TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Programming Distributed Systems

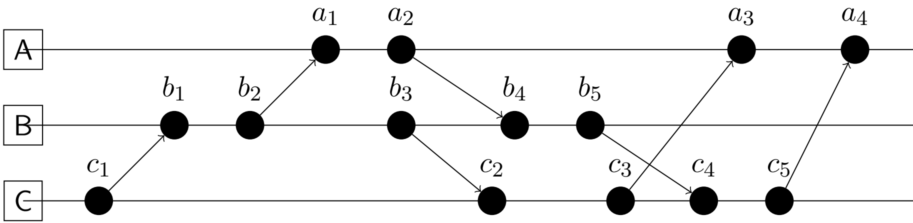
Introduction to Erlang

Peter Zeller, Annette Bieniusa

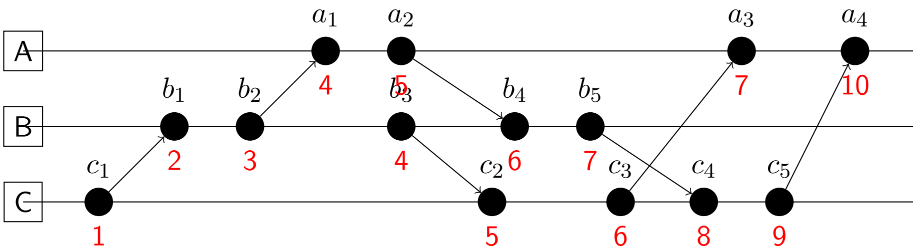
AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2018

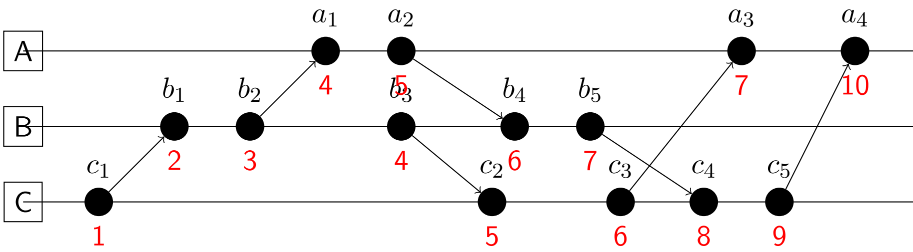
Logical Clocks



Logical Clocks

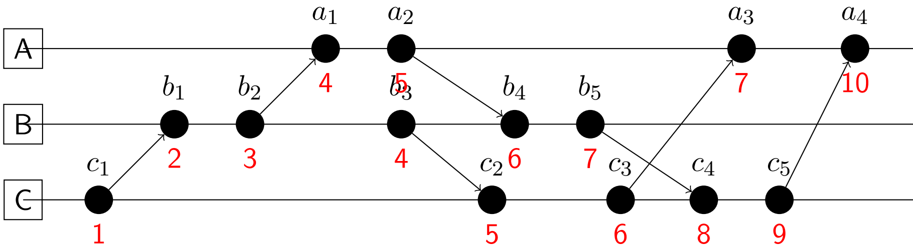


Logical Clocks



Give an example execution that shows: $t(e_1) < t(e_2)$ does not imply that $e_1 \rightarrow e_2$.

Logical Clocks



Give an example execution that shows: $t(e_1) < t(e_2)$ does not imply that $e_1 \rightarrow e_2$.

b5 and c3

3c)

Assume $e_1 \rightarrow e_2$ and show $t(e_1) < t(e_2)$. Proof by induction over the inductive definition of the happens before-relation:

- Ca If e_1 and e_2 are events in the same process and e_1 comes before e_2:
  Since l_p is strongly monotonically increasing for each event, we have $t(e_1) < t(e_2)$.

- Ca If e_1 is the sending of a message by one process and e_2 is the receipt of the same message by another process:
  Then the message must include $t(e_1)$. As $t(e_2) = max(t(e_1), l_p) + 1$, we have $t(e_2) > t(e_1)$.

- Ca Transitivity: There is an event e', such that $e_1 \rightarrow e'$ and $e' \rightarrow e_2$.
  By induction hypothesis, we have $t(e_1) < t(e')$ and $t(e') < t(e_2)$ and because $<$ is transitive on natural numbers, we get $t(e_1) < t(e_2)$.

1a)

Write a function `maximum/2`, which takes two numbers and returns the maximum of the two. Do not use the built-in `max` function. Hint: You can use the **`if`**-expression, **`case`**-expression or guards.

```
maximum(X, Y) when X > Y -> X;
maximum(_, Y) -> Y.
```

1a)

Write a function `maximum/2`, which takes two numbers and returns the maximum of the two. Do not use the built-in `max` function. Hint: You can use the **`if`**-expression, **`case`**-expression or guards.

```
maximum(X, Y) when X > Y -> X;
maximum(_, Y) -> Y.

maximum2(X, Y) ->
    case X > Y of
        true -> X;
        false -> Y
    end.

maximum3(X,Y) ->
    if
        X > Y -> X;
        true -> Y
    end.
```

1b)

Write a function `list_max/1`, which takes a nonempty list of numbers and computes the maximal element in the list. Do not use the built-in function `lists:max`. Use recursion to implement the function.

```
list_max([X]) -> X;
list_max([X|Xs]) -> maximum(X, list_max(Xs)).
```

1b)

Write a function `list_max/1`, which takes a nonempty list of numbers and computes the maximal element in the list. Do not use the built-in function `lists:max`. Use recursion to implement the function.

```
list_max([X]) -> X;
list_max([X|Xs]) -> maximum(X, list_max(Xs)).

% tail-recursive-variant
list_max_tailrec([X|Xs]) -> list_max_h(Xs, X).

list_max_h([], Max) -> Max;
list_max_h([X|Xs], Max) -> list_max_h(Xs, maximum(X, Max)).
```

1c)

Write a function `sorted/1`, which takes a list of numbers and checks, whether it is sorted in ascending order.

```
sorted([]) -> true;
sorted([_]) -> true;
sorted([X,Y|Rest]) -> X =< Y andalso sorted([Y|Rest]).
```

1d)

Write a function `swap/1`, which takes a pair and returns a pair where the two components are swapped.

```
swap({X, Y}) -> {Y,X}.
```

1e)

Write a function `find/2`, which takes a key and a list of key-value pairs. The function should return `{ok, X}`, if `X` is the value of the first pair in the list that has the given key. If no entry with the given key exists, the function should return `error`.

```
find(_, []) -> error;
find(Key, [{Key,Val}|_]) -> {ok, Val};
find(Key, [_| Rest]) -> find(Key, Rest).
```

1f)

Write a function `find_all/2`, which takes a list of keys and a list of key-value pairs. The function should use the `find`-function above to lookup every key from the first in the second list. The result should be a list of all key-value pairs that were found with the same order as they appeared in the given list of keys.

```
find_all([], _) -> [];
find_all([Key|Keys], Dict) ->
    case find(Key, Dict) of
        {ok, Val} -> [{Key, Val}|find_all(Keys, Dict)];
        error -> find_all(Keys, Dict)
    end.
```

g)

Use `lists:filter/2` to write a function `positive/1`, which takes a list of numbers `L` and returns a list of all numbers in `L`, which are greater or equal to 0.

```
positive(L) ->
    lists:filter(fun(X) -> X >= 0 end, L).
```

h)

Use `lists:all/2` to write a function `all_positive/1`, which takes a list of numbers and checks whether all numbers in the list are greater or equal to 0.

```
all_positive(L) ->
    lists:all(fun(X) -> X >= 0 end, L).
```

i)

Use `lists:map/2` to write a function `values/1`, which takes a list of key-value pairs and returns a list of only the values.

```
values(L) ->
    lists:map(fun({_, X}) -> X end, L).
```

j)

Use `lists:foldl/3` to write a function `list_min`, which computes the minimal element of a nonempty list.

```
minimum(X, Y) when X < Y -> X;
minimum(_, Y) -> Y.

list_min([X|Xs]) ->
lists:foldl(fun minimum/2, X, Xs).
```

List comprehensions

```
> L1 = [1,14,7,6].
> L2 = [a, {ok, 3}, {ok, 4}, b].
[2*X || X <- L1].
% [2,28,14,12]

[2*X || {ok, X} <- L2].
% [6,8]

[{ok, 2*X} || X <- L1, X < 10].
% [{ok,2},{ok,14},{ok,12}]

[{X,Y} || X <- L1, Y <- [a,b]].
% [{1,a},{1,b},{14,a},{14,b},{7,a},{7,b},{6,a},{6,b}]
```

In General: `[Expression || Qualifier1, Qualifier2, ...]`

- Generator Qualifier: `Pattern <- ListExpr`
- Filter Qualifier: Boolean expression

Concurrent programming

Processes

Creating a new process:

```
spawn_link(Fun)
spawn_link(Module, Function, Args)
```

Example:

```
F = fun() ->
    timer:sleep(5000), % sleep 5 seconds
    io:format("Hello from process ~p!~n", [self()])
  end.
Pid = spawn_link(F).
```

Messages

Sending messages:

```
Receiver ! Message
```

Receiving messages:

```
receive
    Pattern1 -> Expr1;
    Pattern2 -> Expr2;
    ...
    PatternN -> ExprN
end
```

- takes first message from mailbox that matches one of the patterns
- blocks until matching message available
- FIFO order (messages from same origin are ordered)

Message example 1

```
Pid = spawn_link(fun() ->
    receive
        a -> io:format("Received a~n")
    end,
    receive
        a -> io:format("Received a~n");
        b -> io:format("Received b~n")
    end
  end).
Pid ! b.
Pid ! a.
```

Message example 2

```
Pid = spawn_link(fun() ->
    timer:sleep(10000),
    receive
        a -> io:format("Received a~n");
        b -> io:format("Received b~n")
    end
  end).
Pid ! b.
Pid ! a.
```

Timeouts

Receive with timeouts:

```
receive
    Msg -> ...
after 5000 -> % timeout after 5000ms
    ...
end
```

Use timeout 0 to check if message is already in mailbox without blocking.

Example: Echo server 1

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

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

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

Example: Echo server 1

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

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

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

Problem: What if receiver also gets other messages?

Example: Echo server 2

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

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

Client:

```
EchoServer ! {self(), "Hello World"},
receive
    {EchoServer, Answer} -> ...
end
```

Example: Echo server 3

Solution b): Sending unique reference.

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

Client:

```
Ref = make_ref(),
EchoServer ! {self(), Ref, "Hello World"},
receive
    {Ref, Answer} -> ...
end
```

Example: Counting server

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

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.
```

```
-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.
```

Maps

```
M = #{a => 1, b => 42, c => 3}.
```

Reading entries:

```
#{a := X, c := Y} = M.      % binds X to 1 and Y to 3
maps:get(b, M).             % returns 42
maps:get(x, M).             % exception
maps:get(x, M, 0).          % returns 0
maps:find(b, M).            % returns {ok, 42}
maps:find(x, M).            % returns error
```

Updating and adding entries:

```
M#{a => 2}.    % #{a => 2,b => 42,c => 3}
M#{a := 2}.    % #{a => 2,b => 42,c => 3}
M#{x := 2}.    % exception
M#{x => 2}.    % #{a => 1,b => 42,c => 3,x => 2}
```

More functions at http://erlang.org/doc/man/maps.html