle_event(Type4, Content4, State2, Data) ->
    {keep_state_and_data,
        [postpone, {next_event, TypeB, ContentB}]};
```



A mental picture

- Only event one queue that is an extension of the process mailbox, with:
  - A *head* pointing at the oldest event;
  - A *current* pointing at the next event to be processed.
  - A *tail* pointing at the youngest event;

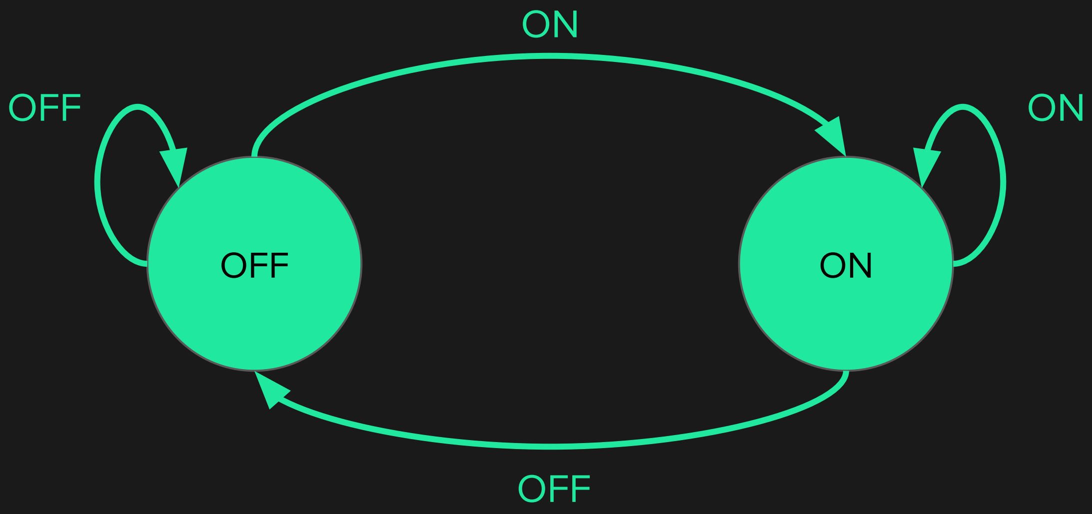
Erlang Solutions

gen_statem

Managing accidental complexity

Erlang Solutions

An archetypical example



```erlang
handle_call(on, _From, {off, Light}) ->
    on = request(on, Light),
    {reply, on, {on, Light}};
handle_call(off, _From, {on, Light}) ->
    off = request(off, Light),
    {reply, off, {off, Light}};
handle_call(on, _From, {on, Light}) ->
    {reply, on, {on, Light}};
handle_call(off, _From, {off, Light}) ->
    {reply, off, {off, Light}}.
```

All requests to the light are *synchronous*

```
handle_call(on, From, {off, undefined, Light}) ->
    Ref = request(on, Light),
    {noreply, {off, {on, Ref, From}, Light}}.
handle_call(off, From, {on, undefined, Light}) ->
    Ref = request(off, Light),
    {noreply, {on, {off, Ref, From}, Light}}.

handle_call(off, _From, {on, {off, _, _}, Light} = State) ->
    {reply, turning_off, State}.  %% ???
handle_call(on, _From, {off, {on, _, _}, Light} = State) ->
    {reply, turning_on, State}. %% ???
handle_call(off, _From, {off, {on, _, _}, Light} = State) ->
    {reply, turning_on_wait, State}.  %% ???
handle_call(on, _From, {on, {off, _, _}, Light} = State) ->
    {reply, turning_off_wait, State}. %% ???

handle_info(Ref, {State, {Request, Ref, From}, Light}) ->
    gen_server:reply(From, Request),
    {noreply, {Request, undefined, Light}}.
```

What do we do with queueing requests now? **Careful with user-level callstacks!**

```
off({call, From}, off, {undefined, Light}) ->
    {keep_state_and_data, [{reply, From, off}]};
off({call, From}, on, {undefined, Light}) ->
    Ref = request(on, Light),
    {keep_state, {{Ref, From}, Light}, []};
off({call, From}, _, _) ->
    {keep_state_and_data, [postpone]};
off(info, {Ref, Response}, {{Ref, From}, Light}) ->
    {next_state, Response, {undefined, Light}, [{reply, From, Response}]}.

on({call, From}, on, {undefined, Light}) ->
    {keep_state_and_data, [{reply, From, on}]};
on({call, From}, off, {undefined, Light}) ->
    Ref = request(off, Light),
    {keep_state, {{Ref, From}, Light}, []};
on({call, From}, _, _) ->
    {keep_state_and_data, [postpone]};
on(info, {Ref, Response}, {{Ref, From}, Light}) ->
    {next_state, Response, {undefined, Light}, [{reply, From, Response}]}.
```

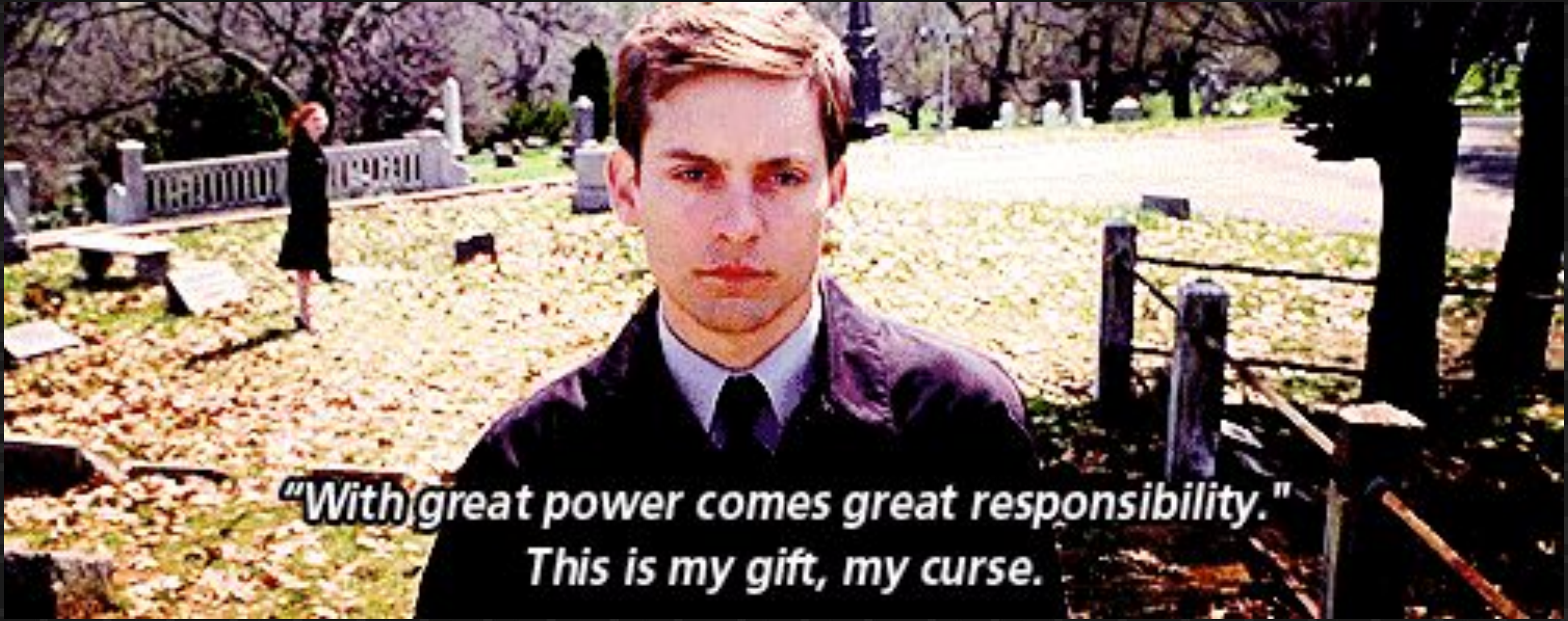
Try with a gen_statem… cool, we can “postpone” events! But it feels repetitive, there is **no code reuse**.

Apparent problems

- There is no global ordering
- Tying yourself to the actual ordering of events, leads to accidental complexity
- Complexity grows relative to the number of possible permutations of event sequences… unless you have a strategy for “reordering events”
- Code reuse becomes practically impossible

Ulf Wiger: Death by accidental complexity

```erlang
handle_event({call, From}, State, State, {undefined, Light}) ->
    {keep_state_and_data, [{reply, From, State}]};
handle_event({call, From}, Request, State, {undefined, Light}) ->
    Ref = request(Request, Light),
    {keep_state, {{Ref, From}, Light}, []};
handle_event({call, _}, _, _, _) ->
    {keep_state_and_data, [postpone]};
handle_event(info, {Ref, Response}, State, {{Ref, From}, Light}) ->
    {next_state, Response, {undefined, Light}, [{reply, From, Response}]}.
```

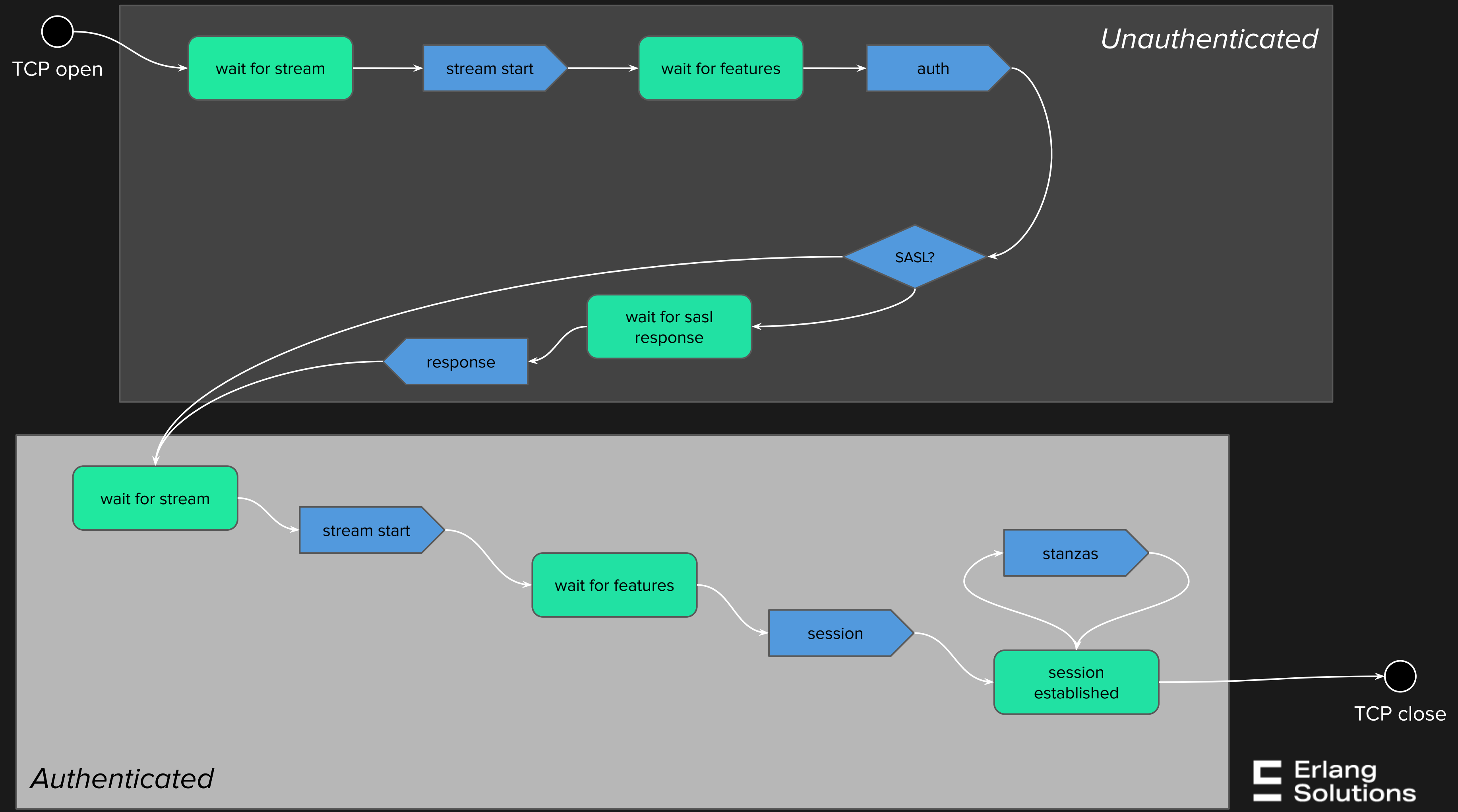


Erlang Solutions

A case study

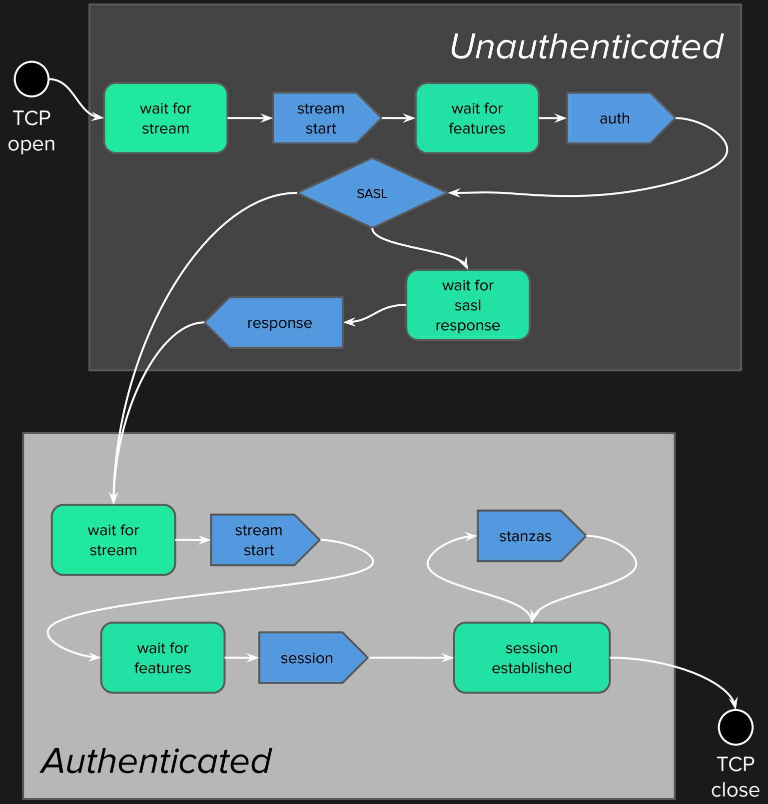
XMPP server and client state machines

Erlang Solutions

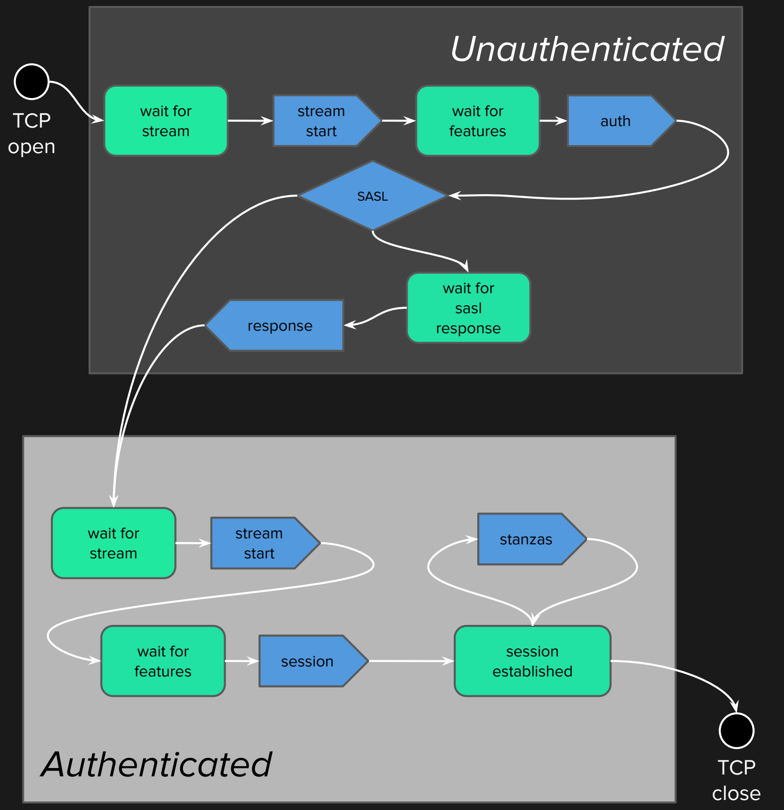


Erlang Solutions

```erlang
wait_for_stream(stream_start, Data#{auth = false}) ->
    {next, wait_for_feature};
wait_for_stream(stream_start, Data#{auth = true}) ->
    {next, wait_for_feature};

wait_for_feature(authenticate, Data#{auth = false}) ->
    {next, wait_for_stream, Data#{auth = true}};
wait_for_feature(session, Data#{auth = true) ->
    {next, session_established};

wait_for_sasl_response(auth_response, Data) ->
    {next, wait_for_stream, Data#data{auth = true}};

session_established(Event, Data) ->
    {next, session_established};

...

handle_info({tls, S, D}, Data#{socket = S}) ->
    Decrypt = tls:decrypt(D),
    XmlEl = exml:parse(Decrypt),
    Fsm ! {xml, XmlEl}.

handle_info({xml, XmlEl}, StName, Data#{socket = S}) ->
    S ! XmlEl;
```

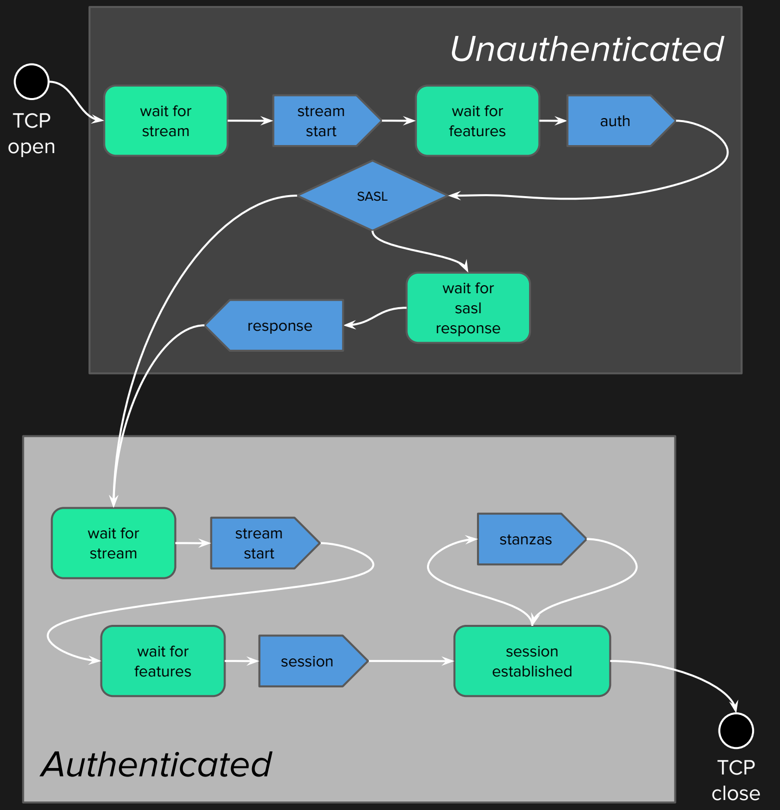


```erlang
wait_for_stream(stream_start, Data#{auth = false}) ->
    {next, wait_for_feature};
wait_for_stream(stream_start, Data#{auth = true}) ->
    {next, wait_for_feature};

wait_for_feature(authenticate, Data#{auth = false}) ->
    {next, wait_for_stream, Data#{auth = true}};
wait_for_feature(session, Data#{auth = true) ->
    {next, session_established};

wait_for_sasl_response(auth_response, Data) ->
    {next, wait_for_stream, Data#data{auth = true}};

session_established(Event, Data) ->
    {next, session_established};

...

handle_info({tls, S, D}, Data#{socket = S}) ->
    Decrypt = tls:decrypt(D),
    XmlEl = exml:parse(Decrypt),
    Fsm ! {xml, XmlEl}.

handle_info({xml, XmlEl}, StName, Data#{socket = S}) ->
    S ! XmlEl;
```



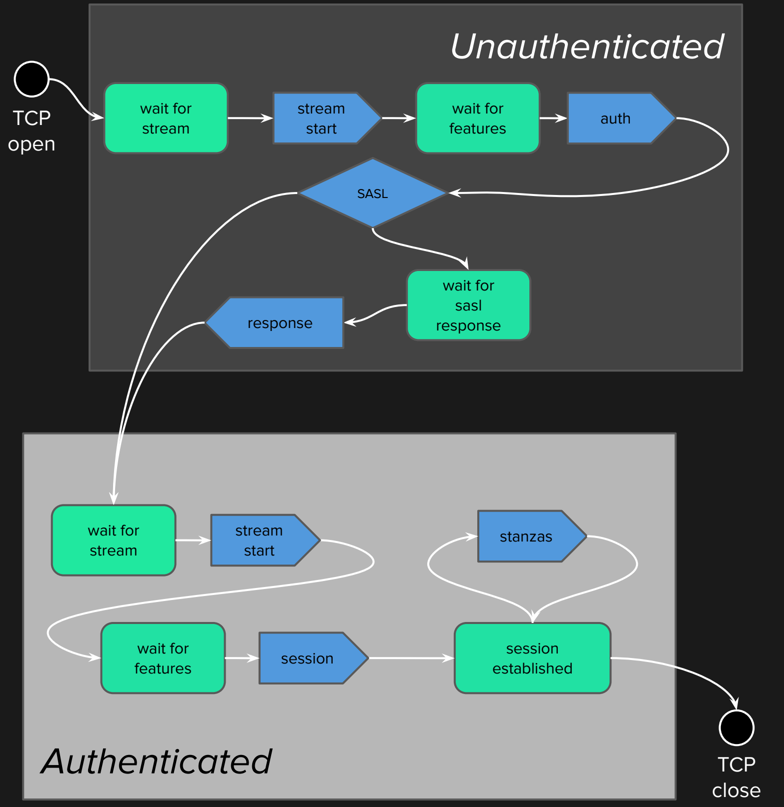
Erlang Solutions

```erlang
wait_for_stream(stream_start, Data#{auth = false}) ->
    {next, wait_for_feature};
wait_for_stream(stream_start, Data#{auth = true}) ->
    {next, wait_for_feature};

wait_for_feature(authenticate, Data#{auth = false}) ->
    {next, wait_for_stream, Data#{auth = true}};
wait_for_feature(session, Data#{auth = true) ->
    {next, session_established};

wait_for_sasl_response(auth_response, Data) ->
    {next, wait_for_stream, Data#data{auth = true}};

session_established(Event, Data) ->
    {next, session_established};

...

handle_info({tls, S, D}, Data#{socket = S}) ->
    Decrypt = tls:decrypt(D),
    XmlEl = exml:parse(Decrypt),
    Fsm ! {xml, XmlEl}.

handle_info({xml, XmlEl}, StName, Data#{socket = S}) ->
    S ! XmlEl;
```



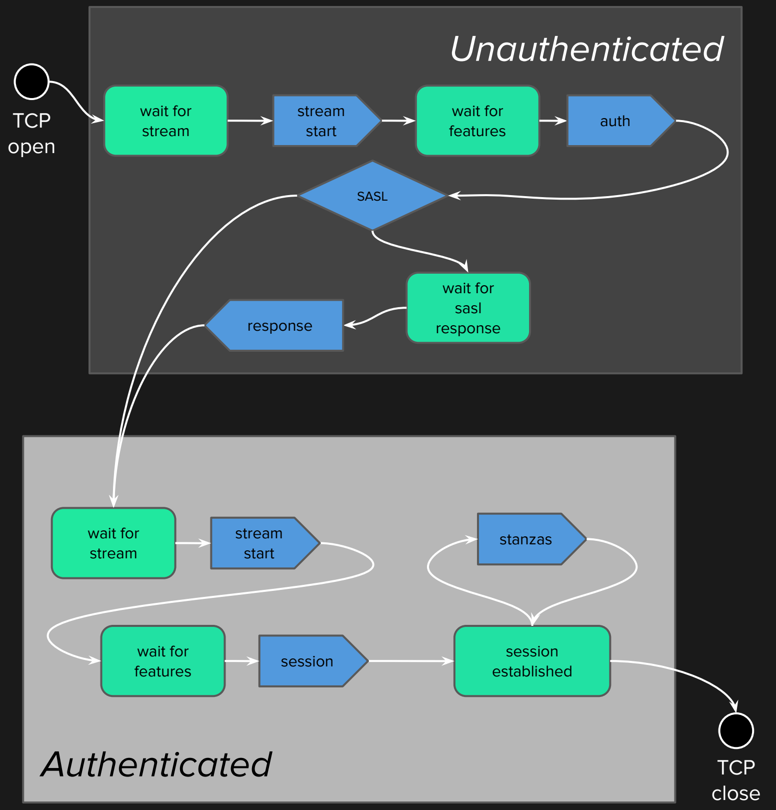
Erlang Solutions

```erlang
wait_for_stream(stream_start, Data#{auth = false}) ->
    {next, wait_for_feature};
wait_for_stream(stream_start, Data#{auth = true}) ->
    {next, wait_for_feature};

wait_for_feature(authenticate, Data#{auth = false}) ->
    {next, wait_for_stream, Data#{auth = true}};
wait_for_feature(session, Data#{auth = true) ->
    {next, session_established};

wait_for_sasl_response(auth_response, Data) ->
    {next, wait_for_stream, Data#data{auth = true}};

session_established(Event, Data) ->
    {next, session_established};

...

handle_info({tls, S, D}, Data#{socket = S}) ->
    Decrypt = tls:decrypt(D),
    XmlEl = exml:parse(Decrypt),
    Fsm ! {xml, XmlEl}.

handle_info({xml, XmlEl}, StName, Data#{socket = S}) ->
    S ! XmlEl;
```



Erlang Solutions

```erlang
wait_for_stream(stream_start, Data#{auth = false}) ->
    {next, wait_for_feature};
wait_for_stream(stream_start, Data#{auth = true}) ->
    {next, wait_for_feature};

wait_for_feature(authenticate, Data#{auth = false}) ->
    {next, wait_for_stream, Data#{auth = true}};
wait_for_feature(session, Data#{auth = true) ->
    {next, session_established};

wait_for_sasl_response(auth_response, Data) ->
    {next, wait_for_stream, Data#data{auth = true}};

session_established(Event, Data) ->
    {next, session_established};

...

handle_info({tls, S, D}, Data#{socket = S}) ->
    Decrypt = tls:decrypt(D),
    XmlEl = exml:parse(Decrypt),
    Fsm ! {xml, XmlEl}.

handle_info({xml, XmlEl}, StName, Data#{socket = S}) ->
    S ! XmlEl;
```



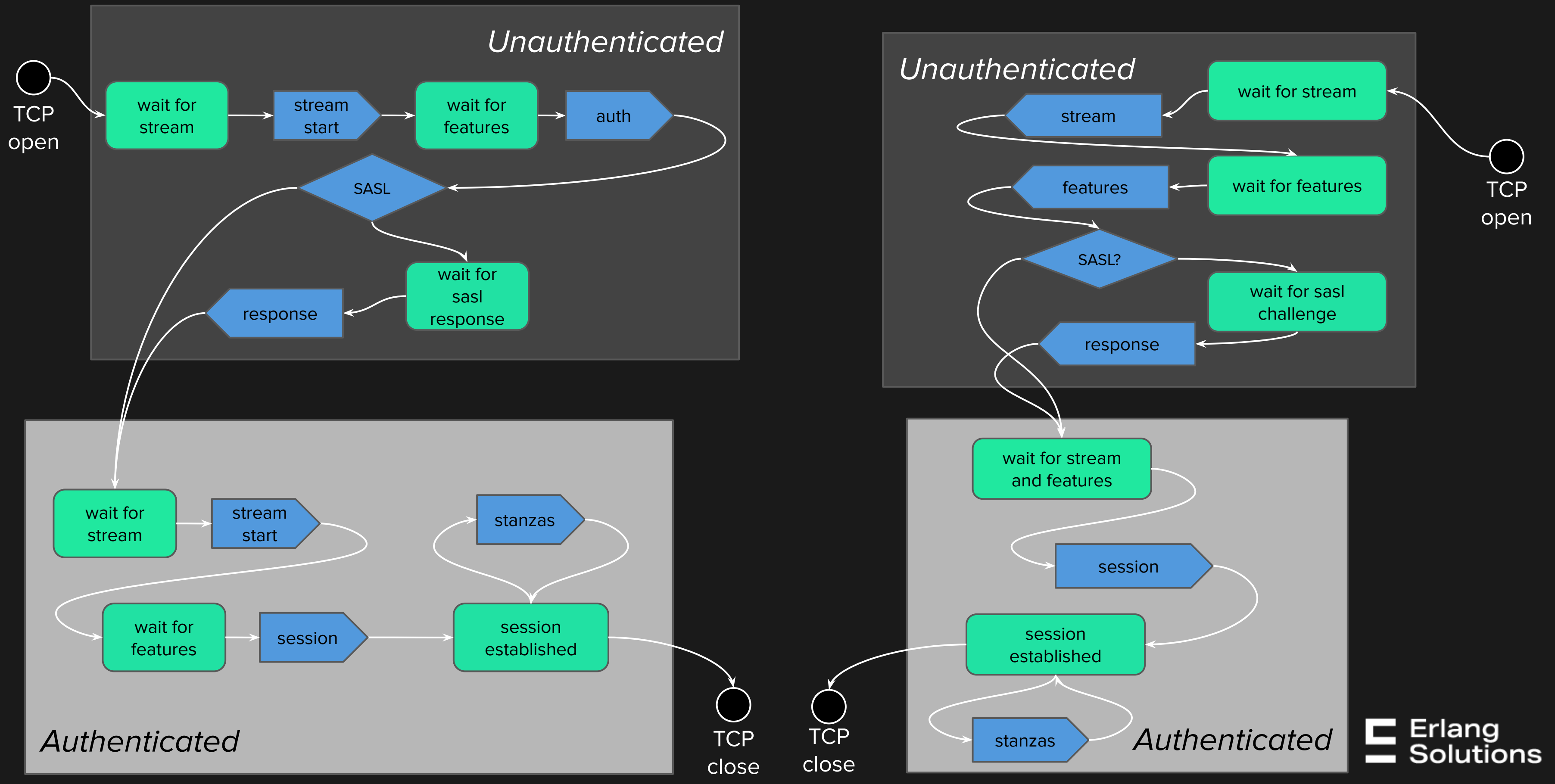
Erlang Solutions

Finite State Machine

We are using an automaton more powerful that is really needed!

Server

Client



Complex States

XMPP client

```
handle_event(Type, stream, [stream, features, sasl_ch, wait_for, AuthData, not_auth], _)

handle_event(Type, features, [features, sasl_ch, wait_for, AuthData, not_auth], _)

handle_event(Type, sasl_ch, [sasl_ch, wait_for, AuthData, not_auth], _)

handle_event(Type, stream, [stream, features, wait_for, Sess, auth], _)

handle_event(Type, features, [features, wait_for, Sess, auth], _)

handle_event(Type, Content, [session_established, Sess, auth], _)
```

Complex States

XMPP server

```
handle_event(Type, stream, [before_auth, wait_for, stream], _)

handle_event(Type, auth, [before_auth, wait_for, features, Retries, AuthData], _)

handle_event(Type, sasl_ch, [during_auth, wait_for, sasl_response, Retries, AuthData], _)

handle_event(Type, stream, [auth, wait_for, stream], _)

handle_event(Type, session, [auth, wait_for, features, Features], _)

handle_event(Type, Content, [auth, session_established], _)
```

Complex Events

XMPP server

```
handle_event(info, {tcp, Socket, Payload), StateName, #{parser = P, crypto = C} = StateData) ->
        {P1, C1, XmlElements} = decode(P, decrypt(C, Payload)),
        StreamEvents = [ {next_event, internal, El} || #xmlel{} = El <- XmlElements ],
        {next_state, StateName, StateData#{parser = P1, crypto = C1}, StreamEvents};
```

Erlang Solutions

Erlang Solutions

@NelsonVides

esl/MongooseIM

MONGOOSEIM

esl/amoc

Nelson Vides
Senior Erlang Consultant and Core MongooseIM developer
nelson.vides@erlang-solutions.com

Erlang Solutions

gen_statem Unveiled

Questions?

Erlang Solutions

Contact us

London | Stockholm | Krakow | Budapest | US Remote

www.erlang-solutions.com

general@erlang-solutions.com