Message Passing

With message passing, processes share *channels*, which is an abstraction of a physical communication network providing a communication path between processes. Channels are accessed via 2 kinds of primitives: **send** and **receive**. To initiate a communication, processes sends a message to a channel; another processes acquires the message by receiving from the channel. Communication is accomplished since data flows from the sender to the receiver, and synchronization is accomplished since a message cannot be read before it is send.

When message passing is used, channels are typically the only objects processes share. This absence of shared variables implies that processes do not need to share memory; in particular, they can be distributed among processors. For this reason, concurrent programs that use message passing are called distributed programs. They can be executed on centralized processors, just as any other concurrent program.

The way channels are provided and named vary. For example, channels can be global to processes, be connected to receivers, or be connected to a single sender and receiver. Channels can provide either uni-directional or bi-directional information flow. Communication can be synchronous, or asynchronous.

There are 4 basic kinds of processes in a distributed program: filters, clients, servers, and peers. A filter is a data transformer, receiving streams of data, transforms this data in some way, and sends the results to its output channels.

A client is a triggering process, a server is a reactive process. Clients make requests that trigger reactions from servers.

A peer is one of a collection of identical process that interact to provide a service or solve a problem. For example, two peers might manage a copy of a replicated file and interact to keep both copies consistent, or they might cooperate to solve part of a parallel programming problem.

Because each process has direct access to only to its own variables, one of the challenges of designing distributed programs is maintaining or determining a global state.

Programming Notation

In JR, the notion of channels (a communication path between processes) is provided through the use of operations. Processes **send** messages to and **receive** messages from operations. Operations are therefore queues of messages. In this role an operation has no corresponding method. Invocations of the method are serviced by **receive** statements; the process executing the **receive** statement delays if no invocation is present.

A **send** invocation of an operation serviced by **receive** statements causes the invocation to be appended to the message queue. The invoker continues immediately after the invocation has been sent.

A **receive** statement specifies an operation and gives a list of zero or more variables separated by commas. It has the following form:

```
        receive op_expr(variable, variable,...);
```

The *op_expr* is any expression that evaluates to an operation: either the operation name or an operation capability.

The variables take the values of the first message in the queue; the variables must have been declared in the scope of the receive statement.

As an example, consider the three process system below. The program uses two operations, `stream1` and `stream2`. The first process sends its numbers (the numbers are ordered), including the end of stream marker `EOS` to `stream1`, the second to `stream2`. The `merge` process first gets a number from each stream. It executes the body of the loop as long as one of the two numbers is not `EOS`. The if statement compares the two numbers, outputs the smaller, and receives the next number in the stream from which the smaller number came. If one stream ends before the other, `v1` or `v2` will be `EOS`. Since `EOS` is larger than any other number in the stream, numbers from the non-terminated stream will be consumed until it is also finished. The loop terminates when both streams have been entirely consumed.

```
public class StreamMerge {

   private static final int EOS = Integer.MAX_VALUE;// end of stream marker
   private static op void stream1(int);
   private static op void stream2(int);

   private static process one {
      int y = 38;
      send stream1(y);
      send stream1(56);
      send stream1(77);
      send stream1(83);
      send stream1(EOS);
   }

   private static process two {
      int y = 14;
      send stream2(y);
      send stream2(44);
      send stream2(98);
      send stream2(EOS);
   }

   private static process merge {
      int v1, v2;
      receive stream1(v1);  receive stream2(v2);
      while (v1 < EOS || v2 < EOS) {
         if (v1 <= v2) {
            System.out.println(v1);
            receive stream1(v1);
         }
         else { // v1 > v2
            System.out.println(v2);
            receive stream2(v2);
         }
      }
      System.out.println(EOS);
   }
   public static void main(String [] args) {}
}
```

Message order

```
import edu.ucdavis.jr.JR;

public class Order {

   private static op void a(int);
   private static op void b(int);

   private static process one {
      send a(1);
      send b(2);
   }


   private static process two {
      int x;
      receive a(x);
      send b(3);
   }


   private static process three {
      int x;
      receive b(x);
      System.out.println("three1 "+x);
      receive b(x);
      System.out.println("three2 "+x);
   }


   public static void main(String [] args) {
   }

}
```

Semaphores – take two

In JR, semaphores are really just abbreviations for operations and send and receive statements. The mapping is summarized in the following table:

Semaphore Primitive	Corresponding Message Passing Primitive
`sem s`	`op void s()`
`P(s)`	`receive s()`
`V(s)`	`send s()`

`sem s =` *`expr`*	`op void s();`
	`{`
	`    int v=`*`expr`*`;`
	`    for (int i=0; i < v; i++) send s();`
	`}`

```
public class CS {
   private static final int N = 20; // number of processes
   private static int x = 0;        // shared variable
   private static op void mutex();  // mutual exlusion for x

   static {
      send mutex(); // initialize mutex to "1"
   }

   private static process p( (int i = 0; i < N; i++) ) {
      non-critical section();
      // critical section
      receive mutex(); // enter critical section
      System.out.println(i);
      x = x + 1;
      send mutex();    // leave critical section
      // non-critical section
   }

   public static void main(String [] args) {
   }
}
```

Shared Operations

Operations can be shared by multiple processes, as illustrated in the semaphore example above. When an operation is shared, processes compete for the operation's invocations. Processes obtain the invocations in a first-come, first-served basis.

A useful application of shared operations is for server work queues. A share operation can be used to permit multiple servers to service the same work queue. As an example of using shared operations for a server work queue, here is a method for finding the area under a curve: adaptive quadrature. Given are a continuous, non-negative function f(x) and two values l and r, l < r. The problem is to compute the area bounded by f(x), the x axis, and the vertical lines through l and r. This corresponds to approximating the integral of f(x) from l to r. The following code uses a shared operation, called `bag` (bag of tasks); each task represents a sub-interval over which the integral of f(x) is to be approximated.

```
public class AQ {
   private static final int N = 20; // number of workers
   private static op void bag(double, double, double, double);
   private static op void result(double);
   private static final double Epsilon = 0.001; // precision

   private static double f(double x) {
      return Math.pow(x,3.0);
   }

   private static double area = 0.0;
   private static process administrator {
      double part;
      double LEFTBOUND = 0.0, RIGHTBOUND = 4.0;  // hardcoded boundaries
      send bag(LEFTBOUND, RIGHTBOUND, f(LEFTBOUND), f(RIGHTBOUND));
      while (true) {
         System.out.println("admin top of loop, area="+area);
         receive result(part);
         area += part;  // sum up the parts
      }
   }

   private static process worker( (int i = 1; i <= N; i++) ) {
      double a, b, m, fofa, fofb, fofm;
      double larea, rarea, tarea, diff;
      while (true) {
         receive bag(a,b,fofa,fofb);
         m = (a+b)/2;  fofm = f(m);
         // compute larea, rarea, and tarea using trapezoidal rule
         larea = (m - a) * (fofa + fofm) / 2.0;
         rarea = (b - m) * (fofm + fofb) / 2.0;
         tarea = (b - a) * (fofa + fofb) / 2.0;
         diff  = Math.abs(tarea - (larea + rarea));
         if (diff <= Epsilon) { /* diff small enough */
            send result(larea + rarea);
         }
         else { // diff > Epsilon /* diff too large */
            send bag(a, m, fofa, fofm);
            send bag(m, b, fofm, fofb);
         }
      }
   }
   public static void main(String [] args) {
      try { JR.registerQuiescenceAction(done);
      } catch (edu.ucdavis.jr.QuiescenceRegistrationException e){e.printStackTrace();}
   }
   private static op void done() {System.out.println("computed area = "+area);}
}
```

```
import edu.ucdavis.jr.JR;
public class Cap1 {
  private static op void f(double);
  private static op void g(double);
  public static void main(String [] args) {
    cap void (double) y, z;
    // make y point to one of f or g, and z point to other
    if (args.length > 0) { y = f; z = g; }
    else                 { y = g; z = f; }
    // invoke what y points to and then what z points to
    send y(4.351);
    send z(8.21);
  }
  private static process pf {
    double d;
    receive f(d);
    System.out.println("pf got "+d);
  }
  private static process pg {
    double d;
    receive g(d);
    System.out.println("pg got "+d);
  }
}
```

```
import edu.ucdavis.jr.JR;
public class Cap2 {
  private static op void f(double);
  private static op void g(double);
  public static void main(String [] args) {
    cap void (double) y, z;
    // make y point to one of f or g, and z point to other
    if (args.length > 0) { y = f; z = g; }
    else                 { y = g; z = f; }
    send f(4.351);
    send g(8.21);
    double d;
    receive y(d);
    System.out.println("rcv1 got "+d);
    receive z(d);
    System.out.println("rcv2 got "+d);
  }
}
```

```
import edu.ucdavis.jr.JR;
public class Cap3 {
  public static op void getcap(cap void (double));
  public static process p {
    op void f(double);
    send getcap(f);
    double d;
    receive f(d);
    System.out.println("rcv got "+d);
  }
  public static process q {
    cap void (double) y;
    receive getcap(y);
    send y(5.78);
  }
  public static void main(String [] args) {
  }
}
```

Client - Server Models

```
import edu.ucdavis.jr.JR;
public class Model1 {
  private static op void request(char);
  private static op void results(double);

  private static process client {
    send request('w');
    // possibly perform some other work
    double d;
    receive results(d);
    System.out.println(" got " + d);
  }
  private static process server {
    while (true) {
      char data;  double ans;
      receive request(data);
      // handle request; put answer in ans
      ans = 222.8;
      send results(ans);
    }
  }
  public static void main(String [] args) {
  }
}
```

```
import edu.ucdavis.jr.JR;
public class Model2 {
  private static final int N = 20; // number of client processes
  private static op void request(int, char);
  private static cap void (double) results[] =
    new cap void (double)[N];
  static {
    for (int i = 0; i < N ; i++ ) {
      results[i] = new op void (double);
    }
  }

  private static process client( (int i = 0; i < N; i++) ) {
    send request(i, 'w');
    // possibly perform some other work
    double d;
    receive results[i](d);
    System.out.println(i + " got " + d);
  }
  private static process server {
    while (true) {
      int id;  char data;  double ans;
      receive request(id, data);
      // handle request; put answer in ans
      ans = id * 2.0;
      send results[id](ans);
    }
  }
  public static void main(String [] args) {
  }
}
```

```
import edu.ucdavis.jr.JR;
public class Model3 {
  private static final int N = 20; // number of client processes
  private static op void request(cap void (double), char);

  private static process client( (int i = 0; i < N; i++) ) {
    op void results(double);
    send request(results, 'w');
    // possibly perform some other work
    double d;
    receive results(d);
    System.out.println(i + " got " + d);
  }
  private static process server {
    while (true) {
      cap void (double) results_cap;  char data;  double ans;
      receive request(results_cap, data);
      // handle request; put answer in ans
      ans = data * 2.0; // not very interesting...
      send results_cap(ans);
    }
  }
  public static void main(String [] args) {
  }
}
```

```
import edu.ucdavis.jr.JR;

public class Model4  {
   private static final int N = 20; // number of client processes

   private static op void request(cap void (double), char);

   private static process client( (int i = 0; i < N; i++) ) {
      op void results(double);
      send request(results, 'w');
      // possibly perform some other work
      double d;
      receive results(d);
      System.out.println(i + " got " + d);
   }

   private static void request (cap void (double) results_cap,
                                 char data) {
      double ans;
      // handle request; put answer in ans
      ans = data * 2.0; // not very interesting...
      send results_cap(ans);
   }

   public static void main(String [] args) {
   }
}
```