University of Southampton

University of Southampton

COMP2212 Programming Language Concepts

Message Passing Concurrency

Dr Julian Rathke

Message Passing

- It is evident that common wisdom is that concurrent programming is hard.
- This stems from the difficulties arising in the shared memory concurrency model and the issues with
    - Race conditions, critical sections, mutexes, deadlock, cache visibility, instruction reordering etc
- However, shared memory concurrency isn't the only game in town.
- Recall that a fundamental principle of concurrent programming is that threads must communicate by sending some sort of messages to each other.
- What if we build our concurrency model and programming languages around that principle instead?
- That is, assuming an efficient, correct implementation of a mechanism by which we can send messages addressed from one thread to another then concurrent programming can be reduced to a series of thread creation and message passing instructions.

Advantages of Message Passing

- If we give up on shared memory concurrency and use message passing instead then we
  - Don't have to worry about critical regions and mutex locks
  - Run less risk of deadlock as we aren't necessarily holding locks
  - Avoid problems with cache visibility as we know which thread is the intended recipient of the message
- Also consider that when programming for distributed systems we have a number of processes running on different machines trying to communicate.
  - Shared memory is not obviously an option in this scenario so message passing is typically used here.
  - By using message passing for concurrent programming too we can use a single model that scales from single machine, to multi/many core to distributed systems.
- Message Passing systems have well established interfaces (e.g. MPI) and well studied theory.

Disadvantages of Message Passing

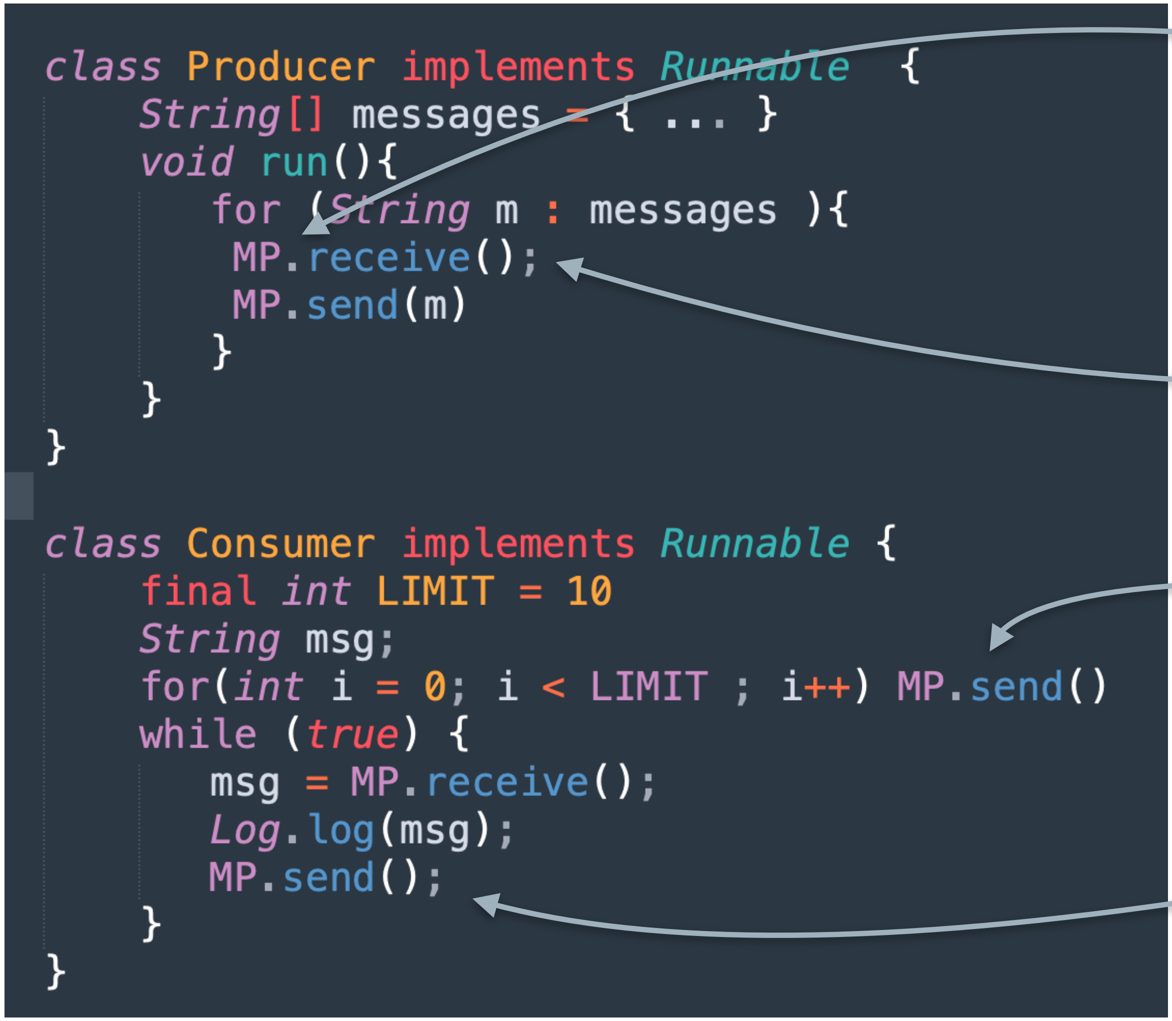
- So with all of these advantages, why is the predominant model of concurrent programming still based on shared memory?
- Speed!
- Shared memory concurrency, with architectural support in the form of test and set lock instructions is very fast when compared to higher-level implementations of message queues.
- Research trends indicate that for large multi/many core systems then with suitable architectural support for direct small messages between cores and a mapping table of threads to cores then message passing could become as efficient as shared memory communication.
- Watch this space!

Synchrony and Asynchrony

- We discussed already the distinction between synchronous and asynchronous communication.
- Synchronous message passing (sender blocks until receive is ready) is popular for hardware design languages and is easier to program with but is difficult (and at worst, impossible) to arrange in a distributed system.
- Asynchronous message passing (sender does not block, messages are buffered) is the popular choice for distributed applications.
- Programming for asynchronous systems can be difficult :
    - Typically the idiom is to send a message and set up a receiver for a callback when the message is received.
    - Callbacks appear "non-linearly" in the program flow and do not help with maintainability and debugging.
    - The flow of messages controls the program flow!

Example Producer/Consumer

- Consider the following Java-like code showing a Producer/Consumer implementation that uses asynchronous message passing to communicate in a bounded way



```
class Producer implements Runnable  {
    String[] messages = { ... }
    void run(){
       for (String m : messages ){
        MP.receive();
        MP.send(m)
       }
    }
}

class Consumer implements Runnable {
    final int LIMIT = 10
    String msg;
    for(int i = 0; i < LIMIT ; i++) MP.send()
    while (true) {
       msg = MP.receive();
       Log.log(msg);
       MP.send();
    }
}
```

MP is assumed to be a message passing implementation - the buffering happens in here.

The producer waits to receive notice of a "slot", then writes the next message.

The consumer advertises the number of available "slots" in the communication buffer

After receiving each message, another "slot" becomes available.

Channels

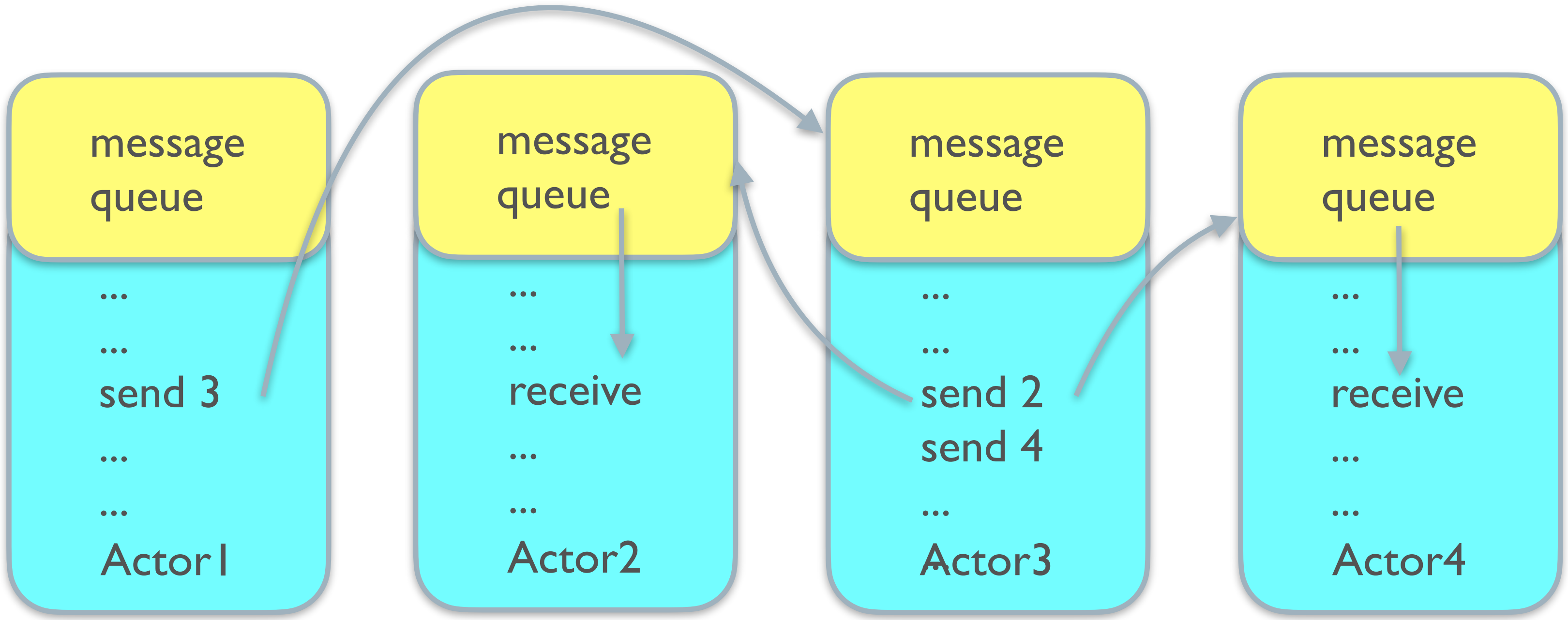
- In the previous example we assumed a single point through which to send and receive messages.
- This clearly does not scale well for many threads.
- In practice, messages are
    - a **handshake** between just two threads
    - group **multicast** between many threads
    - **broadcast** to all threads in the system
- For the former two at least, in order to achieve this it is common to use a naming scheme to identify a **channel** of communication. Sending and receiving of messages is done via a named channel.
- A common alternative approach is to implement channel objects, sometimes with separate endpoints for just sending or just receiving messages.

The Actor Model

- The Actor Model is a model of concurrent programming inspired from ideas in physics in quantum mechanics and general relativity and the programming languages Lisp, Simula and Smalltalk.
- Actors were first proposed in 1973 by Hewitt et al
- Since then they have been implemented as primitives in many programming languages and, most recently, been used as the concurrency model for
  - Erlang, Scala, Dart, Swift and Ruby
- Many languages have Actor libraries rather than direct language support:
  - Python, Java (Akka), C, C++, Rust, Haskell etc
- As such they become a viable option for programming both concurrent and distributed activity.

Actors

- The starting point is that all computation is performed by Actors
- Actors do not share memory
- Actors have unique addresses to which they can be sent messages
- Actors have a thread of control
- Actors have a "mailbox" that acts as a FIFO queue for incoming messages.
- Actors can send messages, or values, to other actors and receive messages from their own mailbox.

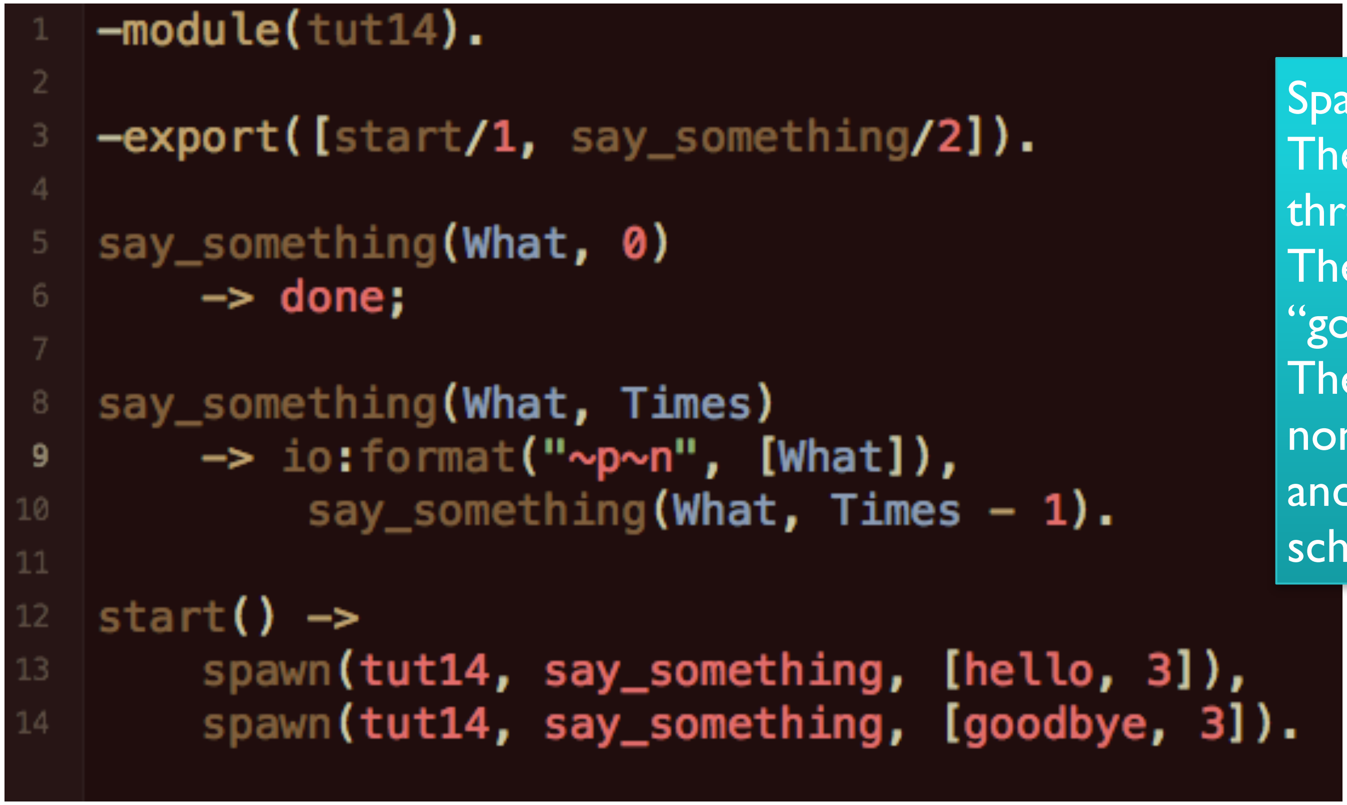


Actor Messages

- Actor messages consist of values - these can be typed or untyped.
- Values can include the addresses of other actors - this means the underlying communication topology is dynamic.
- In response to receiving a message, an actor can
    - Send a finite number of messages to other actors
    - Create, or spawn, a finite number of new actors
    - Change its own behavioural state.
- Actors communicate "locally" this means actors only send to other actors for which
    - They already know the actors address
    - They have received the actor's address in a previous message
    - They are the creator of the actor
- It may of course be possible to "guess" addresses to send to but a decent address allocation algorithm should avoid this.

Actor Example 1

- Let's first look at an example in Erlang in which there are no messages sent or received but new Actors are spawned.
- The instance of tut14 has an entry point of start(), this then spawns two more instances of tut14 with different entry points and parameters.



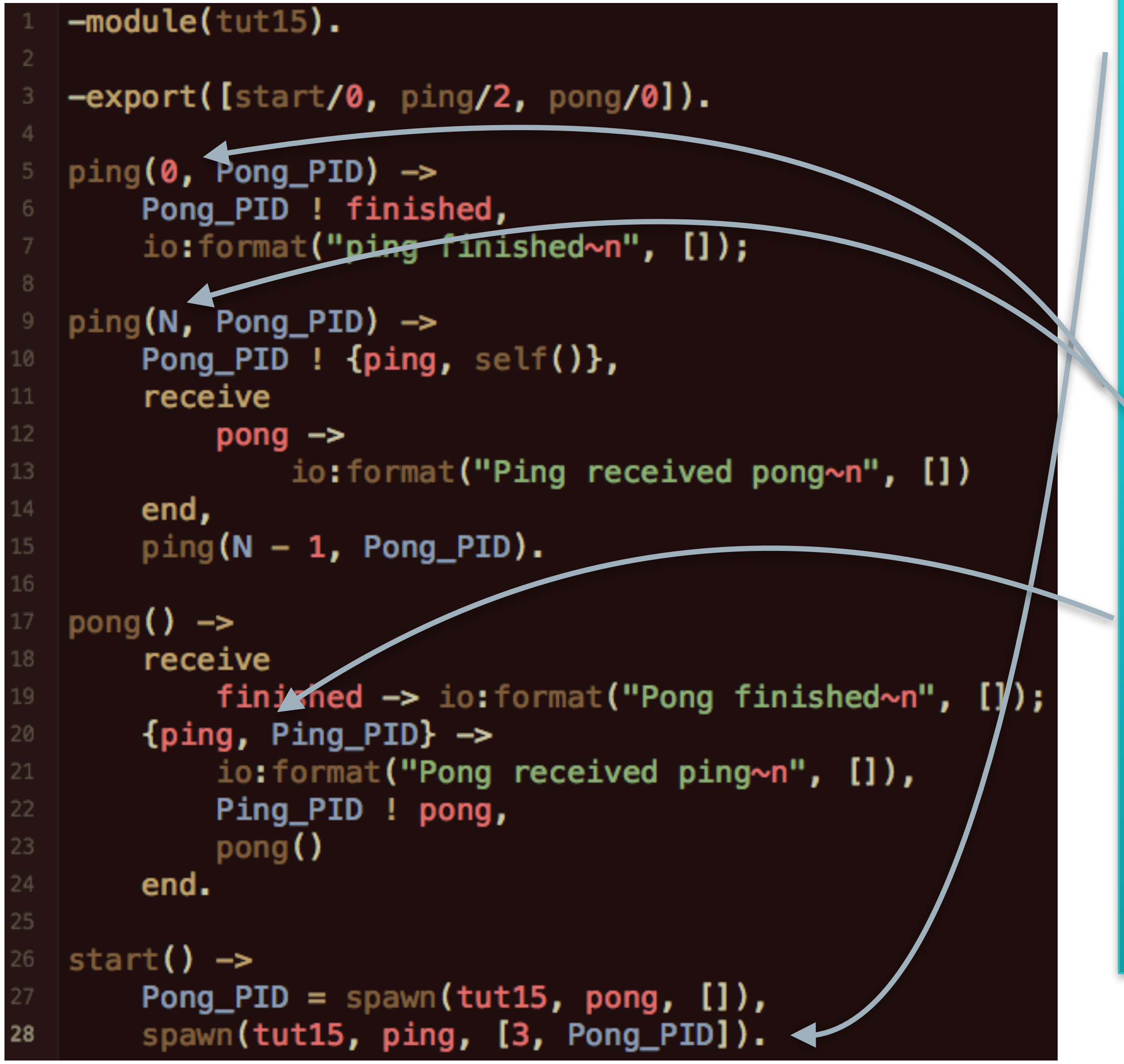
```erlang
-module(tut14).

-export([start/1, say_something/2]).

say_something(What, 0)
    -> done;

say_something(What, Times)
    -> io:format("~p~n", [What]),
        say_something(What, Times - 1).

start() ->
    spawn(tut14, say_something, [hello, 3]),
    spawn(tut14, say_something, [goodbye, 3]).
```

Spawns two threads:
The first writes "hello" three times,
The second writes "goodbye" three times.
The interleaving is nondeterministic and depends on the scheduler.

http://www.erlang.org/download/getting_started-5.4.pdf

Actor Example 2

- The next Erlang example involves message passing



```erlang
-module(tut15).

-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished -> io:format("Pong finished~n", []);
    {ping, Ping_PID} ->
        io:format("Pong received ping~n", []),
        Ping_PID ! pong,
        pong()
    end.

start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```

Spawns two actors: the first calls pong and the second, ping. Pong_PID stores the address of pong.

The ping pattern matches its inputs, if non-zero, it sends a message to Pong_PID and its own address then waits for a reply message called "pong".

The pong actor receives messages and pattern matches the content. For "ping" messages that come with a reply address, the actor responds with a "pong" message to that address.

http://www.erlang.org/download/getting_started-5.4.pdf

Hierarchical Actors

- Another feature of some Actor implementations is the ability to group Actors in a hierarchy. The Scala Akka library supports this.
- Each Actor has a unique parent Actor who created it and the parent Actor supervises various features of the behaviour.
- For example, the parent can delegate actions to its sub-Actors
- Error handling in Actors can also be handled hierarchically:
    - Failures (exceptions) in lower Actors can be passed up to higher Actors in the hierarchy if not handled.
    - Handling can be specified as a
        - "One For One" strategy - handler applies to failed children only
        - "All for One" strategy - handler applies to all children.
    - Possible Handler actions to an Actor are
        - Resume, Restart, Stop, Escalate

Next Lecture

Coroutines