Aalto University
School of Science

Concurrency Frameworks, Actors, Akka, Fault Tolerance

T-106.5600 Concurrent Programming

Keijo Heljanko

Department of Computer Science
Aalto University School of Science

November 9th, 2016

Slides by Keijo Heljanko

Concurrency Frameworks - Shared Memory

Shared Memory Concurrency

- Very high performance when done right
- Traditionally uses locks to protect data structures
  - If a thread crashes while holding a lock, other threads will block on the lock
  - Partial fixes: Use short lock regions that are robust against exceptions, release held locks in exception handlers
- Use libraries with lock-free/wait-free data structures that do not block after one of the threads has crashed
- Shared memory is not location transparent - concurrency is limited to a single machine shared memory
- Lots of freedom for programmer - No single way to implement fault tolerance, automatic parallelization

Concurrency Frameworks - Message Passing

Message Passing Concurrency

- Instead of directly accessing memory, communicate through sending messages
- Allows easy way to implement location transparency - The messages can be easily sent/received to/from another machine over the network
- These frameworks are used for distributed computing
- Fault tolerance can be tricky to implement - Often done through periodic snapshots
- Some easy things in shared memory concurrency become very tricky: Creating a consistent snapshot of the system state, detecting when algorithm has terminated
- Example: MPI used for high performance computing

Concurrency Frameworks - Functional Programming

Functional Programming Concurrency

- Fault tolerance can be implemented by the programming framework - Recomputing a function on input data will yield the same result
- Allows for automatic parallelization and load balancing - If there are a large number of functions to be computed all of them can be done in parallel
- Computing is done in isolation - Each function does not communicate with other functions / Communication heavy algorithms such as many iterative algorithms can become slow. Also lack of data mutations can make some algorithms slower
- The framework can handle all concurrency related issues - dataflow, synchronization, fault tolerance
- Some frameworks: MapReduce, Apache Spark, Scala parallel collections

Concurrency Frameworks - Actors

Actors Concurrency

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

Reactive Manifesto

The Reactive Manifesto gives the philosophy behind actor frameworks

- The Reactive Manifesto `http://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 Actor Framework

- An Actor framework for Scala and Java
- 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
- Akka documentation states one gigabyte of memory can support up to 2.7 million actors
- Internally each JVM thread will support a large number of actors
- Fault tolerance strategies supported by the framework

Actors

Actors are Not Objects

- Actors have no public methods
- Actors have no publically accessible data
- There is no synchronous communication with actors
- The only way to talk to them is by sending messages
- The only way to access their data is by sending them messages

Defining an Actor in Akka
Defining an Actor

```
1  import akka.actor.UntypedActor;
2  import akka.event.Logging;
3  import akka.event.LoggingAdapter;
4
5  public class MyUntypedActor extends UntypedActor {
6      LoggingAdapter log = Logging.getLogger(getContext().system(), this);
7
8      public void onReceive(Object message) throws Exception {
9          if (message instanceof String) {
10             log.info("Received␣String␣message:␣{}", message);
11             getSender().tell(message, getSelf());
12         } else
13             unhandled(message);
14     }
15 }
```

Sending Messages in Akka

There are different ways to send messages:

- `tell` is a fire-and-forget way of sending messages, it sends a message asynchronously and returns immediately, the second argument is the sender of the message: `target.tell(message, getSelf())`
- `ask` is a way to implement sending a message and generating a `Future` representing a possible reply. A timeout for the reply can be specified with the ask method, see Akka documentation for further information
- `onReceive` method is called at the receiver end for each received message

Akka Futures

Akka Futures:

- Used to implement asynronous wait for message reply
- A timeout is usually used to bound this wait
- The `Await.result` can be used to obtain the result of a future, and `Await.ready` can be used to check would is the result ready, and the `Await.result` would block
- Notice that blocking will prohibit other actors from running on the same thread, and thus blocking is not recommended for performance reasons

Message Processing by Actors

- Actors communicate with each other through asynchronous immutable messages
- Asynchronous means that the processing of a message might happen later than its send, the message is queued in the meantime
- 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 are event driven: Activated at the message receive time
- Handling failures is done through the “Let it crash”-philosophy: Isolate the failure and continue with the non-faulty parts, restarting failed subsystems when needed

Message Processing by Actors (cnt.)

- Actors themselves are single threaded by design - If concurrency is needed, deploy several actors
- Different invocations of an actor can happen in different threads at different times
- An actor can also dynamically create other actors and change its own behaviour
- Because actors operate on local state that is not supposed to be exposed to the outside world, no locking of the actors internal state is needed
- The incoming messages of an actor are stored in an unbounded queue (FIFO by default) called the Mailbox
- If an actor fails, the messages in its Mailbox will be lost

Akka - Getting Started

Akka Documentation

- The Akka documentation is available from:
  `http://doc.akka.io/docs/akka/2.4.0/java.html`
- See e.g., the following page for the different install options:
  `http://doc.akka.io/docs/akka/2.4.0/intro/getting-started.html`
- Note that latest versions of Akka require Java 8 (or newer)

Creating an Actor System
Creating an Actor System

```
1  import akka.actor.ActorRef;
2  import akka.actor.ActorSystem;
3
4  // ActorSystem is a heavy object: create only one per application
5  final ActorSystem system = ActorSystem.create("MySystem");
6  final ActorRef myActor = system.actorOf(Props.create(MyUntypedActor.class),
7                                                    "myactor");
8
9  // The following will create a child actor
10
11 class A extends UntypedActor {
12     final ActorRef child =
13         getContext().actorOf(Props.create(MyUntypedActor.class), "myChild");
14     // plus some behavior ...
15 }
```

Akka - Creating Children

Creating Child Actors

- An Actor can create children, potentially processing subtasks on its behalf
- 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)

Akka - Fault Tolerance

Akka fault tolerance - Supervision strategies

- The main Akka fault tolerance notion is based on the idea of 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

Supervision strategies

Supervision strategies

- Main supervision strategies for different errors (e.g., thrown exceptions) are:
  - Resume child, keeping accumulated internal state
  - Restart child, clearing accumulated internal stateo
  - Stop the child permanently
  - Escalate the failure, failing the actor itself
- Note that because of the hierarchical nature of actors, e.g., stopping an actor will all stop all its Children in a recursive manner
- Note: Supervision messages have their own system mailboxes and are not ordered with the normal messages!

Restarting Actors

Reasons to restart Actors

- Systematic error (exception) on processing a message
- Transient failure of an external resource
- Corrupt internal state of an actor - Has to be suspected if the first two can not be identified

Actor References

Actor References

- `ActorRef`, the reference to an actor, gives the possibility to send it messages
- Each actor has its own reference available through the `self` field
- This reference is also by default used as the message sender reference by default
- Each received message has a `sender` actor reference attached to it
- Note that actor references can sometimes also refer to actors in remote computers

Obtaining Actor References

Obtaining Actor References

- One can create actors using an `ActorSystem` to obtain a reference to a new actor
- Each actor has an actor path consisting of the hierarchical naming of actor instances, and references can be looked up by the hierarchical name of the actor

Location Transparency

- Everything is distributed by default
- 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. This is transparent to the programmer

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 always safe to share between threads

Akka Message Delivery Reliability

Akka promises the following:

- At-most-once delivery: A message is never delivered once (but it might get lost) - Best effort message delivery, least possible overhead!
- Message ordering by send-receive pair
- You have to design applications to expect the potential of message loss
- This is a design decision that has implications to the application developer
- There are libraries to help implementing at-least-once delivery on top of the basic Akka messaging
- Akka persistence is a collection of libraries allowing actors to persist internal state in an external database to allow for better fault tolerance

Akka Cluster

Akka cluster gives additional functionality:

- Fault-tolerant decentralized peer-to-peer based cluster membership service
- No single point of failure
- Uses peer-to-peer gossip protocols to implement a failure detector
- Can be used to do fault tolerant scalable systems
- Fairly new, not used for homework 2!

Akka.io

Akka.io is a library for doing Web services

- High scalability to large number of concurrent connections
- Low latency
- Optional throttling functionality

Akka.io Streams

Akka.io Streams:

- A way to implement stream processing in Akka
- Deploys back pressure algorithms to avoid overflowing message buffers when fast producers are run with slow consumers
- Still experimental
- Documentation:
  `http://doc.akka.io/docs/akka-stream-and-http-experimental/current/java.html`

Akka Actors

Akka Actors give us:

- High performance reactive programming framework
- Designed for distributed processing
- Based on the Actors philosophy - See Reactive manifesto
  - Location transparent
  - No shared state - lockless
  - Fault is the norm - functionality to do hierarchical actor supervision: resume, restart, stop, escalate
- Based on at most once (lossy) message delivery but also has e.g., persistence features allowing for more reliability
- Lots of additional features to do Web services
- See tutorials and home assignment 2 for more info