Lecture 22
Programming in akka Java
Programming in Java

• In single-threaded setting, synchronous!
  – Computation happens via method invocation
  – When a method is called, caller waits until method is done

• In multi-threaded setting, multiple method calls can be active at same time
  – General model is still synchronous within threads, though; threads launch methods, wait for results
  – Futures, etc. give mechanism for separating call, response, however
Programming in akka

• Actors-based programming is sometimes referred to as reactive programming
  – Actors compute by reacting to messages
  – Message sending is asynchronous

• Promotes “server-like” programming model
  – Actors are like servers
  – You send messages to servers to get them to do something
  – The server may send a response, or not; it depends on how it is programmed
  – You should be clear about the communications protocol in your application
    • Which interactions are “fire-and-forget”
    • Which interactions involve a “request-response” model?
    • Etc.
Sample Protocols

- Request-response interaction
  - Like asynchronous method call
  - Messaging
    - Send message to “server”
    - Process return message from server
    - Since return message can be mixed in with other messages, return message needs some detail to tell recipient what to do
  - Variation: delegated response
    - Send message to “server” as before
    - Response may come from a different server

- Trigger interaction
  - Like `exec.execute()` in Java executors
  - Messaging:
    - Send message to “launch server”
    - Note that assumptions cannot be made about when the interaction is finished
Designing Communications Protocols for Actors

• Two similar graphical notations for representing communications protocols
  – **Sequence Diagrams (SDs)**
    • Part of Unified Modeling Language (UML)
    • Used for describing interactions among general objects
  – **Message Sequence Charts (MSCs)**
    • International Telecommunications Union (ITU) standard
    • Used for describing message-passing interactions

• We can use them to write down how we want actors to exchange messages
  – We will refer to the diagrams as sequence diagrams
  – They will not strictly adhere to the UML standard, and will include some MSC-style notation
Sample Sequence Diagram

- Client
- Server
- Request
- Response

- actor instances
- message sends
- message types
- life line for Client actor
- life line for Server actor
- termination indicator
Components of a Sequence Diagram

- Collection of actors, each with a (named) lifeline
  - Name given in box at top of lifeline
  - Lifeline represents execution flow for the given actor
    - Execution starts at top, goes to bottom
    - Execution may terminate (e.g. Client in previous example), or keep going (e.g. Server)

- Message passing arrows
  - Arrows go from lifeline of sender to lifeline of receiver
  - Array labeled by the type of message (i.e. what message is for)
Actor Creation

- Dashed line indicates that Server creates an instance of Transaction Handler
- Position of arrow on lifeline of Server indicates when this happens
Example

• Setting
  – We have a system of “integer chains”, i.e. actors that each store an integer and their downstream neighbor, if they have one
  – We would like a message-passing protocol for computing the sum of all integers in a chain

• General solution
  – Node in a chain receives a message from its upstream neighbor with partial sum
  – Node updates partial sum, sends message to its downstream neighbor
  – Final node in chain returns result
Sequence Diagram for Sum Protocol
What Sum Sequence Diagram Says

• “User” is the actor who triggers the protocol by sending a SumRequest message to the first node
• First node sends SumResult message (carrying its value, but this is not explicit in the diagram) to the next node
• Each intermediate node, upon receiving a SumResult message, adds in its value and sends the resulting SumResult message to its downstream neighbor
• Final node (Node n) sends SumResult message back to “User”
Another Example

- Another operation on chains: adding a new node at the end of a chain
- How this should work
  - Request message comes in to first node
  - Nodes forward message to neighbors until final node reached
  - Final node creates new node
Sequence Diagram for Add Protocol

User -> Node 1 -> Node 2 -> ... -> Node n

AddRequest

Node n+1
What Add Sequence Diagram Says

- “User” is the actor who triggers the protocol by sending an AddRequest message (with value to put in new node, but this is not explicit in diagram) to the first node
- First node sends AddRequest message to the next node
- Each intermediate node, upon receiving an AddRequest message, sends the message to its downstream neighbor
- Final node (Node n) creates the new node (which holds value in AddRequest message, although diagram does not say this explicitly)
Using Sequence Diagrams to Design Actor Systems

• Sequence diagrams make two things clear
  – What types of messages will be exchanged
  – For each actor, what types of incoming messages it needs to be able to process

• In the Integer Chain system example:
  – Three kinds of messages: SumRequest, SumResult, AddRequest
  – Node actors can receive all three kinds of messages
  – Consequently, code for onReceive() method in node actors needs to deal with each of these three kinds of messages
Messages in akka

• Recall header for `onReceive()` in `UntypedActor`
  
  ```java
  void onReceive(Object arg0)
  ```
  
  – Type of message is `Object`!
  – To do anything useful with message, it must be cast to a type at runtime
  – Messages should also convey information to recipient actors about what they are for

• Good practice: use different classes for different message types
  
  – Ensure messages are all in message classes
  – Only send messages that are instance of message classes
  – This helps remind you what they are for and makes processing easier

• Tips
  
  – Put message class files in one package, actors in another
  – Distinguish “Request” and “Response” (or “Result”) message types when appropriate
  – Base names of classes on sequence diagrams, if these exist
Designing Actors

- **Main control structure of** `onReceive()`: if ... else if ... else if ...
  - Do case analysis on message type
  - Final else clause should call `unhandled()` with message (unhandled() is instance method in UntypedActor)

- **Example**

```java
if (msg instanceof MType1) { // msg has type MType1
    MType1 payload = (MType1)msg;  // cast needed to get type
    ...
}
else if (msg instanceof MType2) {
    MType2 payload = (MType2)msg;
    ...
}
else {
    System.out.println ...; // Print unhandled message
    unhandled(msg);
}
```

- **Tips**
  - Put agent implementations in packages separate from messages
  - Include static Props-producing factory method in actor classes to ease production of actors from actor classes
Testing

• Start with single-threaded tests
  – Interact with actor system using messages sent from Java via Patterns.ask()
  – Check that correct results are being returned
• Then try doing multiple simultaneous tests
  – This can be done by creating special “test actors” that are run inside the actor system and interact with the “regular actors”
  – You can also create multiple threads in Java and have them each execute single-threaded tests as above
• Point of simultaneous tests: make sure that actors do not get confused when messages involving multiple interactions are being passed around
  – Possibility of multiple simultaneous interactions has implications for message-class design
  – For example, depending on your application:
    • You may want information regarding eventual recipient of data, etc.
    • You may want to include full intermediate results of a computation inside message