Aalto University
School of Science

Concurrency frameworks

CS-E4110 Concurrent Programming
Autumn 2021 (II), week 4

Vesa Hirvisalo

Week 4 in a nutshell

- Computing in a large scale
  - Scalability, resilience, efficiency
  - Distributed systems, datacenters, etc.
- Introduction to concurrency frameworks
  - Working on a higher abstraction level
    - Note: usually does not exclude using the lower levels!
- Akka actors
  - A message passing system for Scala
    - The system supports actor-local shared memory computing
  - With (some) fault tolerance and natural distribution

Computing in large scale

Distributed systems

- In larger scale we need
  - Scalability, resilience, efficiency, etc.
    => Several computers / computing nodes
    Highly concurrent distributed systems
- Network environments and distributed environments
  - Network: the individual computers a visible to the user
  - Distributed: viewed as a single environment by the user
- Distributed computing is challenging
  - There is no global state (by definition)
    - The system state is update by exchanging message
  - Without a global state, understanding of correctness is hard (we will omit a detailed view into these problems)

Distribution and synchronization

- Synchronization is about the (correct) execution order(s)
  - We allow only those with the correct result
  - The execution order is related to time
- What is the time now?
  - The answer comes always with a delay
  - Distributing a clock is difficult
  - And similarly, being synchronous is difficult
- The correct order of events?
  - Assuming no global time (in a system)
  - The system cannot tell (with means of the system itself)
  - Only partial orders are possible

Datacenters as computers

- Efficient and scalable computing
  - Need plenty of resources => make computing a utility
  - Energy consumption => highly flexible and concurrent code
- Typical datacenters are warehouse size computers
  - Built around a set of networks
- Usually, there is
  - A set of networks
  - A hierarchy of memory and storage
  - The processing units
  - In addition
    - Plenty issues on power, cooling, etc.
    - It really is a house!

Introduction to frameworks

Shared memory frameworks

- Traditionally uses locks (etc.) to synchronize and protect data structures
  - If a thread crashes while holding a lock, other threads will block on the lock
  - Concurrency is limited to a single machine shared memory
- A number of frameworks for parallelization
  - More abstract than raw threads, e.g., OpenMP
- Distribution, fault-tolerance, etc.?
  - No single (or direct) way to do such
  - E.g., partially solving fault-tolerance: use short lock regions that are robust against exceptions, release held locks in exception handlers

Message passing frameworks

- Have an abstract and integrated approach
  - Instead of simply having only some send/receive methods
  - E.g., MPI used for high performance computing
- Message passing frameworks are a natural choice for distributed computing
  - Distributed systems are often defined as systems that base their communication on messages (shared channels instead of shared memory)
- However, some things are not easy
  - Especially, understanding system state
    - E.g., detecting when an algorithm has terminated

Functional programming frameworks

- Several aspects can be implemented by a framework
  - Dataflow, synchronization, fault tolerance
    - E.g., recomputing a function on unmodified input data will yield the same result
  - Natural way to parallelization and load balancing
  - Some frameworks: MapReduce, Apache Spark, Scala parallel collections
- However, functions are by default computed in isolation
  - Communication heavy algorithms such as many iterative algorithms can become slow
  - Lack of data mutations can make some algorithms slower

Actor frameworks

- Actor frameworks are a mix between shared memory and message passing concurrency
  - Each actor has its own mutable state
  - Actors communicate by sending immutable messages
  - The actor frameworks give ways to do implement application specific load balancing and fault tolerance strategies with help from the framework
- Actors are made as lightweight as possible
  - Much more lightweight than threads
  - The message send/receive is location transparent
  - The same messaging primitives are used for both local messages and messages over the network

Reactive manifesto

- The Manifesto (`www.reactivemanifesto.org`) says that reactive systems are:
  - Responsive: Systems respond in a timely manner
  - Resilient: The system stays responsive in the face of failure
  - Elastic: The system stays responsive under varying workload
  - Message driven: Systems rely on asynchronous message passing: loose coupling, isolation, location transparency, delegates errors as messages

Akka actors (in Scala)

The Akka Actor framework

- Design heavily influence by telecommunications software design style - Erlang
  - Used to implement reactive server software that responds to client requests
- Very light weight actors
  - Usually one actor used for each incoming request / connection
  - The Akka documentation is available from `akka.io`: `https://doc.akka.io/docs/akka/current/index-classic.html`
  - Note that there are several versions of Akka
- Actor framework for Scala and Java
  - **Actors are not objects**
  - Actors have no public methods or data
  - The only way using them is by sending messages

An example

Defining a simple Actor (Classic Akka)

```
1  import akka.actor.Actor
2  import akka.actor.Props
3  import akka.event.Logging
4
5  class MyActor extends Actor {
6    val log = Logging(context.system, this)
7
8    def receive = {
9      case "test" => log.info("received␣test")
10     case _ => log.info("received␣unknown␣message")
11   }
12 }
```

- Note that `this` and `self` are different
  - `self` is used to refer to the `ActorRef` of the `Actor`
  - A received message has a `sender` actor reference attached

Akka Futures*

- In futures are a generic concurrency primitive
  - They are for waiting thing that will eventually become available
- In Akka: asynchronous waiting for message reply
- A timeout is usually used to limit the wait
  - Otherwise you should be sure
  - `Await.result` obtains the result of a future
  - `Await.ready` can be used to check is the result ready
- Notice that blocking will prohibit other actors from running on the same thread
  - I.e., blocking is not recommended for performance reasons

*) *Next week, we will discuss futures in a more generic way*

Sending messages

- `tell` sends a message asynchronously and returns immediately (basically: `targetActorRef ! Message`)
- `ask` is a way to implement sending a message and generating a `Future` representing a possible reply
  - *E.g.*: val future = myActor ? "hello"
  - A timeout for the reply can be specified with the ask method
  - `val f = myActor.ask("msg")(5 seconds)`
- At the receiver end for each received message, the `receive` is invoked

Processing messages

- Actors are event driven: Activated at the receive time
  - Multiple messages are queued per actor (FIFO Mailbox)
  - Actors should be using non-blocking I/O. If blocking I/O is needed, special care must be taken not to block other actors from proceeding during the I/O wait
  - Actors themselves are single threaded by design - If concurrency is needed, deploy several actors
- Fault-tolerance through the “Let it crash”-philosophy
  - Isolate the failure and continue with the non-faulty parts, restarting failed subsystems when needed
- Actors operate on local state
  - not to be exposed ouside, no internal locking, etc.

Children and supervision

- An Actor can create children
  - The spawned actors form a hierarchy, where each actor is supervising its child actors
  - If errors happen in a child actor, its parent is notified, and it can take supervision action (e.g., restarting the child actor)
- Fault-tolerance is implemented by supervision strategies
  - Each actor has a strategy for supervising its children
  - If different strategies are needed for different children, one needs to create an additional layer of actors to do the different supervision strategies for different children
- As a whole, a system is defined
  - This includes multiple aspects (e.g., scheduling)

Actor Systems

An Example of creating a system (Classic Akka)

```
1  // Modified from "Scala Cookbook":
2  import akka.actor.Actor
3  import akka.actor.ActorSystem
4  import akka.actor.Props
5  class HelloActor extends Actor { // Code removed
6  }
7
8  object Main extends App {
9      // an actor needs an ActorSystem
10     val system = ActorSystem("HelloSystem")
11     // create and start the actor
12     val helloActor = system.actorOf(Props[HelloActor], name = "helloactor")
13     // shut down the system
14     system.shutdown
15 }
```

Supervision strategies

- Main supervision strategies for different errors (e.g., thrown exceptions) are:
  - Resume child, keeping accumulated internal state
  - Restart child, clearing accumulated internal state
  - Stop the child permanently
  - Escalate the failure, failing the actor itself
- Note the hierarchical nature of actors
  - E.g., stopping an actor will all stop all its children
- Some notes
  - Supervision messages have their own system mailboxes and are not ordered with the normal messages
  - Messages `Kill`, `stop` and `PoisonPill` do differ

Location transparency

- Actor references
  - Can refer to actors in remote computers
  - Each actor has an actor path consisting of the hierarchical naming of actor instances
- Everything is distributed by default
  - The distribution is transparent to the programmer
  - All message passing is asynchronous
  - Because messages pass over real network connections, the possibility of losing messages is much higher than within a single machine
  - Sometimes message passing can be optimized by the framework if the sender and receiver are on the same computer

Akka and Java Memory Model

- The happens-before guarantees made by Akka:
  - The send of a message by an actor happens-before the receive of that message by the same actor
  - Processing of one message happens-before processing of the next message by the same actor
  - Thus there is no need to mark internal state volatile
  - However, the messages sent to other actors need to be immutable and properly constructed to not cause problems in data transfer between actors
- Properly constructed immutable data is safe to share between threads

Akka Message Delivery

- Akka takes care of the following:
  - At-most-once delivery
    - Never deliver twice but messages **can get lost!**
    - Applications have to take care of the lost ones
    - There are libraries to help implementing at-least-once delivery
  - Best effort message delivery (i.e., go for low overhead)
  - Message ordering by send-receive pair
- Akka persistence
  - A collection of libraries allowing actors to persist internal state in an external database to allow for better fault tolerance
  - There are also many other tools available

Related readings and exercises

=> Check A+