TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Programming Distributed Systems

03 Causality, Vector clocks, OTP

Annette Bieniusa, Peter Zeller

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2018

Motivation

- Causality is fundamental to many problems occurring in distributed computing
- *Examples*: Determining a consistent recovery point, detecting race conditions, exploitation of parallelism
- The happens-before relation of events is often also called *causality relation* (Schwarz and Mattern 1994).

An event e may causally affect another event e' if and only if $e \rightarrow e'$.

- The happens-before order $\rightarrow$ indicates only *potential* causal relationship.
- Tracking whether an event indeed is a cause of another event is much more involved and requires more complex dependency analysis.

Overview

- Causality Tracking with Vector clocks
- Causal Broadcast revisited
- Erlang OTP

Causality tracking with Vector clocks

Causal Histories

- We here distinguish three types of events occurring in a process:
  - Send events
  - Receive events
  - Local / internal events
- Let E_i denote the set of events occurring at process p_i and E the set of all executed events:

$$E = E_1 \cup \cdots \cup E_n$$

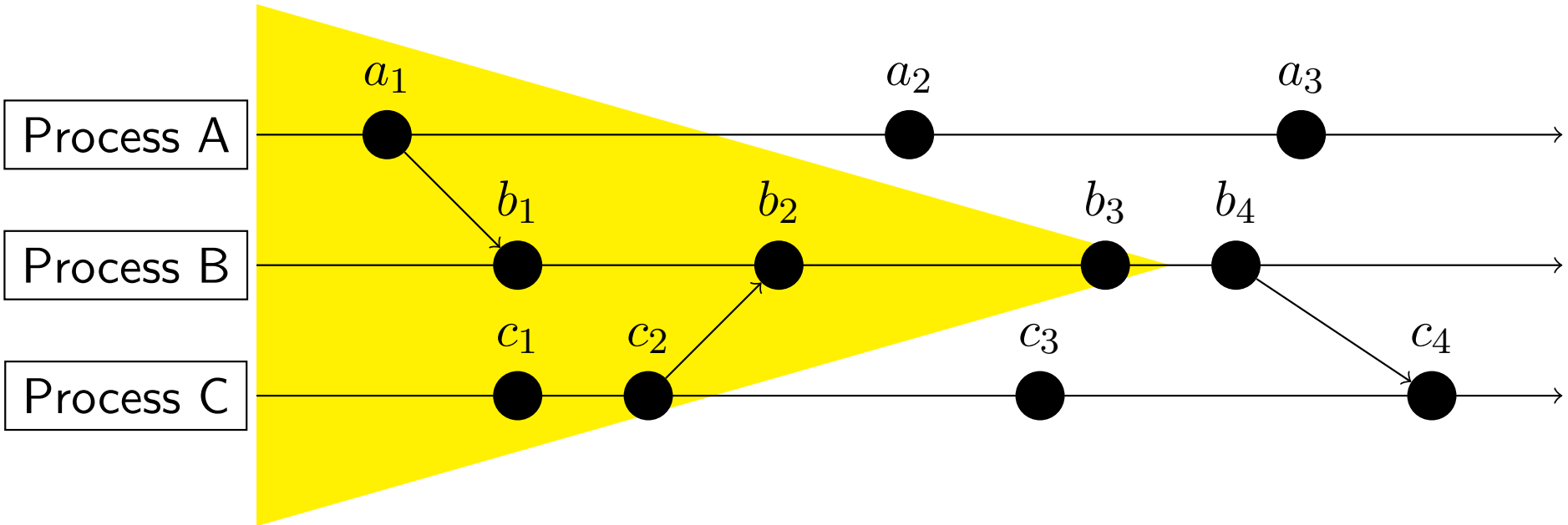
- The *causal history* of an event $e \in E$ is defined as

$$C(e) = \{e' \in E \mid e' \rightarrow e\} \cup \{e\}$$

- Note: Just a different representation of happens-before:

$$e' \rightarrow e \quad \Leftrightarrow \quad e' \neq e \wedge e' \in C(e)$$

Example: Causal history of b_3



$$C(b_3) = \{a_1, b_1, b_2, b_3, c_1, c_2\}$$

Tracking causal histories

Each process p_i stores current causal history as set of events C_i.

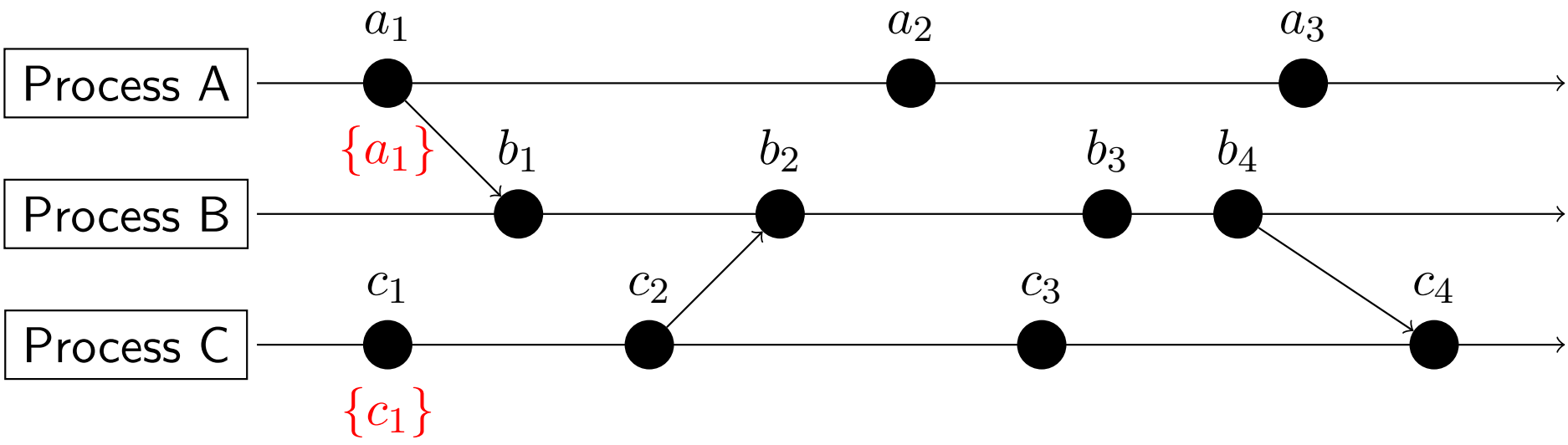
- Initially, $C_i \leftarrow \emptyset$
- On each local event e at process p_i, the event is added to the set:

$$C_i \leftarrow C_i \cup \{e\}$$

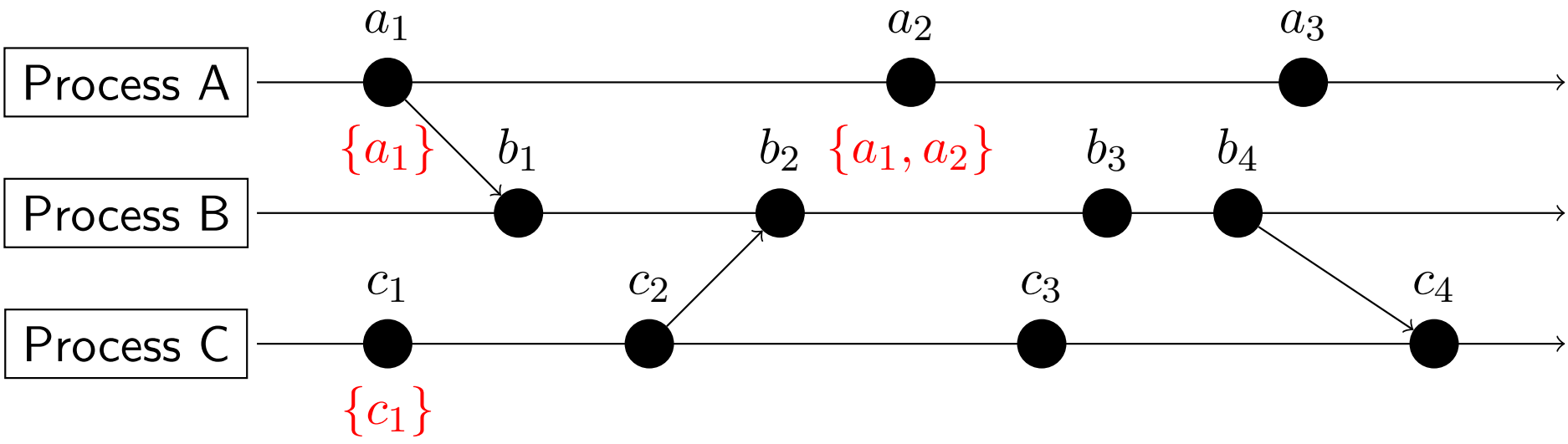
- On sending a message m, p_i updates C_i as for a local event and attaches the new value of C_i to m.
- On receiving message m with causal history $C(m)$, p_i updates C as for a local event. Next, p_i adds the causal history from C(m):

$$C_i \leftarrow C_i \cup C(m)$$

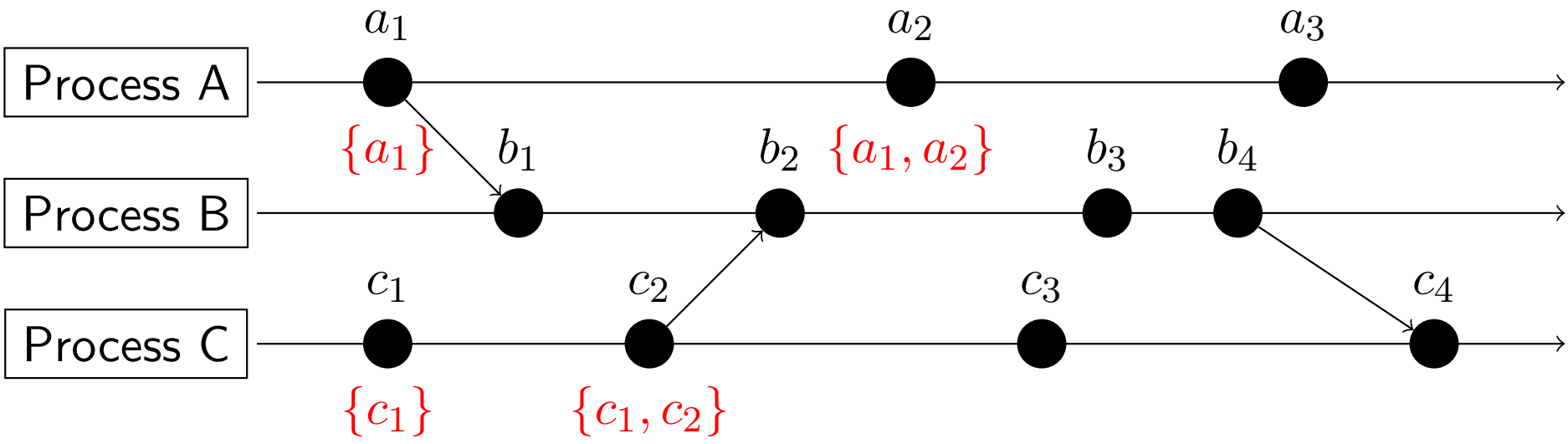
Example: Causal histories



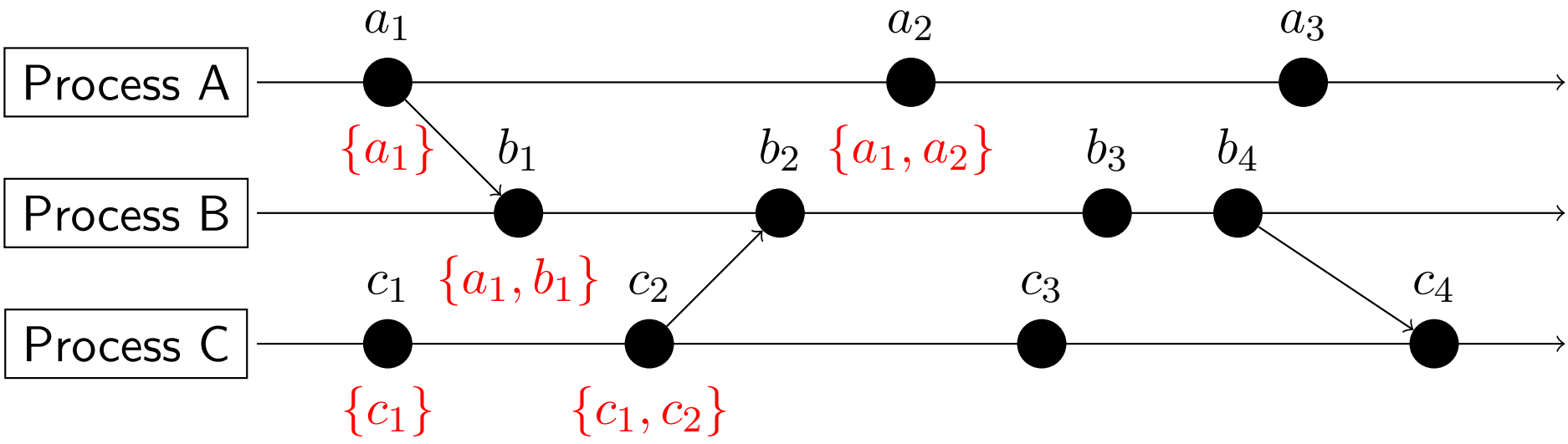
Example: Causal histories



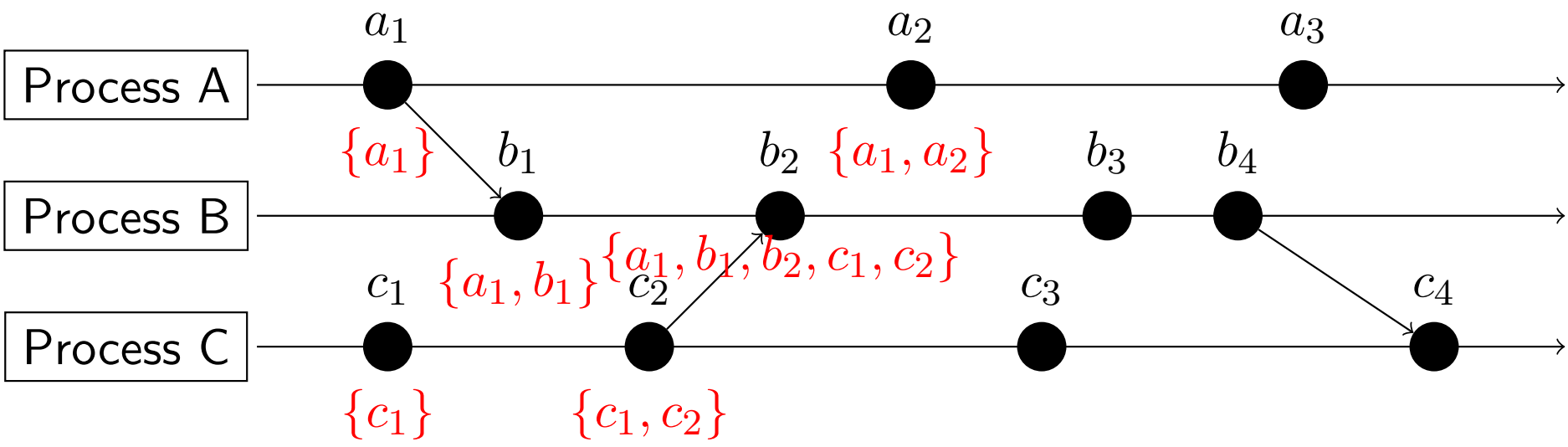
Example: Causal histories



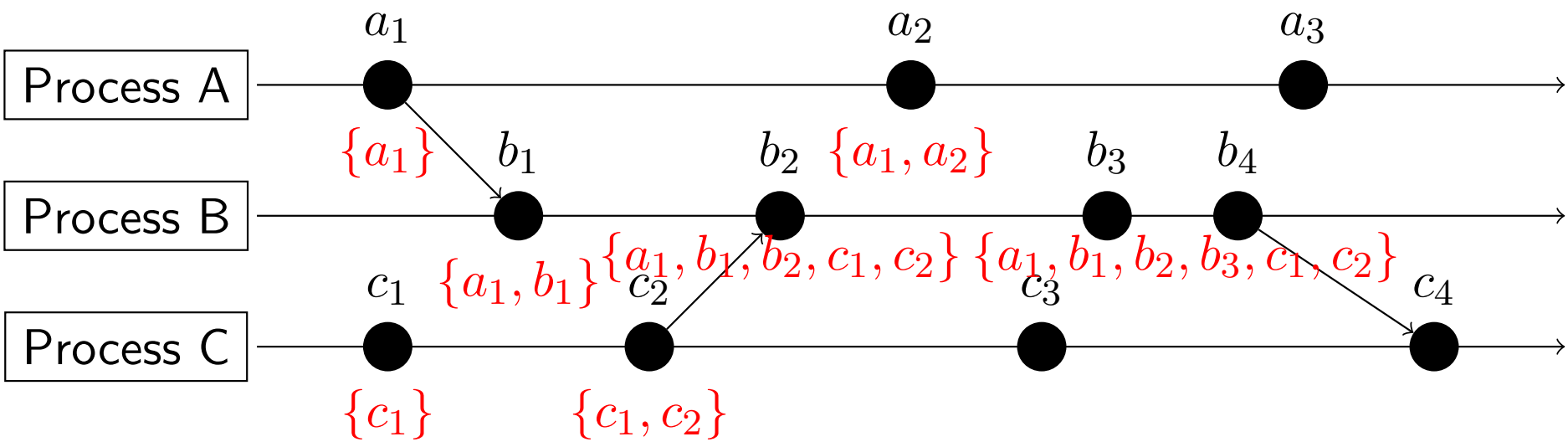
Example: Causal histories



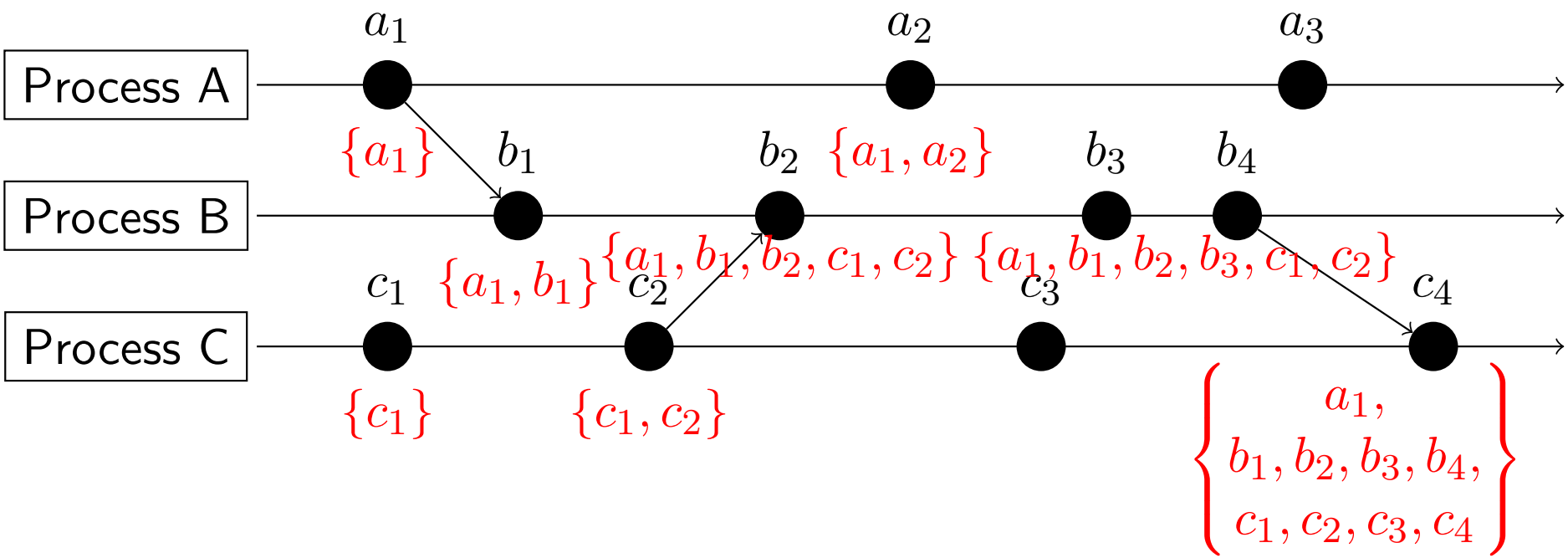
Example: Causal histories



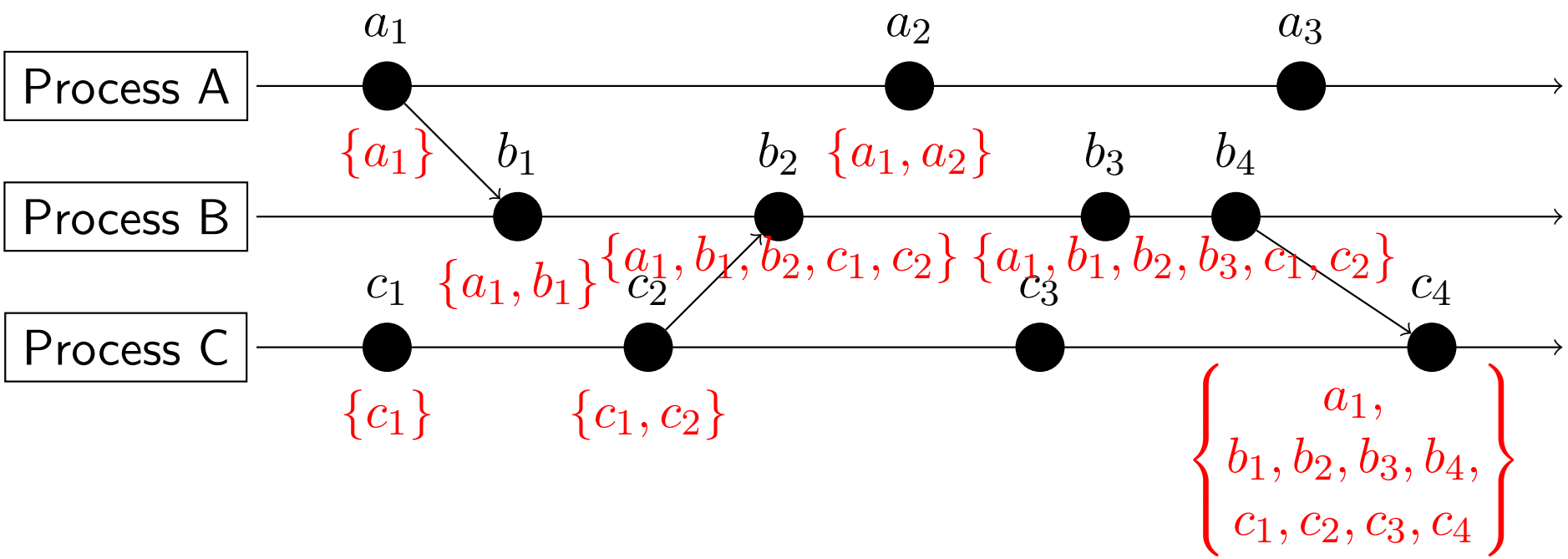
Example: Causal histories



Example: Causal histories

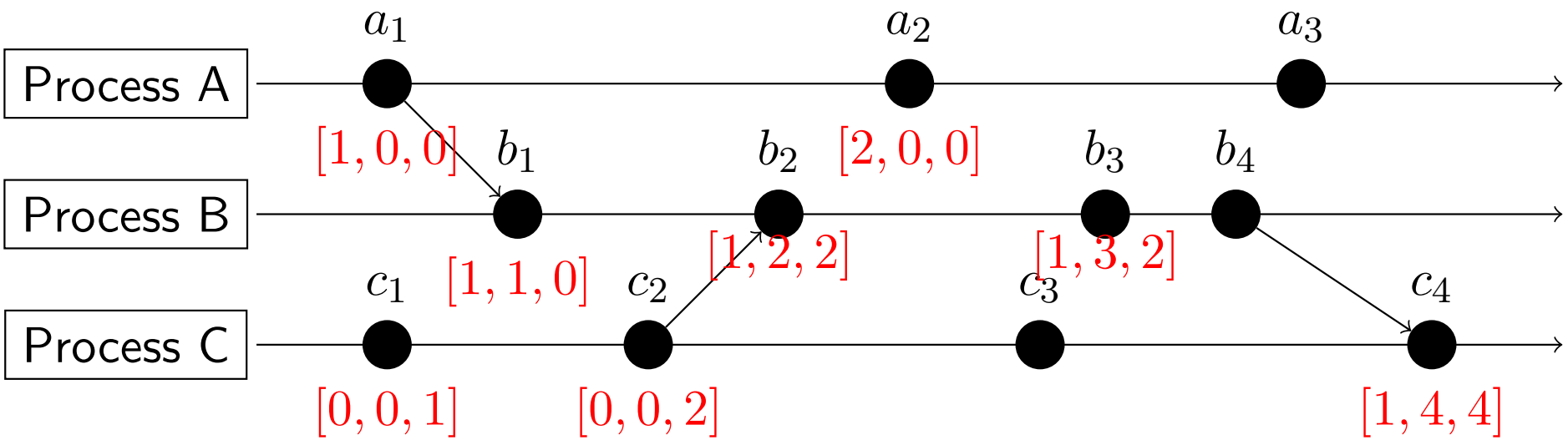


Example: Causal histories



Can we represent causal histories more efficiently?

Example: Efficient representation of causal histories



Efficient representation of causal histories

- Vector clock $V(e)$ as efficient representation of $C(e)$.
- Vector clock is a mapping from processes to natural numbers:
  - Example: $[p_1 \mapsto 3, p_2 \mapsto 4, p_3 \mapsto 1]$
  - If processes are numbered $1, \ldots, n$, this mapping can be represented as a vector, e.g. [3, 4, 1]
  - Intuitively: $p_1 \mapsto 3$ means "observed 3 events from process p_1"

Formal Construction

- Assume processes are numbered 1, . . . , n
- Let $E_k = \{e_{k_1}, e_{k_2}, \dots\}$ be the events of process k
  - Totally ordered: $e_{k_1} \rightarrow e_{k_2}, e_{k_2} \rightarrow e_{k_3}, \dots$
- Let $C(e)[k] = C(e) \cap E_k$ denote the projection of $C(E)$ on process k.

$$C(e) = C(e)[1] \cup \dots \cup C(e)[n]$$

- Now, if $e_{k_j} \in C(e)[k]$, then by definition it holds that $e_{k_1}, \dots, e_{k_j} \in C(e)[k]$
- The set $C(e)[k]$ is thus sufficiently characterized by the largest index of its events, i.e. its cardinality!
- Summarize $C(e)$ by an n-dimensional vector $V(e)$ such that for $k = 1, \dots, n$:

$$V(e)[k] = |C(e)[k]|$$

Note: Both representations are lattices with a lower bound

Operator	Causal history	Vector clock
$\bot$	$\emptyset$	$\lambda i.\ 0$
$A \leq B$	$A \subseteq B$	$\forall i.\ A[i] \leq B[i]$
$A \geq B$	$A \supseteq B$	$\forall i.\ A[i] \geq B[i]$
$A \sqcup B$	$A \cup B$	$\lambda i.\ max(A[i], B[i])$
$A \sqcap B$	$A \cap B$	$\lambda i.\ min(A[i], B[i])$

- $\bot$: bottom, or smallest element
- $A \sqcup B$: least upper bound, or join, or supremum
- $A \sqcap B$: greatest lower bound, or meet, or infimum

Tracking causal histories

Each process p_i stores current causal history as set of events C_i.

- Initially, $C_i \leftarrow \emptyset$
- On each local event e at process p_i, the event is added to the set: $C_i \leftarrow C_i \cup \{e\}$
- On sending a message m, p_i updates C_i as for a local event and attaches the new value of C_i to m.
- On receiving message m with causal history $C(m)$, p_i updates C_i as for a local event. Next, p_i adds the causal history from C(m):

$$C_i \leftarrow C_i \cup C(m)$$

Tracking causal histories

Each process p_i stores current causal history as set of events C_i.

- Initially, $C_i \leftarrow \perp$
- On each local event e at process p_i, the event is added to the set: $C_i \leftarrow C_i \cup \{e\}$
- On sending a message m, p_i updates C_i as for a local event and attaches the new value of C_i to m.
- On receiving message m with causal history $C(m)$, p_i updates C_i as for a local event. Next, p_i adds the causal history from C(m):

$$C_i \leftarrow C_i \sqcup C(m)$$

Vector time

Each process p_i stores current causal history as a vector clock V_i.

- Initially, $V_i[k] \leftarrow \bot$
- On each local event, process p_i increments its on entry in V_i as follows: $V_i[i] \leftarrow V_i[i] + 1$
- On sending a message m, p_i updates V_i as for a local event and attaches new value of V_i to m.
- On receiving message m with vector time $V(m)$, p_i increments its own entry as for a local event. Next, p_i updates its current V_i by joining $V(m)$ and V_i:

$$V_i \leftarrow V_i[k] \sqcup V(m)$$

Relating vector times

Let u, v denote time vectors. We say that

- $u \leq v$ iff $u[k] \leq u[k]$ for $k = 1, \ldots, n$
- $u < v$ iff $u \leq v$ and $u \neq v$
- $u \parallel v$ iff neither $u \leq v$ nor $v \leq u$

For two events e and e', it holds that $e \rightarrow e' \quad \Leftrightarrow \quad V(e) < V(e')$

- Proof: By construction.

How does vector time relate to Lamport timestamps?

- Both are logical clocks, counting events.
- Lamport time (and real time) are insufficient to characterize causality and can't be used to prove that events are not causally related

Causal Broadcast (RCO): Algorithm 2 (Waiting)

```
State:
  pending //set of messages that cannot be delivered yet
  VC // vector clock
Upon Init do:
  pending <- ∅;
  forall p_i ∈ Π do: VC[p_i] <- 0;

Upon rco-Broadcast(m) do
  trigger rco-Deliver(self, m);
  trigger rb-Broadcast(VC, m);
  VC[self] <- VC[self] + 1;

Upon rb-Deliver(p, VC_m, m) do
  if ( p ≠ self ) then
    pending <- pending ∪ {(p, VC_m, m)};
    while exists (q, VC_{m_q}, m_q) ∈ pending, such that VC ≥ VC_{m_q} do
        pending <- pending \ {(q, VC_{m_q}, m_q)};
        trigger rco-Deliver(q, m_q);
        VC[q] <- VC[q] + 1;
```

Causal Broadcast (RCO): Algorithm 2 (Waiting)

```
State:
  pending //set of messages that cannot be delivered yet
  VC // vector clock
Upon Init do:
  pending <- ∅;
  forall p_i ∈ Π do: VC[p_i] <- 0;

Upon rco-Broadcast(m) do
  trigger rco-Deliver(self, m);
  trigger rb-Broadcast(VC, m);
  VC[self] <- VC[self] + 1;

Upon rb-Deliver(p, VC_m, m) do
  if ( p ≠ self ) then
    pending <- pending ∪ {(p, VC_m, m)};
    while exists (q, VC_{m_q}, m_q) ∈ pending, such that VC ≥ VC_{m_q} do
        pending <- pending \ {(q, VC_{m_q}, m_q)};
        trigger rco-Deliver(q, m_q);
        VC[q] <- VC[q] + 1;
```

Question: Why is it called “waiting”?

Limits of Causal Broadcast

- Processes can observe messages in different order
- Example: Replicated database handling bank accounts
- Initially, account A holds 1000 Euro.
- User deposits 150 Euro, triggers broadcast of message
  $m_1 =$ `'add 150 Euro to A'`
- Concurrently, bank initiates broadcast of message
  $m_2 =$ `'add 2% interest to A'`
- Diverging state!

$\Rightarrow$ Next lecture: Atomic broadcast!

Summary

- Causality important for many scenarios
- Causality not always sufficient
- Vector clocks:
  - Efficient representation of causal histories / happens-before
  - How many events from which process?
- Causal broadcast: Use vector clocks to deliver in causal order

Erlang OTP

Example: Echo server 1

```
-module(echo).
-export([start_link/0]).

start_link() ->
    {ok, spawn_link(fun() -> loop() end)}.

loop() ->
    receive
        {From, Msg} ->
            From ! Msg,
            loop();
        stop ->
            true
    end.
```

Example: Echo server client 1

```
-module(echo_client).

-export([test/0]).

test() ->
    {ok, Server1} = echo:start_link(),
    {ok, Server2} = echo:start_link(),

    Server1 ! {self(), hello},
    Server2 ! {self(), world},

    receive
        Msg1 -> io:format("Server 1 responded: ~p~n", [Msg1])
    end,
    receive
        Msg2 -> io:format("Server 2 responded: ~p~n", [Msg2])
    end.
```

Does this always work correctly?

Example: Echo server 2

```
-module(echo2).
-export([start_link/0]).

start_link() ->
    {ok, spawn_link(fun() -> loop() end)}.

loop() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            loop();
        stop ->
            true
    end.
```

Sending own process-id (`self()`), so that receiver can match answer to request.

Example: Echo client 2

```
-module(echo_client2).

-export([test/0]).

test() ->
    {ok, Server1} = echo2:start_link(),
    {ok, Server2} = echo2:start_link(),

    Server1 ! {self(), hello},
    Server2 ! {self(), world},

    receive
        {Server1, Msg1} -> io:format("1 responded: ~p~n", [Msg1])
    end,
    receive
        {Server2, Msg2} -> io:format("2 responded: ~p~n", [Msg2])
    end.
```

Example: Counting server

```
-module(counter).
-export([start_link/0, loop/1]).

start_link() ->
    {ok, spawn_link(?MODULE, loop, [0])}.

loop(Counter) ->
    receive
        {From, increment} ->
            From ! {self(), ok},
            loop(Counter + 1);
        {From, read} ->
            From ! {self(), Counter},
            loop(Counter);
        stop ->
            true
    end.
```

Records: Organizing complex state in a server

```
-record(person, {name, age, hobbies = []}).
```

Creating instances:

```
P = #person{name = "Hans", age = 7}
```

Accessing fields:

```
P#person.name
P#person.age
```

Updating record fields:

```
P#person{age = 8}
```

Pattern matching with records:

```
#person{name = Name, age = Age} = P
```

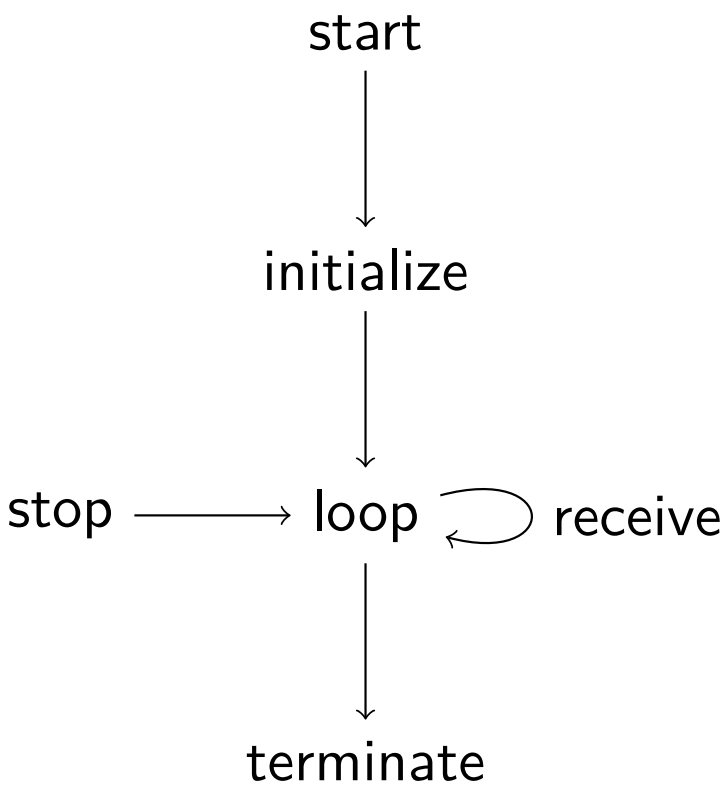
Using records as process state

```
-module(bounded_counter).
-export([start_link/1, loop/1, increment/1, read/1]).
-record(state, {limit, count}).

start_link(Limit) ->
    State = #state{limit = Limit, count = 0},
    {ok, spawn_link(?MODULE, loop, [State])}.

loop(State = #state{count = Counter, limit = Limit}) ->
    receive
        {From, increment} when Counter < Limit ->
            From ! {self(), ok},
            loop(State#state{count = Counter + 1});
        {From, increment} ->
            From ! {self(), {error, limit_reached}},
            loop(State);
        {From, read} ->
            From ! {self(), Counter},
            loop(State);
        stop ->
            true
    end.
```

Bounded Counter API (synchronous call)

```
increment(Server) ->
    Server ! {self(), increment},
    receive
        {Server, Msg} -> Msg
    end.

read(Server) ->
    Server ! {self(), read},
    receive
        {Server, Msg} -> Msg
    end.
```

Generic Client/Servers



Separating generic and specific parts

Generic	Specific (Counter)
Spawning the server	Initial State:
Storing the loop data	#state{limit = Limit, count = 0}
Sending requests to server	Handling of requests (increment, read)
Sending replies to client	
Receiving server replies	
Stopping	(cleaning up)

Separating generic and specific parts

Generic	Specific (Counter)
Spawning the server	Initial State:
Storing the loop data	#state{limit = Limit, count = 0}
Sending requests to server	Handling of requests (increment, read)
Sending replies to client	
Receiving server replies	
Stopping	(cleaning up)

Implement generic part once, use callbacks for specific parts

Specific part

```
-module(bounded_counter2).
-export([start_link/1, increment/1, read/1]).
-export([init/1, handle_call/3]).

-record(state, {limit, count}).

start_link(Limit) ->
    my_gen_server:start_link(?MODULE, [Limit], []).

increment(Server) ->
    my_gen_server:call(Server, increment).

read(Server) ->
    my_gen_server:call(Server, read).

init([Limit]) ->
    {ok, #state{limit = Limit, count = 0}}.

handle_call(increment, _From, State = #state{count = Counter, limit = Limit}) ->
    case Counter < Limit of
        true -> {reply, ok, State#state{count = Counter + 1}};
        false -> {reply, {error, limit_reached}, State}
    end;
handle_call(read, _From, State) ->
    {reply, State#state.count, State}.
```

Simple generic server

```
-module(my_gen_server).
-export([start_link/3, call/2]).

start_link(Module, Args, _Options) ->
    {ok, InitialState} = Module:init(Args),
    {ok, spawn_link(fun() -> loop(Module, InitialState) end)}.

call(P, Msg) ->
    P ! {call, self(), Msg},
    receive
        {reply, P, Response} ->
            Response
    end.

loop(Module, State) ->
    receive
        {call, From, Msg} ->
            {reply, Reply, NewState} =
                Module:handle_call(Msg, From, State),
            From ! {reply, self(), Reply},
            loop(Module, NewState)
    end.
```

Implementation in standard library: gen_server

- More robust than `my_gen_server`
  - Timeouts and monitors to handle failures
- Init called in new process
- More events:
  - `handle_call` and `gen_server:call` for synchronous requests
  - `handle_cast` and `gen_server:cast` for asynchronous requests
  - `handle_info` for other messages
- `handle_call` can reply later (e.g. handle reply in other process)
- callback `terminate` for cleaning up
- callback `code_change` for handling dynamic code reloading

Example: gen_server (1/2)

```
-module(bounded_counter3).
-behavior(gen_server).
-export([start_link/1, increment/1, read/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-record(state, {limit, count}).

start_link(Limit) ->
    gen_server:start_link(?MODULE, [Limit], []).


increment(Server) ->
    gen_server:call(Server, increment).

read(Server) ->
    gen_server:call(Server, read).

init([Limit]) ->
    {ok, #state{limit = Limit, count = 0}}.
```

Example: gen_server (2/2)

```
handle_call(increment, _From,
            State = #state{count = Counter, limit = Limit}) ->
    case Counter < Limit of
        true -> {reply, ok, State#state{count = Counter + 1}};
        false -> {reply, {error, limit_reached}, State}
    end;
handle_call(read, _From, State) ->
    {reply, State#state.count, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(_Msg, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Error handling in Erlang

Two kinds of errors:

- Predictable errors
  - Wrong user input, connection problem, error reading file
  - Often handled with special return values, e.g.

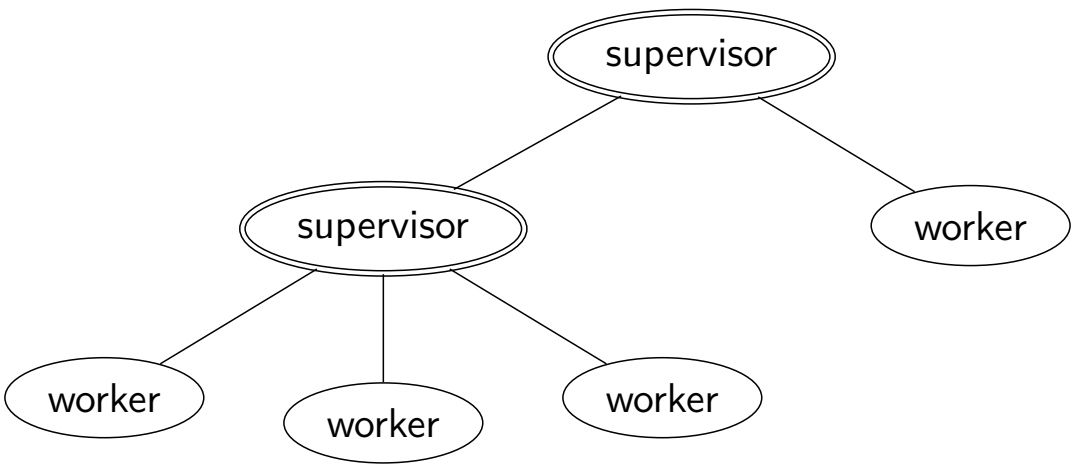
    ```
    read_file(Filename)-> {ok, Binary} | {error, Reason}
    ```
  - Sometimes handled with exceptions
- Unpredictable errors
  - Software bugs, corrupt state, system resources exhausted
  - Handled by monitoring whole processes ($\Rightarrow$ supervisors)

Linked processes and monitoring

- Processes can be linked
  - A link has no direction
  - `spawn_link` spawns a new process and links it to the current
    - Also: `link` and `unlink` functions
  - If a process terminates, all linked processed are notified:
    - by default linked process terminates as well (with same reason)
    - if `process_flag(trap_exit, true)` is set, a special message `{'EXIT', Pid, Reason}` is sent instead
- Processes can be monitored
  - Only one direction
  - If monitored process terminates, monitoring process receives message `{'DOWN', MonitorRef, Type, Object, Info}`

Supervisors

- Start child processes (with link)
- Trap exits
- Handle termination of child processes (e.g. restart)
- Cleanly terminate applications
- Usually organized hierarchical



Generic Supervisor

Just implement callback `init/1` to specify the supervisor.

```
{ok, {SupFlags,[ChildSpec]}}.
```

SupFlags is a tuple `{RestartStrategy, MaxRestart, MaxTime}`

Restart strategies:

- `one_for_one`: Restart only terminated process
- `one_for_all`: Restart all child processes
- `rest_for_one`: Restart all processes, that were started after the terminating process
- `simple_one_for_one`: Like `one_for_one`, but all children run the same code

MaxRestart and MaxTime:

- If more than `MaxRestart` restarts happen within `MaxTime` seconds, the supervisor terminates its children and then itself.

Supervisor Children

ChildSpec is a tuple

```
{ChildId, StartFunc, Restart, Shutdown, Type, Modules}
```

- `ChildId`: Name of the child
- `StartFunc`: Tuple `{Module, Func, Args}` to call for initialization
- `Restart`:
  - `permanent`: always restart
  - `temporary`: never restart
  - `transient`: restart only after crash
- `Shutdown`: How long to wait until children have properly shut down
- `Type`: `worker` or `supervisor`
- `Modules`: `[ModuleName]` or `dynamic` (used for managing releases)

Children can be dynamically added and removed:

- `start_child(SupRef, ChildSpec)`
- `delete_child(SupRef, Id)`

Supervisor example

```
-module(example_sup).
-behaviour(supervisor).
-export([start_link/0, init/1]).
-export([stop/0]).

start_link() ->
    supervisor:start_link(?MODULE, []).

stop(Pid) ->
    exit(Pid, shutdown).

init(_) ->
    ChildSpecList = [child(service1), child(service2)],
    {ok,{{one_for_one, 2, 3600}, ChildSpecList}}.

child(Module) ->
    {Module, {Module, start_link, []},
        permanent, 2000, worker, [Module]}.
```

Erlang OTP

- Generic servers (`gen_server`)
- Generic Supervisors (`supervisor`)

More features:

- Generic state machine behavior `gen_statem` (different states accept different messages)
- Generic event handling behavior `gen_event` (multiple event handlers receive notification for one event)
- Applications, releases and release handling

Further reading

Schwarz, Reinhard, and Friedemann Mattern. 1994. "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail." *Distributed Computing* 7 (3):149–74. https://doi.org/10.1007/BF02277859.