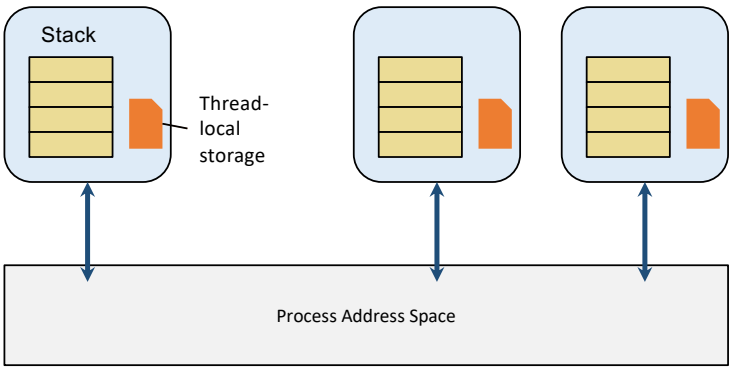
Introducing Actors with Akka and Java

Traditional Threading Model

- Multiple threads sharing a single address space

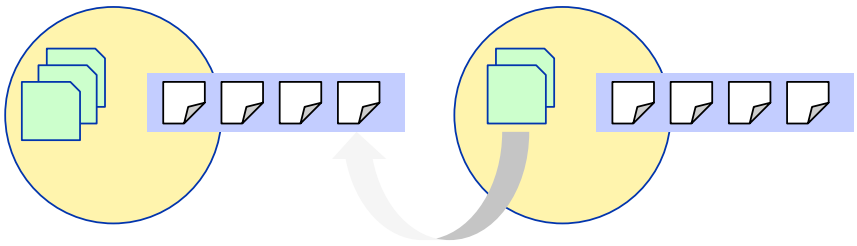


Issues With The Traditional Model

- Threads no longer viewed as lightweight
  - stack size 512K to 2MB
  - limits number of threads that can be created
- Protection of shared mutable state is hard
  - locking very difficult to get right
  - based on notion of blocking and context switching
  - many problems are timing related
- Much boiler plate needed
  - low level constructs need management

Actors

- An alternative approach to concurrency and distribution
- Actor is a small, self-contained processing unit
  - contains state, behaviour and mailbox
- Actors communicate by sending messages
  - asynchronously



Actors...

- Should not share any mutable state
  - can have mutable state internally but nothing exposed
- Should communicate using immutable messages
- Should communicate asynchronously
- Behave reactively
  - Only perform calculations in response to messages
- Can exist within one process or across processes
  - also across machines
- Should provide a safe model for handling failures

A Simple Example

- Two Actors implementing "TickTock" example
- Message types
  - usually defined as classes

```
public class TickMessage {
}
```

```
public class TockMessage {
}
```

```
public class StartTickingMessage {

  private ActorRef dest;

  public StartTickingMessage(ActorRef d) {
    dest = d;
  }

  public ActorRef getDest() {
    return dest;
  }
}
```

A Simple Example

- The "Tick" Actor

```
public class TickActor extends UntypedActor {

  LoggingAdapter log = Logging.getLogger(
                                    getContext().system(), this);

  @Override
  public void onReceive(Object msg) throws Exception {
    if ( msg instanceof StartTickingMessage ) {
      log.info("Starting");
      ActorRef tocker = ((StartTickingMessage)msg).getDest();
      tocker.tell(new TockMessage(), getSelf());
    } else if ( msg instanceof TickMessage ) {
      log.info("Tick");
      ActorRef sender = getSender();
      sender.tell(new TockMessage(), getSelf());
    }
  }
}
```


A Simple Example

- The "Tock" Actor

```
public class TockActor extends UntypedActor {
  LoggingAdapter log = Logging.getLogger(
                                getContext().system(), this);

  @Override
  public void onReceive(Object msg) throws Exception {
    if ( msg instanceof TockMessage ) {
      log.info("Tock");
      ActorRef sender = getSender();
      sender.tell(new TickMessage(), getSelf());
    }
  }

}
```

A Simple Example

- The driver application

```
public class TickTockProg {

  public static void main(String[] args) {
    ActorSystem aSystem = ActorSystem.create("TickTock");
    ActorRef ticker = aSystem.actorOf(new Props(TickActor.class),
                                                    "ticker");
    ActorRef tocker = aSystem.actorOf(new Props(TockActor.class),
                                                    "tocker");

    StartTickingMessage msg = new StartTickingMessage(tocker);
    ticker.tell(msg, null);

    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) { }
    aSystem.shutdown();
  }
}
```

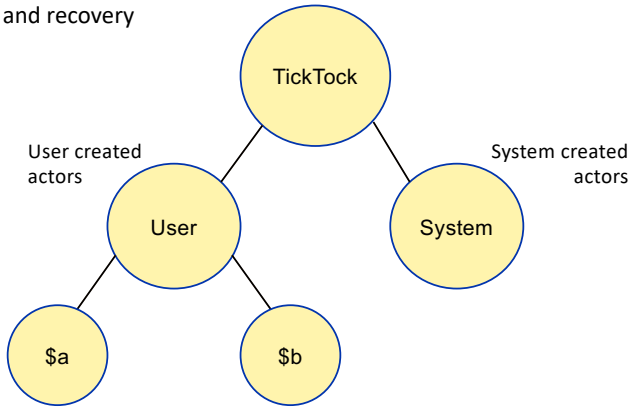

A Simple Example

- Running the application

```
[INFO] [09/03/2013 21:09:44.246] … [akka://TickTock/user/ticker] Starting
[INFO] [09/03/2013 21:09:44.247] … [akka://TickTock/user/tocker] Tock
[INFO] [09/03/2013 21:09:44.449] … [akka://TickTock/user/ticker] Tick
[INFO] [09/03/2013 21:09:44.650] … [akka://TickTock/user/tocker] Tock
[INFO] [09/03/2013 21:09:44.852] … [akka://TickTock/user/ticker] Tick
[INFO] [09/03/2013 21:09:45.053] … [akka://TickTock/user/tocker] Tock
[INFO] [09/03/2013 21:09:45.254] … [akka://TickTock/user/ticker] Tick
[INFO] [09/03/2013 21:09:45.456] … [akka://TickTock/user/tocker] Tock
[INFO] [09/03/2013 21:09:45.657] … [akka://TickTock/user/ticker] Tick
[INFO] [09/03/2013 21:09:45.858] … [akka://TickTock/user/tocker] Tock
[INFO] [09/03/2013 21:09:46.060] … [akka://TickTock/user/ticker] Tick
[INFO] [09/03/2013 21:09:46.261] … [akka://TickTock/user/tocker] Tock
…
```

Actor Application Structure and Naming

- Actors exist in a hierarchy
  - Important for error handling and recovery
- Pathname identifies individual actors



Request/Response Operation

- Actor communication encouraged to be asynchronous
  - "fire and forget"
  - no implicit reply
- Request/response communications possible
  - use `ask` method rather than `tell` method
  - still asynch under the hood
- Leverages Futures for handling replies

Request/Response Example

- Actor generates and sends a random `Int` value between 0 and 100

```
public class GetRandomInt { }
```

```
public class RandomNumActor extends UntypedActor {

  LoggingAdapter log = Logging.getLogger(
                                getContext().system(), this);
  java.util.Random rGen = new java.util.Random();

  @Override
  public void onReceive(Object msg) throws Exception {
    if ( msg instanceof GetRandomInt ) {
      int rNum = Math.abs(rGen.nextInt() % 100);
      log.info("Returning: " + rNum);
      ActorRef sender = getSender();
      sender.tell(rNum, null);
    }
  }
}
```


Request/Response Example

- Send request and handle response as `Future<Int>`

```
public class RandomActorProg {

  public static void main(String[] args) {
    ActorSystem aSystem = ActorSystem.create("RandomInt");
    ActorRef rGen = aSystem.actorOf(
                           new Props(RandomNumActor.class), "rGen");
    Timeout t = new Timeout(Duration.create(5, TimeUnit.SECONDS));

    Future<Object> response = ask(rGen, new GetRandomInt(), 1000);
    try {
      Integer rInt = (Integer)Await.result(response, t.duration());
      System.out.println("Got result: " + rInt);
      Thread.sleep(2000);
    } catch (Exception e) { }
    aSystem.shutdown();
  }
}
```

Request/Response Example

- Asynchronous operation
  - Use callback on `Future`

```
…
    Future<Object> response = ask(rGen, new GetRandomInt(), t);
    response.onSuccess(new OnSuccess<Object>() {
        @Override
        public final void onSuccess(Object t) {
          log.info("Got Result: " + t);
        }
    }, aSystem.dispatcher());
…
```


Additional Akka Features

- Higher Level APIs
  - Streams
  - HTTP
- "Let it crash" failure management
  - based on hierarchical actor structure
  - highly flexible recovery
- Dynamic reconfiguration of actors
  - changing behaviour while application is running
- Flexible dispatching of requests to actors
  - "routers"
- Clustering support