Lecture 20
The Actor Framework
Recall

• Concurrency
  Several operations may be in progress at the same time

• Parallelism
  Several operations may be executing simultaneously

• “Distributed-ness”
  Several machines may be working at the same time for the same application
So Far We Have Concentrated On:

• Concurrency in Java
  – Threads
  – Locks
  – Etc.

• Parallelism in Java
  – Performance tuning
  – Fork/Join
  – Etc.

• Focus has been on threaded applications running inside a single process (= single instance of JVM)
Recall Threads vs. Process

• Threads
  – Independent control flows, stacks
  – Shared heap

• Processes
  – Independent flows, stacks
  – Independent heaps
Distributed Computing

• Distributed systems have multiple processes
  – No shared memory
  – So, no data races!
  – But, need explicit IPC (Inter-Process Communication) mechanisms

• In case of distributed computing, network communication is typically used
Some Distributed System Terminology

- **Host**
  Computer running in a distributed environment

- **Port**
  Communication channel used by hosts to exchange messages

- **Network**
  System consisting of hosts, equipment used to connect hosts

- **IP address**
  Internet Protocol address: number assigned to a host connected to the internet so that other hosts may communicate with it

- **MAC address**
  Media Access Control address: number assigned to a host on a local-area network (LAN) so that other hosts on LAN may communicate with it.
The Actors Model

• A system model supporting a multi-process programming paradigm
  – Model assumes no shared memory
  – No assumptions about distributed / non-distributed

• Systems consist of multiple *actors*
  – An actor is an independent sequential ("= single-process") computation
  – Each actor has a "mailbox" from which it extracts messages that it then processes
  – Actors communicate by sending each other messages
An Actor System

- Actor 1
- Actor 2
- Actor 3

- Computation
- send
- recv
- mailbox

©2015 University of Maryland
General Actor Behavior

• Actors wait until there is a message in their mailbox
• They remove message from mailbox and process it
• Processing may involve sending of messages to other actors
• When execution is complete, they retrieve next message from mailbox and repeat
Message Passing

- Recall: actors communicate via message passing
- Different actor frameworks provide different guarantees about message delivery.
- Here are the ones we will use (conform to akka)
  - **Asynchronous**: senders do not know when messages are received
  - **At-most-once delivery**: every message sent is eventually received at most once (could be lost, but not duplicated)
  - **Locally FIFO**: messages sent by one actor directly to another are received in the order sent, lost messages excepted
Actors History

• Originally proposed by Carl Hewitt in 1970s as basic model of distributed computing
• Theory studied in 1980s / early 1990s by researchers
• Mid-1990s: first serious language implementation (Erlang, Ericsson)
  – Used in implementation of telephone switches
  – Key features: light-weight (more like tasks than threads), high degree of concurrency, resiliency in face of failure
• Mid-2000s: Scala language targeting JVM includes actors
• Late 2000s: akka open-source actor library for Scala, Java
akka Java Library

• Provides implementation of actor model for Java
• Key features
  – Basic actor framework
    • Special actor objects
    • Communication via message-passing methods
  – Lightweight
    • Actors resemble tasks more than threads
    • 650 bytes of overhead to create task
  – Location transparency
    • Actors programmed identically, whether local or remote host
    • Differences captured in configuration file
  – Fault tolerance via hierarchy
    • Actors arranged in parent/child hierarchy
    • Parents handle failures of children
Installing akka for Java

- akka libraries need to be downloaded, installed on Java build path
- Eclipse-based directions
  1. Download latest (2.4.0) Standalone Distribution of akka for Java from http://akka.io/downloads/
  2. Extract all files from the downloaded file akka_2.11-2.4.0.zip. This creates a directory akka_2.11-2.4.0
  3. For each project in Eclipse using akka, you need to add following from this directory to build path:
     • lib/scala-library-2.11.7.jar
     • lib/akka/akka-actor_2.11-2.4.0.jar
     • lib/akka/config-1.3.0.jar
  4. To add a file to project build path in Eclipse:
     • Right-click on project, then select Build Path → Add External Archives
     • Use resulting file dialog to locate above .jar files and add.
akka Documentation

• General: http://doc.akka.io/
  – There are links for the full documentation of Java version of akka
  – The “snapshot” documentation is also useful

• Javadoc:
  http://doc.akka.io/japi/akka/snapshot/
  This summarizes the classes and methods in the akka distribution
Basics of akka Java

• akka actors live in an **actor system**
  – Actor system provides actor execution (think “threads”), message-passing infrastructure
  – To create actors, you must first create an actor system
  – The relevant Java class: `ActorSystem`

• So, first line of Hello World `main()` method is:

```java
ActorSystem actorSystem = ActorSystem.create("Message_Printer");
```

• “Message_Printer” is name of actor system (required)
• akka actor system names must not have spaces or punctuation other than - or _!
Creating Actors in akka Java (1/4)

• Actors are objects (of course!)
• Objects are typically in a subclass of the akka library class `UntypedActor`
• **Step 1 in creating actors:** define class of actors
  – In Hello World example, the class of actors is `MessagePrinterActor`
  – Here is the relevant import / class declaration
    ```java
    import akka.actor.UntypedActor;
    ...
    public class MessagePrinterActor extends UntypedActor ...
    ```
Creating Actors in akka Java (2/4)

- **Step 2 in creating actors:** finish implementation of actor class
  - akka UntypedActor needs instance method
    ```java
    public void onReceive(Object msg)
    ```
  - This method describes how a message object should be processed

- **Hello World example**
  ```java
  @Override
  public void onReceive(Object msg) throws Exception {
    if (msg instanceof String) {
      System.out.printf("Message is: %s", msg);
    }
  }
  ```

- **Observations**
  - Messages are objects!
  - Processing a message requires determining which class to which it belongs
  - More on messages later
Creating Actors in akka Java (3/4)

- In akka, actors can only be created in the context of an `ActorSystem`
  - Relevant instance method in `ActorSystem` is
    ```java
    ActorRef actorOf(Props p, String name);
    ```
  - Return type `ActorRef` is class of “references to actors” (more later on this notion)
  - String parameter is actor name (no spaces or non-alphanumeric characters other than -,_,!)  
    - “Props”? 

- In akka, actors have various configuration information
  - Type of mailbox data structure
  - How messages actually get delivered to mailbox (“dispatching”)
  - Etc.

- This information is encapsulated in a `Props` object for a given class of actors

- To create actors in a class, a `Props` object for the class must be constructed

- **Step 3 in creating actors**: create `Props` object for actors class.

- This is done in the Hello World `main()` using a factory method in akka `Props` class that has reasonable defaults (unbounded FIFO queues for mailboxes, etc.)
  ```java
  Props mpProps = Props.create(MessagePrinterActor.class);
  ```
Creating Actors in akka Java (4/4)

- **Step 4 in creating actors:** call `actorOf()` method in relevant `ActorSystem`

- In Hello World example:

  ```java
  ActorRef mpNode = actorSystem.actorOf(mpProps, "MP_Node");
  ```

  - This creates and launches a single actor in `actorSystem`

  - Actor is now ready to receive, process messages
Communicating with Actors

• Actors compute by processing messages
• To send a message to an actor, use ActorRef instance method `tell(Object msg, ActorRef sender)`
  – `tell()` takes message (payload) and sender as arguments
    • sender parameter allows return communication
    • If no return communication desired, specify null for sender field
  – `tell()` is often said to implement “fire and forget” communication
    • Method call returns as soon as message handed off to infrastructure
    • No waiting to see if recipient actually receives it
• In Hello World example:
  `mpNode.tell("Hello World", null);`
Shutting Down an ActorSystem

- ActorSystem objects use worker threads internally to execute actors
- These threads must be killed off before an actor-based application can terminate
- This is done by shutting down the ActorSystem using instance method terminate()
- From Hello World example:
  ```
  actorSystem.terminate();
  ```
Moving Information from ActorSystem to Java

• The `tell()` method permits messages to be sent to actors
  – In Hello World, this was how information was passed from “rest of Java” into actor
  – Actors can also send messages to each other inside an actor system

• How can actors communicate with outside world?
  Outside world (i.e. “rest of Java”) is not an actor, so `tell()` cannot be used!

• **Solution**: `Patterns.ask()`
Patterns.ask()

- **Patterns**: a class in akka supporting the creation of different communication patterns
- **ask()** is a static method in Patterns that supports “call-response” communication
  
  **Header**
  
  ```scala
  public static scala.concurrent.Future<java.lang.Object> ask(ActorRef actor, Object msg, long timeoutMillis)
  ```

  **Behavior**

  - `ask(actor, msg, timeout)` sends `msg` to `actor`, just like `tell()`
  - It returns a (Scala, not Java!) Future holding a return message from actor.
  - If return message not available by `timeout`, `AskTimeoutException` thrown
  - To get return message from Future `f`, need to do Scala equivalent of `f.get()`:
    ```scala
    Await.result(f, timeout.duration())
    ```
    - `Await` is Scala class of static blocking methods
    - `timeout` is object in Scala Timeout class; `duration()` is instance method for this class

- `ask()` can be used between actors, or between a non-actor and an actor
ask() Example: ToAndFrom

• Goal: have simple “call-response” involving `main()`, `actor`
  – `main()` sends message to `actor`
  – `Actor prints message, sends response`
  – `main()` prints response

• Key classes
  – `MessageAcknowledgerActor`
  – `ToAndFrom (has main())`
public class MessageAcknowledgerActor extends UntypedActor {

    @Override
    public void onReceive(Object msg) throws Exception {
        if (msg instanceof String) {
            ActorRef sender = getSender();
            String payload = (String)msg;
            System.out.printf("Message is:  %s\n", payload);
            sender.tell(payload + " message received", sender);
        }
    }
}
getSender()?

• Instance method in ActorRef
• Returns ActorRef for sender of current message being processed in onReceive()
Actor Communication

• Actor(Ref)s communicate by sending each other messages
• To send a message to recipient \( r \), a sender \( s \) needs to invoke \( r \cdot \text{tell}() \)
• This means the sender needs to know \( r \)!
• Different ways to do this
  – Send a message to \( s \) containing \( r \) as payload
  – Send message to \( s \) with \( r \) as sender
  – In constructor associated with \( s \), include \( r \) as parameter
PingPong Example

• Goal: have actor system containing two actors that send message back and forth
  – One prints “Ping ... “ when it gets message
  – Other prints “Pong”

• They stop after a set number of exchanges
public class PongActor extends UntypedActor {

    @Override
    public void onReceive(Object msg) throws Exception {
        if (msg instanceof String) {
            String payload = (String)msg;
            if (payload.equals("stop")) { // Game over
                System.out.println(getSelf().path().name() + ": OK");
            } else if (payload.equals("start")) {
                System.out.println(getSelf().path().name() + ": Let's do it.");
                getSender().tell("go", getSelf());
            } else { // Next stroke
                System.out.println("Pong");
                getSender().tell("go", getSelf());
            }
        }
    }
}
public class PingActor extends UntypedActor {

    private int numHitsLeft;
    private ActorRef partner;

    public PingActor(int numHits) {
        this.numHitsLeft = numHits;
    }

    @Override
    public void onReceive(Object msg) throws Exception {
        if (msg instanceof ActorRef) {
            partner = (ActorRef)msg;
            System.out.println(getSelf().path().name() + "\n\t: Game on!");
            partner.tell("start", getSelf());
        } else if (msg instanceof String) {
            ...
        }
    }

    – If msg is an ActorRef, this is assigned to the partner field
    – This is how PingActor knows to whom to send messages!
public class PingPong {

    public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("Ping_Pong");
        Props pingProps = Props.create(PingActor.class, 5);
        Props pongProps = Props.create(PongActor.class);
        ActorRef pingNode = actorSystem.actorOf(pingProps, "Ping_Node");
        ActorRef pongNode = actorSystem.actorOf(pongProps, "Pong_Node");
        pingNode.tell(pongNode, null);
        actorSystem.terminate();
    }
}

– In pingProps definition, the “5” is the argument to the PingActor constructor that will be used
– Note that main() is sending pongNode to pingNode to start system off!
Messages

• Message are objects
• Valid classes of messages must match `Serializable` interface
  – Serializable objects can be converted into bytes
  – This is needed for actors to communicate over communication networks, which just transmit bytes
• They should also be *immutable*
  – Objects are properly constructed
  – Fields are private, final
  – State never changes