# Practical Concurrent and Parallel Programming XII

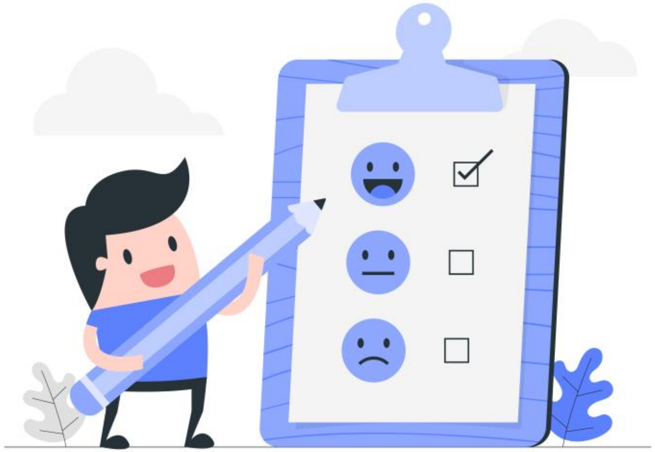
# Message Passing I

Raúl Pardo

# Course Evaluation Survey



Please participate in the course evaluation



# Agenda

- Problems in shared memory concurrency (revisited)
- Actors
- Erlang
- Example systems
  - Turnstile (counter)
  - Broadcaster
  - Bounded Buffer

# Problems in shared memory concurrency



*"Writing thread-safe code is, at its core, about* ***<u>managing access to shared mutable data</u>****"*

Goetz

# Problems in shared memory concurrency



*“Writing thread-safe code is, at its core, about* ***managing access to shared mutable data****”*

Goetz

What problems have we seen in concurrent access to shared memory?

# Problems in shared memory concurrency



*"Writing thread-safe code is, at its core, about* ***managing access to shared mutable data****"*

Goetz

- Race conditions
- Data races
- Visibility
- Reasoning is tricky
  - Specially lock-free computation 😁

# Problems in shared memory concurrency



*"Writing thread-safe code is, at its core, about* ***managing access to shared mutable data****"*

Goetz

What solutions have we seen to the problems in concurrent access to shared memory?

# Problems in shared memory concurrency

*"Writing thread-safe code is, at its core, about **managing access to shared mutable data**"*

Goetz

- Happens-before reasoning
- Linearizability reasoning
- For race conditions and data races:
  - Ensuring mutual exclusion
    - Trying to ensure progress
  - Immutability
  - Compare and Swap (CAS) algorithms
    - Trying to ensure progress
- For visibility:
  - Volatile and final variables, idioms for safe publication, etc

# Problems in shared memory concurrency

*"Writing thread-safe code is, at its core, about* ***__managing access to shared mutable data__****"*

Goetz

- Happens-before reasoning
- Linearizability reasoning
- For race conditions and data races:
  - Ensuring mutual exclusion
    - Trying to ensure progress
  - Immutability
  - Compare and Swap (CAS) algorithms
    - Trying to ensure progress
- For visibility:
  - Volatile and final variables, idioms for safe publication, etc.

**Why don't we simply avoid sharing state?**
**This is the idea behind message passing!**

# Message passing concurrency



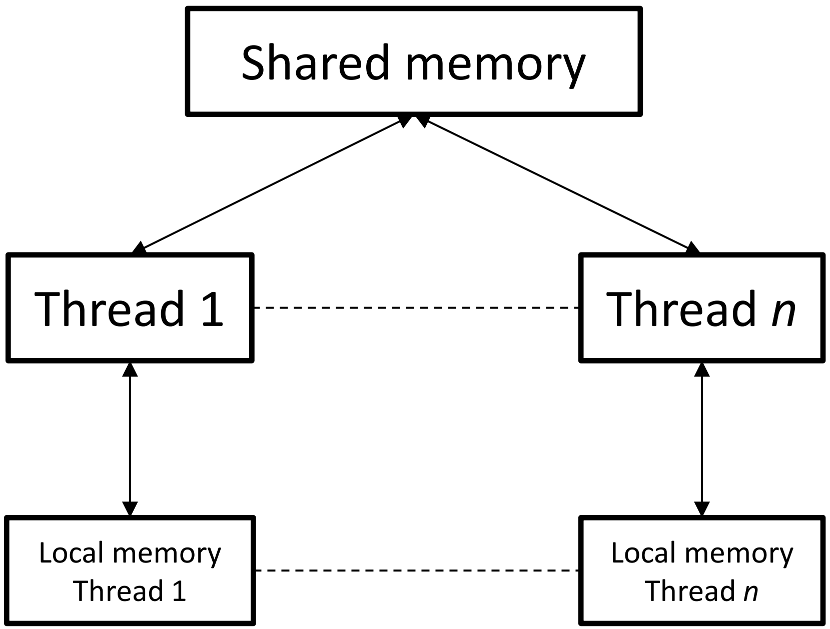
One of the designers of Erlang

- Threads do not share state
- If threads need to share data, then data must be communicated by sending messages
- Threads work only on their own local memory

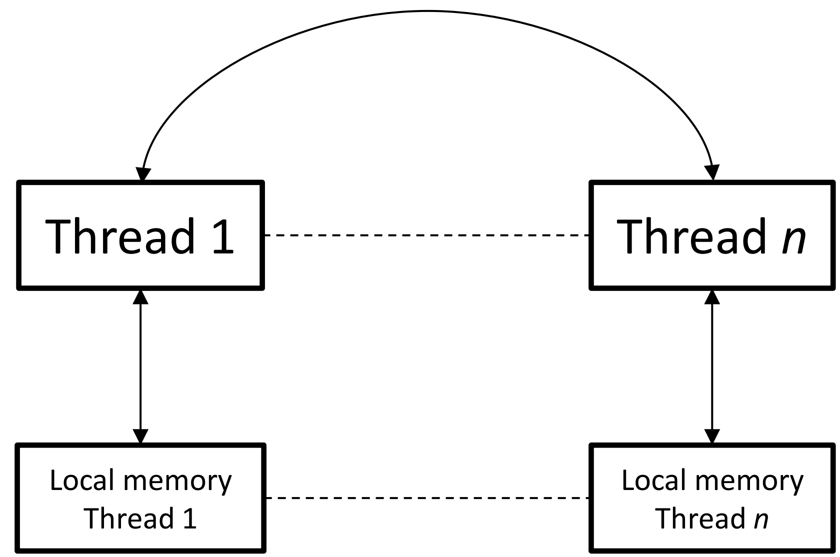
# Shared memory vs Message Passing



- Shared Memory
  - Synchronisation by writing in shared memory




- Message Passing
  - Synchronisation by sending messages

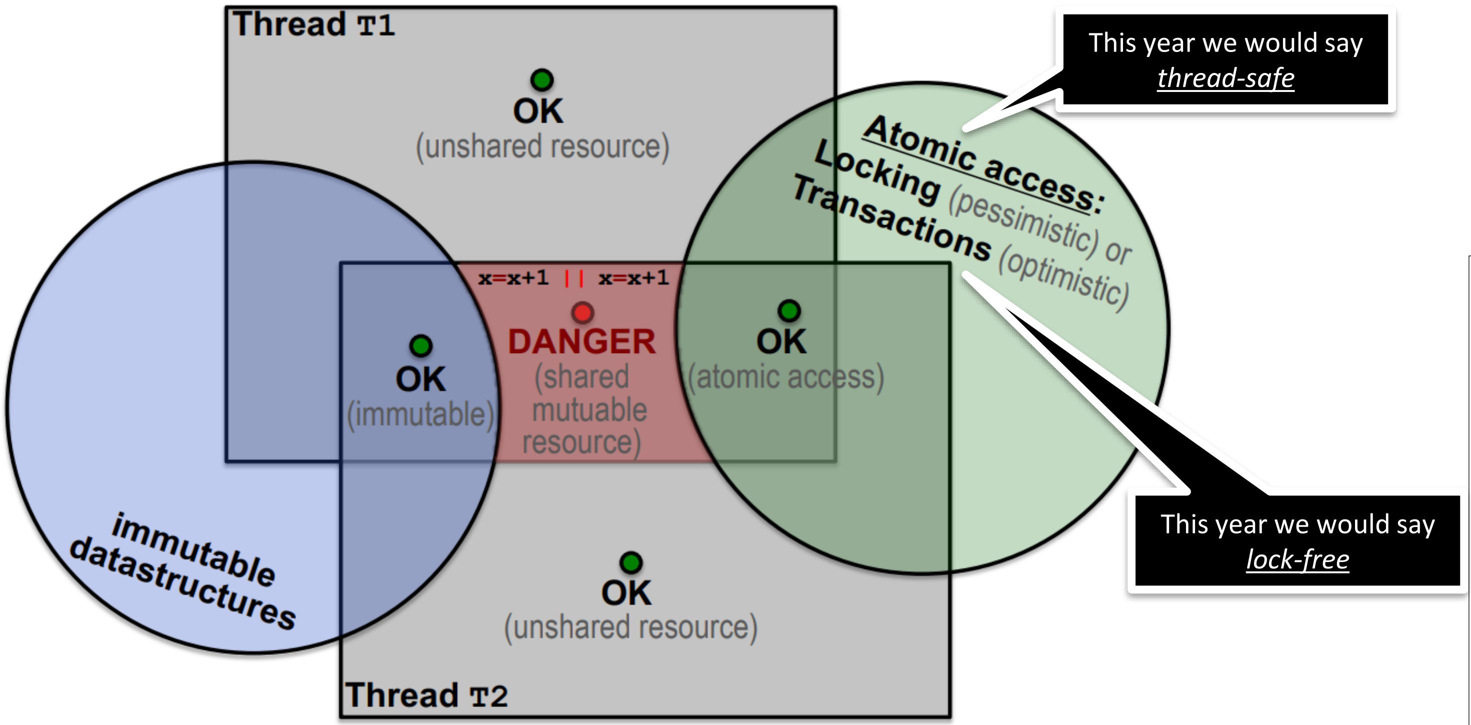


# Message passing concurrency



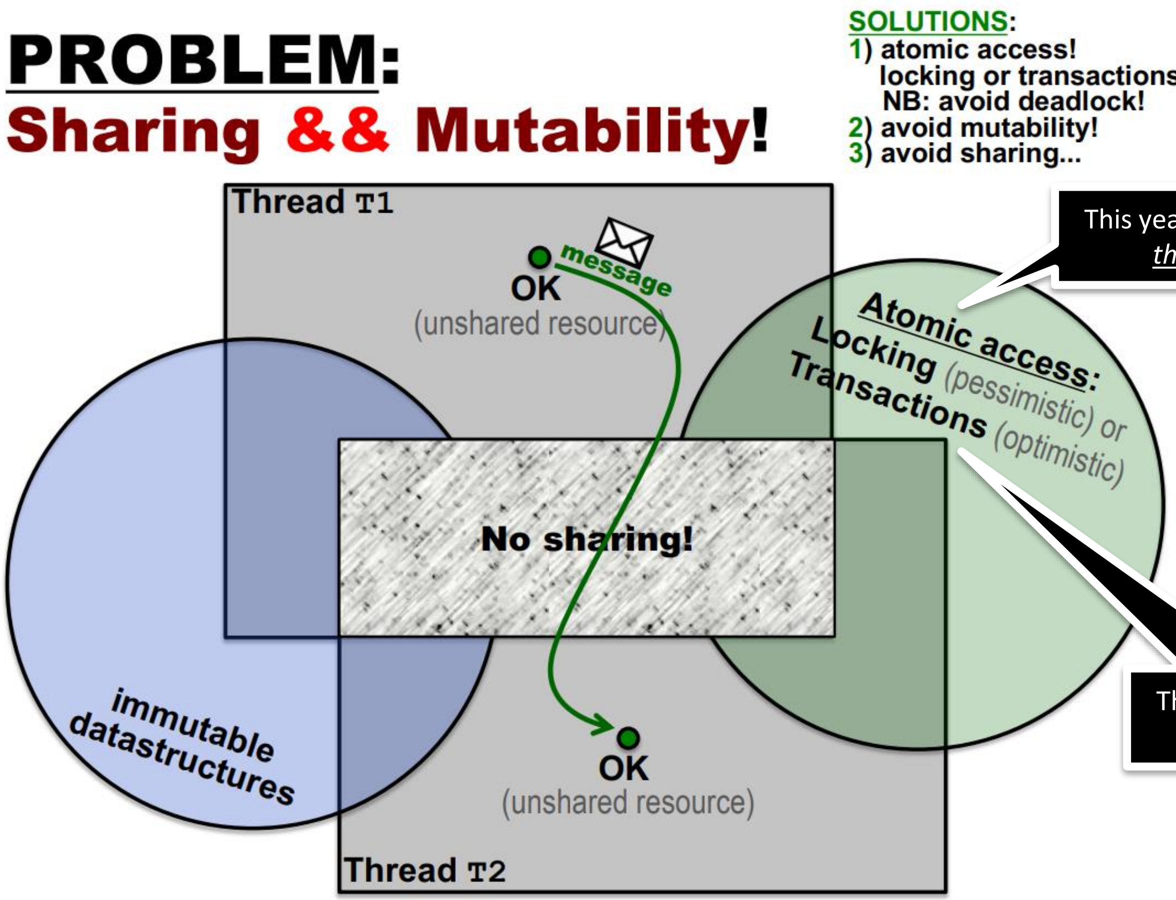
# PROBLEM:
## Sharing && Mutability!

**SOLUTIONS:**
1) atomic access!
   locking or transactions
   NB: avoid deadlock!
2) avoid mutability!
3) avoid sharing...




Slide from lecture 13 in PCPP 2019

# Message passing concurrency




This year we would say *thread-safe*

This year we would say *lock-free*

Slide from lecture 13 in PCPP 2019

# Message passing concurrency



- How should we implement message passing concurrency?

- A possible solution is use standard communication systems
  - Sockets (*last week*)
  - Remote Procedure Calls (RPC)
  - Java Remote Method Invocation (RMI)
  - Message passing interfaces (MPI)

combined with concurrency as we have seen so far

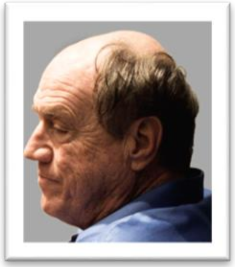
# Message passing concurrency



- How should we implement message passing concurrency?

- Another option is to ***use a concurrency model with message passing built-in***
  - That is, the ***actor model***!

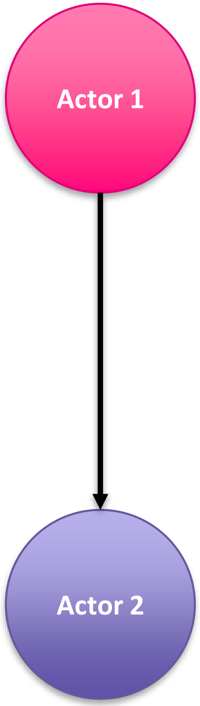
- The actors model was first introduced by [Hewitt'73] and later formalized by [Agha'85] (part of the readings)
  - [Hewitt'73] - Carl Hewitt, Peter Bishop & Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence. 1973.
  - [Agha'85] – Gul A. Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press. 1985.



# Actor model

# What is an Actor? (Bird's eye)

- An actor can be seen as a sequential unit of computation
  - Although, formally, the model allows for parallelism within the actor, one can safely assume that there are not concurrency issues within the actor.
  - You can think of an actor as a thread
- Actors can send messages to other actors

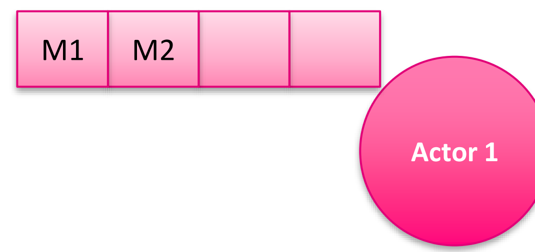


# Actor – Specification

- An actor is an abstraction of a thread (intuitively)

- An actors can only execute any of these 4 actions
  1. *Receive messages from other actors*
  2. *Send asynchronous messages* to other actors
  3. *Create new actors*
  4. *Change its behaviour* (local state and/or message handlers)

- Actors *do not share memory*
  - They only have access to:
    – Their *local state* (local memory)
    – Their *mailbox* (multiset with received messages)
    – By default: i) the mailbox is unbounded, but with a fixed size at each point of the execution;
      ii) messages are ordered in a FIFO style



# Actors and Messages

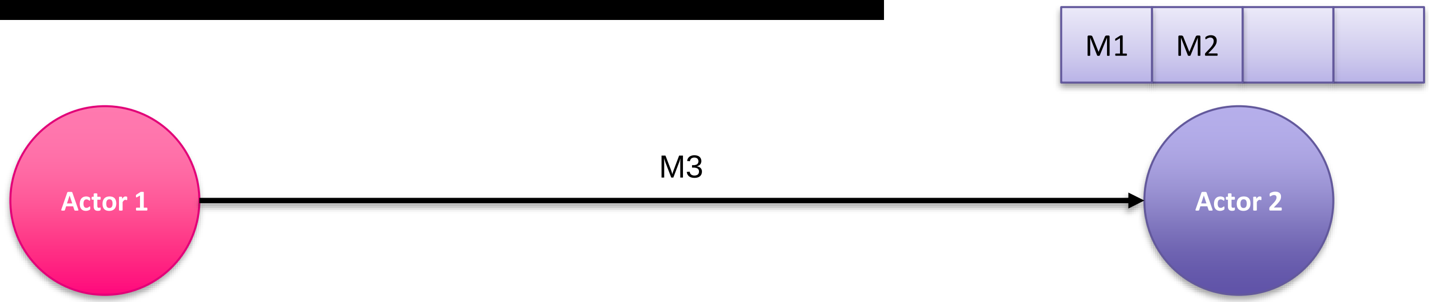


- Every actor in the system has a ***unique identifier***
  - A.k.a. ***mail address*** or ***actor reference***

- Actors can
  - Send (finitely many) messages
  - Receive (finitely many) message
    – Received messages are placed in the actor's mailbox (asynchronous communication, see next slide)

- Messages include
  - Content of the message (arbitrary payload)

# Asynchronous communication




- Asynchronous *send:*
  - The sender places the message in the mailbox of the receiver
  - It is *non-blocking*

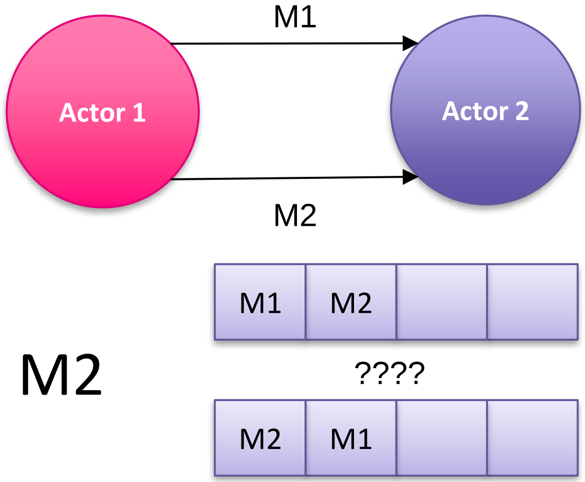
- Asynchronous *receive:*
  - The receiver takes a message from the mailbox
  - The receiver *blocks* if the mailbox is empty

# No requirements on message arrival order



- No assumptions should be made about the order of arrival of messages

- For instance, consider this sequence of operations
    1. Actor1 sends message M1 to Actor2
    2. Actor1 sends message M2 to Actor2




- It is *not* guaranteed that M1 arrives before M2

# Erlang (Actors implementation)

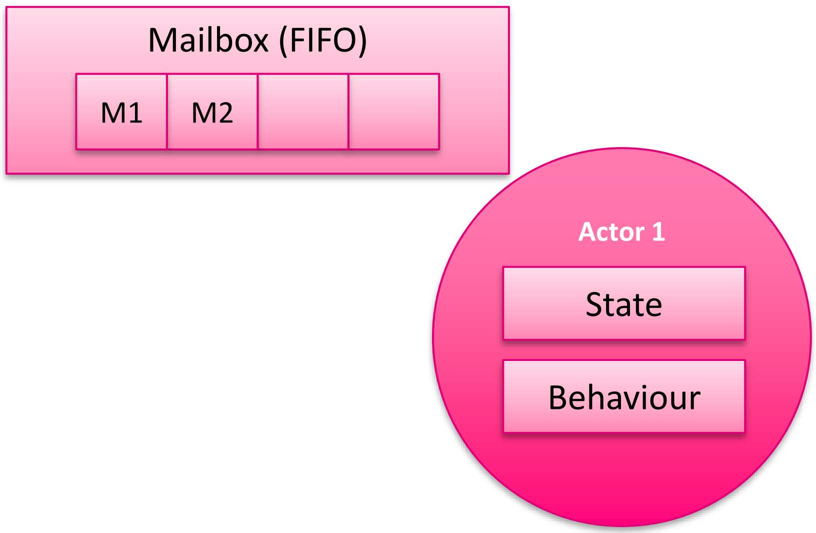
# Erlang

- Developed by Joe Armstrong, Robert Virding, and Mike Williams in 1986
  - Open-sourced in 1998; despites Ericsson's attempts to prevent it
- Erlang = **<u>Er</u>**icsson **<u>Lang</u>**uage
  - (Presumably) named after the Danish mathematician Agner Krarup Erlang (1878 –1929) for his pioneering and influential work in the field of telecommunications
- Language developed for telephony applications
  - Erlang/OTP is supported and maintained by the Open Telecom Platform (OTP) product unit at Ericsson.
- Famously used at WhatsApp (among many other companies)
  - In 2014, there were only 32 engineers at WhatsApp developing/maintaining software for 450 million users
- Multiple companies use Erlang in production
  - [https://erlang-companies.org/](https://erlang-companies.org/)

...

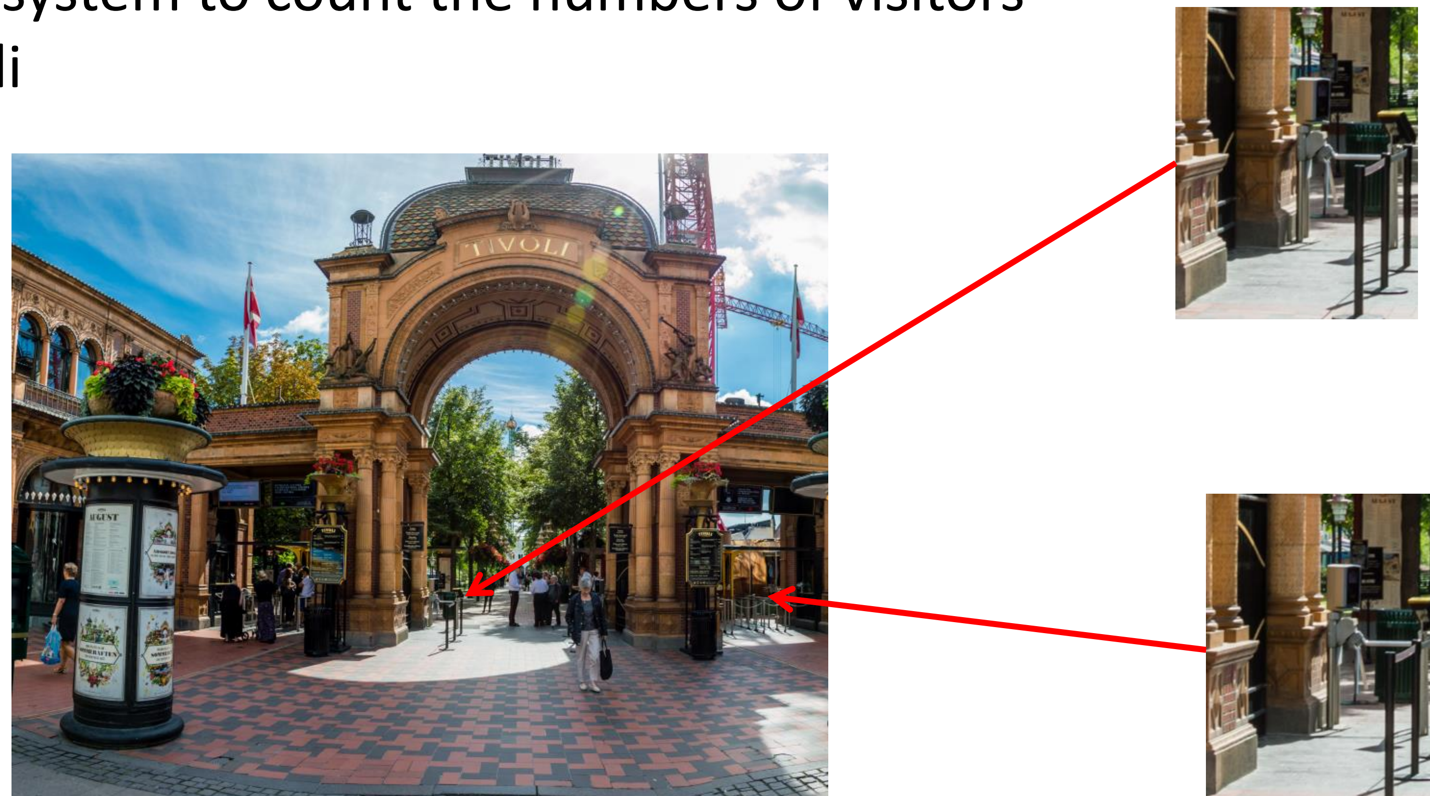
# Erlang actors (processes)





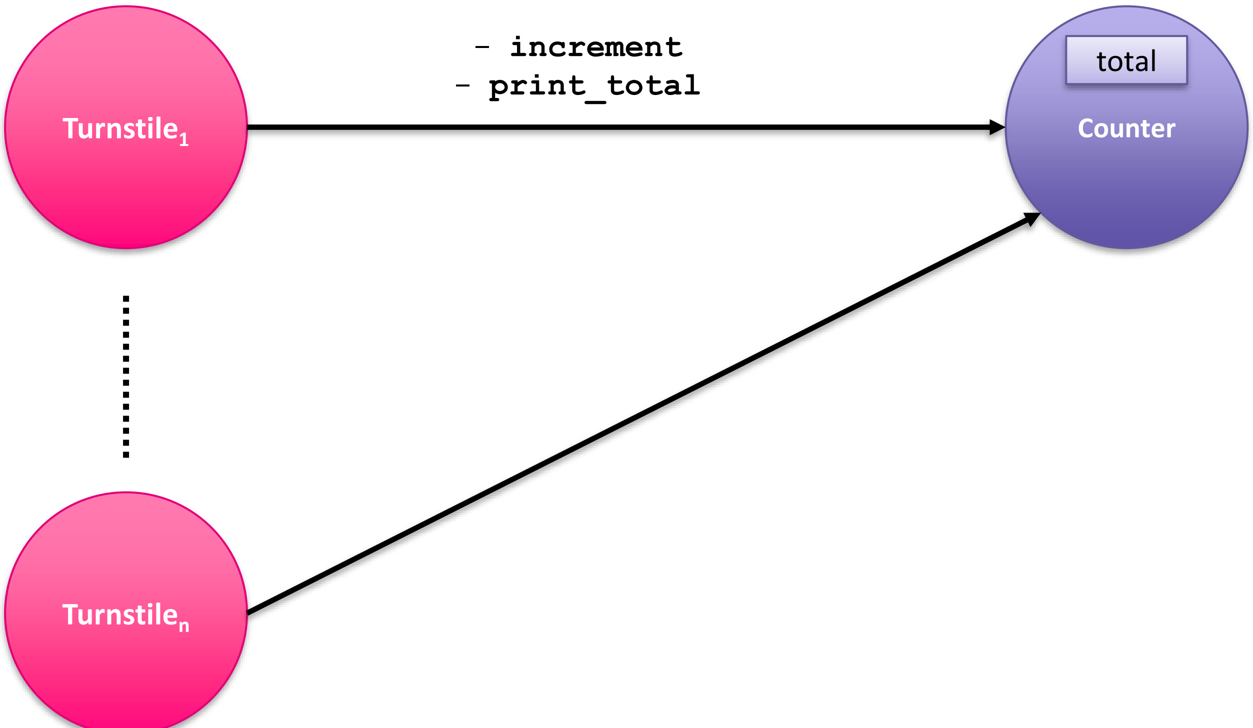
# Tivoli Turnstiles with Actors!

- Actors system to count the numbers of visitors in Tivoli



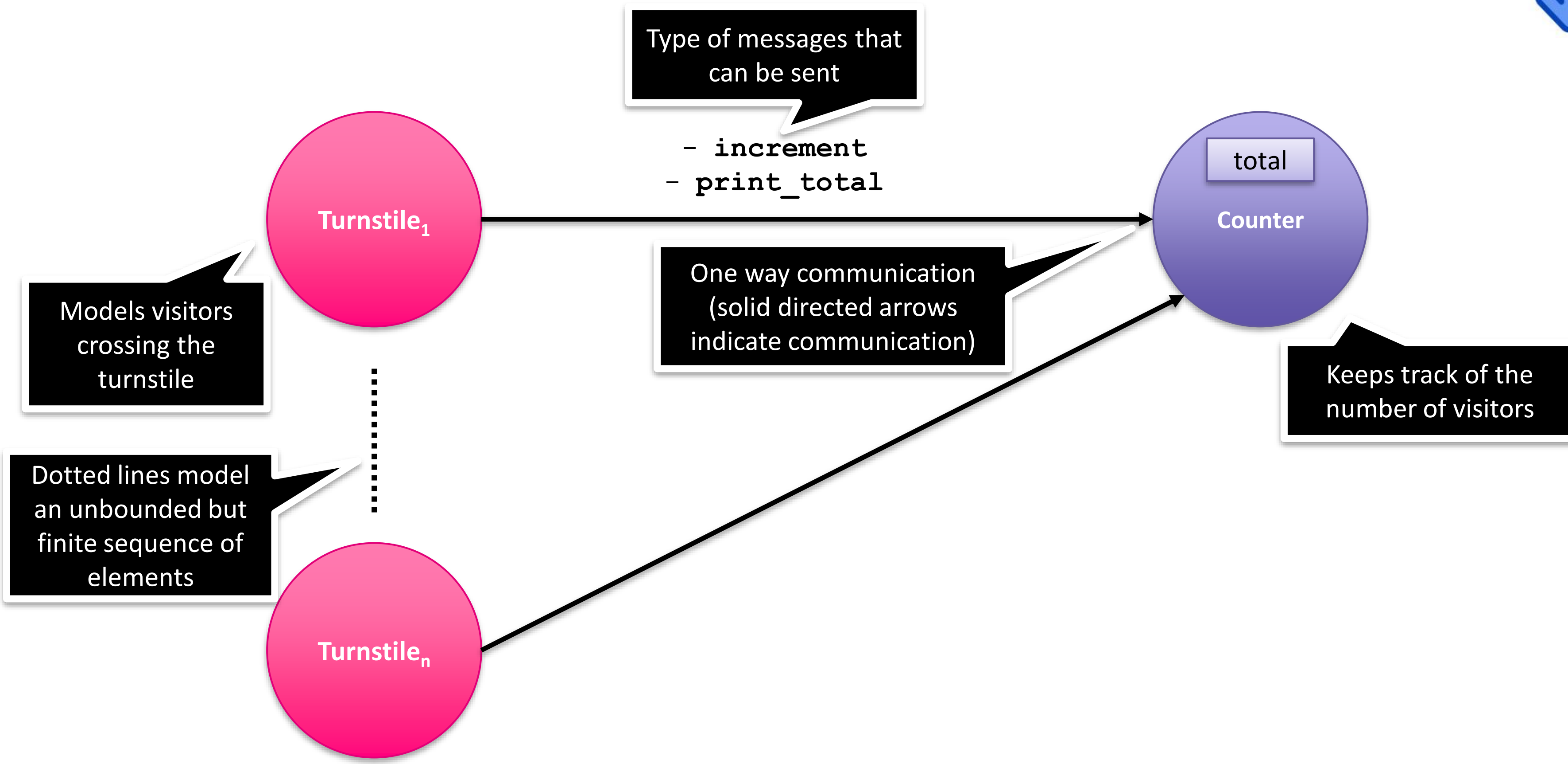
# Turnstile with Actors - Design





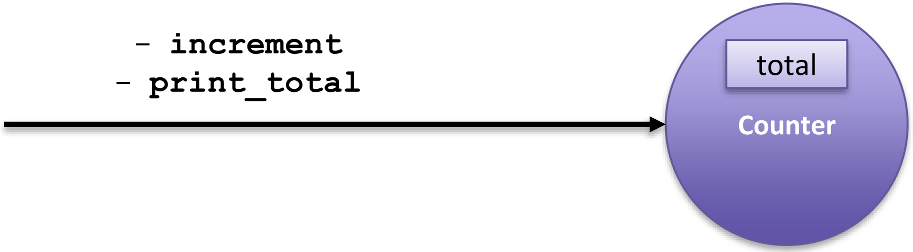
# Turnstile with Actors - Design





# Turnstile with Actors - Implementation

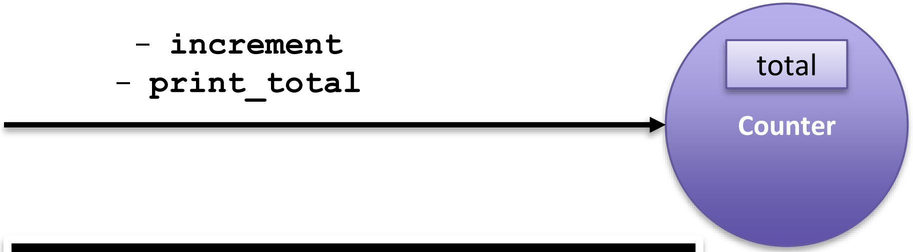
```
-module(counter).
-export([start/0, init/1]).


% State of the actor
-record(counter_state, {total}).


% Function to create counter actors
start() ->
    spawn(?MODULE, init, [0]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```



# Turnstile with Actors - Implementation



```erlang
-module(counter).
-export([start/0, init/1]).


% State of the actor
-record(counter_state, {total}).


% Function to create counter actors
start() ->
    spawn(?MODULE, init, [0]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```



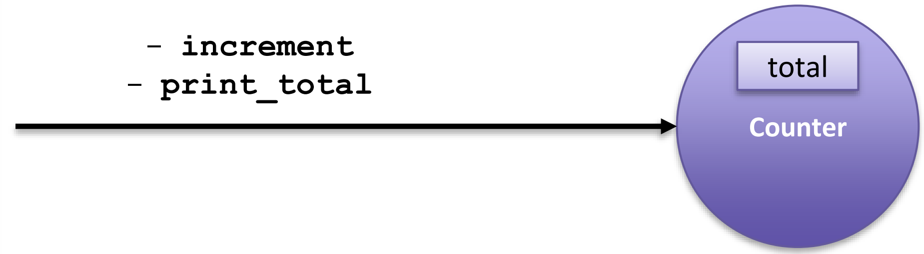

**Actors vs Threads**

Each actor is defined in its own module. This is similar to a Thread class for a Java thread.

Remember that the module name must match the file name, in this case `counter.erl`

# Turnstile with Actors - Implementation

```
-module(counter).
-export([start/0, init/1]).


% State of the actor
-record(counter_state, {total}).


% Function to create counter actors
start() ->
    spawn(?MODULE, init, [0]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```



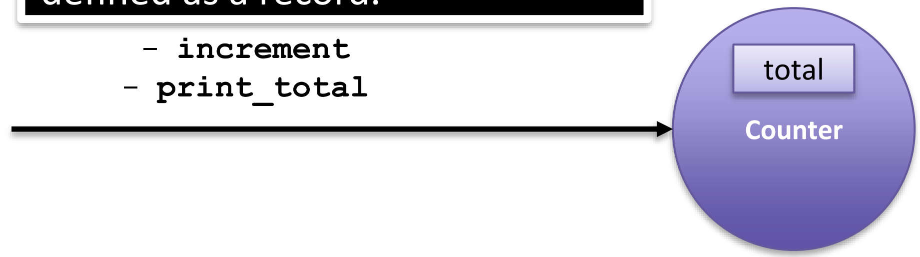

We must export the functions necessary to create and interact with the actor.

# Turnstile with Actors - Implementation

```
-module(counter).
-export([start/0, init/1]).


% State of the actor
-record(counter_state, {total}).


% Function to create counter actors
start() ->
    spawn(?MODULE, init, [0]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```

**Actors vs Threads**

- Like threads actors' have a local state. The local state is commonly defined as a record.



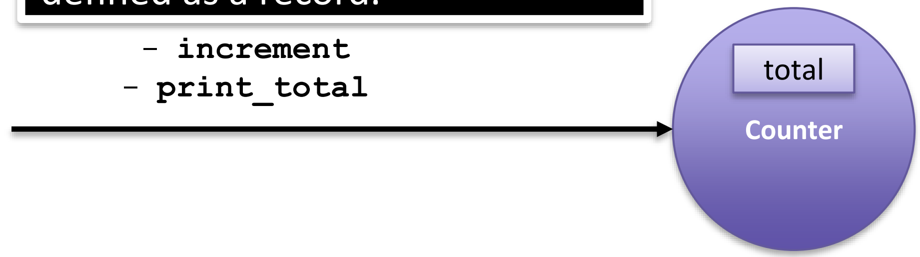
# Turnstile with Actors - Implementation



```
-module(counter).
-export([start/0, init/1]).


% State of the actor
-record(counter_state, {total}).


% Function to create counter actors
start() ->
    spawn(?MODULE, init, [0]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```

<u>Actors vs Threads</u>

- Like threads actors' have a local state. The local state is commonly defined as a record.

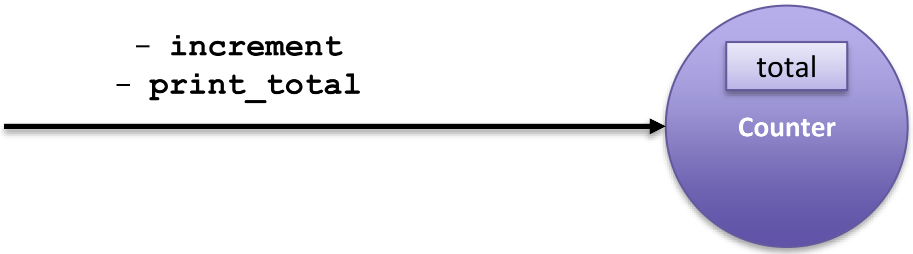



Are there visibility issues in the actor state?

# Turnstile with Actors - Implementation



```
-module(counter).
-export([start/0, init/1]).



% State of the actor
-record(counter_state, {total}).



% Function to create counter actors
start() ->
    spawn(?MODULE, init, [0]).



% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).



% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```




<u>Actors vs Threads</u>

- It is common to define a function `init` to initialize the state of the actor and its behavior (specified in the `loop` function that we will explain the in the coming slides)

This is similar to the constructor of a Thread class for a Java thread.

# Turnstile with Actors - Implementation



```erlang
-module(counter).
-export([start/0, init/1]).


% State of the actor
-record(counter_state, {total}).


% Function to create counter actors
start() ->
    spawn(?MODULE, init, [0]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```


Actors vs Threads

- To start an actor, we use the built-in function (BIF) **`spawn`**. This function takes:
    - i) an atom indicating the module where the **`init`** function is
    - ii) The **`init`** function
    - iii) a list of parameter for the **`init`** function

This is similar to the function **`start()`** in a Java Thread class.

# Turnstile with Actors - Implementation



```
-module(counter).
-export([start/0, init/1]).


% State of the actor
-record(counter_state, {total}).


% Function to create counter
start() ->
    spawn(?MODULE, init, [0]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```

**?MODULE** is a predefined macro that expands to the module atom at compile time


<u>Actors vs Threads</u>

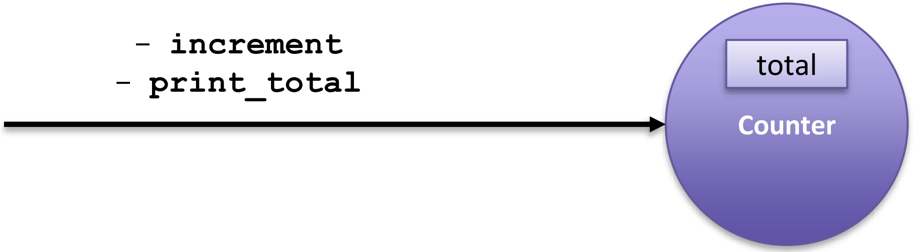
- To start an actor, we use the built-in function (BIF) **spawn**. This function takes:
    i) an atom indicating the module where the **init** function is
    ii) The **init** function
    iii) a list of parameter for the **init** function

This is similar to the function **start()** in a Java Thread class.

# Turnstile with Actors - Implementation



```
-module(counter).
-export([start/0, init/1]).


% State of the actor
-record(counter_state, {total}).


% Function to create counter actors
start() ->
    spawn(?MODULE, init, [0]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```

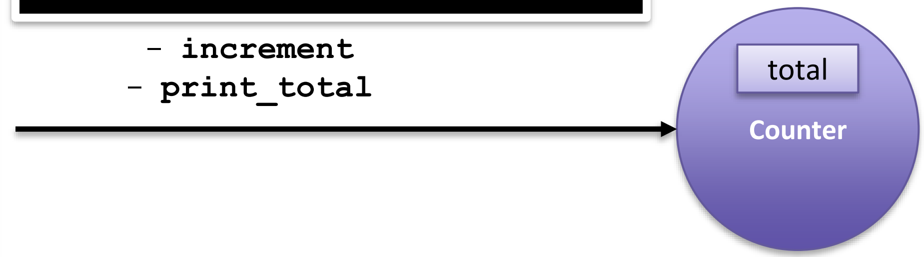



- The loop function defines the behavior of the thread upon receiving messages
- It takes as a parameter the current state of the thread; as it might be needed in processing incoming messages.

# Turnstile with Actors - Implementation

```
-module(counter).
-export([start/0, init/1]).


% State of the actor
-record(counter_state, {total}).


% Function to create counter actors
start() ->
    spawn(?MODULE, init, [0]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```

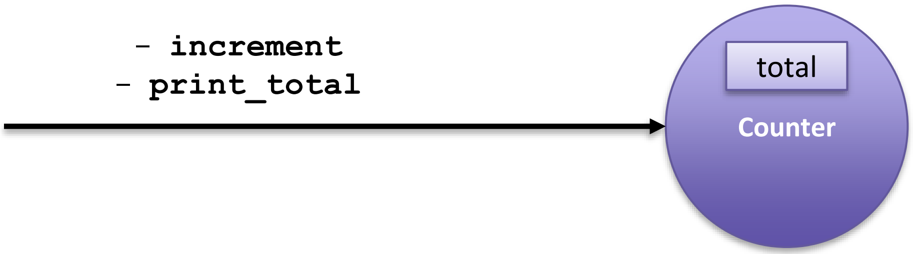
- The body of the loop function is a receive statement with one case per type of message that the actor can handle.




- **`receive`** is a *blocking* statement that waits for incoming messages to the actor's mailbox that match any of the specified cases.
- Messages that do not match the cases in receive may add up in the mailbox, which may lead to slow processing times.

# Turnstile with Actors - Implementation

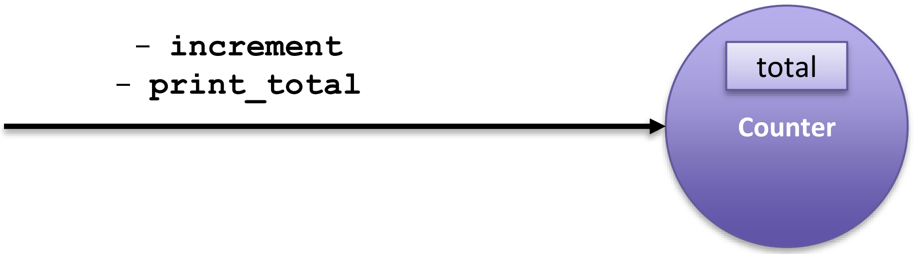
```
-module(counter).
-export([start/0, init/1]).


% State of the actor
-record(counter_state, {total}).


% Function to create counter actors
start() ->
    spawn(?MODULE, init, [0]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```




- For **increment** messages we increment the value of the counter
- For the **print_total** message we print the value

# Turnstile with Actors - Implementation

```
-module(counter).
-export([start/0, init/1]).


% State of the actor
-record(counter_state, {total}).


% Function to create counter actors
start() ->
    spawn(?MODULE, init, [0]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(InitialValue) ->
    InitialState = #counter_state{total = InitialValue},
    loop(InitialState).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        increment ->
            CurrentTotal = State#counter_state.total,
            NewState = State#counter_state{total = CurrentTotal + 1},
            io:format("A visitor arrived 🤗~n"),
            loop(NewState);
        print_total ->
            io:format("The counter value is ~p~n",
                      [State#counter_state.total]),
            loop(State)
    end.
```

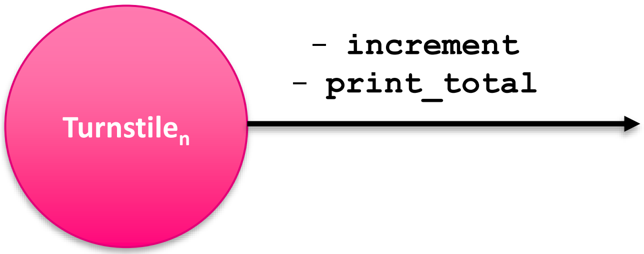



- For **increment**, we wait again for messages but using the **NewState**
- For the **print_total**, we wait for with the old state **State**

# Turnstile with Actors - Implementation



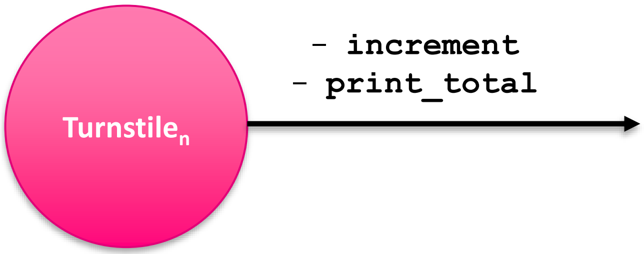
```erlang
-module(turnstile).
-export([start/1, init/1]).


% State of the actor
-record(turnstile_state, {counter_server}).


% Function to create turnstile actors
start(CounterPID) ->
    spawn(?MODULE, init, [CounterPID]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(CounterPID) ->
    State = #turnstile_state{counter_server = CounterPID},
    loop(State).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        person_crossing ->
            State#turnstile_state.counter_server ! increment,
            loop(State)
    end.
```




- The state of the turnstile actor is simply a variable to store the identifier of the actor, typically referred to in Erlang as “process identifier” (PID)

# Turnstile with Actors - Implementation

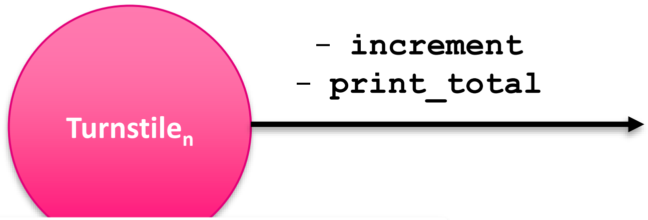
```
-module(turnstile).
-export([start/1, init/1]).


% State of the actor
-record(turnstile_state, {counter_server}).


% Function to create turnstile actors
start(CounterPID) ->
    spawn(?MODULE, init, [CounterPID]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(CounterPID) ->
    State = #turnstile_state{counter_server = CounterPID},
    loop(State).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        person_crossing ->
            State#turnstile_state.counter_server ! increment,
            loop(State)
    end.
```




- Initialization is the same as in the counter actor. The only difference is that the start and init functions now take an initialization parameter from the user, namely the CounterPID, which refers to the PID of the counter process.

# Turnstile with Actors - Implementation

```
-module(turnstile).
-export([start/1, init/1]).



% State of the actor
-record(turnstile_state, {counter_server}).



% Function to create turnstile actors
start(CounterPID) ->
    spawn(?MODULE, init, [CounterPID]).



% Function to initialize the state and the actors behavior
% upon receiving messages
init(CounterPID) ->
    State = #turnstile_state{counter_server = CounterPID},
    loop(State).



% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        person_crossing ->
            State#turnstile_state.counter_server ! increment,
            loop(State)
    end.
```




- In the loop function, we simply consider one case modelling a person crossing the turnstile.
- In this case, the turnstile actor sends an increment message to the counter server.
- This is done using the operation

**`PID ! Message`**

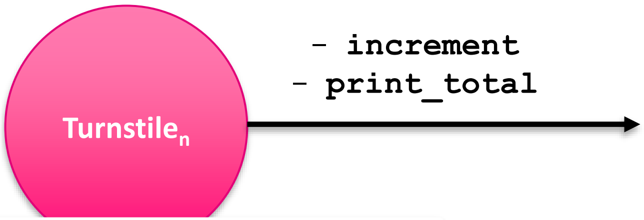
that sends an *asynchronous* message, **`Message`** to the process with identifier **`PID`**

# Turnstile with Actors - Implementation



```
-module(turnstile).
-export([start/1, init/1]).


% State of the actor
-record(turnstile_state, {counter_server}).


% Function to create turnstile actors
start(CounterPID) ->
    spawn(?MODULE, init, [CounterPID]).


% Function to initialize the state and the actors behavior
% upon receiving messages
init(CounterPID) ->
    State = #turnstile_state{counter_server = CounterPID},
    loop(State).


% Function defining the behavior upon receiving messages
loop(State) ->
    receive
        person_crossing ->
            State#turnstile_state.counter_server ! increment,
            loop(State)
    end.
```

Let’s run the code in the `turnstile` folder!




- In the loop function, we simply consider one case modelling a person crossing the turnstile.
- In this case, the turnstile actor sends an increment message to the counter server.
- This is done using the operation

**`PID ! Message`**

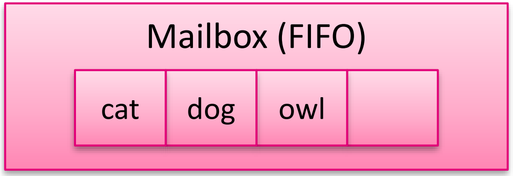
that sends an *asynchronous* message, **`Message`** to the process with identifier **`PID`**

# A technical note on receive in Erlang



- Messages when executing receive match from top to bottom
  - Similar to function cases
- Furthermore, the mailbox always preserves the order in which messages arrive, i.e., FIFO

```
loop() ->
  receive
      dog ->     …;
      cat ->     …;
      snake ->   …;
      Animal -> …
  end,
  loop().
```

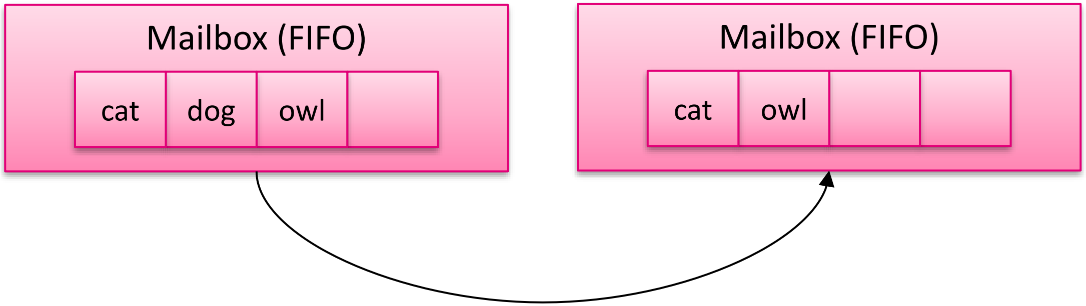


# A technical note on receive in Erlang



- Messages when executing receive match from top to bottom
  - Similar to function cases
- Furthermore, the mailbox always preserves the order in which messages arrive, i.e., FIFO

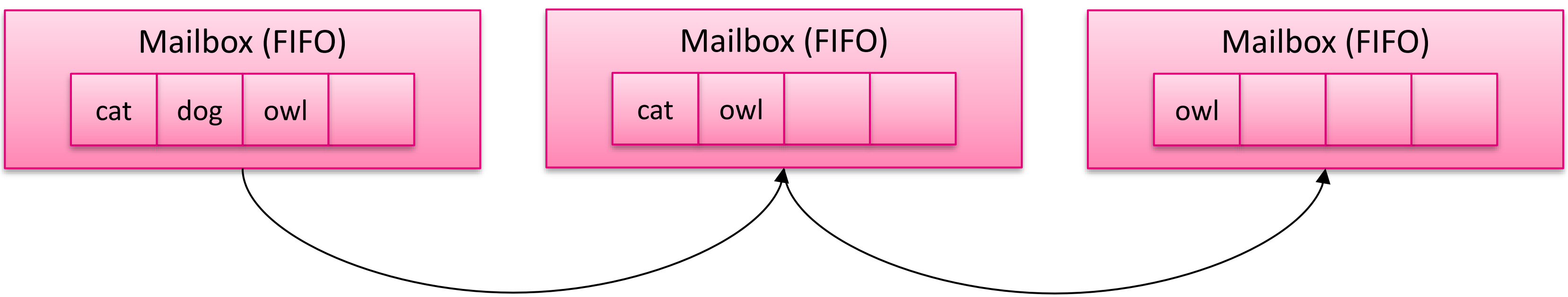
```
loop() ->
  receive
      dog ->     …;
      cat ->     …;
      snake ->   …;
      Animal ->  …
  end,
  loop().
```



# A technical note on receive in Erlang

- Messages when executing receive match from top to bottom
  - Similar to function cases
- Furthermore, the mailbox always preserves the order in which messages arrive, i.e., FIFO

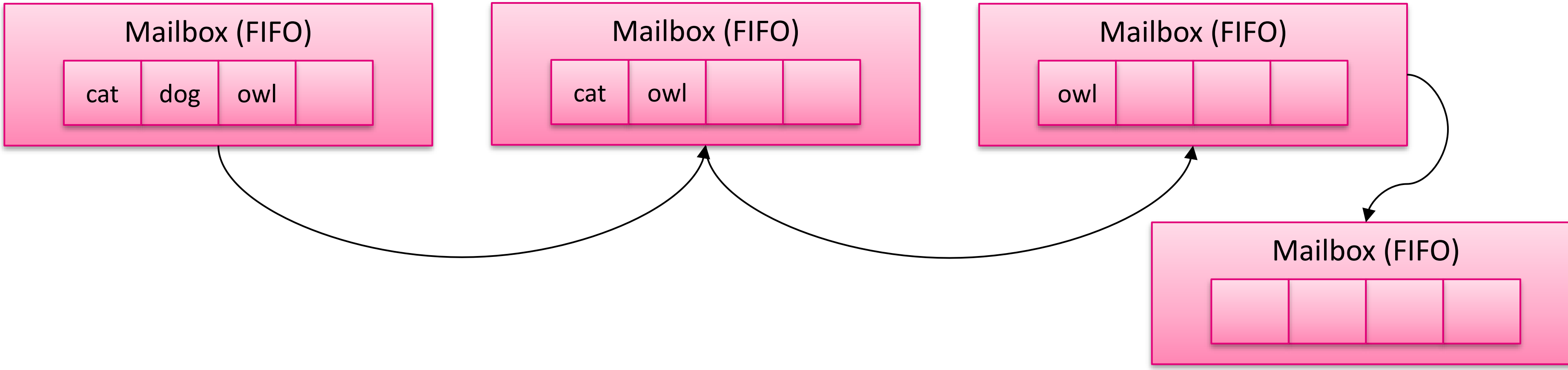
```
loop() ->
  receive
       dog ->     …;
       cat ->     …;
       snake ->   …;
       Animal ->  …
  end,
  loop().
```



# A technical note on receive in Erlang

- Messages when executing receive match from top to bottom
  - Similar to function cases
- Furthermore, the mailbox always preserves the order in which messages arrive, i.e., FIFO

```
loop() ->
  receive
      dog ->      …;
      cat ->      …;
      snake ->    …;
      Animal ->   …
  end,
  loop().
```

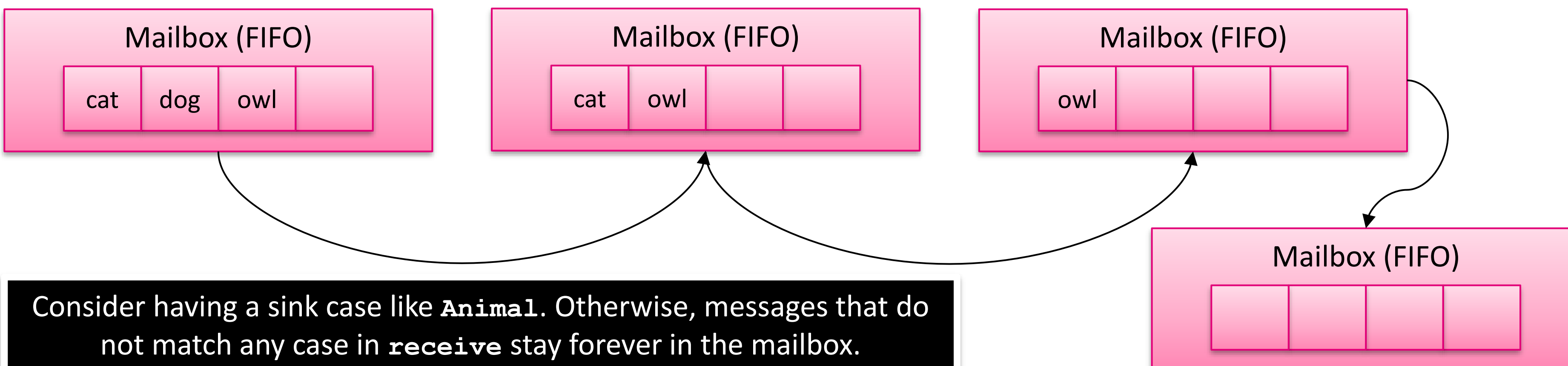


# A technical note on receive in Erlang



- Messages when executing receive match from top to bottom
  - Similar to function cases
- Furthermore, the mailbox always preserves the order in which messages arrive, i.e., FIFO

```
loop() ->
  receive
      dog ->      …;
      cat ->      …;
      snake ->    …;
      Animal ->   …
  end,
  loop().
```




Consider having a sink case like **Animal**. Otherwise, messages that do not match any case in **receive** stay forever in the mailbox.

# Erlang actor module – Summary



- In summary an Erlang actor module should have (at least) these elements:
  1. State
  2. Start function(s)
  3. Init function(s)
  4. Loop function
- Optionally it is also recommended to have (coming in the next slides):
  5. Message handling functions
  6. Actor API
- You may notice that the files in the code-lecture folder have the structure on the right to make it easier to write actor modules

```
-module(<module_atom>).
-export([…]).

% 1. State of the actor
…

% 2. Function(s) to create turnstile actors
…

% 3. Function(s) to initialize the state and the actors
% behavior upon receiving messages
…

% 4. Function defining the behavior upon receiving messages
…

% 5. Message handlers
…

% 6. API
…
```

# Erlang actors



- There is a one-to-one correspondence of the basic actor operations and concepts in Erlang

| Actors Model | Erlang |
|---|---|
| Actor | Module |
| Mailbox Address | Process identifier (PID) |
| Message | Erlang term (typically an atom or tuple) |
| State | Erlang term (typically a record) |
| Behaviour | loop() |
| Create actor | spawn |
| Send message | PID ! Message |
| Receive message | receive … end |

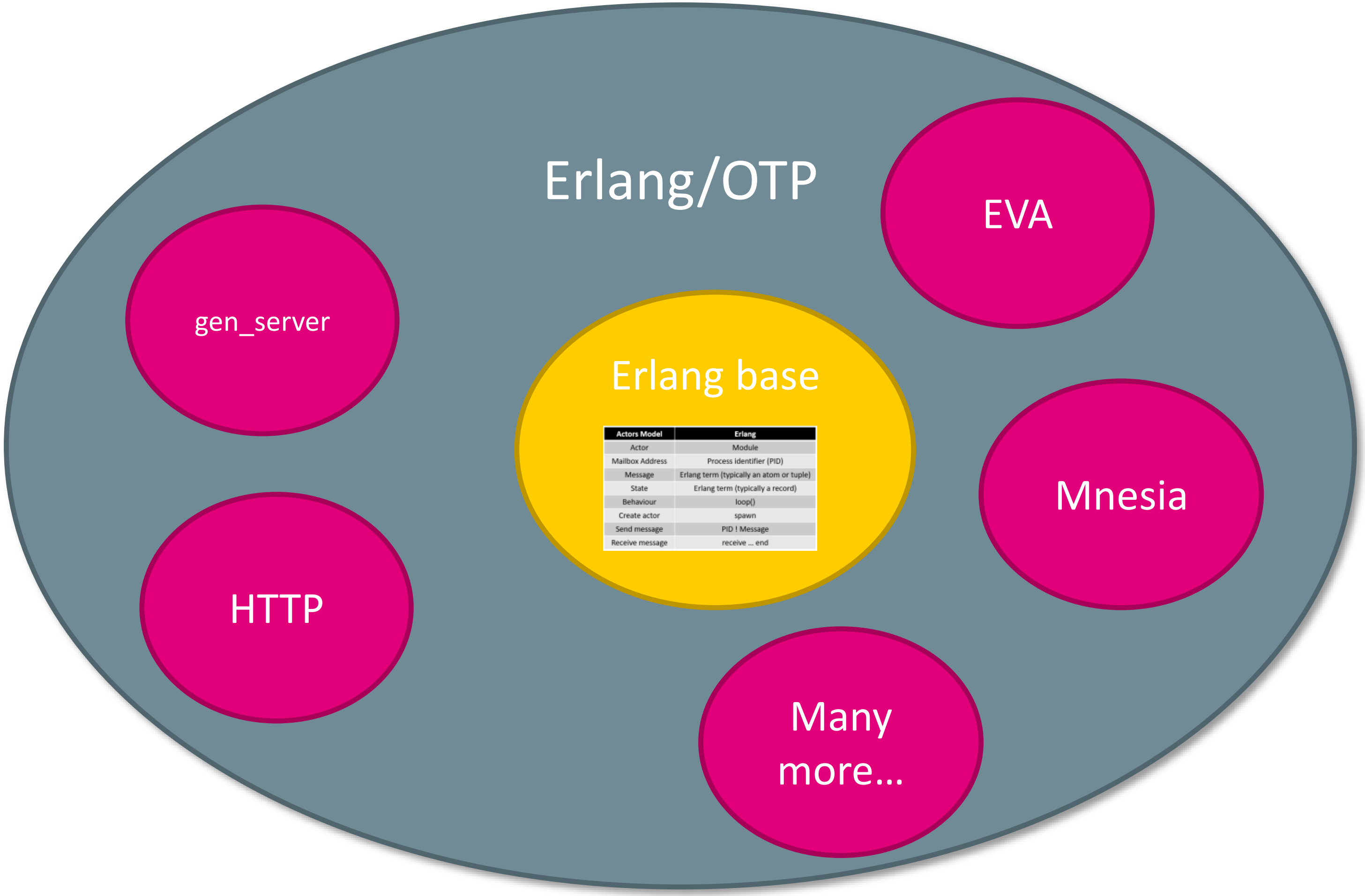
# How do we use Erlang actors?



- You may  interact with actors directly from the shell
  - For instance, `TurnstilePID ! person_crossing`
  - Let’s try with the turnstile example
- To perform a predefined set of operations, you may write a function in a system utilities module
  - The functions in this module can be executed directly in the shell
  - This module does not necessarily need to be an actor
  - Let’s look at the function `example_execution(N)` in the module `system_utilities` in folder `turnstile` for an example
- Recall that we have a guide to use Erlang for exercises
  - https://github.itu.dk/jst/PCPP2024-Public/blob/main/general-info/guide-using-erlang-for-exercises.md

# We only used a tiny bit of Erlang



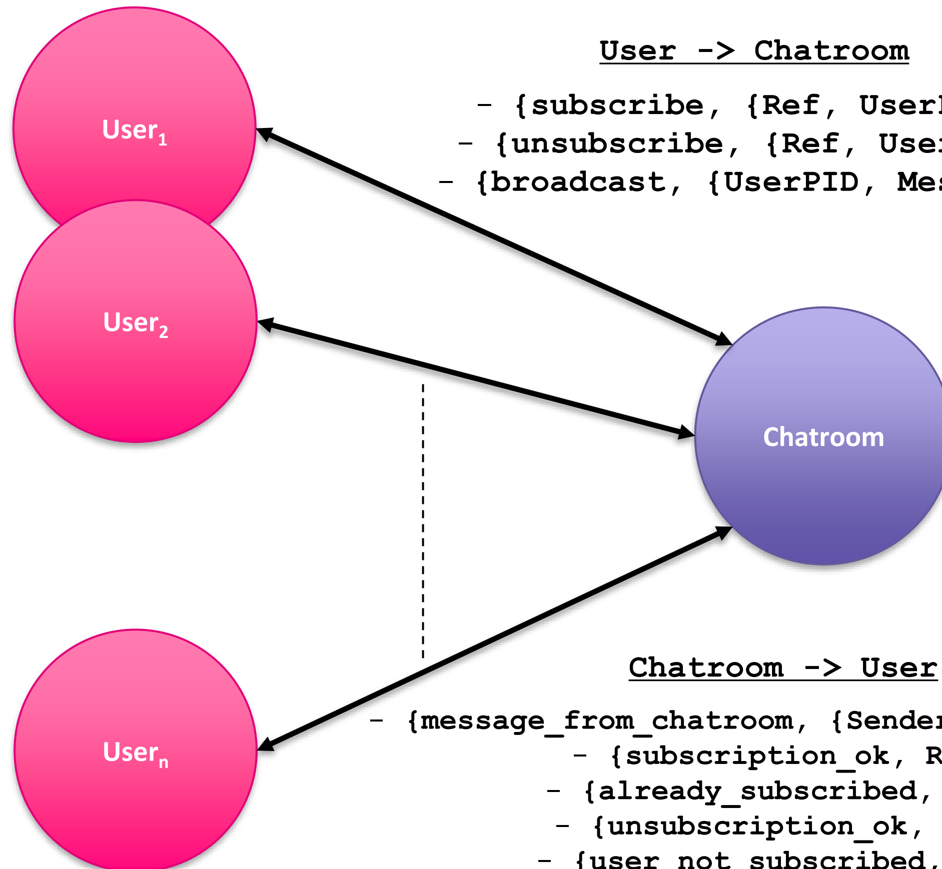



| Actors Model | Erlang |
| --- | --- |
| Actor | Module |
| Mailbox Address | Process identifier (PID) |
| Message | Erlang term (typically an atom or tuple) |
| State | Erlang term (typically a record) |
| Behaviour | loop() |
| Create actor | spawn |
| Send message | PID ! Message |
| Receive message | receive … end |

# A broadcast chatroom

# Broadcaster

- A set of user actors may subscribe to a chatroom actor
  - The chatroom must confirm the subscription
- Users may emit messages that the chatroom broadcasts to all subscribers (except for the sender)
- Users may unsubscribe
  - The chatroom must confirm the un-subscription.




**User -> Chatroom**

```
- {subscribe, {Ref, UserPID}}
- {unsubscribe, {Ref, UserPID})
- {broadcast, {UserPID, Message}}
```

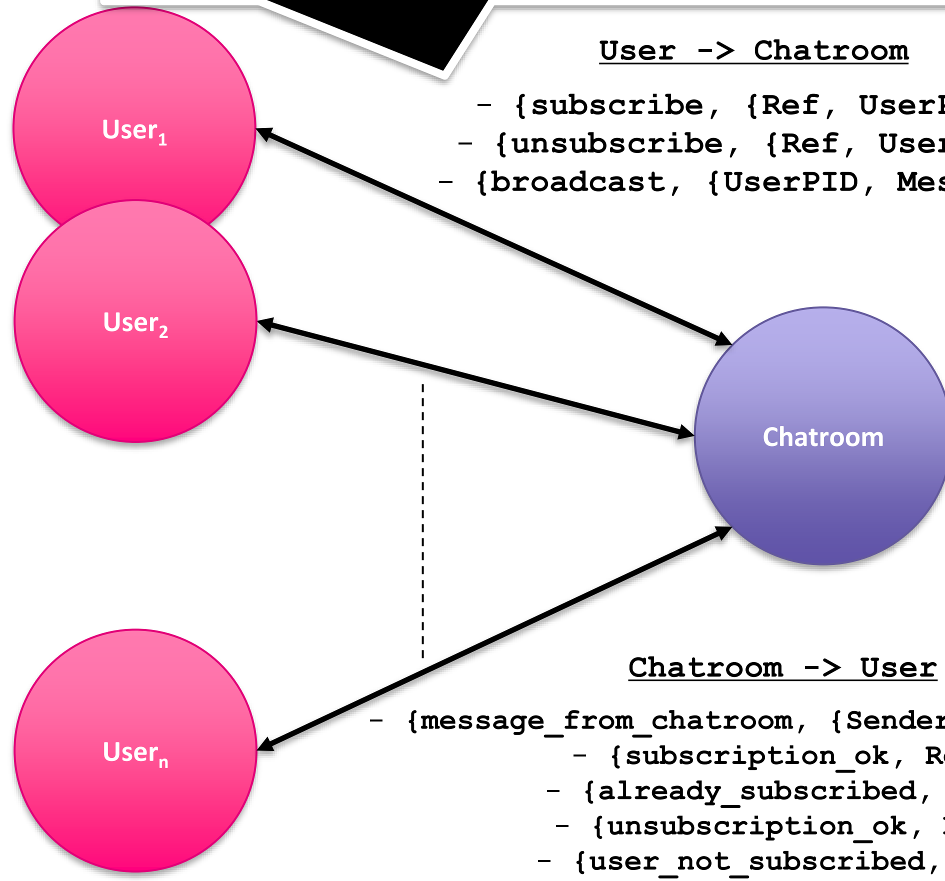
**Chatroom -> User**

```
- {message_from_chatroom, {SenderPID, Message}}
- {subscription_ok, Ref}
- {already_subscribed, Ref}
- {unsubscription_ok, Ref}
- {user_not_subscribed, Ref}
```

# Broadcaster

- A set of user actors may subscribe to a chatroom actor
  - The chatroom must confirm the subscription
- Users may emit messages that the chatroom broadcasts to all subscribers (except for the sender)
- Users may unsubscribe
  - The chatroom must confirm the un-subscription.

Important detail, messages do not contain information about the sender. If, for instance, the sender needs a reply, the message must contain a reference to the sender




**User -> Chatroom**

- `{subscribe, {Ref, UserPID}}`
- `{unsubscribe, {Ref, UserPID})`
- `{broadcast, {UserPID, Message}}`

**Chatroom -> User**

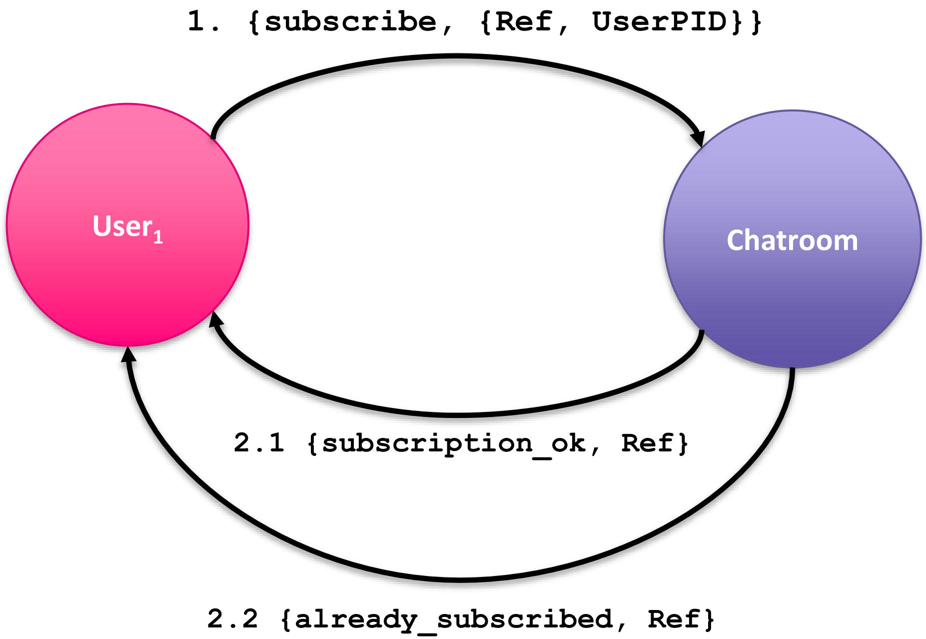
- `{message_from_chatroom, {SenderPID, Message}}`
- `{subscription_ok, Ref}`
- `{already_subscribed, Ref}`
- `{unsubscription_ok, Ref}`
- `{user_not_subscribed, Ref}`

# Synchronous communication - Broadcaster



- Subscriptions require <u>synchronous communication</u>
  - After the user actor sends subscribe it must <u>wait</u> until it receives a reply from the server

```erlang
%% User actor
handle_subscribe(ChatroomPID, State) ->
    Ref = make_ref(),
    ChatroomPID ! {subscribe, {Ref, self()}},
    receive
        {subscription_ok, Ref} ->
            NewState = State#user_state{chatroom=ChatroomPID},
            io:format("Successfully subscribed in the chatroom!~n"),
            loop(NewState);
        {already_subscribed, Ref} ->
            io:format("The chatroom says I am already subscribed.~n"),
            loop(State)
    end.

%% Chatroom actor
handle_subscribe(UserPID, Ref, State) ->
    Users = State#cr_state.users,
    case lists:member(UserPID, Users) of
        false ->
            NewState = State#cr_state{users= [UserPID|Users]},
            UserPID ! {subscription_ok, Ref},
            loop(NewState);
        true ->
            UserPID ! {already_subscribed, Ref},
            loop(State)
    end.
```



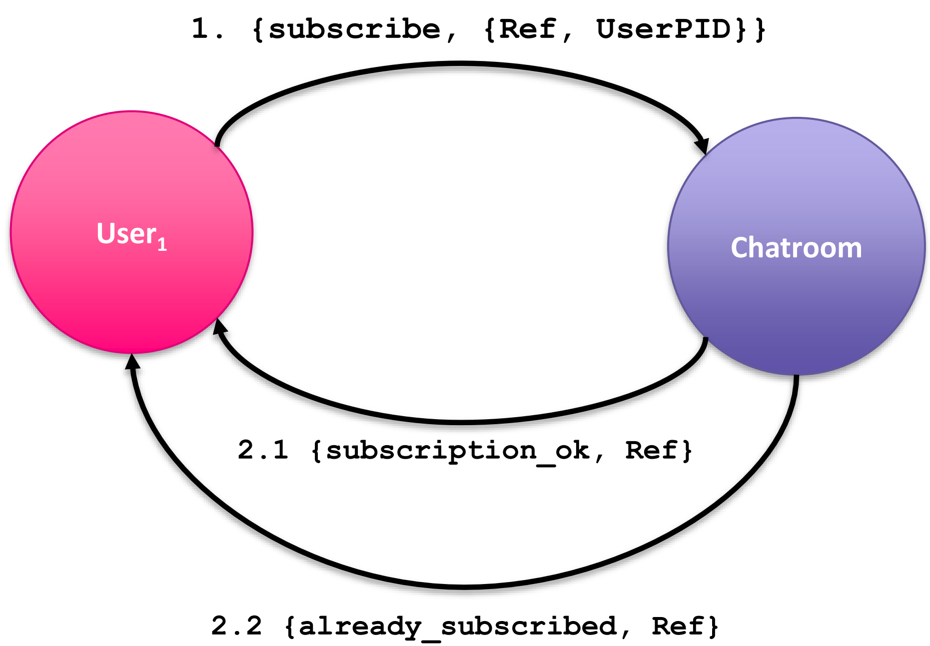
# Synchronous communicati



This is not a problem, as we can implement synchronous communication using the asynchronous send and receive from Erlang

- Subscriptions require synchronous communication
  - After the user actor sends subscribe it must wait until it receives a reply from the server

```
%% User actor
handle_subscribe(ChatroomPID, State) ->
    Ref = make_ref(),
    ChatroomPID ! {subscribe, {Ref, self()}},
    receive
        {subscription_ok, Ref} ->
            NewState = State#user_state{chatroom=ChatroomPID},
            io:format("Successfully subscribed in the chatroom!~n"),
            loop(NewState);
        {already_subscribed, Ref} ->
            io:format("The chatroom says I am already subscribed.~n"),
            loop(State)
    end.

%% Chatroom actor
handle_subscribe(UserPID, Ref, State) ->
    Users = State#cr_state.users,
    case lists:member(UserPID, Users) of
        false ->
            NewState = State#cr_state{users= [UserPID|Users]},
            UserPID ! {subscription_ok, Ref},
            loop(NewState);
        true ->
            UserPID ! {already_subscribed, Ref},
            loop(State)
    end.
```



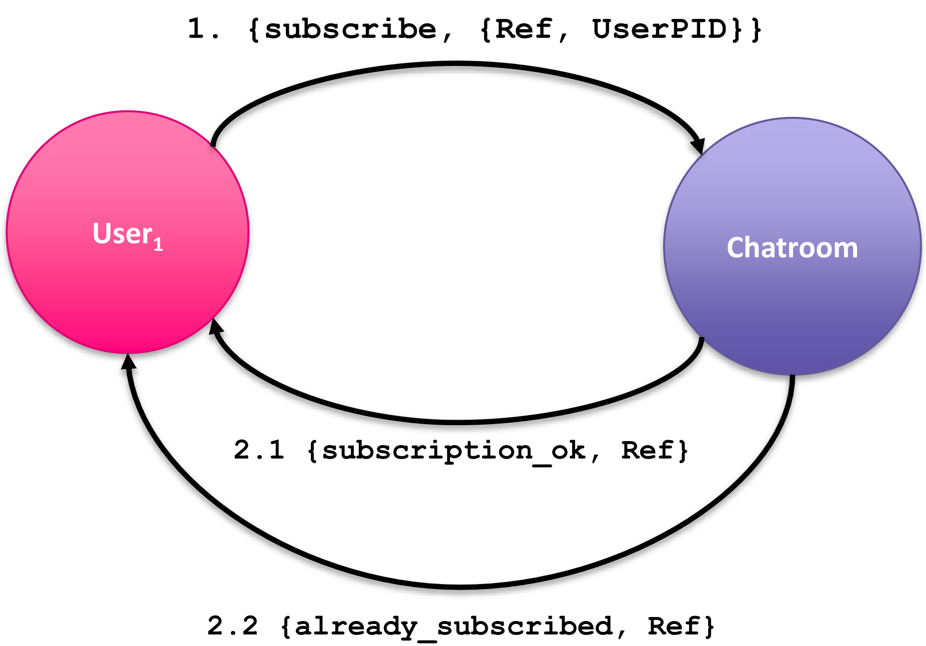
# Synchronous communicati

This is not a problem, as we can implement synchronous communication using the asynchronous send and receive from Erlang

- Subscriptions require synchronous communication
- …ubscribe it must wait until it receives a

References (via **`make_ref()`**) create a unique number that we can use to make sure that we wait for the reply to the message we sent

```
%% User act
handle_subscribe(ChatroomPID, State) ->
    Ref = make_ref(),
    ChatroomPID ! {subscribe, {Ref, self()}},
    receive
        {subscription_ok, Ref} ->
            NewState = State#user_state{chatroom=ChatroomPID},
            io:format("Successfully subscribed in the chatroom!~n"),
            loop(NewState);
        {already_subscribed, Ref} ->
            io:format("The chatroom says I am already subscribed.~n"),
            loop(State)
    end.

%% Chatroom actor
handle_subscribe(UserPID, Ref, State) ->
    Users = State#cr_state.users,
    case lists:member(UserPID, Users) of
        false ->
            NewState = State#cr_state{users= [UserPID|Users]},
            UserPID ! {subscription_ok, Ref},
            loop(NewState);
        true ->
            UserPID ! {already_subscribed, Ref},
            loop(State)
    end.
```



# Synchronous communicati

This is not a problem, as we can implement synchronous communication using the asynchronous send and receive from Erlang

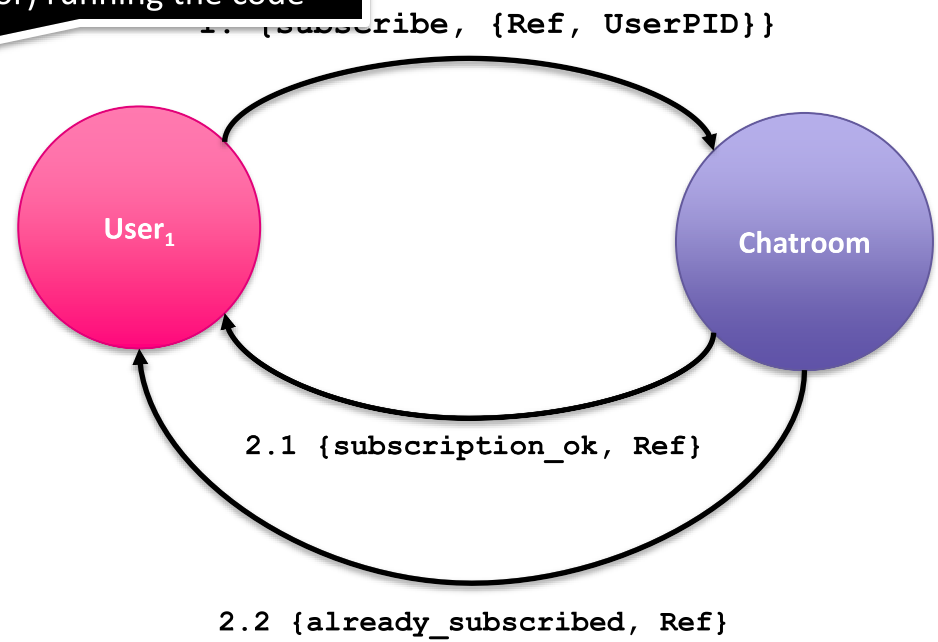


- Subscriptions require synchronous communication
- ... il it receives a

References (via **`make_ref()`**) create a unique number that we can use to make sure that we wait for the reply to the message we sent

The function **`self()`** returns the PID of the Erlang process (actor) running the code

```
%% User act
handle_subscribe(ChatroomPID, State) ->
    Ref = make_ref(),
    ChatroomPID ! {subscribe, {Ref, self()}},
    receive
        {subscription_ok, Ref} ->
            NewState = State#user_state{chatroom=ChatroomPID},
            io:format("Successfully subscribed in the chatroom!~n"),
            loop(NewState);
        {already_subscribed, Ref} ->
            io:format("The chatroom says I am already subscribed.~n"),
            loop(State)
    end.

%% Chatroom actor
handle_subscribe(UserPID, Ref, State) ->
    Users = State#cr_state.users,
    case lists:member(UserPID, Users) of
        false ->
            NewState = State#cr_state{users= [UserPID|Users]},
            UserPID ! {subscription_ok, Ref},
            loop(NewState);
        true ->
            UserPID ! {already_subscribed, Ref},
            loop(State)
    end.
```

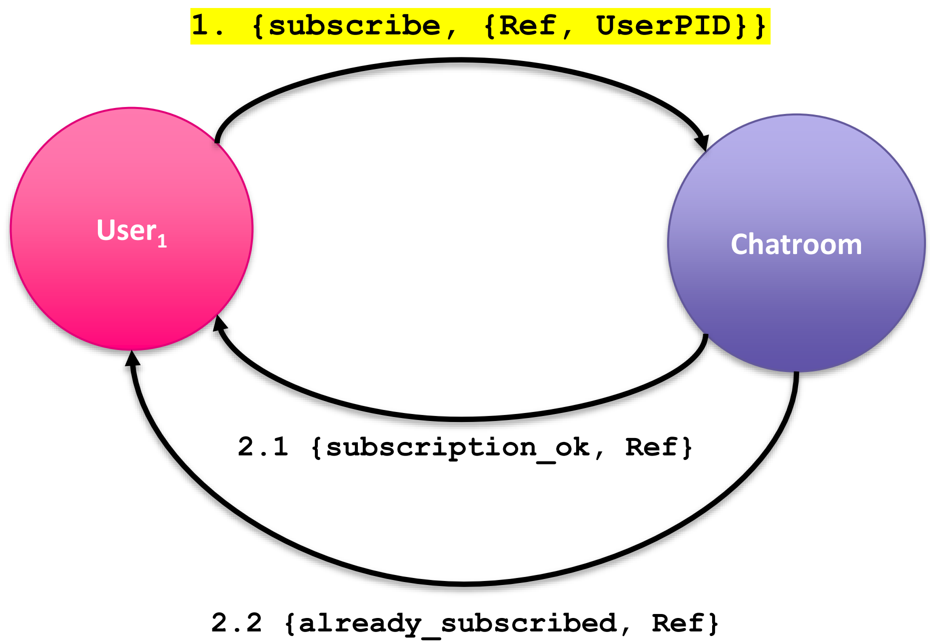


# Synchronous communication - Broadcaster



- Subscriptions require synchronous communication
  - After the user actor sends subscribe it must wait until it receives a reply from the server

```
%% User actor
handle_subscribe(ChatroomPID, State) ->
    Ref = make_ref(),
    ChatroomPID ! {subscribe, {Ref, self()}},
    receive
        {subscription_ok, Ref} ->
            NewState = State#user_state{chatroom=ChatroomPID},
            io:format("Successfully subscribed in the chatroom!~n"),
            loop(NewState);
        {already_subscribed, Ref} ->
            io:format("The chatroom says I am already subscribed.~n"),
            loop(State)
    end.

%% Chatroom actor
handle_subscribe(UserPID, Ref, State) ->
    Users = State#cr_state.users,
    case lists:member(UserPID, Users) of
        false ->
            NewState = State#cr_state{users= [UserPID|Users]},
            UserPID ! {subscription_ok, Ref},
            loop(NewState);
        true ->
            UserPID ! {already_subscribed, Ref},
            loop(State)
    end.
```

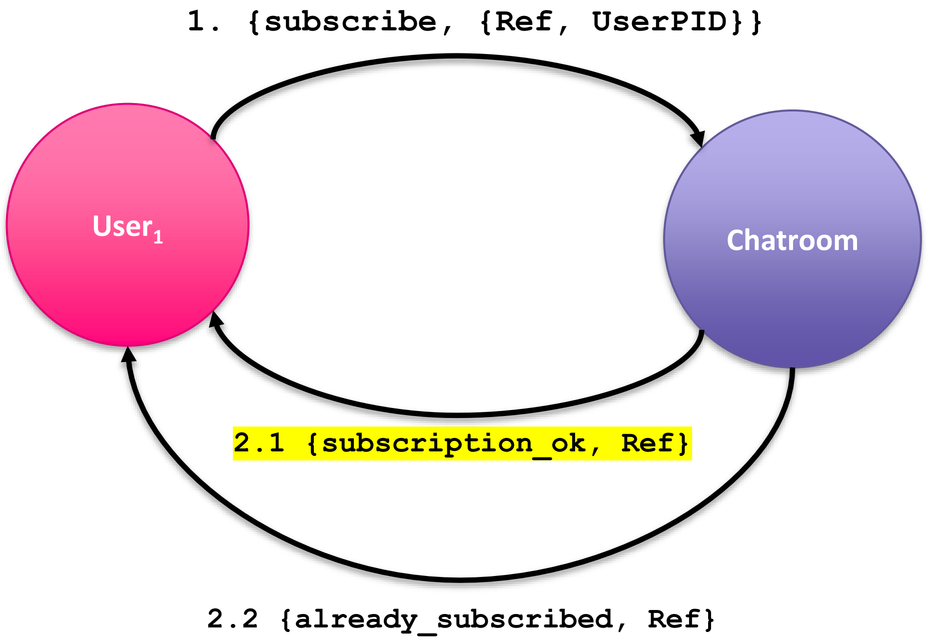


# Synchronous communication - Broadcaster



- Subscriptions require <u>synchronous communication</u>
  - After the user actor sends subscribe it must <u>wait</u> until it receives a reply from the server

```
%% User actor
handle_subscribe(ChatroomPID, State) ->
    Ref = make_ref(),
    ChatroomPID ! {subscribe, {Ref, self()}},
    receive
        {subscription_ok, Ref} ->
            NewState = State#user_state{chatroom=ChatroomPID},
            io:format("Successfully subscribed in the chatroom!~n"),
            loop(NewState);
        {already_subscribed, Ref} ->
            io:format("The chatroom says I am already subscribed.~n"),
            loop(State)
    end.

%% Chatroom actor
handle_subscribe(UserPID, Ref, State) ->
    Users = State#cr_state.users,
    case lists:member(UserPID, Users) of
        false ->
            NewState = State#cr_state{users= [UserPID|Users]},
            UserPID ! {subscription_ok, Ref},
            loop(NewState);
        true ->
            UserPID ! {already_subscribed, Ref},
            loop(State)
    end.
```

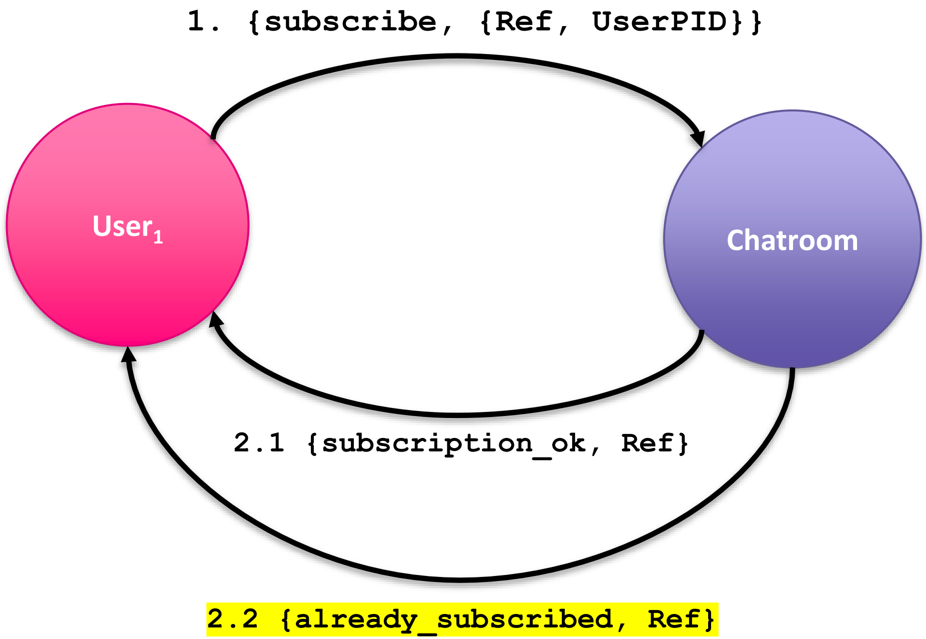


# Synchronous communication - Broadcaster



- Subscriptions require synchronous communication
  - After the user actor sends subscribe it must wait until it receives a reply from the server

```
%% User actor
handle_subscribe(ChatroomPID, State) ->
    Ref = make_ref(),
    ChatroomPID ! {subscribe, {Ref, self()}},
    receive
        {subscription_ok, Ref} ->
            NewState = State#user_state{chatroom=ChatroomPID},
            io:format("Successfully subscribed in the chatroom!~n"),
            loop(NewState);
        {already_subscribed, Ref} ->
            io:format("The chatroom says I am already subscribed.~n"),
            loop(State)
    end.

%% Chatroom actor
handle_subscribe(UserPID, Ref, State) ->
    Users = State#cr_state.users,
    case lists:member(UserPID, Users) of
        false ->
            NewState = State#cr_state{users= [UserPID|Users]},
            UserPID ! {subscription_ok, Ref},
            loop(NewState);
        true ->
            UserPID ! {already_subscribed, Ref},
            loop(State)
    end.
```



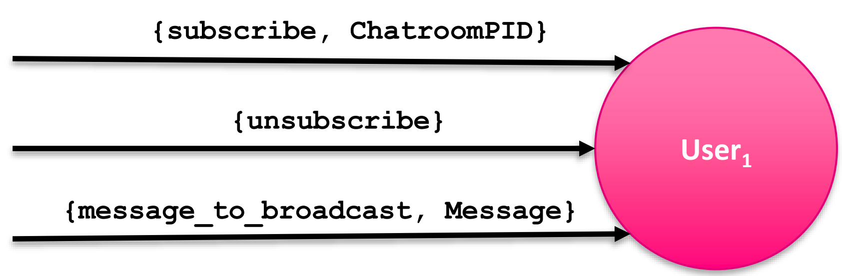
# User API - Broadcaster

- Often it is a good idea to hide the message protocol from the users of our Erlang programs
  - To this end, define set of API functions which trigger the necessary messages

```
% 6. API
subscribe_to_chatroom(UserPID, ChatroomPID) ->
    UserPID ! {subscribe, ChatroomPID}.

unsubscribe_from_chatroom(UserPID) ->
    UserPID ! {unsubscribe}.

send_message_to_broadcast(UserPID, Message) ->
    UserPID ! {message_to_broadcast, Message}.
```

```
loop(State) ->
    receive
        {subscribe, ChatroomPID} ->
            handle_subscribe(ChatroomPID, State);
        {unsubscribe} ->
            handle_unsubscribe(State);
        {message_to_broadcast, Message} ->
            handle_message_to_broadcast(Message, State);
        …
end.
```

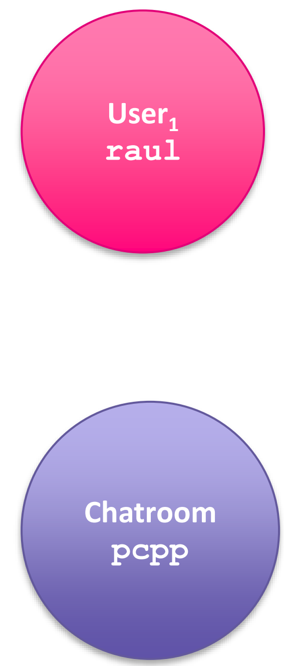
```
> user:subscribe_to_chatroom(UserPID, ChatroomPID)
> user:unsubscribe_from_chatroom(UserPID)
> user:send_message_to_broadcast(UserPID, Message)
```



# Registering processes - Broadcaster

- In Erlang, it is possible to register processes using an atom
  - This allows to send messages using said atom
  - For the broadcaster, you can use the **`start_reg`** function

```
%% User actor
start_reg(Username) ->
    PID = spawn(?MODULE, init, [Username]),
    register(Username, PID).


%% Chatroom actor
start_reg(ChatroomName) ->
    PID = spawn(?MODULE, init, [ChatroomName]),
    register(ChatroomName, PID).
```

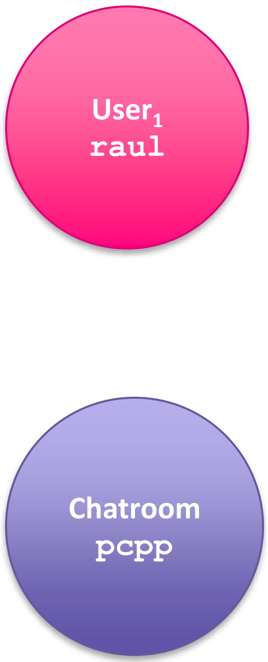


# Registering processes - Broadcaster

- In Erlang, it is possible to register processes using an atom
  - This allows to send messages using said atom
  - For the broadcaster, you can use the **`start_reg`** function

```
%% User actor
start_reg(Username) ->
    PID = spawn(?MODULE, init, [Username]),
    register(Username, PID).


%% Chatroom actor
start_reg(ChatroomName) ->
    PID = spawn(?MODULE, init, [ChatroomName]),
    register(ChatroomName, PID).
```

Demo time!



# Broadcaster interesting executions

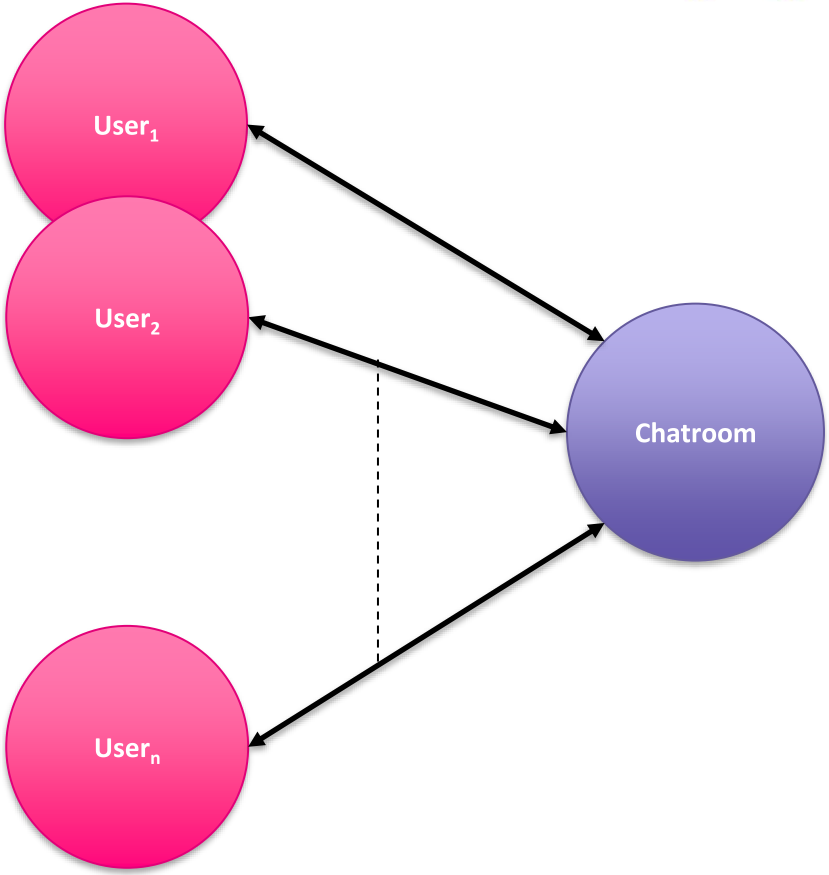


Recall: FIFO mailboxes

- Consider this execution

1. user1 sends subscription to Chatroom
2. user2 sends subscription to Chatroom
3. …

What actor will receive first subscription_ok?



# Broadcaster interesting executions

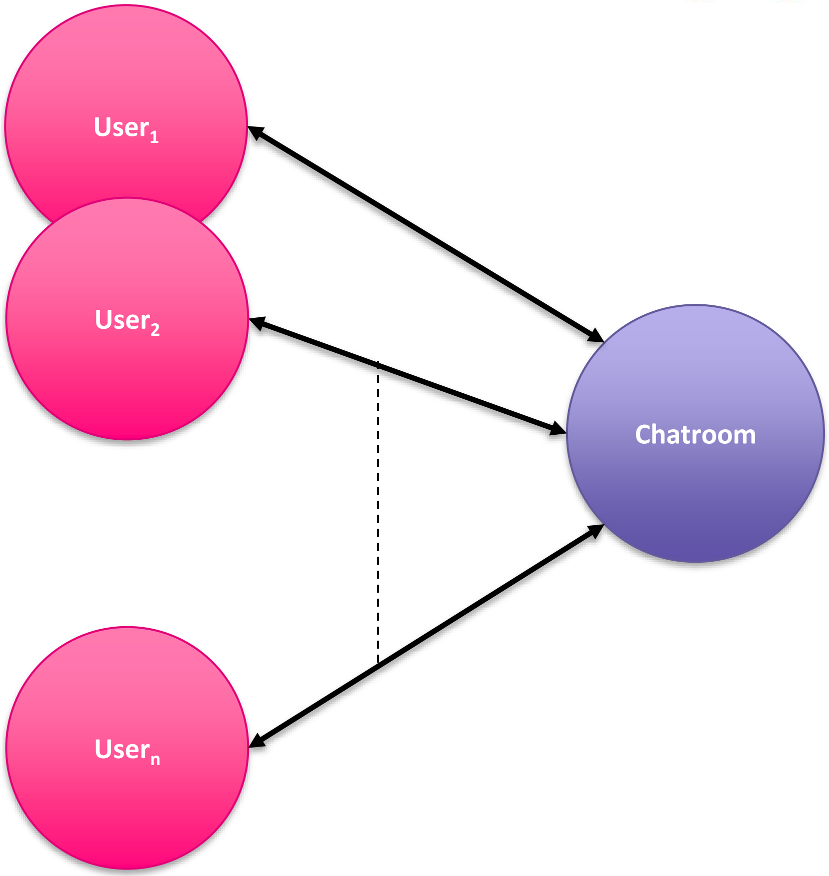


Recall: FIFO mailboxes

- Consider this execution

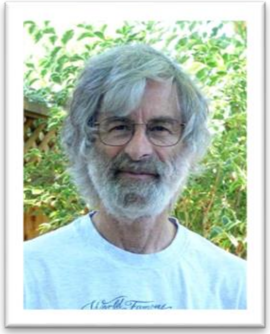
1. user1 sends subscription to chatroom
2. chatroom replies subscription_ok to user1
3. user1 emits message to chatroom
4. user2 sends subscription to chatroom
5. …

Can user2 receive the message sent by user1 in step 3?



# Happened-before in distributed systems

- Note that in the previous questions the behaviour of the systems depends on the reception of messages



- Thus, the happened-before relation defined by Lamport is useful in reasoning about actor systems
  - An action *a* happens-before an action *b* if they belong to the same actor and *a* was executed before *b*
  - A send(m) action happens-before its corresponding receive(m)

- Note the similarity with the happens-before relation of the Java memory model
  - We reason about message exchange instead of locking (but *inherent coordination problems remain*)
  - Visibility issues disappear as actors only access local memory

# A bounded buffer

# Producer-consumer problem | Intuition

- Perhaps more intuitive example

Consumers

Producers

Shared data structure of fixed size

# Producer-consumer problem | Intuition

Producers

Consumers

Shared data structure of fixed size

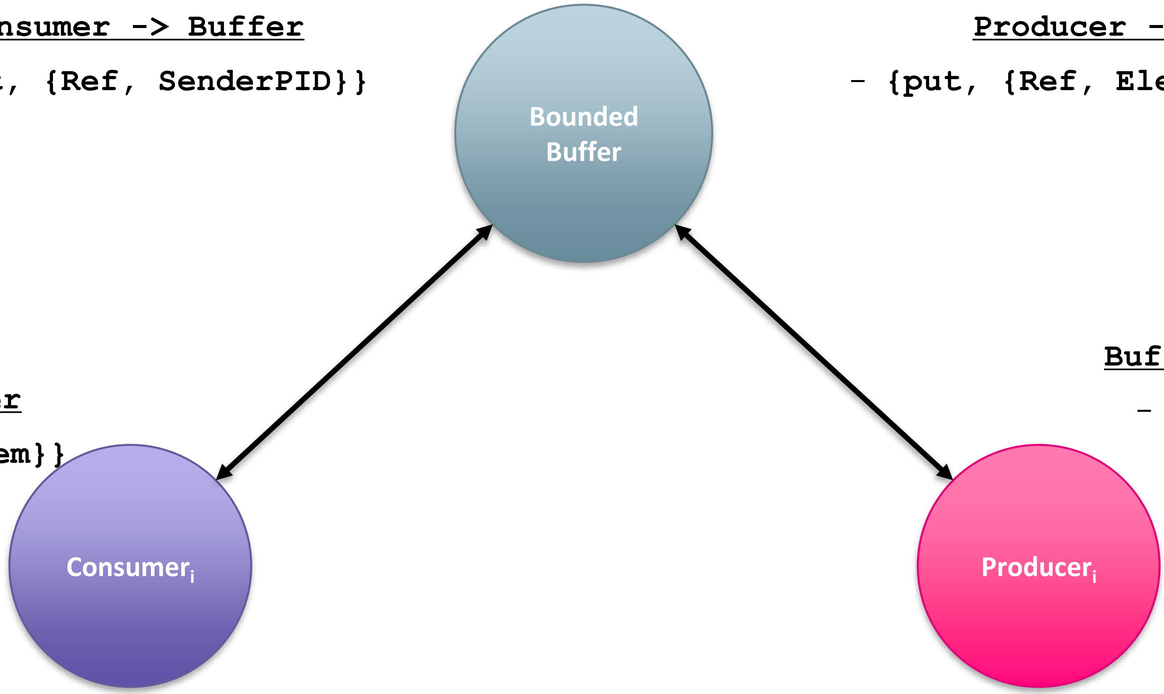
# Bounded Buffer with Actors

**<u>`Consumer -> Buffer`</u>**

- `{get, {Ref, SenderPID}}`

**<u>`Producer -> Buffer`</u>**

- `{put, {Ref, Elem, SenderPID}}`




**<u>`Buffer -> Producer`</u>**

- `{put_ok, Ref}`

**<u>`Buffer -> Consumer`</u>**

- `{get_ok, {Ref, elem}}`

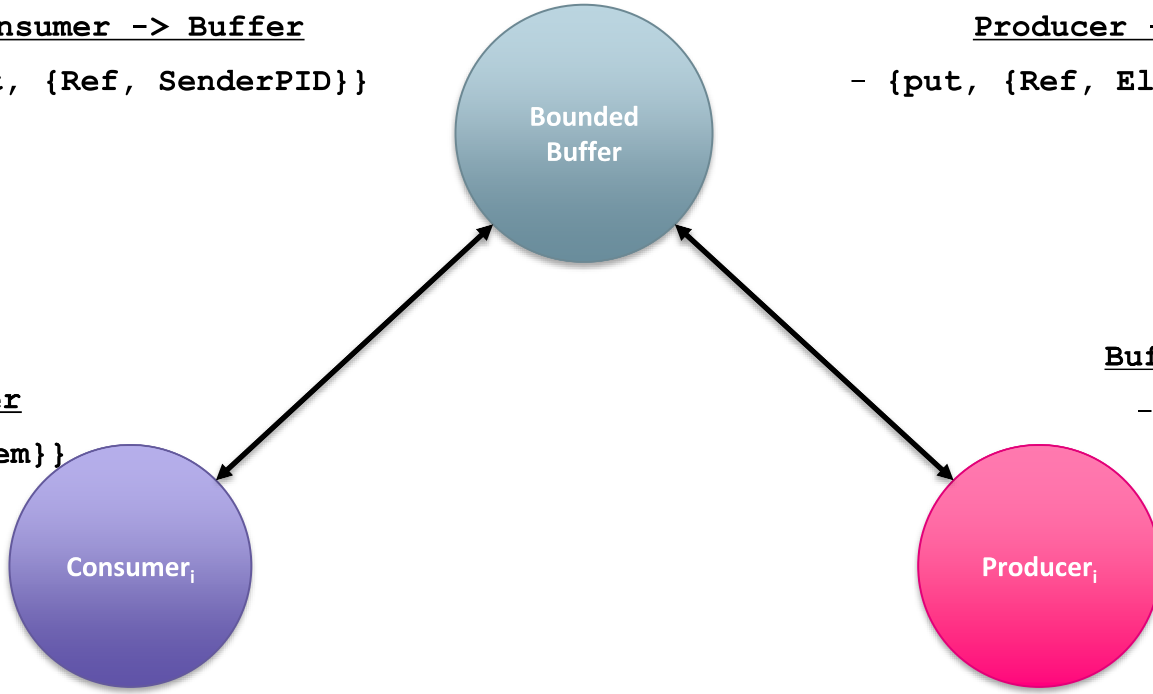
# Bounded Buffer with Actors

**Consumer -> Buffer**

`- {get, {Ref, SenderPID}}`

**Producer -> Buffer**

`- {put, {Ref, Elem, SenderPID}}`




**Buffer -> Producer**

`- {put_ok, Ref}`

**Buffer -> Consumer**

`- {get_ok, {Ref, elem}}`

Let's look at the code (bounded_buffer package)

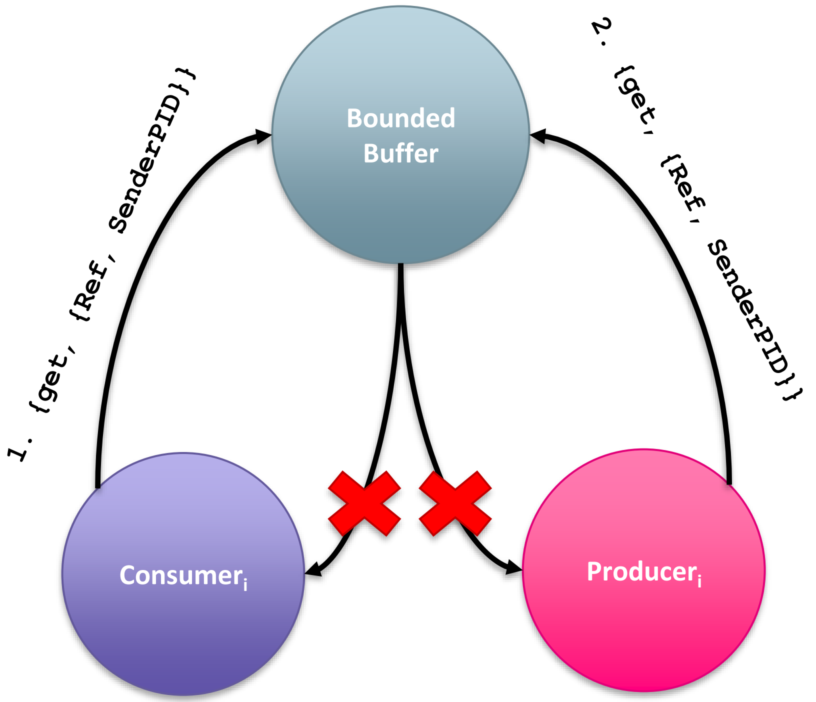
- After a request to consume or produce, the actors must wait for the reply from the bounded buffer (i.e., synchronous communication)
- Consumers wait if the buffer is empty
- Producers wait if the buffer is full

# Deadlocks in Actor Systems – Bounded Buffer



- An actor system may end up in a deadlock state

```
deadlock_situation() ->
    BufferPID = buffer:start(1),
    A1 =
        fun () ->
            buffer:get(BufferPID),
            buffer:put(BufferPID, something),
            io:format("Done~n")
        end,
    A2 =
        fun () ->
            buffer:get(BufferPID),
            buffer:put(BufferPID, something),
            io:format("Done~n")
        end,
    lists:foreach(fun (A) -> spawn(A) end, [A1,A2]).
```



# Actors in distributed systems



The actors model has natural mapping in distributed systems

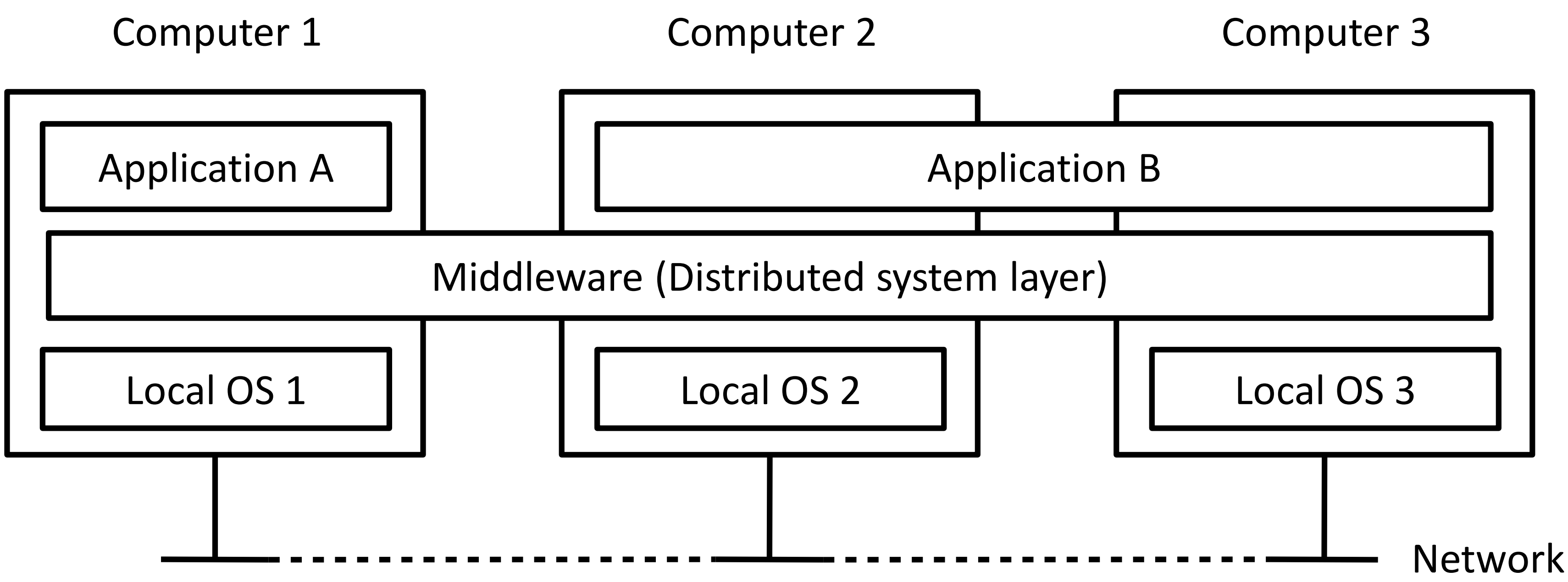
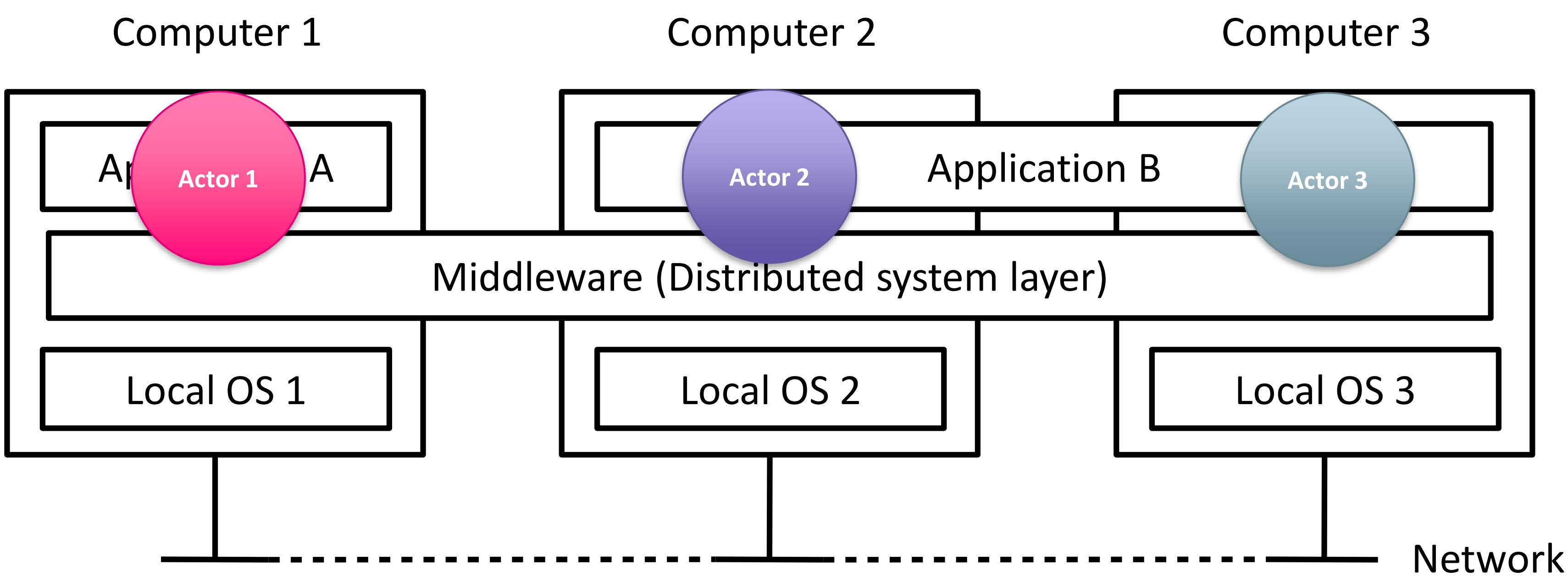



Figure taken from -> Distributed Systems: Principles and Paradigms. Andrew S. Tanenbaum and Maarten Van Steen. 2007.

# Actors in distributed systems

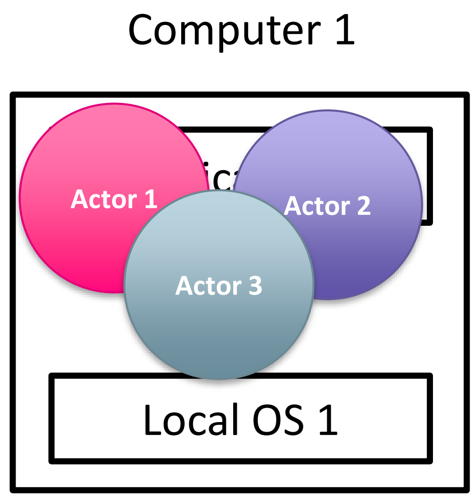
The actors model has natural mapping in distributed systems



# Actors in a single computer



The actors model is applicable in a single computer as well




**In this lecture, we focus on this type of actor system**

# Agenda



- Problems in shared memory concurrency (revisited)
- Actors
- Erlang
- Example systems
  - Turnstile (counter)
  - Broadcaster
  - Bounded Buffer