Lecture 21
akka Java in Detail
Recall akka

• Open-source implementation of actors model
  – Originally developed for Scala language
  – Ported also to Java

• Key concepts
  – ActorSystem
  – UntypedActor
  – onReceive()
  – tell()
  – Patterns.ask()
Dynamic Actor Creation

• In Java we saw that tasks can create other tasks
• In akka Java, actors can also create other actors!
  
  – Actor creation so far has been done using calls to actorOf() method of ActorSystem object
  
  – It may also be done by calling actorOf() method of ActorContext object

• An ActorContext object is the environment surrounding an actor
• To get the ActorContext of an UntypedActor actor, call getContext() instance method
Supervision

• Every actor has exactly one supervising actor
  – When one actor creates another using first actor’s context, first actor is supervisor of second
    • First actor often also called parent
    • Second usually called child or subordinate
  – What about actors created via ActorSystem actorOf()?
    • Every actor system three top-level actors (called guardians) that are started automatically
      – / The root guardian
      – /system The System guardian (child of /)
      – /user The Guardian Actor (child of /)
    • When an object is created using actorOf() in ActorSystem, it is by default made a child of /user

• What supervisors do
  – Delegate tasks to children
  – Take remedial action when children fail

• Supervision is basis of fault tolerance in akka
Getting Supervisory Information

• **ActorContext** has methods for retrieving parent, child information
  - `ActorRef parent()`  
    Return parent of actor associated with context
  - `java.lang.Iterable<ActorRef> getChildren()`  
    Return children as a Java Iterable
  - `ActorRef getChild(String name)`  
    Return child having given name, or null if there is no such child

• To find parent of given actor, invoke following in body of actor definition:  
  `getContext().parent()`
Supervisory Hierarchy

- Supervision relationship induces a tree
  - Every actor (except /) has exactly one parent
  - Every actor has $\geq 0$ children
- Every actor can be identified via path \( \text{ActorPath} \) in tree
- To get path of \texttt{ActorRef}, use \texttt{path()} instance method
- For actorA
  - Parent: user
  - Children: actorB1, actorB2
  - Path: /user/actorA
How an Actor Can Find Its Name

- `getName()`? `name()`? No
  No such instance methods in `UntypedActor`
- `getSelf().getName()`? `getSelf().name()`? No
  No such instance methods in `ActorRef`
- `getContext().getName()`?
  `getContext().name()`? No
  No such instance methods in `ActorContext`
- **Solution:** go through `ActorPath`
  - `ActorPath` objects have `name()` method returning name (String) of actor at that path
  - So, `getSelf().path().name()` returns name of yourself
Supervision in Detail

• When an actor fails (i.e. throws an exception) a special system message is sent to its parent
  – Systems messages have their own message queue; they are not handled by `onReceive()`
  – No guarantees about precedence of system messages over regular messages

• Parent actor has four choices in akka
  1. _Resume_ the failed child in child’s accumulated internal state
  2. _Restart_ the failed child in its initial state
  3. _Stop_ the failed child permanently
  4. _Escalate_ (i.e. fail itself, handing off responsibility to its own parent)

• Communication associated with these choices is via system messages that are handled by special system-message queue
  This queue is only used for supervision (i.e. parent-child) communication
Resumption of Failed Child

• `onReceive()` method in child is re-invoked
  – Message being processed when failure occurred is lost
  – Processing of messages in child’s message queue resumes

• When to do this?
  – Maybe if transient system fault caused failure
  – Maybe if there is a bug in child that doesn’t affect its ability to process future messages
Restarting a Failed Child

- **Idea**
  - Create new actor instance
  - Replace actor instance in ActorRef for failed child with new instance
    - Path unchanged
    - So is name
  - Invoke onReceive() method of new actor instance to start processing messages in message queue
- Message processed during failure is lost, but no pending messages in failed child’s mailbox are
Stopping an Actor

- Stopping a child during supervision involves a general actor-stopping technique.
- `ActorContext` objects include following method:
  ```java
  void stop(ActorRef actor)
  ```
  - Stops actor
  - Processing of current message completes first, however
- What about messages in mailbox when actor is stopped? And those sent to stopped actor?
  - These are called *dead letters*
  - akka uses a special actor (`/deadLetters`) to handle these
  - There are also mechanisms for retrieving them
- What about children?
  - They are stopped also,
  - This percolates downwards through supervision hierarchy, to children’s children, children’s children’s children, etc.
Actors Can Stop Other Actors ...

- ... even themselves!
- If following is executed in UntypedActor ...
  `get Context().stop(getSelf())`
- ... then it stops itself! (And consequently its children, grandchildren, etc.)
  - When an actor is stopped, its supervisor is notified
  - So are other actors that are monitoring this actor
  - akka buzzwords for this: DeathWatch, DeathPact
    - Special `Terminated` messages (these are not system messages, so are delivered to regular mailboxes) are sent to actors that have registered with stopped actor
    - Registration is done via `watch()` method in `ActorContext`
    - De-registration: `unwatch()` method in same class
Failure Escalation

• As name suggests, escalation in response to child failure means that parent fails by throwing same exception as child
• Parent’s parent then must handle failure
Details of Supervision

• Each UntypedActor object contains a SupervisorStrategy object
  – To obtain SupervisorStrategy object, execute actor’s supervisorStrategy() instance method
  – This method may be overridden in order to customize supervision approach

• The SupervisorStrategy determines how failures of children will be handled
Two Kinds of SupervisorStrategy

• **AllForOneStrategy** *(subclass of SupervisorStrategy)*
  – If one child fails, apply supervision strategy to all of the children, not just the failing one
  – Used if children are tightly coupled

• **OneForOneStrategy** *(also subclass of SupervisorStrategy)*
  – Apply supervision strategy only to failing child; other children left unaffected
  – Used if children are largely independent
Deciders

• **Core of a** SupervisionStrategy: *decider*
  – A decider maps exception classes to directives, which describe which of four mechanisms to use to recover
  – A directive has one of four forms: Escalate, Restart, Resume, Stop

• You may customize a **SupervisionStrategy** by changing the decider

• There is also a default decider
akka and the Java Memory Model

• Actors do not (intentionally) share memory
• In a local application (single JVM), one still needs to worry about visibility
• akka guarantees the following
  – If one actor sends a message to another, then pending writes before the send are guaranteed to be visible after the receipt
  – Pending writes after an actor reads a message are visible when the actor reads the next message