Actors, Active Objects and Asynchronous Communication

Andrea Pferscher

14.10.2024

University of Oslo

Part 2: Message Passing

Structure

- Part 1: Shared Memory (and Java)
- **Part 2:** Message Passing (and Go)
- Part 3: Analyses and Tool Support (and maybe Rust)

Content of this part:

- Synchronous and asynchronous message passing
- Channels, actors, go-routines, asynchrounous programming

Outline Today

- Actors
- Futures and promises
- Active objects
- Asynchronous communication with `await`-statement

Repetition

Message Passing and Channels

- Shared memory vs. distributed memory
- Synchronous and asynchronous message passing, the high level picture
- *Asynchronous message passing*: channels, messages, primitives
- Example: filters and sorting networks
- Comparison of message passing and monitors
- Basics *synchronous message passing*

Actors

Async. Communication without Channels

Channels

- Need additional primitives for concurrency; `send` and `receive`
- Channels are explicit while process/objects are implicit
- Complex typing disciplines

Can we do asynchronous communication without explicit channels?

- Actors: Messages between objects
- Active Objects: Messages between objects with cooperative scheduling
- Async/Await in mainstream languages: Using (lightweight) threads (with shared memory)

Actors

- Actors: a programming concept for distributed concurrency which combines a number of topics we have discussed in the course;
    - active monitors,
    - objects and encapsulation,
    - race-free (no race conditions on shared state)
- Examples of programming languages that implement actors:
  Erlang, Scala's Akka library, Dart, Swift, etc.

Object-Oriented Programming and Language Design

What are objects

How do OO programs fit into the design of programming languages?

- **State space**: local or global?
- **Thread** interaction and **objects**
- **Communication**: shared variables, channels or messages?
- **Communication**: synchronous or asynchronous?
- **Dynamic state allocation**: object creation

What can we do to protect objects against races?

Can we combine objects with ideas from monitors?

- Passive monitors vs. active monitors
- A method is *active*, if a statement in the method is executed by some thread

Passive Monitors – Repetition

Await

```
monitor name {
 monitor variables
 ## monitor invariant
 initialization code
 procedures
}
```

- Threads *communicate* by calling monitor methods
- Threads do not need to know all the implementation details: only the procedure names are visible from outside the monitor
- Statements *inside* a monitor: *no* access to variables *outside* the monitor
- Statements *outside* a monitor: *no* access to variables *inside* the monitor
- **Monitor variables:** *initialized* before the monitor is used
- **Monitor invariant:** describes a condition on the inner state
- The monitor invariant can be analyzed by sequential reasoning inside the monitor

Passive Monitors: Synchronization with condition variables – Repetition

- Monitors contain *special* type of variables: **cond** (condition)
- Used for synchronization/to delay processes
- Each such variable is associated with a *wait condition*
- The value of a condition variable: *queue* of delayed threads
- Not directly accessible by programmer, instead, manipulated by special operations

```
cond cv;        # declares a condition variable cv
empty(cv);      # asks if the queue on cv is empty
wait(cv);       # causes thread to wait in the cv queue
signal(cv);     # wakes up a thread in the queue to cv
signal_all(cv); # wakes up all threads in the cv queue
```

Passive Monitors – Repetition

Await

```
monitor Mon { // monitor invariant: r ≥ 0
  int r := 0 // number of resources
  cond res; // wait condition variable

  procedure Acquire() {
    while (r=0) { wait (res) };
    r := r − 1 }

  procedure Release() {
    r := r+1;
    signal (res); }}
```

- `wait` and `signal`: *FIFO signaling strategy*
- A thread in the monitor can execute `signal(cv)`.
  If there is a waiting thread, do we get *two active methods* in the monitor?

Objects as Passive Monitors in Java

Java

```
class Mon { // class invariant: this.r >= 0
  int r = 0; // number of resources
  Condition res; // wait condition
  public synchronized void Acquire() {
    while( r == 0 ) { res.await(); };
    r = r - 1; }
  public synchronized void Release() {
    r = r + 1;
    res.signal(); }}
```

- How do condition variables and synchronized methods relate?

Actors

Fundamental idea: Decouple communication and control.

Capabilities of Actors

An actor reacts to incoming messages to

- change its state,
- send a finite number of messages to other actors, and
- create a finite number of new actors.

Intuition

We can think of an actor as an object that can only communicate asynchronously.
Some actor models can also pattern match over its message queue of incoming messages.

Implementation of Actors in Programming Languages

- Supported by numerous languages and frameworks
- Not always strictly OO: Erlang, ...
- Sometimes as library, not part of language: Akka actors, ...
- Numerous differences on how basic capabilities are implemented or extended

- Type safety: Can we guaranteed statically whether messages can be processed?
- Integration with OO: Are messages methods? Do actors have a class?
- Integration with other primitives: Can actors share state?
- Integration with error handling: What happens when an actor fails?
- Here: foundations

Actors: Communication & Concurrency

Actors

- Recipients of messages are identified by name (no channels).
- An actor can only communicate with actors that it knows.
- An actor can obtain names from messages that it receives, or because it has created the actor

The actor model is characterized by

- inherent concurrency among actors
- dynamic creation of actors,
- inclusion of actor names in messages, and
- interaction only through direct asynchronous message passing with no restriction on message arrival order.
- *message servers* might be implemented by matching messages from the queue to procedures

Example: Erlang-style Actors - Matching Messages
Publish and Subscribe Server

```
runServer(Subs) ->
  receive
    {sub,from} -> runServer(Subs + from); % subscribe
    {publish,value} ->   % publish
                for(id in Subs) id!{value}, % broadcast value
                runServer(Subs);
    _ -> runServer(Subs); % ignore other messages

Server { % publish and subscribe server
  start() -> spawn(fun() -> runServer([])).} % start the server

Client { % send requests to the server
  start() -> Server!{sub,self}, Server!{publish,10}.}
```

Example: Erlang-style Actors

- State as argument to recursive calls
- We can dynamically change the message server
- An actor can match different messages in different states
- ... but tricky to detect errors in message servers

```
runServer1(Subs) -> receive % subscribe when there is space
    {sub,from} -> if(size(Subs) >= 9) runServer2(Subs + from)
                  else runServer1(Subs + from);
    {unsub,from} -> runServer1(Subs - from);
    ...
```

Example: Erlang-style Actors – Handling Return Values between Actors

```
id1 = spawn(fun() -> func1([])); id2 = spawn(fun() -> func2([]))
id1!{step1, 42, id2};
...
func1(history) -> receive
    {step1, data, other} -> newData = doSomethingFirst(data),
                            other!{step2, newData, self},
                            func1(insert(history,data));
    {step3, data, other} -> newData = doSomethingThird(data),
                            other!{step4, newData, self},
                            func1(insert(history,data));

func2(history) -> receive
    {step2, data, other} -> newData = doSomethingSecond(data),
                            other!{step3, newData, self},
                            func2(insert(history,data)); ...
```

Futures

Futures – Handling Return Values between Actors

Welcome to “callback hell”!

- Problem: Logically related code is scattered in program
- We need a way to identify callback messages
- We also need a way to wait for a result
- Solution: futures, special mailboxes transmit return values

Reminder in Java:

Java

```
ExecutorService service = Executors.newFixedThreadPool(2);
Future<Int> f = service.submit(() -> { /* do */ return 1;});
...
Int x = f.get(); //essentially a join
```

Futures and Promises

Futures

- It is a handle for the caller of a process. It will contain the result value once computed
- It can be read multiple times
- It can be used by the caller to synchronize with the callee

Java

```java
Future<Int> f = service.submit(() -> { return 1;});
...
Int x = f.get();
```

Promises

- What if the value will be computed somewhere else?
- A *promise* is a future for which it is not clear who computes the value

Promises

A promise:

- May be eventually completed (but maybe by somebody else)
- Must be completed (written) only once
- Deadlock/starvation occurs if it is never completed
- It can be seen as a handle for the callee and the callee does not synchronise with the caller

Java calls promises *CompletableFutures*:

Java

```
CompletableFuture<Integer> f = new CompletableFuture<>();
service.submit(() -> { f.complete(1); return null;});
...
Int x = f.get();
```

Promises – Example: Service Delegation

Java

```
/* the function casts a promise as a future */
/* from outside the future can only be retrieved */
Future<Integer> callAsync() ... {
 CompletableFuture<Integer> completableFuture = new CompletableFuture<>();
 service1.submit(() -> {
   if(/* service1 cannot process, then it delegates to service2 */ )
     service2.submit(() ->
      { /* compute */ completableFuture.complete(1); return null })
   else { /* process the request */
     /* compute */ completableFuture.complete(1); }
   return null;
 });
 return completableFuture;}
```

Composition Futures/Promises

Logically related Futures/Promises scattered in the code.

Java

```
CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(() -> 1);
  ...
CompletableFuture<Integer> f2
  = CompletableFuture.supplyAsync(() -> f1.get + 1);
```

Connecting Futures/Promises (composition)

Java

```
CompletableFuture<Integer> f
  = CompletableFuture.supplyAsync(() -> 1)
                          .thenApply((res) -> res + 1);
```

Very similar patterns are common in web development with JavaScript

Interpreting Futures/Promises as Channels

Channel-view on single-read futures

- Create channel and send it via an asynchronous message
- For the caller, the channel behaves as a future:
  caller waits on the channel for a return (caller side does not write on the channel).
- For the callee, the channel behaves as a promise:
  it can be passed around, and eventually someone will write on it *exactly once*
  (callee side does not read on the channel)

Limits of this view

- Futures may be read more than once
- "immediately creating and sharing a channel" may be more complex and its implementation is delegated to the programmer

Active Objects

Motivation

- How to combine monitors and actors?
- How to make signalling less error-prone?
- How to make conditions/invariants easier to use?
- How to connect futures/promises with actors?

Active Objects

An active object[a] is an actor with an *implicit* message server, that only communicates asynchronously, but allows internal message handlers to use *cooperative scheduling*.

- One process/thread per object
- Messages identified with methods
- Implicit queue of tasks (procedures in the methods)
- Explicit synchronization

[a] ABS is a modelling language to run simulations of distributed systems.
The simulation tool is maintained by the PSY group at IFI: `https://abs-models.org/`

Cooperative concurrency

Active Monitors as Active Objects

- Cooperative concurrency:
  constructs to suspend and resume execution (=task) of a local method
- External cooperation (operations on futures)
  - Send is asynchronous: Fut $\langle$ T $\rangle$ f = o!m(...); ... ;
  - Retrieve value is blocking: x = f.**get**;
  - Check for value is suspending: **await** f?
- Interaction patterns between methods
  - Fut $\langle$ T $\rangle$ f = o!m(...);x = f.**get**;
  - Fut $\langle$ T $\rangle$ f = o!m(...); ...; x = f.**get**;
  - Fut $\langle$ T $\rangle$ f = o!m(...); ...; **await** f?; x = f.**get**;

Cooperative Scheduling – Example: The Diner

- Each object runs one thread and each method call spawns a *task*
- Thread is responsible to schedule tasks in some order
- Waiting on future suspends the task, not the thread!
- Reading on future potentially blocks task and thread – no other task can run

ABS

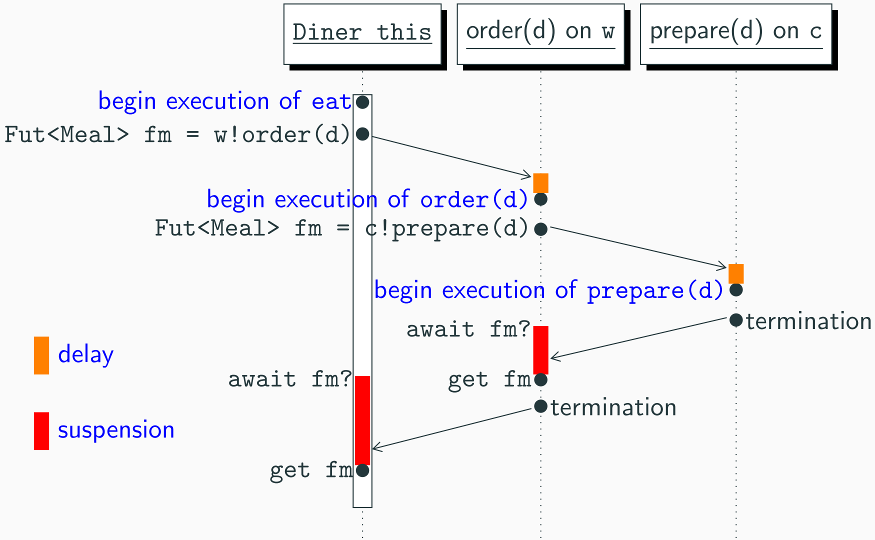
```
class Diner(IWaiter w) implements IDiner {
    Unit eat(Dish d) {
        Fut<Meal> fm = w!order(d); // place order with waiter
        await fm?; // while waiting do something else, e.g, take a phone call
        Meal m = fm.get; // receive meal
        Fut<Unit> fc = this!consume(m);
        Fut<Unit> fp = w!pay(this, d); // eating, paying in some order
        await fc? & fp?; // eaten and paid — ready to leave!
    }
    Unit takeCall(){ ... }
    Int takeMoney(Int a) { ...}
    ...
}
```

Example (Continuation): – The Waiter

ABS

```
class Waiter(ICook c, Int purse) implements IWaiter {
    Meal order(Dish d) {
        Fut<Meal> fm = c!prepare(d); // place order with cook
        await fm?; // waiter serves other guests while meal is cooked
        Meal m = fm.get; // receive meal reuse names for local variables
        return m; // ready to serve the meal!
    }
    Unit pay(IDiner g, Dish d) {
        Int amount = price(d); // lookup price in the menu
        Int a = g.takeMoney(amount); // synchronous (blocking) call, no wait
        this.purse = this.purse + amount; // no data race possible
    }
}
```

Example (Continuation) – The Restaurant Experience



Condition Synchronization

- Condition variables can be derived from monitor invariant
- Or can be bound to some other condition
- Error-prone implementations
- Active Object approach: condition synchronization as primitive

ABS

```
class C () {
  int i = 0;
  Unit inc() { i = i+1; return; }
  Int isGreaterThanTen(){ await i > 10; return i; }
}
```

- Condition variables: explicit suspension instead of busy waiting
- Every time the object is idle, the object thread evaluates all conditions of suspended tasks, otherwise it waits for new messages to arrive

Objects as Passive Monitors (reminder) – Example

Java

```java
class Mon {
  int r = 0;
  Condition res;

  public synchronized void Acquire() {
    while( r == 0 ) { res.await(); };
      r = r - 1;
    }

  public synchronized void Release() {
    r = r + 1;
    res.signal();
   }
 }
```

Monitors with active objects – Example

ABS

```
class Mon {
  int r = 0

  Unit Acquire() {
    await (r!=0);
    r = r − 1;
  }

  Unit Release() {
    r = r+1;
  }
}
```

- With cooperative concurrency, we can avoid error-prone signaling in the monitor.
- The active object only has one queue, but reactivation of Acquire methods can only happen when the await-condition holds

Bounded Buffer Synchronization with Active Objects(1)

Let us now solve the bounded buffer problem with active objects

Bounded buffer synchronization

- buffer of size n (“channel”, “pipe”)
- producer: performs `put` operations on the buffer.
- consumer: performs `getVal` operations on the buffer.
- two access operations (“methods”)
    - `put` operations must wait if buffer full
    - `getVal` operations must wait if buffer empty

Bounded Buffer Synchronization with Active Objects (2)

ABS

```
class Bounded Buffer (Int n) {
    List<T> buf = [];
    Unit put(T data){
        await (length(buf) < n);
        buf = appendright(buf,data);
    }
    T getVal() {
        await (length(buf) > 0);
        T tmp = head(buf); buf = tail(buf); return tmp;
    }
}
```

Deadlocks

What is a deadlock?

A system is deadlocked if it is *stuck*: It cannot continue execution, and it has not finished its execution.

A system is deadlocked if there is a circular dependency: There is a sequence of components $C_1, \ldots, C_n$, such that C_i depends on C_{i+1} before it can continue and C_n depends on C_1.

- Actors without futures/channels cannot deadlock – they can always continue execution
  ...but there can be messages that cannot be processed with the current message server!
- In some concurrency models, a system can only get stuck because of a circular dependency

Local Dependencies – Between the Object and its Tasks

ABS

```
class C (){

        Unit m(){
                Fut<T> f = this!n();
                f.get; // deadlock
        }

        T n(){ /* do some computation */ return value; }

}
```

Dependencies due to Synchronization Between Tasks

- A task depends on another task if it waits for its future

ABS

```
class C {
        Fut<Unit> f1;
        Unit store(Fut<Unit> fut) { f1 = fut; }
        Unit m(){ await f1?; return; }} //depends on d.n

class D(C c) {
        Unit n(){ Fut<Unit> f2= c!m();
                  await f2?; //depends on c.m
                  return; } }
{ // Main block
        C c = new C(); D d = new D(c);
        Fut<Unit> f;
        await c!store(f); f= d!n(); // deadlock
}
```

Dependencies Related to the State of an Object

- In a given state a task t_1, that might be stuck on condition e_1, depends on another task t_2, that might be stuck on condition e_2.
- Here e_1 and e_2 are conditions related to the state of an object, which create dependencies between the tasks.

ABS

```
class D { // here exclamation mark is negation
  Bool b1 = false; Bool b2 = false;
  Unit m(){ b1 = true; await b2; b1 = !b2;}
  Unit n(){ await !b1; b2 = !b1;}
}
```

- What happens if we call n() and then m() on a D-object?
- There is no procedure to decide whether an arbitrary program *ever* deadlocks because it depends on the scheduling of tasks

Outlook: Analysis and Modelling

Reasoning

Monitors and actors are well-suited for manual and automatic reasoning

- Builtin mutex ensures that between interaction points, code can be seen as sequential
- Sequential reasoning has to be extended only at these points
- Full concurrency requires non-local reasoning at every point

Programming is Modelling

A program can be used to model a part of the world.

- A program analysis then can be used to derive properties over the world
- For example, 5 philosophers programs are *executable* models
- Allows analysis for deadlock freedom.

Async/Await

Recap on Message Passing

Message Passing So Far

- Channels: Asynchronous shared entities
- Actors: Monitors that send asynchronous messages
- Active Objects: Monitors with their own thread that send asynchronous messages

Java and Async/Await

Reminder in Java:

Java

```
ExecutorService service = Executors.newFixedThreadPool(2);
Future<Int> f = service.submit(() -> { /* do */ return 1;});
...
Int x = f.get(); //essentially a join
```

- Executed function disconnected from classes
- Much boilerplate code, especially when call-backs are involved
- Asynchronous code (library) does not mirror synchronous code (language constructs)

C# and Async/Await

C#'s Asynchronous Concurrency

- Better abstraction to handle Futures/Tasks.
- Concurrency as first-class construct of language

- Methods annotated with `async` can only be called asynchronously
- Methods annotated with `async` return a `Task`
- Only methods annotated with `async` can perform an `await`
- Expression `await` suspends the thread until the task has finished.

C# and Async/Await – Example: Comp. Two Numbers

- Example: Reading two numbers from user and performing some long-lasting computation
- Synchronous version
- Await version: Note that `Method` must be `async` to use `await`
- Asynchronous version: Now both reads can be concurrent

C#

```
class C{
  void async Method() {
    Task<Int> t1 = GetFirstNumber(); Task<Int> t2 = GetSecondNumber();
    Int i1 = await t1; Int i2 = await t2;
    Int res = await Compute(i1,i2);
  }
  Task<Int> async GetFirstNumber() {...}
   ...
}
```

Pros and Cons of Async/Await

What Color is your Function

- Only `async` methods can access results of `async` methods
- Separates whole program into two sets of methods
  that can only interact at specific points
- Sometimes called colored-function problem, after a popular blog entry[a]

[a] https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

- Forces programmer to think about concurrency
- Can still use threads and tasks directly to circumvent all this

Wrap-Up

Today's Lecture

- Actors – Monitors with message passing
- Futures/Promises – Handling asynchronous results
- Active Objects – Actors with cooperative concurrency and futures
- Async/Await – Language-integrated asynchronicity with threads and futures

Next Lectures

- Next Block: How to type channels?

Note: ABS example courtesy of Reiner Hähnle