Actor Acknowledgments

- Actor Model - Wikipedia
- “Actor Concurrency” by Alex Miller
  http://www.slideshare.net/alexmiller/actor-concurrency
- “The Actor Model - Towards Better Concurrency” by Dror Bereznitsky
  http://www.slideshare.net/drorbr/the-actor-model-towards-better-concurrency
- “Actors in Scala” by Philipp Haller and Frank Sommers
The Actor Model (Recap)

- An actor may:
  - process messages
  - read/write local state
  - create a new actor
  - start a new actor
  - send messages to other actors
  - terminate

- An actor processes messages sequentially
  - guaranteed mutual exclusion on accesses to local state
Actor Model

- A message-based concurrency model to manage mutable shared state
- First defined in 1973 by Carl Hewitt
  Further theoretical development by Henry Baker and Gul Agha
- Key Ideas:
  - Everything is an Actor!
  - Analogous to “everything is an object” in OOP
  - Encapsulate shared state in Actors
  - Mutable state is not shared
- Other important features (we will get to these later)
  - Asynchronous message passing
  - Non-deterministic ordering of messages
Actors’ Behavior

- Actors are passive and lazy
  - Only respond if messages are sent to them
    - Messages may come from other actors or from main program (environment)
  - Only process one message at a time
    - Pending messages are stored in a “mailbox”
    - Parallelism comes from multiple actors processing messages in parallel
  - Mutate local state only while processing a message
  - Mutating local state can result in actor responding differently to subsequent messages
Actor

mailbox

local state

process one message at a time
Actor Analogy - Email

- Email accounts are a good simple analogy to Actors
- To notify some information to (i.e. change some state of) A1 another account A2 sends an email (i.e. sends a message) to A1
- A1 has a mailbox to store all incoming messages
- A1 can read (i.e. process) one email at a time
  - At least that is what normal people do :) 
- Reading an email can change how you respond to a subsequent email
  - e.g. receiving pleasant news while reading current email can affect the response to a subsequent email
- Actor creation (stretching the analogy)
  - Create a new email account that can send/receive messages
Actor Life Cycle

Actor states

- **New**: Actor has been created
  - e.g., email account has been created
- **Started**: Actor can receive and process messages
  - e.g., email account has been activated
- **Terminated**: Actor will no longer processes messages
  - e.g., termination of email account after graduation
Using Actors in HJ

- Create your custom class which extends hj.lang.Actor<Object> , and implement the void process() method

```java
class MyActor extends Actor<Object> {
    protected void process(Object message) {
        System.out.println("Processing " + message);
    }
}
```

- Instantiate and start your actor
  ```java
  Actor<Object> anActor = new MyActor(); anActor.start();
  ```

- Send messages to the actor
  ```java
  anActor.send(aMessage); // aMessage can be any object in general
  ```

- Use a special message to terminate an actor
  ```java
  protected void process(Object message) {
      if (message.someCondition()) exit();
  }
  ```

- Actor execution implemented as async tasks in HJ

- Can use finish to await their completion
Hello World Example

1. public class HelloWorld {
2.     public static void main(String[] args) {
3.         EchoActor actor = new EchoActor();
4.         actor.start(); // don't forget to start the actor
5.         actor.send("Hello"); // asynchronous send (returns immediately)
6.         actor.send("World");
7.         actor.send(EchoActor.STOP_MSG);
8.     }
9. }
10. class EchoActor extends Actor<Object> {
11.     static final Object STOP_MSG = new Object();
12.     private int messageCount = 0;
13.     protected void process(final Object msg) {
14.         if (STOP_MSG.equals(msg)) {
15.             println("Message-" + messageCount + ": terminating.");
16.             exit(); // never forget to terminate an actor
17.         } else {
18.             messageCount += 1;
19.             println("Message-" + messageCount + ": " + msg);
20.         }
21.     }
22. }

Sends are asynchronous in actor model, but HJ Actor library preserves order of messages between same sender and receiver.
Pi Computation Example

\[ \pi = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k + 1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots. \]

*Use Master-Worker technique:

- **Master** accumulates approximation of PI

  - Series 1 - 100
  - Series 101 - 200
  - Series 1301 - 1400
  - Series 1401 - 1500
  - Series 1501 - 1600
  - Series 2501 - 2600
  - Series 1201 - 1300

  - **Worker-1**
  - **Worker-2**
  - **Worker-n**

Source: [http://www.enotes.com/topic/Pi](http://www.enotes.com/topic/Pi)
class Master extends Actor<Object> {
  private double result = 0; private int nrMsgsReceived = 0;
  private Worker[] workers;
  Master(nrWrkrs, nrEls, nrMsgs) {...} // constructor
  void start() {
    super.start(); // Starts the master actor
    // Create and start workers
    workers = new Worker[nrWrkrs];
    for (int i = 0; i < nrwrkrs; i++) {
      workers[i] = new Worker();
      workers[i].start();
    }
    // Send messages to workers
    for (int j = 0; j < nrMsgs; j++) {
      someWrkr = ... ; // Select worker for message j
      someWrkr.send(new Work(...));
    }
  } // start()
Pi Calculation --- Master Actor (contd)

19.  void exit() {
20.     for (int i = 0; i < nrWrkrs; i++) workers[i].send(new Stop());
21.     super.exit(); // Terminates the actor
22. } // exit()
23.  void process(final Object msg) {
24.     if (msg instanceof Result) {
25.         result += ((Result) msg).result;
26.         nrMsgsReceived += 1;
27.         if (nrMsgsReceived == nrMsgs) exit();
28.     }
29.     // Handle other message cases here
30. } // process()
31.} // Master
32. . . .
33. // Main program
34. Master master = new Master(w, e, m);
35. finish master.start();
36. println("PI = " + master.getResult());
Pi Calculation --- Worker Actor

1. class Worker extends Actor<Object> {
2.     void process(Object msg) {
3.         if (msg instanceof Stop) exit();
4.         else if (msg instanceof Work) {
5.             Work wm = (Work) msg;
6.             double result = calculatePiFor(wm.start, wm.end)
7.                 master.send(new ResultMessage(result));
8.         } // process()
9.     } // process()
10. }
11. private double calculatePiFor(int start, int end) {
12.     double acc = 0.0;
13.     for (int i = start; i < end; i++) {
14.         acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1);
15.     }
16.     return acc;
17. } // Worker
Simple Pipeline

A Simple pipeline with 3 stages

Stage-1
Filter even length strings

Stage-2
Filter lowercase strings

Stage-3
Print results
Simple Pipeline using HJ Actors

1.   // Main program
2.   finish {
3.       Actor<Object> firstStage =
4.           new EvenLengthFilter(
5.               new LowerCaseFilter(
6.                   new LastStage()));
7.       firstStage.start(); // starts others
8.       firstStage.send("pipeline");
9.       firstStage.send(new StopMessage());
10.   }
11. }
12.class LastStage extends Actor {
13.   protected void process(Object msg) {
14.       if (msg instanceof StopMessage) {
15.           exit();
16.       } else if (msg instanceof String) {
17.           System.out.println(msg);
18.     }   }   }
Simple Pipeline using HJ Actors (contd)

19. class LowerCaseFilter extends Actor {
20.     protected void process(Object msg) {
21.         if (msg instanceof StopMessage) {
22.             exit(); nextStage.send(msg);
23.         } else if (msg instanceof String) {
24.             String str = (String) msg;
25.             if (str.toLowerCase().equals(str)) {
26.                 nextStage.send(str);
27.         }
28.     }
29. }
30. class EvenLengthFilter extends Actor {
31.     protected void process(Object msg) {
32.         if (msg instanceof StopMessage) {
33.             nextStage.send(msg);
34.             exit();
35.         } else if (msg instanceof String) {
36.             String msgStr = (String) msg;
37.             if (msgStr.length() % 2 == 0) {
38.                 nextStage.send(msgStr);
39.             }
40.     }
41. }
ThreadRing (Coordination) Example

1. finish {
2.   int numThreads = 4;
3.   int numberOfHops = 10;
4.   ThreadRingActor[] ring =
5.     new ThreadRingActor[numThreads];
6.   for(int i=numThreads-1;i>=0; i--) {
7.     ring[i] = new ThreadRingActor(i);
8.     ring[i].start();
9.     if (i < numThreads - 1) {
10.        ring[i].nextActor(ring[i + 1]);
11.   }
12.   ring[numThreads-1].nextActor(ring[0]);
13.   ring[0].send(numberOfHops);
14. }

14. class ThreadRingActor
15.   extends Actor<Object> {
16.     private Actor<Object> nextActor;
17.     private final int id;
18.     ...
19.   public void nextActor(
20.     Actor<Object> nextActor) {...}
21.   void process(Object theMsg) {
22.     if (theMsg instanceof Integer) {
23.        Integer n = (Integer) theMsg;
24.        if (n > 0) {
25.           println("Thread-" + id +
26.             " active, remaining = " + n);
27.           nextActor.send(n - 1);
28.         } else {
29.           println("Exiting Thread-" +
30.             id);
31.         }
32.      } else {
33.        /* ERROR - handle appropriately */
34.    } }
}
Integer Counter Example

Without Actors:

1. int counter = 0;
2. public void foo() {
3.    // do something
4.    isolated {
5.        counter++;
6.    }
7.    // do something else
8. }
9. public void bar() {
10.    // do something
11.    isolated {
12.        counter--;
13.    }
14.}

With Actors:

14. class Counter extends Actor {
15.    private int counter = 0; // local state
16.    public void process(Message msg) {
17. if (msg instanceof IncMessage) {
18.        counter++;
19. } else if (msg instanceof DecMessage){
20.        counter--;  
21. }
22. }

• Can also use atomic variables instead of isolated

14. Counter counter = new Counter();
15. public void foo() {
16.    // do something
17.    counter.send(new IncrementMessage(1));
18.    // do something else
19. }
20. public void bar() {
21.    // do something
22.    counter.send(new DecrementMessage(1));
23.}
Limitations of Actor Model

- Deadlocks possible
  - Deadlock occurs when all started (but non-terminated) actors have empty mailboxes
- Data races possible when messages include shared objects
- Simulating synchronous replies requires some effort
  - e.g., does not support `addAndGet()`
- Implementing truly concurrent data structures is hard
  - No parallel reads, no reductions/accumulators
- Difficult to achieve global consensus
  - Finish and barriers not supported as first-class primitives

==> These limitations can be overcome by using a hybrid model that combines task parallelism with actors
Actors - Simulating synchronous replies

- Actors are inherently asynchronous
- Synchronous replies require blocking operations e.g., async await

```java
class CountMessage {
    ... ddf = new DataDrivenFuture();
    int localCount = 0;

    static int getAndIncrement(
        CounterActor counterActor)
    {
        ... msg = new CountMessage();
        counterActor.send(msg);
        // use ddf to wait for response
        // THREAD-BLOCKING
        finish { async await(msg.ddf) { }}
        // return count from the message
        return msg.localCount;
    }
}

class CounterActor extends Actor {
    int counter = 0;
    void process(Object m) {
        if (m instanceof CountMessage){
            CountMessage msg = ...
            counter++;
            msg.localCount = counter;
            msg.ddf.put(true);
        }...
    }
```
Actors – Global Consensus

- Global consensus is a variant of synchronous reply
- More complicated because a group of actors is involved
- Non-blocking solutions can be complex e.g.,
  - First send message from master actor to participant actors signaling intention
  - Wait for all participants to reply they are ready. Participants start ignoring messages sent to them apart from the master
  - Once master confirms all participants are ready, master sends the request to each participant and waits for reply from each
  - Master notifies participants that consensus has been reached, everyone can go back to normal functioning
Parallelizing Actors in HJ

- Two techniques:
  - Use finish construct to wrap asyncs in message processing body
    - Finish ensures all spawned asyncs complete before next message is
  - Allow escaping asyncs inside process() message
    - WAIT! Won't escaping asyncs violate the one-message-at-a-time rule in actors
    - Solution: Use pause and resume
Actors: pause and resume

- Paused state: actor will not process subsequent messages until it is resumed.
- Pause an actor before returning from message processing body with escaping asyncs.
- Resume actor when it is safe to process the next message.
- Akin to Java’s wait/notify operations with locks.
Synchronous Reply using Async-Await (without pause/resume)

1. class SynchronousReplyActor1 extends Actor {
2.     void process(Message msg) {
3.         if (msg instanceof Ping) {
4.             finish {
5.                 DataDrivenFuture<T> ddf = new DataDrivenFuture<T>();
6.                 otherActor.send(ddf);
7.                 async await(ddf) {
8.                     T synchronousReply = ddf.get();
9.                     // do some processing with synchronous reply
10.                }
11.            }
12.         } else if (msg instanceof ...) { ... } } }

Synchronous Reply using Pause/Resume

1. class SynchronousReplyActor2 extends Actor {
2.     void process(Message msg) {
3.         if (msg instanceof Ping) {
4.             DataDrivenFuture<T> ddf = new DataDrivenFuture<T>();
5.             otherActor.send(ddf);
6.             async await(ddf) { // this async processes synchronous reply
7.                 T synchronousReply = ddf.get();
8.                 // do some processing with synchronous reply
9.                 resume(); // allow actor to process next message
10.             }
11.             pause(); // when paused, the actor doesn't process messages
12.         } else if (msg instanceof ...) { ... } }
}
Other uses of hybrid actor+task parallelism

- Can use finish to detect actor termination
- Event-driven tasks
- Stateless Actors
  - If an actor has no state, it can process multiple messages in parallelism
- Pipeline Parallelism
  - Actors represent pipeline stages
  - Use tasks to balance pipeline by parallelizing slower stages
Summary of Mutual Exclusion approaches in HJ

- Isolated --- analogous to critical sections
- Object-based isolation, isolated(a, b, ...)
- Single object in list --- like monitor operations on object
- Multiple objects in list --- deadlock-free mutual exclusion on sets of objects
- Java atomic variables --- optimized implementation of object-based isolation
- Java concurrent collections --- optimized implementation of monitors
- Actors --- different paradigm from task parallelism (mutual exclusion by default)