

A Java actor library for parallel execution

Modernize common concurrency patterns with μJavaActors, a lightweight Java actor library

Barry A. Feigenbaum Ph.D.
(barryfeigenbaum@gmail.com)
Software Engineer
Dell

Skill Level: Intermediate

Date: 30 May 2012

The Java™ platform doesn't currently support actor concurrency, but you can still use actors in your Java code. In this article, Barry Feigenbaum introduces the μJavaActors library: a lightweight, Java-based actor package for highly parallel execution in traditional Java applications. This tutorial includes complete source code for the μJavaActors library and hands-on examples using actors for Java standby patterns like Command, Producer/Consumer, and Map/Reduce.

To act, or not to act? That is the question!

Even with concurrency updates in Java 6 and Java 7, the Java language doesn't make parallel programming particularly easy. Java threads, `synchronized` blocks, `wait/notify`, and the `java.util.concurrent` package all have their place, but Java developers pressed to meet the capacity of multi-core systems are turning to techniques pioneered in other languages. The actor model is one such technique, implemented in Erlang, Groovy, and Scala. For developers who want to experiment with actors but continue writing Java code, this article presents the μJavaActors library.

Three more actor libraries for the JVM

See "[Table 1: Comparing JVM actor libraries](#)" for a quick overview of three popular actor libraries for the JVM as compared to μJavaActors.

The [μJavaActors library](#) is a compact library for implementing actor-based systems on the Java platform (μ represents the Greek letter Μμ, which signifies "micro"). In this article, I use μJavaActors to find out how actors work in common design patterns such as Producer/Consumer and Map/Reduce.

You can [download the source code](#) for the `µJavaActors` library at any time.

Actor concurrency on the Java platform

What's in a name? An Actor by any other name would work as well!

Actor-based systems make parallel processing easier to code by implementing a *message-passing scheme*. In this scheme, each actor in the system can receive messages; perform actions requested by the messages; and send messages to other actors, including themselves, to perform complex sequences of operations. All messages between actors are *asynchronous*, meaning that the sender continues processing before any reply is received. An actor's life may be thus spent in an indefinite loop of receiving and processing messages.

When multiple actors are used, independent activities are easily distributed across multiple threads (and thus processors) that can execute messages in parallel. In general, each actor processes messages on a separate thread, allowing for parallel execution up to the number of actors. Some actor systems assign threads to actors statically; others, such as the system introduced in this article, assign them dynamically.

Introducing `µJavaActors`

`µJavaActors` is a simple Java implementation of an actor system. At approximately 1,200 lines of code, `µJavaActors` is small but powerful. In the following exercises, you will learn how to use `µJavaActors` to create and manage actors dynamically and deliver messages to them.

`µJavaActors` is built around three core interfaces:

- **Message** is a message sent between actors. `Message` is a container for three (optional) values and some behavior:
 - `source` is the sending actor.
 - `subject` is a string defining the meaning of the message (also known as a *command*).
 - `data` is any parameter data for the message; often a map, list, or array. Parameters can be data to process and/or other actors to interact with.
 - `subjectMatches()` checks to see if the message subject matches a string or regular expression.

The default message class for the `µJavaActors` package is `DefaultMessage`.

- **ActorManager** is a manager of actors. It is responsible for allocating threads (and thus processors) to actors to process messages. `ActorManager` has the following key behaviors or characteristics:
 - `createActor()` creates an actor and associates it with this manager.
 - `startActor()` starts an actor.

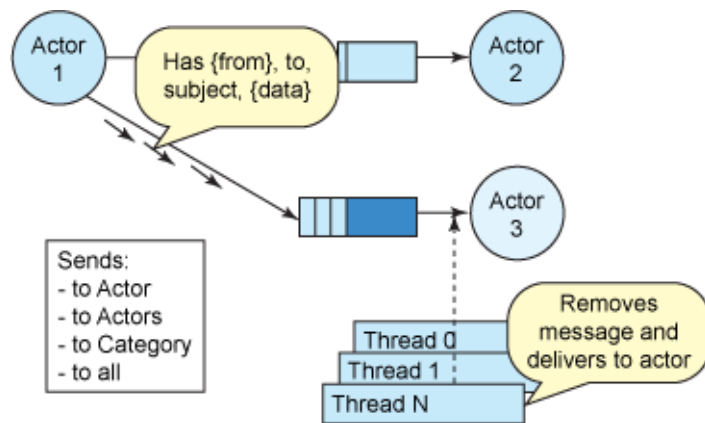
- `detachActor()` stops an actor and disassociates it from this manager.
- `send()/broadcast()` sends a message to an actor, a set of actors, any actor for a category, or all actors.

In most programs, there is a single `ActorManager`, although multiple are allowed if you want to manage multiple thread and/or actor pools. The default implementation of this interface is `DefaultActorManager`.

- **Actor** is a unit of execution that processes messages one at a time. `Actors` have the following key behaviors or characteristics:
 - Each actor has a `name`, which must be unique per `ActorManager`.
 - Each actor belongs to a `category`; categories are a means to send messages to one member of a group of actors. An actor can belong to only one category at a time.
 - `receive()` is called whenever the `ActorManager` can provide a thread to execute the actor on. It is called only when a message for the actor exists. To be most effective, an actor should process messages quickly and not enter long waits (such as for human input).
 - `willReceive()` allows the actor to filter potential message subjects.
 - `peek()` allows the actor and others to see if there are pending messages, possibly for select subjects.
 - `remove()` allows the actor and others to remove or cancel any yet unprocessed messages.
 - `getMessageCount()` allows the actor and others to get the number of pending messages.
 - `getMaxMessageCount()` allows the actor to limit how many pending messages are supported; this method can be used to prevent runaway sends.

Most programs have many actors, often of different types. Actors can be created at the start of a program or created (and destroyed) as a program executes. The [actor package](#) in this article includes an abstract class called `AbstractActor`, on which actor implementations are based.

Figure 1 shows the relationship between actors. Each actor can send messages to other actors. The messages are held in a message queue (also known as a *mail box*; conceptually one per actor) and when the `ActorManager` sees that a thread is available to process a message, the message is removed from the queue and delivered to the actor running under a thread to process that message.

Figure 1. The relationship between actors

Parallel execution with μ JavaActors

The play's the thing!

Now you are ready to begin using μ JavaActors for parallel execution. You'll start by creating a set of actors. These are simple actors in that all they do is delay for a small amount of time and send messages to other actors. The effect is to create a storm of messages, which quiets down over time and eventually stops. In the following demonstration, you will first see how to create the actors, then how they are gradually dispatched to process the messages.

There are two message types:

- `initialization` (`init`) causes the actor to initialize. Sent only once per actor.
- `repeat` causes the actor to send $N-1$ messages, where N is an incoming message parameter.

`TestActor` in Listing 1 implements abstract methods inherited from `AbstractActor`. The `activate` and `deactivate` methods inform the actor about its lifetime; nothing additional is done in this example. The `runBody` method is called when the actor is first created, before any messages are received. It is typically used to bootstrap the first messages to the actor. The `testMessage` method is called when the actor is about to receive a message; here the actor can reject or accept the message. In this case, the actor uses the inherited `testMessage` method to test for acceptance; thus all messages are accepted.

Listing 1. `TestActor`

```
class TestActor extends AbstractActor {

    @Override
    public void activate() {
        super.activate();
    }

    @Override
    public void deactivate() {
        super.deactivate();
    }
}
```

```

@Override
protected void runBody() {
    sleeper(1); // delay up to 1 second
    DefaultMessage dm = new DefaultMessage("init", 8);
    getManager().send(dm, null, this);
}

@Override
protected Message testMessage() {
    return super.testMessage();
}

```

The `loopBody` method, shown in Listing 2, is called when the actor receives a message. After a brief delay to simulate some generic processing, the message is processed. If the message is "repeat" then the actor starts the process of sending N-1 more messages, based on the `count` parameter. The messages are sent to a random actor by invoking the actor manager's `send` method.

Listing 2. `loopBody()`

```

@Override
protected void loopBody(Message m) {
    sleeper(1);
    String subject = m.getSubject();
    if ("repeat".equals(subject)) {
        int count = (Integer) m.getData();
        if (count > 0) {
            DefaultMessage dm = new DefaultMessage("repeat", count - 1);
            String toName = "actor" + rand.nextInt(TEST_ACTOR_COUNT);
            Actor to = testActors.get(toName);
            getManager().send(dm, this, to);
        }
    }
}

```

If the message is "init" then the actor starts the `repeat` message sequence by sending two sets of messages to either randomly selected actors or an actor of the common category. Some messages can be processed immediately (actually as soon as the actor is ready to receive them and a thread is available); others must wait until a few seconds in the future before they can run. Such delayed message processing is not critical to this example, but it can be used to implement polling for long-running processes such as waiting for user input or perhaps for a response to a network request to arrive.

Listing 3. An initialization sequence

```

else if ("init".equals(subject)) {
    int count = (Integer) m.getData();
    count = rand.nextInt(count) + 1;
    for (int i = 0; i < count; i++) {
        DefaultMessage dm = new DefaultMessage("repeat", count);
        String toName = "actor" + rand.nextInt(TEST_ACTOR_COUNT);
        Actor to = testActors.get(toName);
        getManager().send(dm, this, to);

        dm = new DefaultMessage("repeat", count);
        dm.setDelayUntil(new Date().getTime() + (rand.nextInt(5) + 1) * 1000);
        getManager().send(dm, this, "common");
    }
}

```

Otherwise, the message is inappropriate and an error is reported:

```

    else {
        System.out.printf("TestActor:%s loopBody unknown subject: %s%n",
            getName(), subject);
    }
}
}

```

The main program contains the code in Listing 4, which creates two actors in the common category and five in the default category and then starts them. `main` then waits up to 120 seconds (`sleeper` waits for its argument value times ~1000ms), periodically displaying progress messages.

Listing 4. `createActor`, `startActor`

```

DefaultActorManager am = DefaultActorManager.getDefaultInstance();
:
Map<String, Actor> testActors = new HashMap<String, Actor>();
for (int i = 0; i < 2; i++) {
    Actor a = am.createActor(TestActor.class, "common" + i);
    a.setCategory("common");
    testActors.put(a.getName(), a);
}
for (int i = 0; i < 5; i++) {
    Actor a = am.createActor(TestActor.class, "actor" + i);
    testActors.put(a.getName(), a);
}
for (String key : testActors.keySet()) {
    am.startActor(testActors.get(key));
}
for (int i = 120; i > 0; i--) {
    if (i < 10 || i % 10 == 0) {
        System.out.printf("main waiting: %d...%n", i);
    }
    sleeper(1);
}
:
am.terminateAndWait();

```

Trace output

To understand the process just executed, let's look at some tracing output from the actors. (Note that the output can be different for each execution because random numbers are used for counts and delays.) In Listing 5, you see the messages that occurred near the start of the program. The left column (in brackets) is the name of the thread that was executing. In this run, there were 25 threads available to process messages. The rest of the line is (abridged) trace output showing each message as it was received. Notice that the repeat count — that is, the parameter data — is reduced over time. (Also note that the thread name, while starting with `actor`, has nothing to do with the actor's name.)

Listing 5. Trace output: Start of program

```

[main      ] - main waiting: 120...
[actor17   ] - TestActor:actor4 repeat(4)
[actor0    ] - TestActor:actor1 repeat(4)
[actor10   ] - TestActor:common1 repeat(4)
[actor1    ] - TestActor:actor2 repeat(4)

```

```

[actor3      ] - TestActor:actor0 init(8)
[actor22     ] - TestActor:actor3 repeat(4)
[actor17     ] - TestActor:actor4 init(7)
[actor20     ] - TestActor:common0 repeat(4)
[actor24     ] - TestActor:actor0 repeat(4)
[actor0      ] - TestActor:actor1 init(3)
[actor1      ] - TestActor:actor2 repeat(4)
[actor20     ] - TestActor:common0 repeat(4)
[actor17     ] - TestActor:actor4 repeat(4)
[actor17     ] - TestActor:actor4 repeat(3)
[actor0      ] - TestActor:actor1 repeat(8)
[actor10     ] - TestActor:common1 repeat(4)
[actor24     ] - TestActor:actor0 repeat(8)
[actor0      ] - TestActor:actor1 repeat(8)
[actor24     ] - TestActor:actor0 repeat(7)
[actor22     ] - TestActor:actor3 repeat(4)
[actor1      ] - TestActor:actor2 repeat(3)
[actor20     ] - TestActor:common0 repeat(4)
[actor22     ] - TestActor:actor3 init(5)
[actor24     ] - TestActor:actor0 repeat(7)
[actor10     ] - TestActor:common1 repeat(4)
[actor17     ] - TestActor:actor4 repeat(8)
[actor1      ] - TestActor:actor2 repeat(3)
[actor17     ] - TestActor:actor4 repeat(8)
[actor0      ] - TestActor:actor1 repeat(8)
[actor10     ] - TestActor:common1 repeat(4)
[actor22     ] - TestActor:actor3 repeat(8)
[actor0      ] - TestActor:actor1 repeat(7)
[actor1      ] - TestActor:actor2 repeat(3)
[actor0      ] - TestActor:actor1 repeat(3)
[actor20     ] - TestActor:common0 repeat(4)
[actor24     ] - TestActor:actor0 repeat(7)
[actor24     ] - TestActor:actor0 repeat(6)
[actor10     ] - TestActor:common1 repeat(8)
[actor17     ] - TestActor:actor4 repeat(7)

```

In Listing 6, you see the messages that occurred near the end of the program, when the repeat counts have grown smaller. If you were watching this program execute, you would be able to observe a gradual slowing of the rate at which lines were generated; this is because the number of generated messages decreases over time. Given a sufficient waiting time, the message sending to actors would halt completely (as happened to the `common` actors shown in Listing 6). Notice that the message processing is reasonably distributed across the available threads, and that no particular actor is bound to any particular thread.

Listing 6. Trace output: End of program


```

[main        ] - main waiting: 20...
[actor0      ] - TestActor:actor4 repeat(0)
[actor2      ] - TestActor:actor2 repeat(1)
[actor3      ] - TestActor:actor0 repeat(0)
[actor17     ] - TestActor:actor4 repeat(0)
[actor0      ] - TestActor:actor1 repeat(2)
[actor3      ] - TestActor:actor2 repeat(1)
[actor14     ] - TestActor:actor1 repeat(2)
[actor5      ] - TestActor:actor4 repeat(0)
[actor14     ] - TestActor:actor2 repeat(0)
[actor21     ] - TestActor:actor1 repeat(0)
[actor14     ] - TestActor:actor0 repeat(1)
[actor14     ] - TestActor:actor4 repeat(0)
[actor5      ] - TestActor:actor2 repeat(1)
[actor5      ] - TestActor:actor4 repeat(1)
[actor6      ] - TestActor:actor1 repeat(1)

```

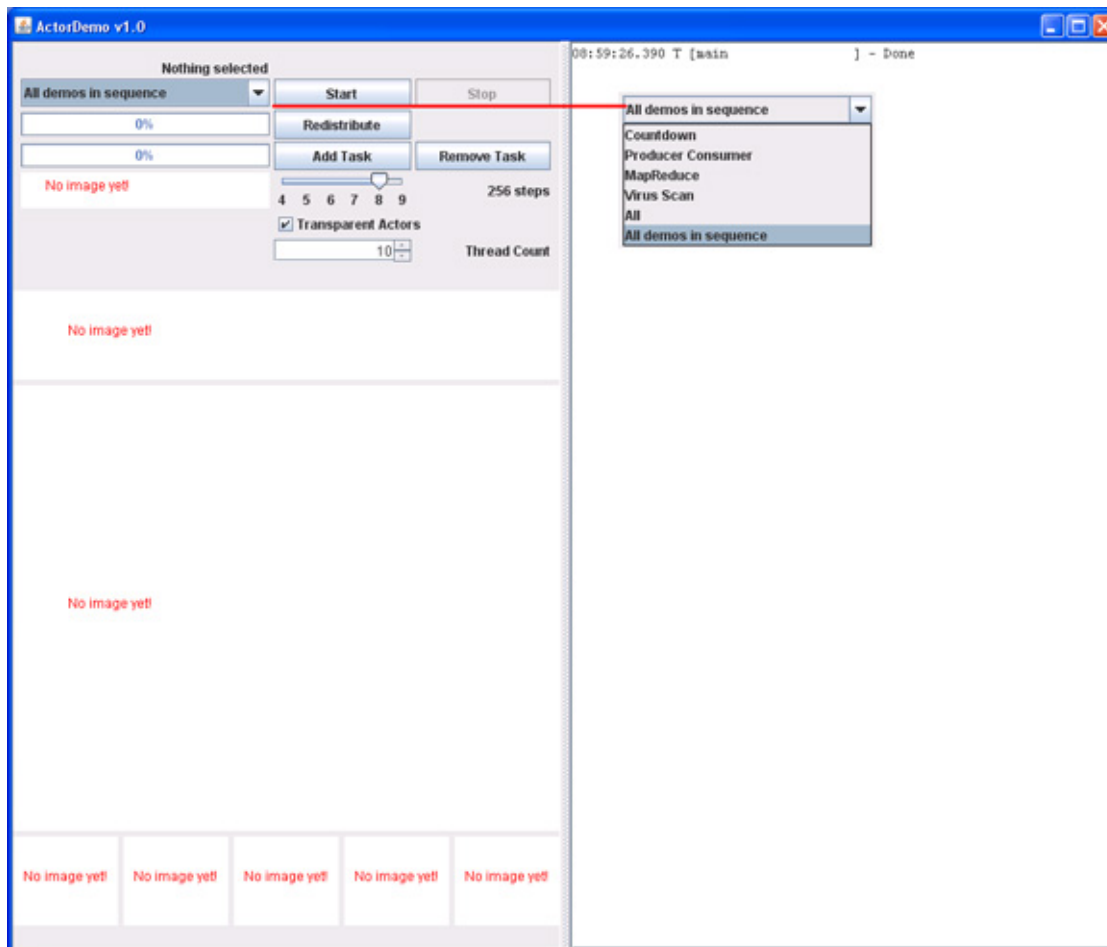
```
[actor5      ] - TestActor:actor3 repeat(0)
[actor6      ] - TestActor:actor2 repeat(1)
[actor4      ] - TestActor:actor0 repeat(0)
[actor5      ] - TestActor:actor4 repeat(1)
[actor12     ] - TestActor:actor1 repeat(0)
[actor20     ] - TestActor:actor2 repeat(2)
[main        ] - main waiting: 10...
[actor7      ] - TestActor:actor4 repeat(2)
[actor23     ] - TestActor:actor1 repeat(0)
[actor13     ] - TestActor:actor2 repeat(1)
[actor8      ] - TestActor:actor0 repeat(0)
[main        ] - main waiting: 9...
[actor2      ] - TestActor:actor1 repeat(0)
[main        ] - main waiting: 8...
[actor7      ] - TestActor:actor2 repeat(0)
[actor13     ] - TestActor:actor1 repeat(0)
[main        ] - main waiting: 7...
[actor2      ] - TestActor:actor2 repeat(2)
[main        ] - main waiting: 6...
[main        ] - main waiting: 5...
[actor18     ] - TestActor:actor1 repeat(1)
[main        ] - main waiting: 4...
[actor15     ] - TestActor:actor2 repeat(0)
[actor16     ] - TestActor:actor1 repeat(1)
[main        ] - main waiting: 3...
[main        ] - main waiting: 2...
[main        ] - main waiting: 1...
[actor4      ] - TestActor:actor1 repeat(0)
[actor6      ] - TestActor:actor2 repeat(0)
```

Simulation screenshots

It's challenging to fully grasp how the actor system behaves from the previous trace, partly because the trace format isn't all that informative. Snapshot images from the execution of a similar actor simulation let you view the same information in a graphical format. Each image shows the simulation after a fixed time period. The following video illustrates some of the Java actor processes not captured by code samples and screenshots. You can view the video inline below or on [YouTube](#), which provides an Interactive Transcript feature that allows you to select specific time signatures as you view. Simply click on the  icon below the video screen to enable it.

View the transcript [here](#).

Figure 2 shows the user interface for the simulation before any simulations have been run. Note the contents of the simulation menu displayed on the right.

Figure 2. Actor simulator before any simulations

View the full figure [here](#).

The top area of the screen displays a menu of simulations with several variations possible; unless noted, the following simulations are shown in the trace output and in the following screenshots:

- A *count-down* simulation (0:15) creates actors that count-down a value to zero and send more requests.
- A *Producer/Consumer* simulation (2:40) creates a variation on the classic Producer/Consumer concurrency problem.
- A *Map/Reduce* simulation (5:28) creates a parallel execution of a sum of squares of 1000 integers.
- A *virus scan* simulation (6:45) scans a disk directory tree for ".txt" files (to limit the number scanned) and detects suspicious content patterns. This non-CPU-bound simulation is not shown in the following screenshots, but it *is* part of the video demo.
- All simulations running concurrently, only in the video demo (8:18).

The video format shows all of these simulations running in sequence with a brief pause between them.

In addition to Start and Stop, the screenshot in Figure 2 also displays the following controls and settings. (Note that Stop does not halt the threads, so some action may occur after stopping.)

- *Redistribute* semi-randomly redistributes the actors in the actor circle (default order is creation order). This can make messages between closely grouped actors easier to see by repositioning the actors. It may also assign new colors to the actors.
- *Add Task* and *Remove Task* add or remove tasks (threads) to/from the starting pool. *Remove Task* will only remove added (not original) tasks.
- *Maximum steps* (in the [log₂](#) of the used value) limits the length of a simulation and is effective only before a simulation starts. Steps are approximately one second long.
- *Show actors as transparent* allows messages between adjacent actors to be more easily seen. Opaque actors can often be seen more easily. It is possible to change this while a simulation is running.
- *Number of threads to use spinner* is effective only before a simulation starts. Many simulations run much faster with more threads.

The display block below the controls shows the current thread usage (as an average over the past second). The large center area shows the simulation. The bottom area shows the simulation history. The area on the right shows the full simulation trace. When running, the simulation frame is configured as follows:

- In the control area are meters updated approximately each second:
 - Message acceptances per second.
 - Message completions per second.
 - Message acceptances vs. completions per second.

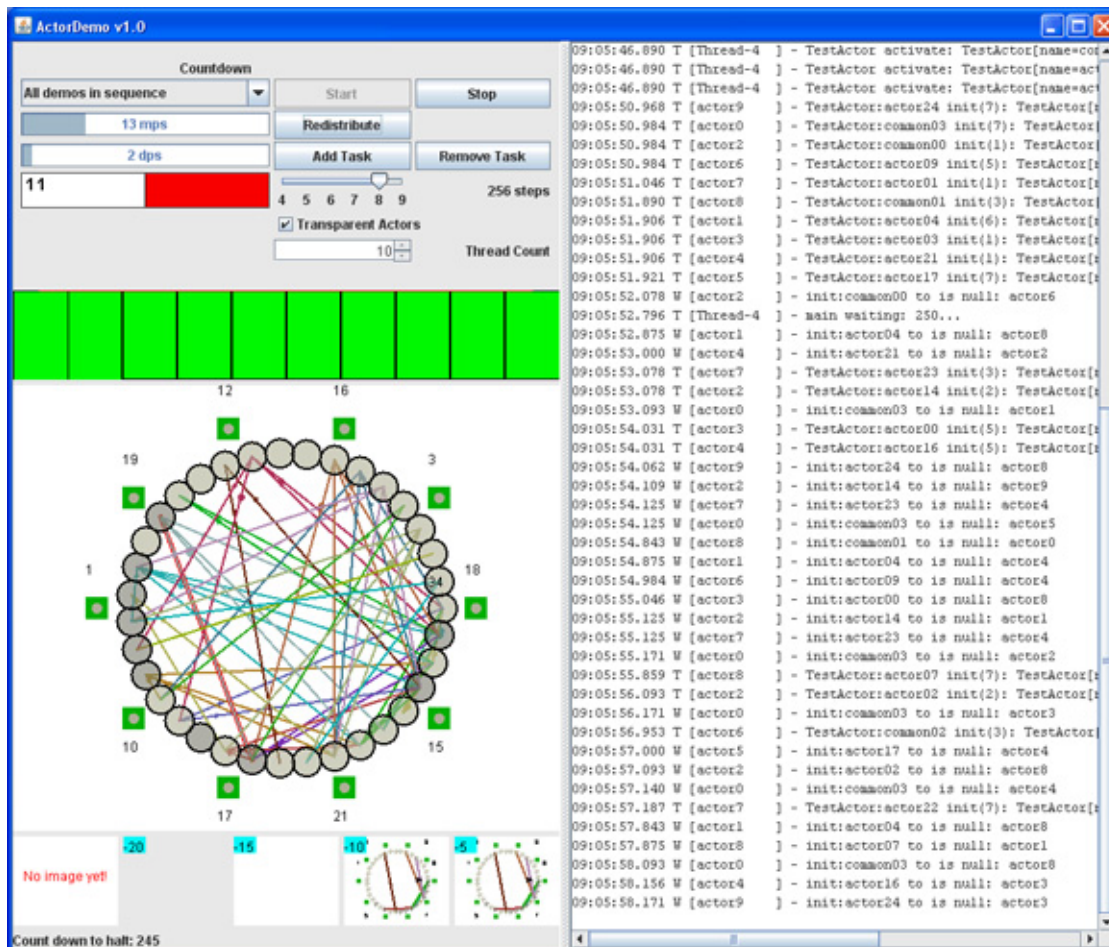
If activity shows on the right side, more messages are arriving than are being processed; eventually, the message buffers will overflow. If activity shows on the left side, more messages are being processed than are arriving; eventually, the system will go idle. A balanced system shows zero or only green levels over long time intervals.
- Above the center area is a grid of green bars; each bar represents a thread (as in the outer circle). A fully green bar means the thread is fully utilized while a fully yellow bar indicates that the thread is fully idle.
- In the center area, the outer ring of squares represents threads (10 in these simulations, 25 in the previous trace). Green threads are attached to an actor to execute a received message; the color of the dot in the center indicates the actor type. The number near the square is the actor number (ordered clockwise from 0 at the left to 360 degrees) currently assigned to this thread. Yellow threads are idle.
- The inner ring of circles represents actors; the color indicates the type (there is only one type in this first example). If the actor is busy processing a message, it is shown in a darker shade (more noticeable if non-transparent actors are

used). The lines between the circles (actors) represent messages. Any bright red lines are new messages sent in the given refresh cycle (the simulation refreshes 10 times per second); the other colors are buffered messages (sent in the past but not yet processed). Buffered lines have a small circle on the receiving end; the circle increases in size as the number of buffered messages increases.

- On the extreme right is a display of the output trace; this trace is similar but more detailed than the one previously discussed.
- At the bottom of the image is a set of smaller circles; each is a scaled down version of the main circle display at intervals in the past. This provides an easy way to see the trend of messages over time. If you observe this history, you will see that the message backlog builds up quickly and then is gradually reduced.

Figure 3 shows the [simulation](#) after approximately 10 seconds of execution. Notice the large number of pending messages, which have built up rapidly. There are 34 actors and only 10 threads, so some actors will necessarily be idle. At this moment, all of the threads are busy processing messages.

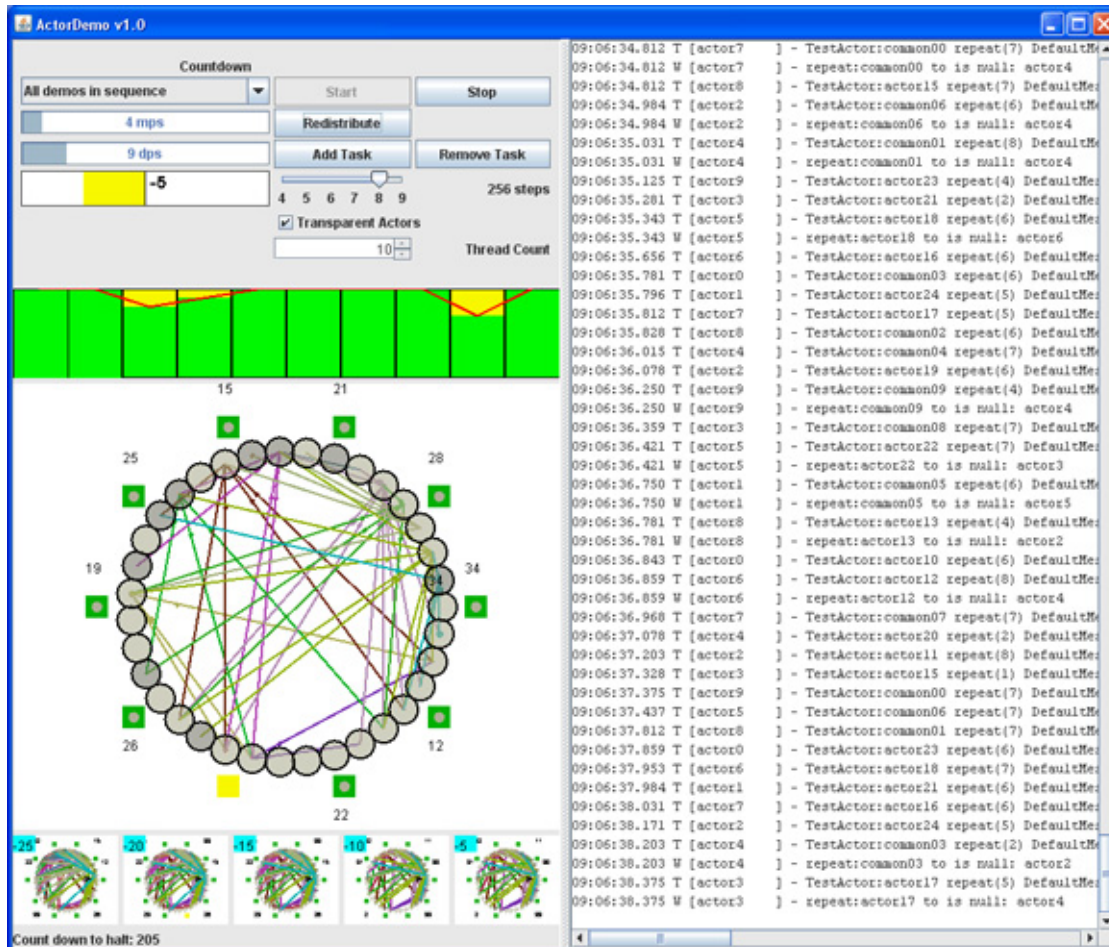
Figure 3. Count-down simulation near the start (0:15)



View the full figure [here](#).

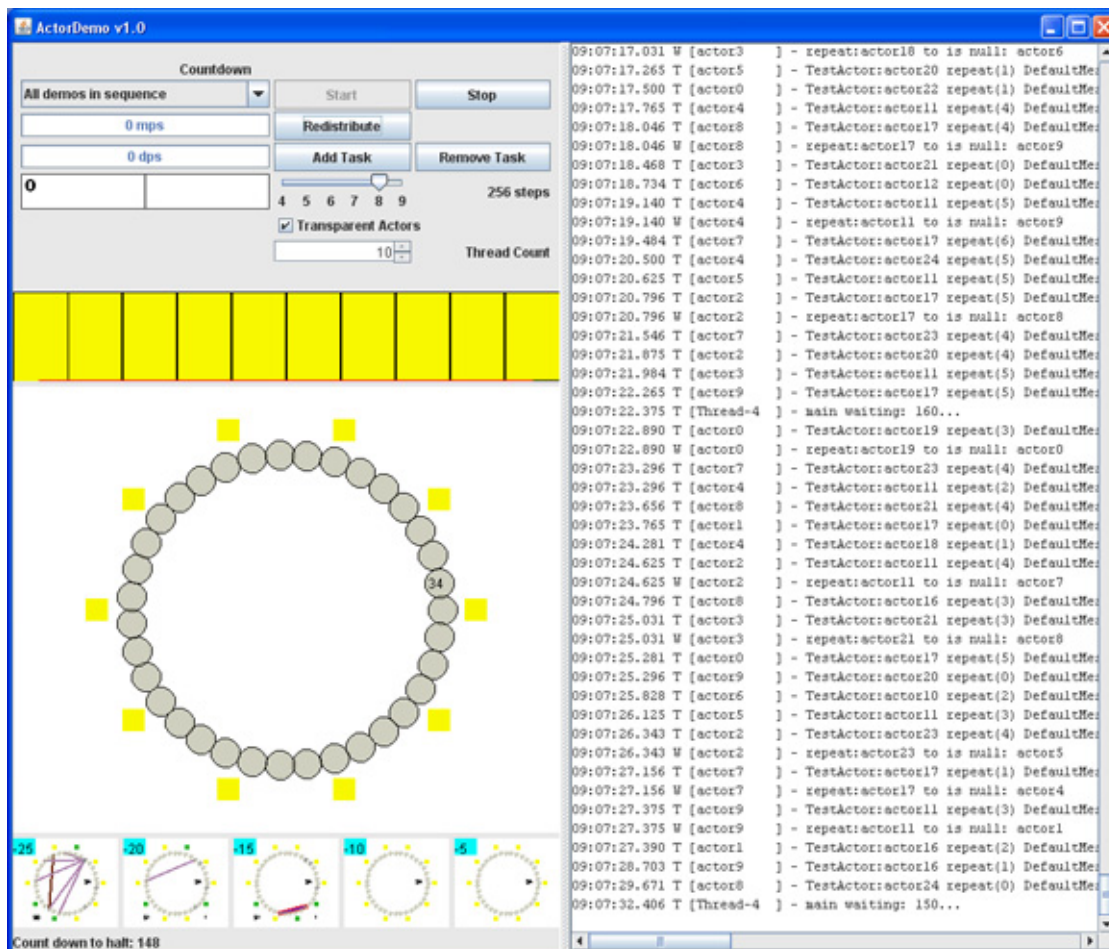
Figure 4 is the simulation after approximately 30 seconds of execution. The number of pending messages has been significantly reduced. Due to a lower message arrival rate, only some of the threads are fully busy processing messages.

Figure 4. Count down simulation in the middle



View the full figure [here](#).

Figure 5 is the simulation after approximately 90 seconds of execution. Now all the pending messages have been processed, thus all the threads are idle.

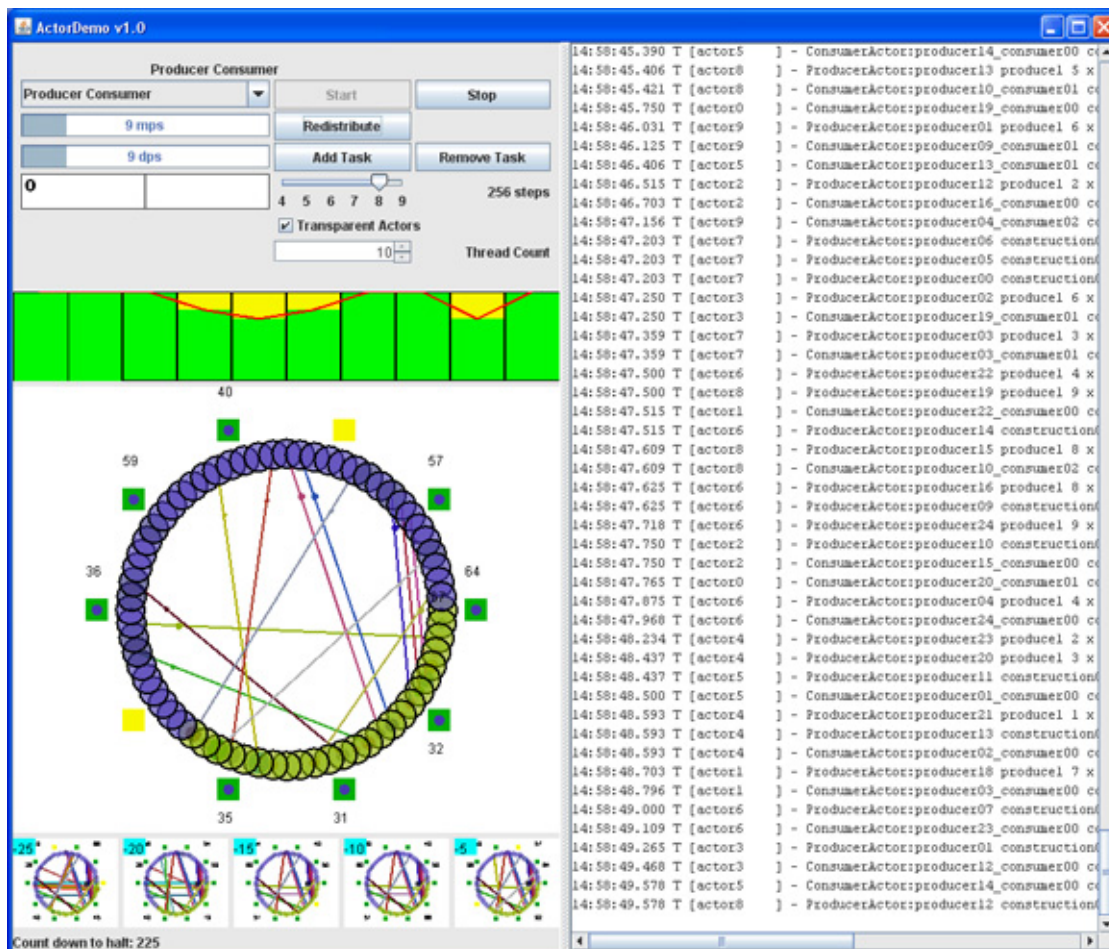
Figure 5. Count down simulation when complete

View the full figure [here](#).

Actors in a Producer/Consumer system

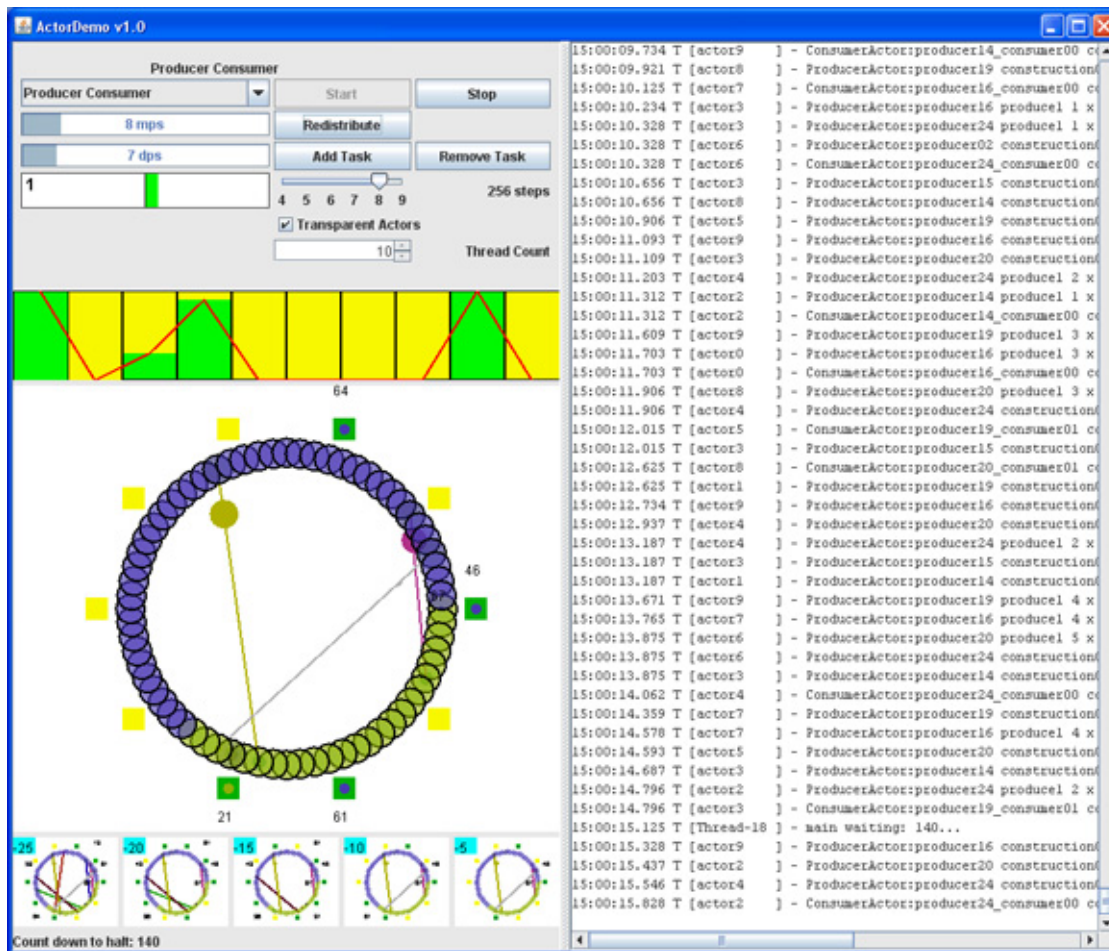
Next, let's look at a demonstration of actors in a Producer/Consumer pattern. Producer/Consumer is one of the most common synchronization patterns for multi-processor systems. In the μ JavaActors demo that follows, producer actors generate requests to consumer actors to create various items. A consumer will create these items (which takes some time), then send a completion message back to the requesting producer.

Figure 6 shows the [video simulation](#) after approximately 30 seconds of execution. Notice that the two actor types are differentiated by color. Producer actors are shown first, on the lower right side of the screen. The producers create the consumers as they run, so they appear next. The workload slowly decreases over time and the threads are mostly busy. Note that the producers complete their tasks so quickly that they rarely show up as active.

Figure 6. Producer/Consumer simulation near the start (2:40)

View the full figure [here](#).

Figure 7 shows the simulation after approximately 115 seconds of execution, which is close to the program's completion. The number of new requests and pending messages has been significantly reduced. In the video demo, you might notice some actors briefly displayed as unfilled circles; these are actors processing messages sent to themselves.

Figure 7. Producer/Consumer simulation near the end

View the full figure [here](#).

ProducerActor

Listing 7 shows the code for the producer actor from the demo. Here the "produceN" message is processed. It is converted into a "produce1" message, which the actor sends to itself. The expected response is recorded as a pending reply count for later verification.

Listing 7. Producer actor

```
public class ProducerActor extends AbstractActor {
    Map<String, Integer> expected = new ConcurrentHashMap<String, Integer>();

    @Override
    protected void loopBody(Message m) {
        String subject = m.getSubject();
        if ("produceN".equals(subject)) {
            Object[] input = (Object[]) m.getData();
            int count = (Integer) input[0];
            if (count > 0) {
                DefaultActorTest.sleeper(1); // this takes some time
                String type = (String) input[1];
                // request the consumers to consume work (i.e., produce)
                Integer mcount = expected.get(type);
            }
        }
    }
}
```

```
        if (mcount == null) {
            mcount = new Integer(0);
        }
        mcount += count;
        expected.put(type, mcount);

        DefaultMessage dm = new DefaultMessage("produce1",
            new Object[] { count, type });
        getManager().send(dm, this, this);
    }
}
```

In Listing 8, the "produce1" message is processed. If the remaining count is greater than zero, it is converted into a "construct" message and sent to a consumer. Note that this logic could have been done as a `for` loop over the count value instead of resending the "produce1" message. Resending the message often creates a better load on threads, especially if the loop body takes significant time.

Listing 8. Processing a producer request

```
} else if ("produce1".equals(subject)) {
    Object[] input = (Object[]) m.getData();
    int count = (Integer) input[0];
    if (count > 0) {
        sleep(100); // take a little time
        String type = (String) input[1];
        m = new DefaultMessage("construct", type);
        getManager().send(m, this, getConsumerCategory());

        m = new DefaultMessage("produce1", new Object[] { count - 1, type });
        getManager().send(m, this, this);
    }
}
```

In Listing 9, the "constructionComplete" message (sent by a consumer) is processed. It decrements the pending-reply count. If all is working correctly, this count will be zero for all actors and type values when the simulation completes.

Listing 9. constructionComplete

```
} else if ("constructionComplete".equals(subject)) {
    String type = (String) m.getData();
    Integer mcount = expected.get(type);
    if (mcount != null) {
        mcount--;
        expected.put(type, mcount);
    }
}
```

The "init" message is processed in Listing 10. The producer creates some consumer actors and then sends several `produceN` requests to itself.

Listing 10. Initialization

```

    } else if ("init".equals(subject)) {
        // create some consumers; 1 to 3 x consumers per producer
        for (int i = 0; i < DefaultActorTest.nextInt(3) + 1; i++) {
            Actor a = getManager().createAndStartActor(ConsumerActor.class,
                String.format("%s_consumer%02d", getName(), i));
            a.setCategory(getConsumerCategory());
            if (actorTest != null) {
                actorTest.getTestActors().put(a.getName(), a);
            }
        }
        // request myself create some work items
        for (int i = 0; i < DefaultActorTest.nextInt(10) + 1; i++) {
            m = new DefaultMessage("produceN", new Object[]
                { DefaultActorTest.nextInt(10) + 1,
                  DefaultActorTest.getItemTypes()[
                      DefaultActorTest.nextInt(DefaultActorTest.getItemTypes().length)] });
            getManager().send(m, this, this);
        }
    }

```

Listing 11 processes invalid messages:

Listing 11. Processing invalid messages

```

    } else {
        System.out.printf("ProducerActor:%s loopBody unknown subject: %s\n",
            getName(), subject);
    }
}

protected String getConsumerCategory() {
    return getName() + "_consumer";
}
}

```

ConsumerActor

The consumer actor is simple. It processes "construct" messages and sends reply messages back to the requester. The code for the consumer actor is shown in Listing 12:

Listing 12. The consumer actor

```

public class ConsumerActor extends AbstractActor {

    @Override
    protected void loopBody(Message m) {
        String subject = m.getSubject();
        if ("construct".equals(subject)) {
            String type = (String) m.getData();
            delay(type); // takes ~ 1 to N seconds

            DefaultMessage dm = new
                DefaultMessage("constructionComplete", type);
            getManager().send(dm, this, m.getSource());
        } else if ("init".equals(subject)) {
            // nothing to do
        } else {
            System.out.printf("ConsumerActor:%s loopBody unknown subject: %s\n",
                getName(), subject);
        }
    }
}

```

The production delay processed in Listing 13 is based on the type of item being constructed. From the traces, you might recall that supported item types are `widget`, `framit`, `frizzle`, `gothca`, and `splat`. Each type takes a different amount of time to construct.

Listing 13. Production delay

```
protected void delay(String type) {
    int delay = 1;
    for (int i = 0; i < DefaultActorTest.getItemTypes().length; i++) {
        if (DefaultActorTest.getItemTypes()[i].equals(type)) {
            break;
        }
        delay++;
    }
    DefaultActorTest.sleeper(DefaultActorTest.nextInt(delay) + 1);
}
```

Actors in a Producer/Consumer pattern

The Producer/Consumer demo shows that creating actor implementations is straightforward. The typical actor decodes received messages and processes them, as if in a case statement. The actual processing is trivial in this example, a mere time delay. It would be more complex in a real application, but no more so than in an implementation using standard Java synchronization techniques; typically, it would be much less complex.

Something else to note from this demo is that complex, and especially repeating, algorithms can be broken down into discrete (and often reusable) steps. Each step can be assigned a different subject name, making the case for each subject very simple. When state is carried in message parameters (such as the count-down value demonstrated previously), many actors can become stateless. Such a program is extremely easy to define and scale (more actors being added to match more threads), yet safe to run in a multithreaded environment; this is similar to using immutable values in functional-style programming.

More patterns for actors

The actors in the Producer/Consumer demo are hard-coded to a specific purpose, but that isn't your only option when coding actors. In this section, you'll learn about the use of actors in more general-purpose patterns, starting with an adaptation of the [Gang of Four Command pattern](#).

The actor in Listing 14 implements a variant of the Command pattern that should be familiar to most Java developers. Here, `CommandActor` supports two messages, "execute" and "executeStatic."

Listing 14. CommandActor

```
public class CommandActor extends AbstractActor {

    @Override
    protected void loopBody(Message m) {
        String subject = m.getSubject();
        if ("execute".equals(subject)) {
            excuteMethod(m, false);
        } else if ("executeStatic".equals(subject)) {
            excuteMethod(m, true);
        } else if ("init".equals(subject)) {
            // nothing to do
        } else {
            System.out.printf("CommandActor:%s loopBody unknown subject: %s",
                              getName(), subject);
        }
    }
}
```

The *executeMethod* method, in Listing 15, loads a parameterized class, invokes a method on that class or an instance of that class, and returns the result of the method or any exception that occurs. You can see how this simple actor can be used to run any service class available on the classpath that has the appropriate execution methods. The *id* parameter is sent by the client so it can correlate the responses back to the requests that created them. Often the replies come back in a different order than they were issued.

Listing 15. Executing a parameterized method

```
private void excuteMethod(Message m, boolean fstatic) {
    Object res = null;
    Object id = null;
    try {
        Object[] params = (Object[]) m.getData();
        id = params[0];
        String className = (String) params[1];
        params = params.length > 2 ? (Object[]) params[2] : null;
        Class<?> clazz = Class.forName(className);
        Method method = clazz.getMethod(fstatic ? "executeStatic"
                                         : "execute", new Class[] { Object.class });
        if (Modifier.isStatic(method.getModifiers()) == fstatic) {
            Object target = fstatic ? null : clazz.newInstance();
            res = method.invoke(target, params);
        }
    } catch (Exception e) {
        res = e;
    }

    DefaultMessage dm = new DefaultMessage("executeComplete", new Object[] {
        id, res });
    getManager().send(dm, this, m.getSource());
}
}
```

Actors in an Event Listener pattern

The *DelegatingActor* in Listing 16 implements a similar generic approach based on the familiar Java Event Listener (or Callback) pattern. It maps each arriving message to an *onMessage* callback on each registered listener until one callback consumes (that is, processes) the event. This delegation approach can significantly reduce the coupling between an actor system and its message processors.

Listing 16. DelegatingActor

```
public class DelegatingActor extends AbstractActor {
    private List<MessageListener> listeners = new LinkedList<MessageListener>();

    public void addMessageListener(MessageListener ml) {
        if (!listeners.contains(ml)) {
            listeners.add(ml);
        }
    }

    public void removeMessageListener(MessageListener ml) {
        listeners.remove(ml);
    }

    protected void fireMessageListeners(MessageEvent me) {
        for (MessageListener ml : listeners) {
            if (me.isConsumed()) {
                break;
            }
            ml.onMessage(me);
        }
    }

    @Override
    protected void loopBody(Message m) {
        fireMessageListeners(new MessageEvent(this, m));
    }
}
```

The `DelegatingActor` class, shown in Listing 17, depends on the `MessageEvent` and `MessageListener` classes:

Listing 17. DelegatingActor

```
/** Defines a message arrival event. */
public static class MessageEvent extends EventObject {
    private Message message;

    public Message getMessage() {
        return message;
    }

    public void setMessage(Message message) {
        this.message = message;
    }

    private boolean consumed;

    public boolean isConsumed() {
        return consumed;
    }

    public void setConsumed(boolean consumed) {
        this.consumed = consumed;
    }

    public MessageEvent(Object source, Message msg) {
        super(source);
        setMessage(msg);
    }
}

/** Defines the message arrival call back. */
public interface MessageListener {
    void onMessage(MessageEvent me);
}
```

```
}
```

An example use of `DelegatingActor` is shown in Listing 18:

Listing 18. Example use of DelegatingActor

```
public static void addDelegate(DelegatingActor da) {
    MessageListener ml = new Echo("Hello world!");
    da.addMessageListener(ml);
}

public class Echo implements MessageListener {
    protected String message;

    public Echo(String message) {
        this.message = message;
    }

    @Override
    public void onMessage(MessageEvent me) {
        if ("echo".equals(me.getMessage().getSubject())) {
            System.out.printf("%s says \"%s\".\n",
                me.getMessage().getSource(), message);
            me.setConsumed(true);
        }
    }
}
```

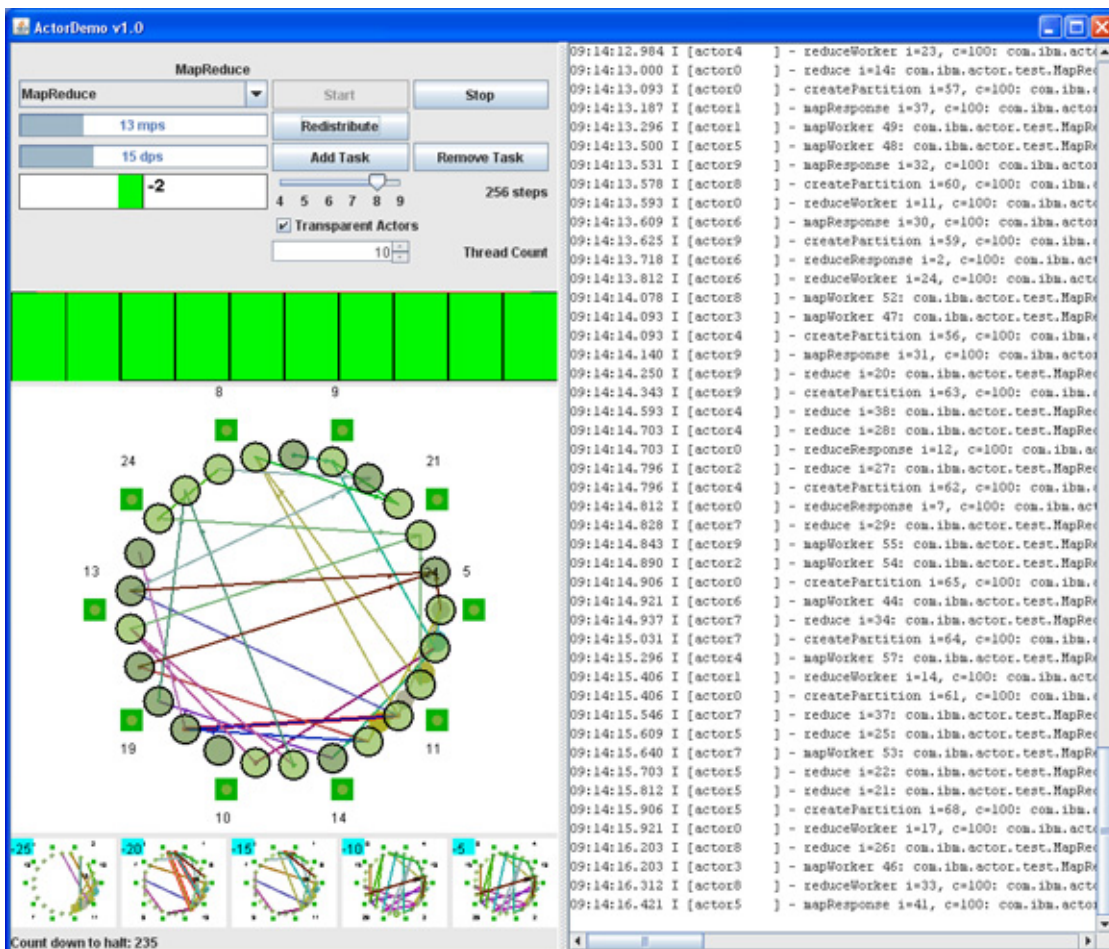
Actors in a Map/Reduce pattern

The sample actors in Listings 14 through 18 are simple and straightforward because messages are sent in only one direction. If the behavior requires feedback (such as when a process cannot proceed until all prior messages have been processed), things can get more complicated. For instance, consider a Map/Reduce implementation in which the `reduce` phase cannot be started until the `map` phase completes.

Map/Reduce is used for parallel processing on programs that process large amounts of data. In the following example, the `map` function takes a large list of items, divides it into partitions, and sends a message to map each partition. I chose to increment a message count on each map request and have the partitioned map processor send a reply that lowers the count. When the count reaches zero, all the mapping is complete and the `reduce` phase can start. Similarly, the `reduce` phase also partitions the list (again for parallelism) and sends messages to reduce the partitions. Like in the `map` phase, `reduce` also counts its messages, so that the completion of the reduction can be detected. The lists of values to process and the counts are passed in each message as parameters.

For this example, I used a single actor type with many subjects. You could also do it with multiple actor types and fewer subjects (down to one) per actor.

Figure 8 is the Map/Reduce simulation after approximately 20 seconds of execution. This is a busy phase of the processing, so the threads are occupied with processing messages.

Figure 8. Map/Reduce near the start (5:28)

View the full figure [here](#).

Map and reduce with MapReducer

Note that this implementation is pluggable; it can run any implementation of the `MapReducer` interface, shown in Listing 19

Listing 19. MapReducer

```
public interface MapReducer {
    /**
     * Map (in place) the elements of an array.
     *
     * @param values elements to map
     * @param start start position in values
     * @param end end position in values
     */
    void map(Object[] values, int start, int end);

    /**
     * Reduce the elements of an array.
     *
     * @param values elements to reduce
     * @param start start position in values
     * @param end end position in values
     * @param target place to set reduced value
     */
}
```

```

    * @param posn position in target to place the value
    */
    void reduce(Object[] values, int start, int end, Object[] target, int posn);
}

```

You could, for instance, use `MapReduceer` to compute the sum of the squares of a set of integers, as in Listing 20:

Listing 20. MapReduceer computing

```

public class SumOfSquaresReducer implements MapReduceer {
    @Override
    public void map(Object[] values, int start, int end) {
        for (int i = start; i <= end; i++) {
            values[i] = ((BigInteger) values[i]).multiply((BigInteger) values[i]);
            sleep(200); // fake taking time
        }
    }

    @Override
    public void reduce(Object[] values, int start, int end, Object[] target, int posn) {
        BigInteger res = new BigInteger("0");
        for (int i = start; i <= end; i++) {
            res = res.add((BigInteger) values[i]);
            sleep(100); // fake taking time
        }
        target[posn] = res;
    }
}

```

MapReduceActor

The Map/Reduce actor is broken down into a number of subjects, each with a simple task. You'll look at each of them in the code samples that follow. I also encourage you to [view the Map/Reduce operation](#) in the video demo; watching the simulation and then studying the code samples will give you a very clear idea of how Map/Reduce is implemented with actors. (Note that the subject order in the following listings could be broken down any number of ways; I designed the sample code with many sends in order to make the video demo more interesting.)

The `mapReduce` subject, shown in Listing 21, starts the Map/Reduce by partitioning the input array, which it does by sending `createPartition` messages. Map and reduce parameters are provided in a `MapReduceParameters` instance, which is cloned and modified as needed, then passed on. Note that time delays are not required for the operation; I added them to ensure that the simulation would be viewable in the user interface.

Listing 21. mapReduce

```

@Override
protected void loopBody(Message m) {
    ActorManager manager = getManager();
    String subject = m.getSubject();
    if ("mapReduce".equals(subject)) {
        try {
            MapReduceParameters p = (MapReduceParameters) m.getData();
            int index = 0;
            int count = (p.end - p.start + 1 + partitionSize - 1) / partitionSize;

```

```

        sleep(1000);
        // split up into partition size chunks
        while (p.end - p.start + 1 >= partitionSize) {
            MapReduceParameters xp = new MapReduceParameters(p);
            xp.end = xp.start + partitionSize - 1;
            DefaultMessage lm = new DefaultMessage("createPartition",
                new Object[] { xp, index, count });
            manager.send(lm, this, getCategory());
            p.start += partitionSize;
            index++;
        }
        if (p.end - p.start + 1 > 0) {
            DefaultMessage lm = new DefaultMessage("createPartition",
                new Object[] { p, index, count });
            manager.send(lm, this, getCategory());
        }
    } catch (Exception e) {
        triageException("mapFailed", m, e);
    }
}

```

The `createPartition` subject creates more actors and forwards the request to a worker, as shown in Listing 22. Note that the `createMapReduceActor` method has an upper bound (currently 25) on the number of actors it will create.

Listing 22. createPartition

```

    } else if ("createPartition".equals(subject)) {
        try {
            Object[] oa = (Object[]) m.getData();
            MapReduceParameters p = (MapReduceParameters) oa[0];
            int index = (Integer) oa[1];
            int count = (Integer) oa[2];
            sleep(500);
            createMapReduceActor(this);
            DefaultMessage lm = new DefaultMessage("mapWorker",
                new Object[] { p, index, count });
            manager.send(lm, this, getCategory());
        } catch (Exception e) {
            triageException("createPartitionFailed", m, e);
        }
    }
}

```

The `mapWorker` subject in Listing 23 invokes the `map` operation on its partition via the supplied `MapReducer`, then replies that the map partition is complete:

Listing 23. mapWorker

```

    } else if ("mapWorker".equals(subject)) {
        try {
            Object[] oa = (Object[]) m.getData();
            MapReduceParameters p = (MapReduceParameters) oa[0];
            int index = (Integer) oa[1];
            int count = (Integer) oa[2];
            sleep(100);
            p.mr.map(p.values, p.start, p.end);
            DefaultMessage rm = new DefaultMessage("mapResponse",
                new Object[] { p, index, count });
            manager.send(rm, this, getCategoryName());
        } catch (Exception e) {
            triageException("mapWorkerFailed", m, e);
        }
    }
}

```


The `mapResponse` subject in Listing 24 then completes the `MapReduceParameters` instance (which holds the count) and starts the reduction process:

Listing 24. `mapResponse`

```

} else if ("mapResponse".equals(subject)) {
    try {
        Object[] oa = (Object[]) m.getData();
        MapReduceParameters p = (MapReduceParameters) oa[0];
        int index = (Integer) oa[1];
        int count = (Integer) oa[2];
        sleep(100);
        p.complete();
        DefaultMessage rm = new DefaultMessage("reduce",
            new Object[] { p, index, count });
        manager.send(rm, this, getCategoryName());
    } catch (Exception e) {
        triageException("mapResponseFailed", m, e);
    }
}

```

Next, the `reduce` message forwards the request to a worker, shown in Listing 25:

Listing 25. `reduce`

```

} else if ("reduce".equals(subject)) {
    try {
        MapReduceParameters p = null;
        int index = 0, count = 0;
        Object o = m.getData();
        if (o instanceof MapReduceParameters) {
            p = (MapReduceParameters) o;
        } else {
            Object[] oa = (Object[]) o;
            p = (MapReduceParameters) oa[0];
            index = (Integer) oa[1];
            count = (Integer) oa[2];
        }
        sleep(100);
        if (p.end - p.start + 1 > 0) {
            createMapReduceActor(this);
            MapReduceParameters xp = new MapReduceParameters(p);
            DefaultMessage lm = new DefaultMessage("reduceWorker",
                new Object[] { xp, index, count });
            manager.send(lm, this, getCategory());
        }
    } catch (Exception e) {
        triageException("reduceFailed", m, e);
    }
}

```

The `reduceWorker` subject in Listing 26 invokes the `reduce` operation on its partition via the supplied `MapReducer` and replies that the reduction is complete. If all reductions are complete, it replies that the Map/Reduce operation is done.

Listing 26. `reduceWorker`

```

} else if ("reduceWorker".equals(subject)) {
    try {
        Object[] oa = (Object[]) m.getData();

```

```

    MapReduceParameters p = (MapReduceParameters) oa[0];
    int index = (Integer) oa[1];
    int count = (Integer) oa[2];
    sleep(100);
    if (index >= 0) {
        p.mr.reduce(p.values, p.start, p.end, p.target, index);
        DefaultMessage rm = new DefaultMessage("reduceResponse",
            new Object[] { p, index, count });
        manager.send(rm, this, getCategory());
    } else {
        Object[] res = new Object[1];
        p.mr.reduce(p.target, 0, count - 1, res, 0);
        DefaultMessage rm = new DefaultMessage("done",
            new Object[] { p, res[0] });
        manager.send(rm, this, getCategory());
    }
} catch (Exception e) {
    triageException("reduceWorkerFailed", m, e);
}
}

```

Next, the `reduceResponse` subject in Listing 27 completes the partition and tests for completion of all partitions and signals it:

Listing 27. reduceResponse

```

} else if ("reduceResponse".equals(subject)) {
    try {
        Object[] oa = (Object[]) m.getData();
        MapReduceParameters p = (MapReduceParameters) oa[0];
        int index = (Integer) oa[1];
        int count = (Integer) oa[2];
        sleep(100);
        p.complete();
        if (p.isSetComplete()) {
            if (count > 0) {
                createMapReduceActor(this);
                MapReduceParameters xp = new MapReduceParameters(p);
                DefaultMessage lm = new DefaultMessage("reduceWorker",
                    new Object[] { xp, -1, count });
                manager.send(lm, this, getCategory());
            }
        }
    } catch (Exception e) {
        triageException("mapResponseFailed", m, e);
    }
}
}

```

Finally, the `done` subject in Listing 28 reports the result:

Listing 28. done

```

} else if ("done".equals(subject)) {
    try {
        Object[] oa = (Object[]) m.getData();
        MapReduceParameters p = (MapReduceParameters) oa[0];
        Object res = oa[1];
        sleep(100);
        System.out.printf("**** mapReduce done with result %s", res);
    } catch (Exception e) {
        triageException("mapResponseFailed", m, e);
    }
}
}

```

Continuing the loop, the `init` subject initiates another Map/Reduce process, shown in Listing 29. Each Map/Reduce is given a different "set" name so that multiple Map/Reduces can be running at the same time.

Listing 29. initialize another Map/Reduce

```

    } else if ("init".equals(subject)) {
        try {
            Object[] params = (Object[]) m.getData();
            if (params != null) {
                Object[] values = (Object[]) params[0];
                Object[] targets = (Object[]) params[1];
                Class clazz = (Class) params[2];
                MapReduceer mr = (MapReduceer) clazz.newInstance();
                sleep(2 * 1000);
                MapReduceParameters p = new MapReduceParameters("mrSet_" + setCount++,
                    values, targets, mr, this);
                DefaultMessage rm = new DefaultMessage("mapReduce", p);
                manager.send(rm, this, getCategoryName());
            }
        } catch (Exception e) {
            triageException("initFailed", m, e);
        }
    } else {
        System.out.printf("**** MapReduceActor:%s loopBody unexpected subject: %s",
            getName(), subject);
    }
}
}

```

Map/Reduce main

The `MapReduceActor` implementation in Listing 30 creates some data values and runs a Map/Reduce on that data. It sets the partition size to 10.

Listing 30. Map/Reduce main

```

BigInteger[] values = new BigInteger[1000];
for (int i = 0; i < values.length; i++) {
    values[i] = new BigInteger(Long.toString((long)rand.nextInt(values.length)));
}
BigInteger[] targets = new BigInteger[Math.max(1, values.length / 10)];

// start at least 5 actors
DefaultActorManager am = new DefaultActorManager();
MapReduceActor.createMapReduceActor(am, 10);
MapReduceActor.createMapReduceActor(am, 10);
MapReduceActor.createMapReduceActor(am, 10);
MapReduceActor.createMapReduceActor(am, 10);
MapReduceActor.createMapReduceActor(am, 10);

DefaultMessage dm = new DefaultMessage("init", new Object[]
    { values, targets, SumOfSquaresReducer.class });
am.send(dm, null, MapReduceActor.getCategoryName());

```

Map/Reduce is one of the most universal of the divide-and-conquer design patterns. It is used from basic functional programming algorithms all the way up to massively parallel processing (of the type that Google does to build its web search engine

index). That the `µJavaActors` library can implement this advanced pattern in such a straightforward way is an indicator of its power, as well as its potential uses.

Inside the `µJavaActors` library

Manager to actor: Don't call me; I'll call you.

You've seen how actors can be used to repurpose some common object-oriented patterns. Now consider the implementation details of the `µJavaActors` system, namely the `AbstractActor` and `DefaultActorManager` classes. I'll only discuss the key methods of each class; you can see the `µJavaActors` [source code](#) for further implementation details.

AbstractActor

Every actor knows the `ActorManager` that manages it. The actor uses the manager to help it send messages to other actors.

In Listing 31, the `receive` method processes a message conditionally. If the `testMessage` method returns `null`, no message will be consumed. Otherwise, the message is removed from the actor's message queue and is processed by calling the `loopBody` method. Every concrete actor subclass must provide this method. In either case, the actor waits for more messages to come by calling the manager's `awaitMessage` method.

Listing 31. `AbstractActor` implements `DefaultActorManager`

```
public abstract class AbstractActor implements Actor {
    protected DefaultActorManager manager;

    @Override
    public boolean receive() {
        Message m = testMessage();
        boolean res = m != null;
        if (res) {
            remove(m);
            try {
                loopBody(m);
            } catch (Exception e) {
                System.out.printf("loop exception: %s\n", e);
            }
        }
        manager.awaitMessage(this);
        return res;
    }

    abstract protected void loopBody(Message m);
}
```

Each actor can implement the `willReceive` method to control which message subjects will be accepted (meaning that it will be placed in the messages list); by default, all messages with non-null subjects are accepted. Each actor can also implement the `testMessage` method to check to see if a message is available to process (that is, it is present in the messages list); by default, this oversight is implemented by using the `peekNext` method.

Listing 32. willReceive(), testMessage(), and peekNext()

```
@Override
public boolean willReceive(String subject) {
    return !isEmpty(subject);
}

protected Message testMessage() {
    return getMatch(null, false);
}

protected Message getMatch(String subject, boolean isRegExpr) {
    Message res = null;
    synchronized (messages) {
        res = peekNext(subject, isRegExpr);
    }
    return res;
}
```

Message capacity

Actors can have either an *unlimited* or a *limited* message capacity. In general, a limited capacity is better as it can help detect runaway message senders. Any clients (but typically the `ActorManager`) can add unscreened messages to an actor. Note that all access to the `messages` list is synchronized.

Listing 33. Message processing

```
public static final int DEFAULT_MAX_MESSAGES = 100;
protected List<DefaultMessage> messages = new LinkedList<DefaultMessage>();

@Override
public int getMessageCount() {
    synchronized (messages) {
        return messages.size();
    }
}

@Override
public int getMaxMessageCount() {
    return DEFAULT_MAX_MESSAGES;
}

public void addMessage(Message message) {
    synchronized (messages) {
        if (messages.size() < getMaxMessageCount()) {
            messages.add(message);
        } else {
            throw new IllegalStateException("too many messages, cannot add");
        }
    }
}

@Override
public boolean remove(Message message) {
    synchronized (messages) {
        return messages.remove(message);
    }
}
```

Message matching

Clients (in particular the actor itself) can check to see if an actor has pending messages. This can be used to process messages out of the sending order, or to

give priority to certain subjects. Message matching is done by testing the message subject for equality with a string value or for matching a regular expression against a parameter value. A `null` subject matches any message. Again, note that all access to the messages list is synchronized.

Listing 34. peekNext()

```
@Override
public Message peekNext() {
    return peekNext(null);
}

@Override
public Message peekNext(String subject) {
    return peekNext(subject, false);
}

@Override
public Message peekNext(String subject, boolean isRegExpr) {
    long now = new Date().getTime();
    Message res = null;
    Pattern p = subject != null ? (isRegExpr ? Pattern.compile(subject) : null) : null;
    synchronized (messages) {
        for (DefaultMessage m : messages) {
            if (m.getDelayUntil() <= now) {
                boolean match = subject == null ||
                    (isRegExpr ? m.subjectMatches(p) : m.subjectMatches(subject));
                if (match) {
                    res = m;
                    break;
                }
            }
        }
    }
    return res;
}
```

Lifecycle methods

Each actor has *lifecycle methods*. The `activate` and `deactivate` methods are called once per association with a particular `ActorManager`. The `run` method is also called once per association with a particular `ActorManager` and it typically bootstraps the actor by self-sending it startup messages. The `run` message starts message processing.

Listing 35. Lifecycle methods

```

@Override
public void activate() {
    // defaults to no action
}

@Override
public void deactivate() {
    // defaults to no action
}

/** Do startup processing. */
protected abstract void runBody();

@Override
public void run() {
    runBody();
    ((DefaultActorManager) getManager()).awaitMessage(this);
}
}

```

DefaultActorManager

The following fields hold an actor manager's state:

- `actors` holds all actors registered to the manager.
- `runnables` holds all actors created that have not yet had their `run` method called.
- `waiters` holds all actors waiting for messages.
- `threads` holds all threads started by the manager.

Note that the use of *LinkedHashMap* is critical (especially for the waiters list); otherwise, some actors may be starved of threads.

Listing 36. DefaultActorManager class and state

```

public class DefaultActorManager implements ActorManager {

    public static final int DEFAULT_ACTOR_THREAD_COUNT = 25;

    protected static DefaultActorManager instance;
    public static DefaultActorManager getDefaultInstance() {
        if (instance == null) {
            instance = new DefaultActorManager();
        }
        return instance;
    }

    protected Map<String , AbstractActor> actors =
        new LinkedHashMap<String , AbstractActor>();

    protected Map<String , AbstractActor> runnables =
        new LinkedHashMap<String , AbstractActor>();

    protected Map<String , AbstractActor> waiters =
        new LinkedHashMap<String , AbstractActor>();

    protected List<Thread> threads = new LinkedList<Thread>();
}

```

The `detachActor` method breaks the association between an actor and its manager:

Listing 37. Actor termination

```
@Override
public void detachActor(Actor actor) {
    synchronized (actors) {
        actor.deactivate();
        ((AbstractActor)actor).setManager(null);
        String name = actor.getName();
        actors.remove(name);
        runnables.remove(name);
        waiters.remove(name);
    }
}
```

Send methods

The send family of methods sends a message to one or more actors. Each message is first checked to see if the actor will accept it. Once the message is queued, `notify` is used to wake up a thread to process the message. When sending to a category, only one actor in the category — the one with the fewest current messages — is actually sent the message. The `awaitMessage` method simply queues the actors on the `waiters` list.

Listing 38. DefaultActorManager class processing a send

```
@Override
public int send(Message message, Actor from, Actor to) {
    int count = 0;
    AbstractActor aa = (AbstractActor) to;
    if (aa != null) {
        if (aa.willReceive(message.getSubject())) {
            DefaultMessage xmessage = (DefaultMessage)
                ((DefaultMessage) message).assignSender(from);
            aa.addMessage(xmessage);
            count++;
            synchronized (actors) {
                actors.notifyAll();
            }
        }
    }
    return count;
}

@Override
public int send(Message message, Actor from, Actor[] to) {
    int count = 0;
    for (Actor a : to) {
        count += send(message, from, a);
    }
    return count;
}

@Override
public int send(Message message, Actor from, Collection<Actor> to) {
    int count = 0;
    for (Actor a : to) {
        count += send(message, from, a);
    }
    return count;
}

@Override
public int send(Message message, Actor from, String category) {
    int count = 0;
    Map<String, Actor> xactors = cloneActors();
```



```

List<Actor> catMembers = new LinkedList<Actor>();
for (String key : xactors.keySet()) {
    Actor to = xactors.get(key);
    if (category.equals(to.getCategory()) &&
        (to.getMessageCount() < to.getMaxMessageCount())) {
        catMembers.add(to);
    }
}
// find an actor with lowest message count
int min = Integer.MAX_VALUE;
Actor amin = null;
for (Actor a : catMembers) {
    int mcount = a.getMessageCount();
    if (mcount < min) {
        min = mcount;
        amin = a;
    }
}
if (amin != null) {
    count += send(message, from, amin);
}
return count;
}

@Override
public int broadcast(Message message, Actor from) {
    int count = 0;
    Map<String, Actor> xactors = cloneActors();
    for (String key : xactors.keySet()) {
        Actor to = xactors.get(key);
        count += send(message, from, to);
    }
    return count;
}

public void awaitMessage(AbstractActor a) {
    synchronized (actors) {
        waiters.put(a.getName(), a);
    }
}
}

```

Thread pool initialization

The manager provides a pool of lower-priority daemon threads to allocate to actors to process received messages. (Note that options processing has been omitted for brevity; it is included in the supplied source.)

Listing 39. DefaultActorManager class initialization

```
protected static int groupCount;

@Override
public void initialize(Map<String, Object> options) {
    int count = getThreadCount(options);
    ThreadGroup tg = new ThreadGroup("ActorManager" + groupCount++);
    for (int i = 0; i < count; i++) {
        Thread t = new Thread(tg, new ActorRunnable(), "actor" + i);
        threads.add(t);
        t.setDaemon(true);
        t.setPriority(Math.max(Thread.MIN_PRIORITY,
            Thread.currentThread().getPriority() - 1));
    }
    running = true;
    for (Thread t : threads) {
        t.start();
    }
}
```

Each actor is dispatched by the `Runnable` implementation in Listing 40. As long as ready actors (that is, actors with pending messages) are available, they are dispatched; otherwise, the thread waits (with a variable timeout) for a message to arrive.

Listing 40. Message processing via a Runnable

```
public class ActorRunnable implements Runnable {
    public void run() {
        int delay = 1;
        while (running) {
            try {
                if (!procesNextActor()) {
                    synchronized (actors) {
                        actors.wait(delay * 1000);
                    }
                    delay = Math.max(5, delay + 1);
                } else {
                    delay = 1;
                }
            } catch (InterruptedException e) {}
            catch (Exception e) {
                System.out.printf("procesNextActor exception %s\n", e);
            }
        }
    }
}
```

The `procesNextActor` method first tests to see if any newly created actors exists, and runs one. Otherwise, it tests for a waiting actor. If there are any, it dispatches one actor to process its next message. At most, one message is processed per call. Note that all synchronization is done with the `actors` field; this reduces the possibility of a deadlock occurring.

Listing 41. Selecting and dispatching the next actor

```
protected boolean procesNextActor() {
    boolean run = false, wait = false, res = false;
    AbstractActor a = null;
    synchronized (actors) {
        for (String key : runnables.keySet()) {
```

```

        a = runnables.remove(key);
        break;
    }
}
if (a != null) {
    run = true;
    a.run();
} else {
    synchronized (actors) {
        for (String key : waiters.keySet()) {
            a = waiters.remove(key);
            break;
        }
    }
    if (a != null) {
        // then waiting for responses
        wait = true;
        res = a.receive();
    }
}
return run || res;
}

```

Terminate methods

Manager termination is requested by calling either the `terminate` or `terminateAndWait` method. `terminate` signals all threads to stop processing as soon as possible. `terminateAndWait` also waits for the threads to complete.

Listing 42. DefaultActorManager class termination

```

@Override
public void terminateAndWait() {
    terminate();
    for (Thread t : threads) {
        try {
            t.join();
        } catch (InterruptedException e) {
        }
    }
}

boolean running;

@Override
public void terminate() {
    running = false;
    for (Thread t : threads) {
        t.interrupt();
    }
    synchronized (actors) {
        for (String key : actors.keySet()) {
            actors.get(key).deactivate();
        }
    }
}
}

```

Create methods

The `create` method family constructs actors and associates them with this manager. A `create` is supplied with the class of the actor, which must have a default constructor. In addition, actors can be started at creation time or later. Note that this implementation requires all actors to extend `AbstractActor`.

Listing 43. Creating and starting actors

```

@Override
public Actor createAndStartActor(Class<? extends Actor> clazz, String name,
    Map<String, Object> options) {
    Actor res = createActor(clazz, name, options);
    startActor(res);
    return res;
}

@Override
public Actor createActor(Class<? extends Actor> clazz, String name,
    Map<String, Object> options) {
    AbstractActor a = null;
    synchronized (actors) {
        if (!actors.containsKey(name)) {
            try {
                a = (AbstractActor) clazz.newInstance();
                a.setName(name);
                a.setManager(this);
            } catch (Exception e) {
                throw e instanceof RuntimeException ?
                    (RuntimeException) e : new RuntimeException(
                        "mapped exception: " + e, e);
            }
        } else {
            throw new IllegalArgumentException("name already in use: " + name);
        }
    }
    return a;
}

@Override
public void startActor(Actor a) {
    a.activate();
    synchronized (actors) {
        String name = a.getName();
        actors.put(name, (AbstractActor) a);
        runnables.put(name, (AbstractActor) a);
    }
}

```

In conclusion

Parting is such sweet sorrow!

In this article, you learned how to use a relatively simple actor system for a variety of common Java programming scenarios and patterns. The `µJavaActors` library is both flexible and dynamic in behavior, offering a Java-based alternative to more heavyweight actor libraries like Akka.

From the code examples and video simulation, it is clear that `µJavaActors` can efficiently distribute actor message-processing across a pool of execution threads. Moreover, the user interface makes it immediately obvious if more threads are needed. The interface also makes it easy to determine which actors are starved for work, or whether some actors are overloaded.

`DefaultActorManager`, the default implementation of the `ActorManager` interface, guarantees that no actor will process more than one message at a time. It thus

relieves the actor author from dealing with any re-entrance considerations. The implementation also does not require synchronization by the actor as long as: (1) the actor only uses private (instance or method local) data and (2) message parameters are written only by message senders, and (3) read only by message receivers.

Two important design parameters of `DefaultActorManager` are the *ratio of threads to actors* and the *total number of threads to use*. There should be at least as many threads as processors on the computer, unless some are reserved for other usage. As threads can be frequently idle (for instance, when waiting on I/O), the correct ratio is often two or more times as many threads as processors. In general, there should be enough actors — really the message rate between actors — to keep the thread pool mostly busy, most of the time. (For best response, some reserve threads should be available; typically an average 75 percent to 80 percent active rate when under load is best.) This means that there usually should be many more actors than threads, as there are times when actors may not have any pending messages to process. Of course, your mileage may vary. Actors that perform actions that wait, such as waiting for a human response, will need more threads. (Threads become dedicated to the actor while waiting and cannot process other messages.)

`DefaultActorManager` makes good use of Java threads in that a thread is only associated with a particular actor while the actor is processing a message; otherwise, it is free to be used by other actors. This allows a fixed-size thread pool to service an unbounded number of actors. As a result, fewer threads need be created for a given workload. This is important because threads are very heavyweight objects that are often limited to a relatively small number of instances by the host operating system. In this, the `µJavaActors` library differentiates itself from actor systems that allocate one thread per actor; doing so effectively idles the thread if the actor has no messages to process and possibly limits the number of actor instances that can exist.

The `µJavaActors` implementation is quite efficient with regard to thread switching. If a new message exists to be processed when message processing is complete, no thread switch occurs; the new message is processed in a repeat of a simple loop. Thus, if there are at least as many messages waiting as threads, no thread becomes idle and thus no switching need occur. If sufficient processors exist (at least one per thread), then it is possible for each thread to be effectively assigned to a processor and never experience a thread switch. Threads will sleep if insufficient buffered messages exist, but this is not significant as the overhead occurs only when no work is pending.

Other actor libraries for the JVM

Other actor solutions for the JVM exist. Table 1 briefly shows how three of them compare with the `µJavaActors` library:

Table 1. Comparing JVM actor libraries with `µJavaActors`

Name	See	Description	Compared with <code>µJavaActors</code>
------	-----	-------------	--

Kilim	http://www.malhar.net/sriram/kilim/	A Java library that supports a multiple-producer, single-consumer mailbox model based on lightweight threads.	Kilim requires byte-code adjustment. In μ JavaActors, each actor is its own mailbox, so separate mailbox objects are not needed.
Akka	http://akka.io/	Attempts to emulate the pattern-matching of actors in functional languages, generally using <code>instanceof</code> type checking (whereas μ JavaActors generally uses string equality or regular expression matching).	Akka is more functional (e.g., supports distributed actors) and thus larger and arguably more complex than μ JavaActors.
GPars	http://gpars.codehaus.org/Actor	Groovy Actor library.	Similar to μ JavaActors but oriented more toward Groovy developers.

Note that some of the JVM actor solutions in Table 1 add synchronous sends (that is, the sender waits for a reply). While convenient, this can result in reduced message-processing fairness and/or possibly re-entrant calls to an actor. μ JavaActors uses POJT (plain old Java threads) and standard thread monitors, which is a more traditional implementation. Some of these other approaches have specialized support to provide their own thread models. μ JavaActors is a pure Java library; in order to use it, you need only ensure that its JAR is on the classpath. No byte-code manipulation or other special actions are required.

Enhancing μ JavaActors

There is, of course, room to improve or extend the μ JavaActors library. I conclude with some possibilities for your interest:

- Redistribution of pending messages in a category: Currently, messages are assigned round-robin when sent, but not rebalanced afterwards.
- Allow for priority-based actor execution: Currently, all actors are executed on threads of equal priority; the system would be more flexible if threads (or thread pools) of different priority existed and actors could be assigned to these threads as conditions change.
- Allow for priority messages: Currently, messages are processed typically in send-order, allowing priority processing would enable more flexible processing.
- Allow actors to process messages from multiple categories: Currently, only one category at a time is allowed.
- Optimize the implementation to reduce thread switches and thus improve potential message processing rates: This would come at the cost of more complexity.
- Distributed actors: Currently, actors must all run in a single JVM; cross-JVM execution would be a powerful extension.

Downloads

Description	Name	Size	Download method
Actor runtime and actor demo source	j-javaactors.jar	104KB	HTTP
Java source files	j-javaactors.zip	47KB	HTTP

[Information about download methods](#)

Resources

Learn

- "[Modernize common concurrency patterns with actors](#)" (Barry Feigenbaum, IBM developerWorks, May 2012) is a short video simulation of actor implementations of four common programming patterns.
- Get a [Wikipedia overview](#) of the actor model, then read the [conceptual paper](#) that introduced actor concurrency to the world (Hewitt, et al.; Proceedings of the third international joint conference on Artificial intelligence, 1973).
- Get hands-on introductions to using actors in JVM languages:
 - "[Resolve common concurrency problems with GParS](#)" (Alex Miller, September 2010)
 - "[Java development 2.0: Introducing Kilim](#)" (Andrew Glover, April 2010)
 - "[The busy Java developer's guide to Scala: Dive deeper into Scala concurrency](#)" (Ted Neward, April 2009)
- [Actor libraries and frameworks](#) (Wikipedia) lists libraries and frameworks for actor-style programming in languages without actor support.
- [Knowledge path: Actor concurrency on the Java platform](#) (developerWorks, May 2012): More resources to help you get started with actor concurrency.
- Learn more about Map/Reduce on developerWorks:
 - "[Java development 2.0: Big data analysis with Hadoop MapReduce](#)" (Andrew Glover, January 2011)
 - "[Solve cloud-related big data problems with MapReduce](#)" (Noah Gift, November 2010)
- Wikipedia provides more detail on [map](#) and [reduce](#) as higher order functions.
- [developerWorks Java technology zone](#): Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- The [Command pattern](#) is one of 23 patterns introduced in the Gang of Four classic: *Design Patterns: Elements of Reusable Object-Oriented Software* (Erich Gamma et al., Addison-Wesley, 1995).

Discuss

- [Participate in the discussion forum for this content.](#)
- Get involved in the [developerWorks community](#). Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Barry A. FeigenbaumPh.D.



Barry Feigenbaum is a software engineer currently working at Dell and previously at IBM and Amazon. He is a Sun (now Oracle) Certified Java Programmer, Developer and Architect. Barry has authored several other developerWorks articles and presented at conferences such as JavaOne, as well as authoring several technical books. He holds a Ph.D. in Computer Engineering.

© Copyright IBM Corporation 2012

(www.ibm.com/legal/copytrade.shtml)

Trademarks

(www.ibm.com/developerworks/ibm/trademarks/)