CSYE 7215: Parallel & Multithreaded Programming

Textbook:

Brian Goetz et al. "Java Concurrency in Practice."

Lecture 14: Akka: Actors, Java Programming

These slides make use of the slides developed by prof. Rance Cleaveland at the University of Maryland. I have received his permission to use these slides in this class. Please do not distribute them to anybody who is not enrolled in this class.

UNIVERSITY OF MARYLAND

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. Processes

- 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

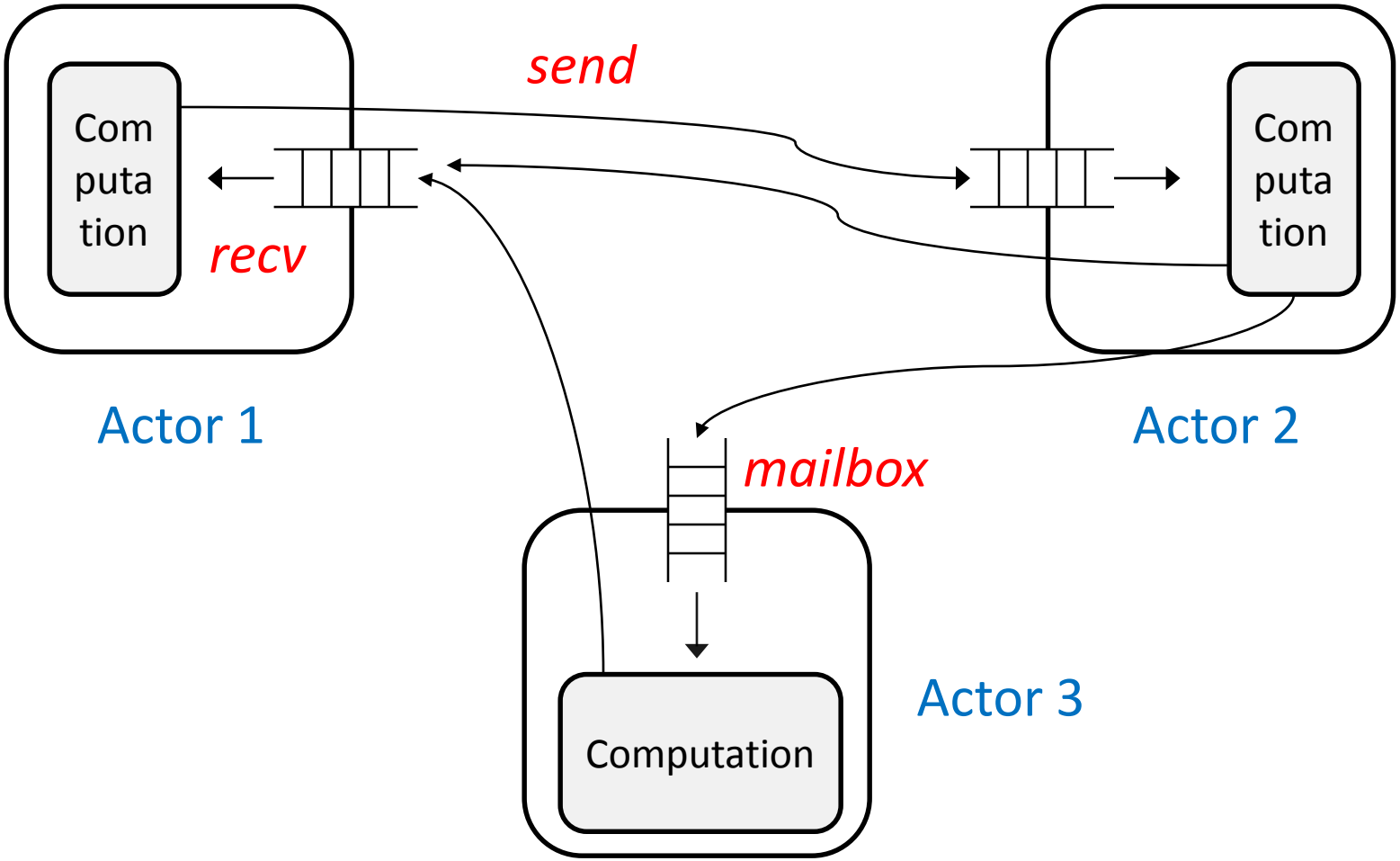
  Internet Protocol address: number assigned to a host connected to the internet so that other hosts may communicate with it
- MAC address

  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-threaded") 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



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 processing 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

Actor 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
    - 300 bytes of overhead per actor
  - 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 (2.4.2) Standalone Distribution of akka for Java from http://akka.io/downloads/
  2. Extract all files from the downloaded file akka_2.11-2.4.2.zip. This creates a directory akka-2.4.2
  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_2.11-2.4.2.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:

```
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

```
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
    `public void onReceive(Object msg)`
  - This method describes how a message object should be processed
- Hello World example

```
@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
    `ActorRef actorOf(Props p, String name);` // method of superclass ActorRefFactory
  - 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
  - This builds `Props` object with reasonable defaults (unbounded queues for mailboxes, etc.)
  - Relevant Hello World code:
    ```
    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:

```
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 down 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

    ```
    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 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()`:

      ```
      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()`)

MessageAcknowledgerActor.java

```
public class MessageAcknowledgerActor extends UntypedActor {
  …
  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()`
  - The sender is the second parameter of the tell() method call corresponding to the current message
  - A more accurate characterization: rather than thinking of this as message sender (it may not be!) think of it as "Reply-To", as in e-mail

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.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

PongActor.java

```
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());
      }
    }
  }
}
```

- `getSelf()` obtains `ActorRef` that `PongActor` is associated with at run time
- `getSelf().path().name()` obtains name assigned to ActorRef at run time

PingActor.java

```
public class PingActor extends UntypedActor {

  private int numHitsLeft;
  private ActorRef partner;

  public PingActor(int numHits) {       // Ping will check numHits, will stop if  == 0
    this.numHitsLeft = numHits;
  }

  @Override
  public void onReceive(Object msg) throws Exception {
    if (msg instanceof ActorRef) {
      partner = (ActorRef)msg;
      System.out.println(getSelf().path().name() +  ": 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!

PingPong.java

```
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

- Messages 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* (because Akka cannot enforce immutability, yet)
  - Objects are properly constructed
  - Fields are private, final
  - State never changes

UNIVERSITY OF MARYLAND

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()`

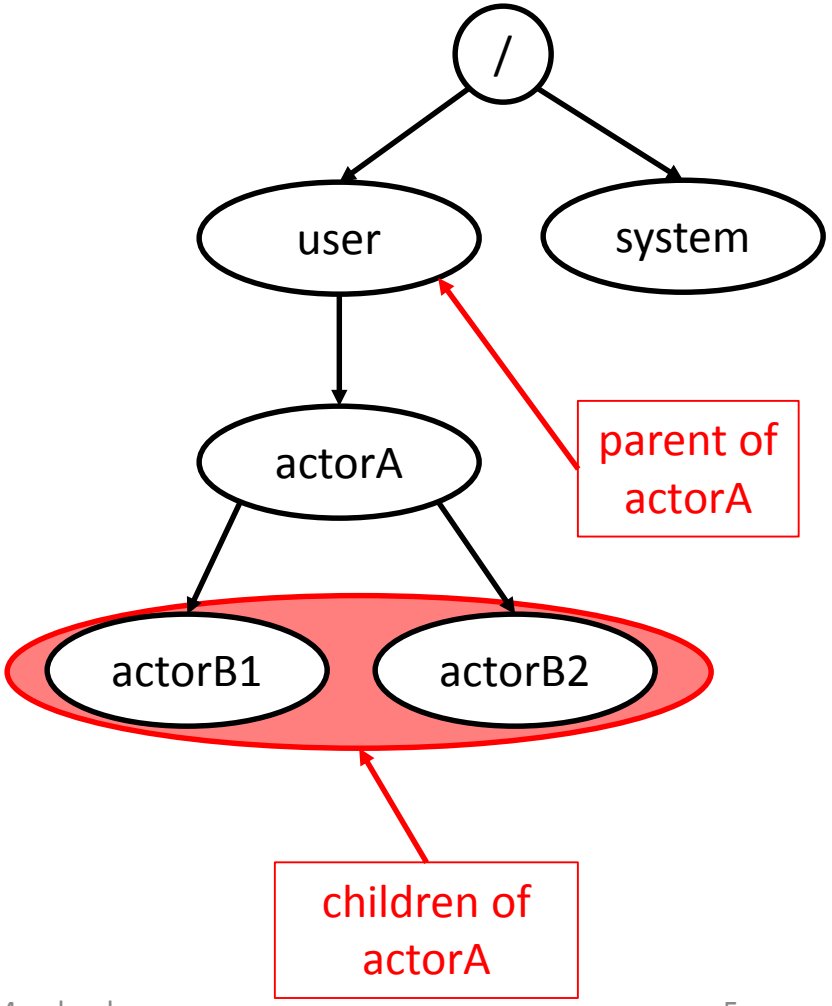
    Return parent of actor associated with context
  - `java.lang.Iterable<ActorRef> getChildren()`

    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 ≥ 0 children
- Every actor can be identified via path (`ActorPath`) in tree
- To get path of `ActorRef`, use `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

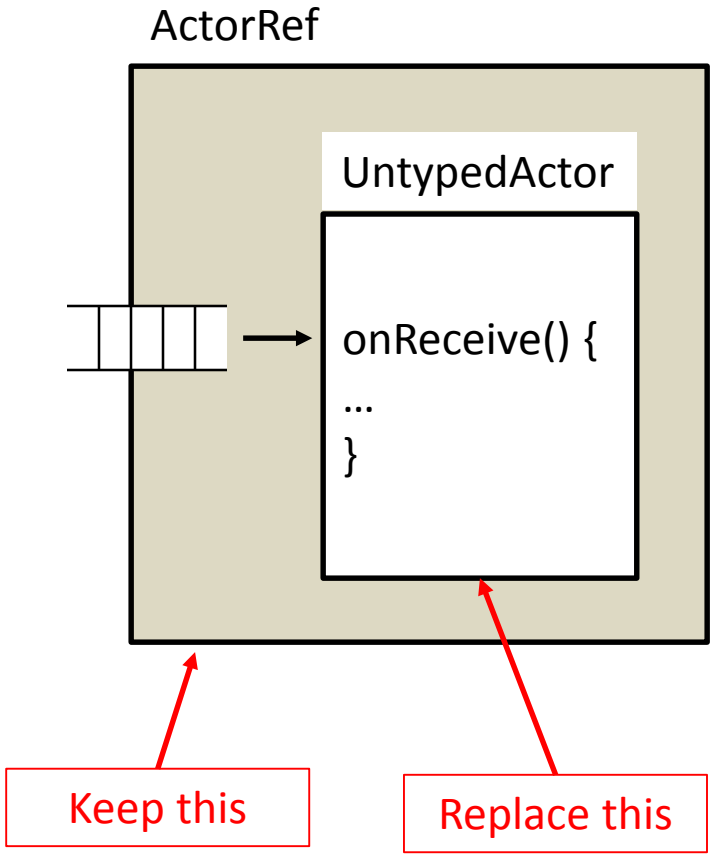
  This queue is only used for supervision (i.e. parent-child) communication

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

  `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 …

```
getContext().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

UNIVERSITY OF MARYLAND

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.

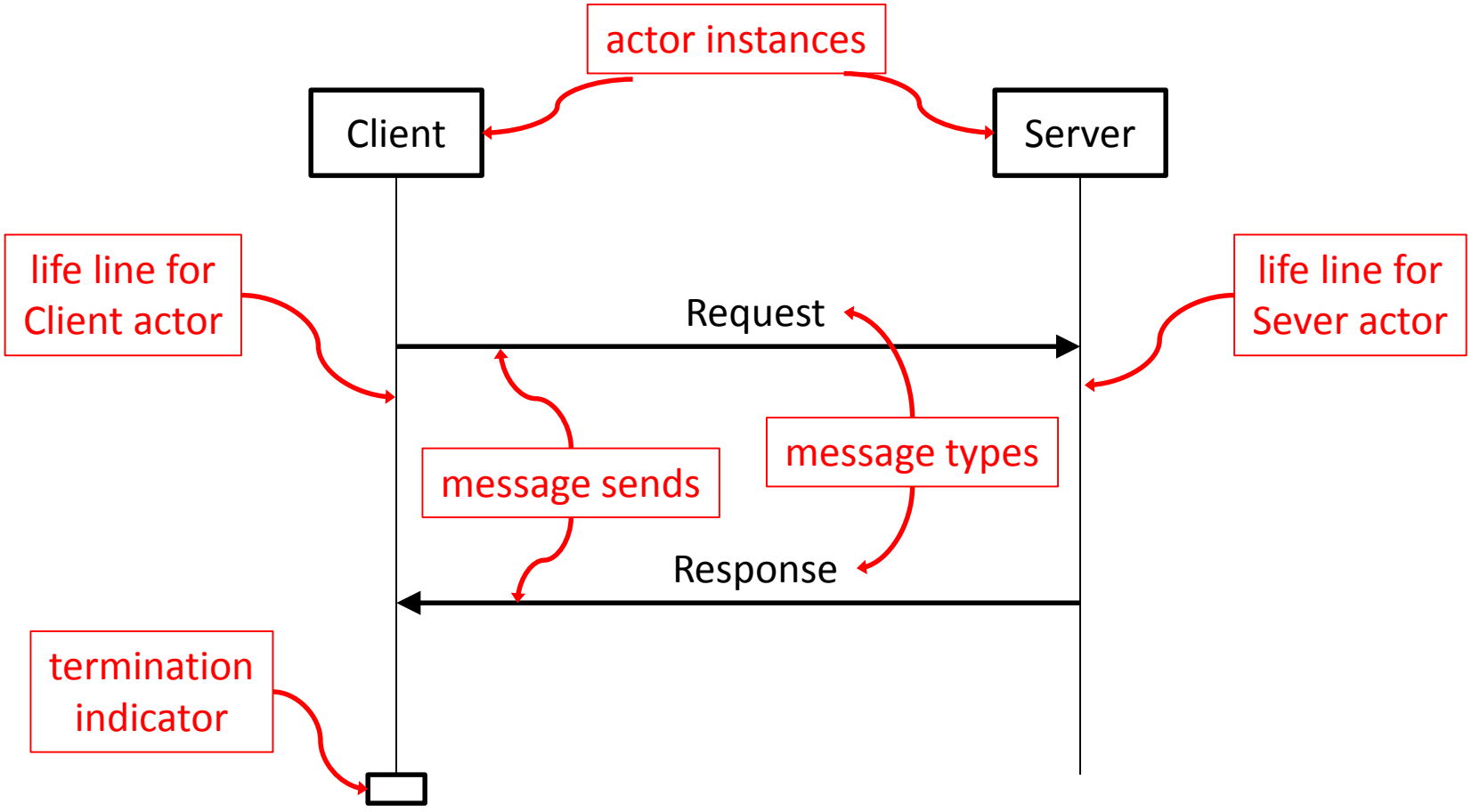
Sample Protocols

- Request-response interaction
  - Like asynchronous method call
  - Messaging
    - Send message to “server”
    - Process return message from method server
    - Since return message can be mixed in with other messages, return message needs some detail to tell recipient what to do
- 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

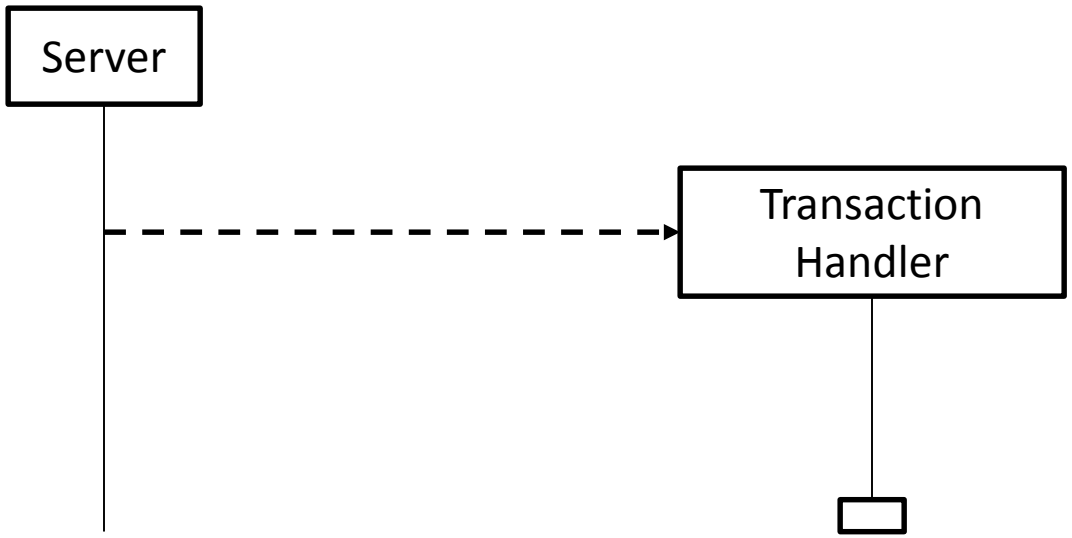
Sample Sequence Diagram



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

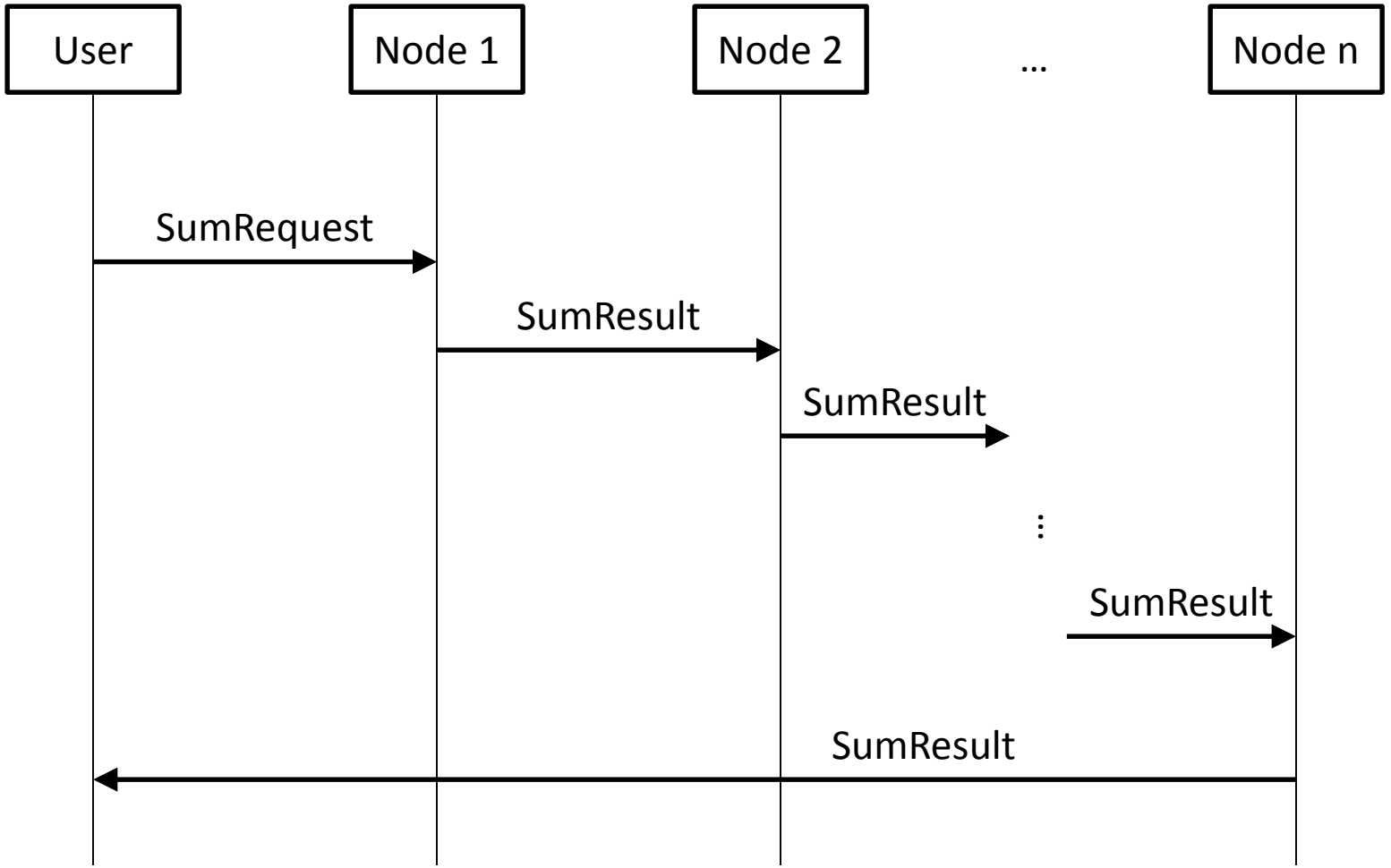


- Dashed line indicates that Server creates an instance of Transaction Handler
- Position of arrow on lifeline of Server indicates when this happens

Example

- Setting
  - We have a system of “integer chains”, i.e. actors that each store an integer and can send messages to their downstream neighbor
  - 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 returns result

Sequence Diagram for Sum Protocol



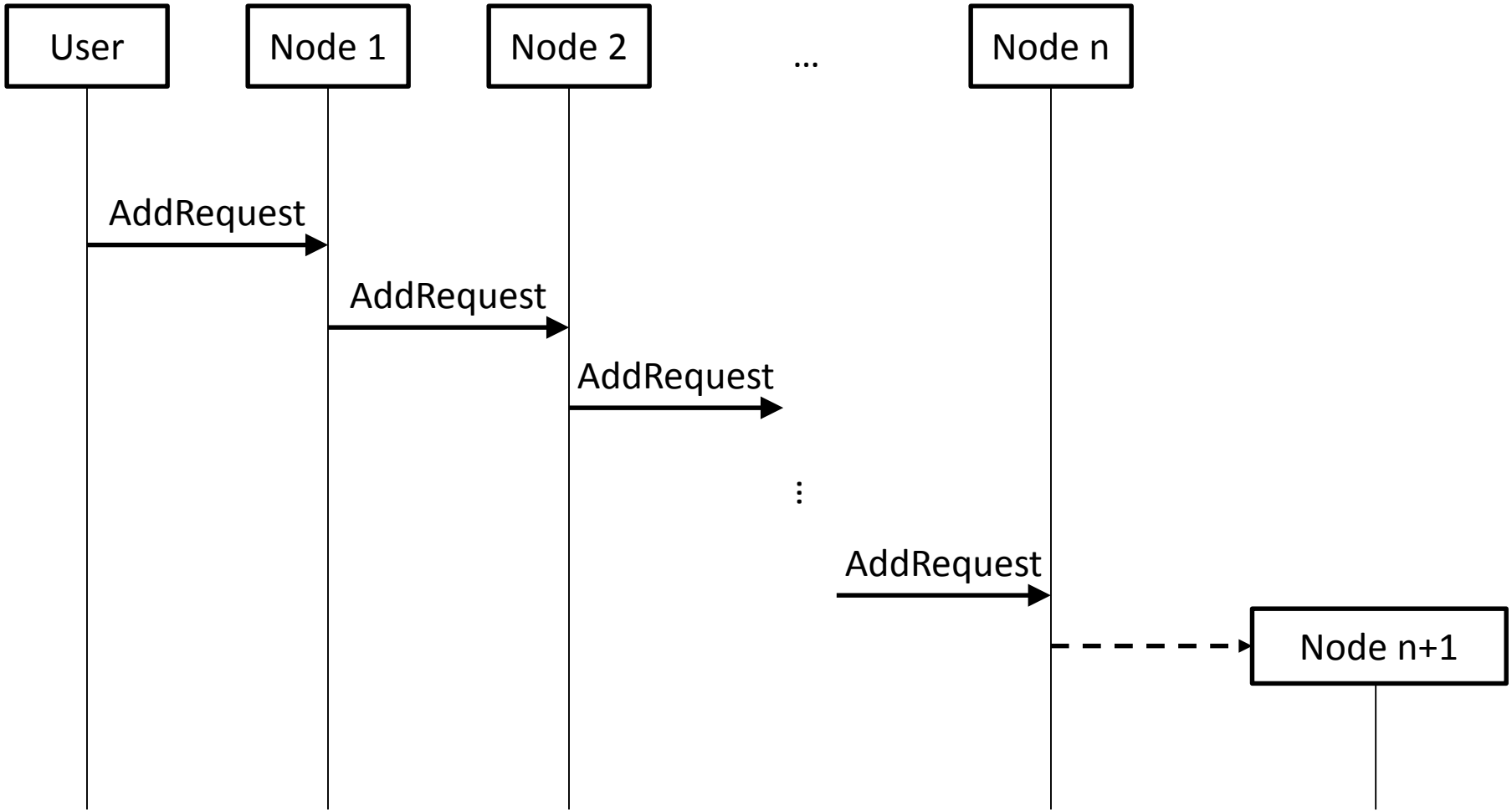
What Sum Sequence Diagram Says

- “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



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`

  ```
  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 “Reponse” (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

// inherited from Actor

```
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