JINQS: An Extensible Library for Simulating Multiclass Queueing Networks V1.0 User Guide

Tony Field

August 20, 2006

Abstract

This document summarises two Java packages (`network` and `tools`) that combine to permit the simulation of multi-class queueing networks. The packages have been designed so as to be easy to use and extend, whilst providing powerful primitives for building queueing networks compositionally and for implementing customer- and/or state-dependent routing and time delays, simultaneous resource possession etc. Additional application-specific functionality can be introduced straightforwardly by subclassing. Examples of each of the library features is presented and some details of their internal implementation is provided where this adds insight to the design. A number of complete case studies is presented at the end.

1 Disclaimer

The software described here is freely available and can be downloaded by following the instructions below. The author accepts no responsibility for problems that may arise from its use. Users may modify the source code in any way they see fit. Any suggestions for improvements, bug reports etc. will be gratefully received; please email all feedback to `ajf@doc.ic.ac.uk`.

2 Getting Started

The following instructions are for the Linux operating system. The procedure is similar for Windows, although details are not provided.

The `network` and `tools` packages, together with JavaDoc documentation, are located in a jar file called `Simulation.jar` which can be downloaded from here:

`http://www.doc.ic.ac.uk/~ajf/Software/Simulation.jar`

Updates to this code base will be made frequently and stored in the file `LatestSimulation.jar` in the same location. You are free to use the latest version, but it is not guaranteed to be consistent with this user guide. The guide will be updated when a new release is announced. A summary of the updates made in the latest version is included in `LatestSumulation.jar`.

Place the `Simulation.jar` file where you want the various packages and documentation to to end up, e.g. `/homes/me/Code`. Now extract the archive. You can optionally delete the archive itself afterwards:

```
prompt% jar xf Simulation.jar
prompt% rm Simulation.jar
```

You will now find the directory `Simulation`, which contains both the source and class directories for the `network` and `tools` packages, together with supporting JavaDoc documentation for the API. An additional source directory `examples` contains the source code for the examples given in Section 5.

```
prompt% ls
META-INF  Simulation
prompt% cd Simulation
prompt% ls
classes  html sources
prompt% ls sources/
examples  network  tools
prompt% ls sources/network/
BoxedQueue.java            OrderedQueueEntry.java
ClassDependentBranch.java  OrderedQueue.java
ClassDependentDelay.java   PreemptiveRestartNode.java
Customer.java              PreemptiveResumeNode.java
Debug.java                 PriorityQueue.java
Delay.java                 ProbabilisticBranch.java
FIFOQueue.java             ProcessorSharingNode.java
InfiniteServerNode.java    QueueingNode.java
LIFOQueue.java             Queue.java
Link.java                  RandomQueue.java
Network.java               ResourcePool.java
Node.java                  Sink.java
NullNode.java              Source.java
Ordered.java
```

and so on. You are now advised to add the location of the class files to your `CLASSPATH`. For example, in your `.cshrc` file, do something like:

```
setenv CLASSPATH BLAH:/homes/me/Code/Simulation/classes
```

where `BLAH` lists any other paths that you have already defined.

When developing code that uses these packages, you simply need to import them at the top of your `.java` files, e.g. for `MyClass.java`:

```
import network.* ;
import tools.* ;

class MyClass {
// Code for MyClass
}
```

To compile `MyClass` invoke `javac` or use your favourite IDE. For example,

```
prompt% javac MyClass.java
```

which compiles `MyClass.java`.

The next section details the use of the `network` package for modelling queueing networks, by means of simple examples. The objectives are to explain the operation of the various classes, and to show how they can be extended to add application-specific functionality. Ideally this should be read sequentially as some examples build on earlier ones. Section 4 summarises the key features of the underlying discrete-event simulation tools that the library builds on.

The archive contains the complete source code for both packages. These are locally documented and JavaDoc documentation for the API is available in the `html` directory above.

3 The Queueing `network` Package

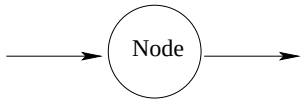
3.1 Nodes

A queueing network is a network of `Node`s. There are several node classes all of which are descended from a `Node` superclass. These can be assembled to form a queueing network and the network can then be simulated. The simulation can be repeated many times, for example to compute a confidence interval. Details of the complete assembly process will be given later. The discussion of the API, and how to extend it, will proceed bottom-up.

Before starting it is important to initialise the `Network` class, which is responsible for setting up a queueing network:

```
Network.initialise() ;
```

As a first example, here's how to build a simple internal node called "Node":

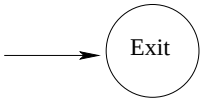



```
Node node = new Node( "Node" ) ;
```

Customers that enter this node are passed directly to the node's output. It's not a very interesting type of node, but more useful behaviour is achieved in the various `Node` subclasses.

3.2 Sinks

Here's how to build a sink node – a subclass of `Node` – called "Exit":




```
Sink exit = new Sink( "Exit" ) ;
```

If a customer enters a sink node, the customer is absorbed. All nodes have to be named explicitly.

3.3 Customers

The objects that flow around the network are `Customer`s. Here's how to build an anonymous customer:

```
Customer c = new Customer() ;
```

Customers can optionally be associated with a particular integer class:

```
Customer c1 = new Customer( 1 ) ;
Customer c2 = new Customer( 2 ) ;
```

Within a class customers can optionally have an additional integer priority:

```
Customer c10 = new Customer( 1, 0 ) ;
Customer c11 = new Customer( 1, 1 ) ;
Customer c12 = new Customer( 1, 2 ) ;
```

All customers have a unique identifier. Their creation time and current location in the network are also recorded internally. There are getter/setter methods for all customer attributes, except for the `arrivalTime` which is read-only, e.g.

```
Customer c53 = new Customer( 5, 3 ) ;
int customerClass = c53.getclass() ;
int customerPriority = c53.getPriority() ;
int id = c53.id() ;
double arrivalTime = c53.arrivalTime() ;
Node location = c53.getLocation() ;
```

It is thus possible for customers to change their class and/or priority as they progress through a network. Note the small 'c' in `getclass()`, which avoids a name clash with Java's existing 'final' method `getClass()`.

We can extend customers with additional application-specific attributes by subclassing the `Customer` class, e.g.:

```
class Patient extends Customer {
  int age ;
  public Patient( int patientClass, int age ) {
    super( patientClass ) ;
    this.age = age ;
  }
  public int getAge() {
    return age ;
  }
}
```

There is a notion of customer ordering that is embodied in the interface `Ordered`:

```
public interface Ordered {
  public boolean smallerThan( Customer e ) ;
}
```

The `Customer` class implements `Ordered`. The default ordering is based on the customer's class.

3.4 Source Nodes

Source nodes can be used to inject customers into a network with a specified inter-arrival time distribution, e.g.

```
Source source = new Source( new Exp( 4 ) ) ;
```

which builds a Poisson process with arrival rate 4. The `Exp( r )` constructor builds an exponential distribution sampler with rate parameter `r` (type `double`). This is provided in the `tools` package, described in more detail in Section 4.

The arrival process may also be batched, with the batch size distribution being specified by an additional parameter, e.g.

```
Source batchedSource = new Source( new Exp( 4 ),
                                   new Geometric( 0.2 ) ) ;
```

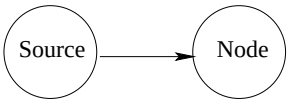
which generates batches of arrivals with a mean size of 5 customers. Internally, this generates individual customers using a protected method `buildCustomer` that may optionally be subclassed. Thus, to inject customers of some subclass of `Customer`, one simply needs to subclass `Source` and redefine this method, e.g.

```
class PatientSource extends Source {
  public PatientSource( String name, DistributionSampler d ) {
    super( name, d ) ;
  }
  protected Customer buildCustomer() {
    return new Patient( 1, 0 ) ;
  }
}
```

This injects `Patients` (see above) with fixed class and age (1 and 0 respectively), although these are easily generalised. The distribution samplers in the `tools` package all return `doubles`; these are truncated down to the nearest integer prior to being used to define the batch size. Note that a batch size of zero causes zero customers to be injected – essentially a non-arrival event.

3.5 Links

Two nodes can be joined together via a link. The simplest link routes a departing customer directly to a specified node, e.g.:




```
Link sourceToNode = new Link( node ) ;
source.setLink( sourceToNode ) ;
```

or

```
source.setLink( new Link( node ) ) ;
```

When two nodes are joined together a customer departing one node is routed automatically to the next. Internally, this is done by invoking a method `enter` defined the `Node` classs; the departing customer is passed as a parameter.

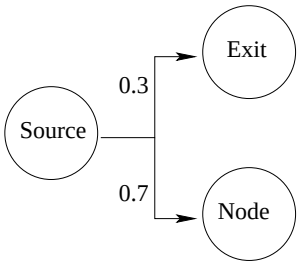
If necessary, the `enter` method can be invoked explicitly. For example, here is how route customer `c` manually to our internal node:

```
node.enter( c ) ;
```

If a network is built simply by wiring together nodes, it will not be necessary to invoke any `enter` methods explicitly as the routing of customers will be performed automatically.

3.5.1 Probabilistic Branching

Several commonly-occurring branching structures are built in as subclasses of `Link`. The simplest is a probabilistic brancher which routes (all) departing customers to one of a number of nodes dependent on associated probabilities, e.g.

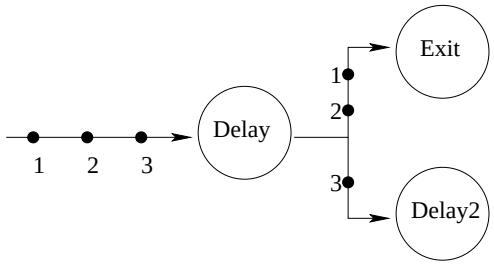



```
ProbabilisticBranch pb
  = new ProbabilisticBranch( new double[] { 0.3, 0.7 },
                             new Node[] { exit, node } ) ;
source.setLink( pb ) ;
```

There can be any number of branches; the probabilities must sum to 1.

3.5.2 Class-dependent Branching

Branching may also be class-dependent, e.g.:




```
ClassDependentBranch cb
  = new ClassDependentBranch( new int[] { 1, 2, 3 },
                              new Node[] { exit, exit, node } ) ;
source.setLink( cb ) ;
```

which routes class 1 and 2 customers to the exit node and class 3 customers to the internal node.

3.6 Infinite Server Nodes

Instances of the `Node` class are not that useful as they simply route customers from the node's input to its output. An `InfiniteServer` node (a subclass of `Node`) delays customers for an amount of time determined by a distribution sampler such as:

```
DistributionSampler e = new Exp( 1.0 ) ;
DistributionSampler w = new Weibull( 1.0, 2.0 ) ;
```

as defined in the `tools` package (Section 4.2). The simplest delay does nothing more than sample a given distribution sampler, e.g.

```
DistributionSampler e = new Exp( 1.0 ) ;
Delay d = new Delay( e ) ;
InfiniteServerNode is = new InfiniteServerNode( "IS", d ) ;
```

or

```
InfiniteServerNode is
  = new InfiniteServerNode( "IS", new Delay( new Exp( 1.0 ) ) ) ;
```

3.6.1 Class-dependent Delays

Delays can depend on the customer. For example, we may want to sample a distribution that depends on the customer's class:

```
int[] cs = { 1, 3 } ;
DistributionSampler[] ds = { new Exp( 1.0 ),
                             new Weibull( 1.0, 2.0 ) } ;
ClassDependentDelay d1 = new ClassDependentDelay( cs, ds ) ;
```

which uses an exponential sampler for class 1 customers and a Weibull sampler for class 3 customers.

3.6.2 Attribute-dependent Delays

Delays can also depend on other state variables or customer attributes defined in a `Customer` subclass. To achieve this we subclass `Delay` and redefine the method `sample`, which takes the customer as a parameter and delivers a sample delay, e.g.:

```
class PatientDelay extends Delay {
  protected double sample( Customer c ) {
    if ( (Patient)c.age < 50 )
      return 10.0 ;
    else
      return w.next() ;
  }
}
```

with the Weibull distribution sampler `w` as above. Then, for example:

```
InfiniteServer is2 = new InfiniteServer( "IS2",
                                         new PatientDelay() ) ;
```

3.7 Resource Pool Nodes

Customers may need to acquire resources in order to progress through a network. A `ResourcePool` node contains a specified integer number of resources. When all resources are busy customers queue in FIFO order. The FIFO queue has infinite capacity – see below how to make this finite and also how to change the queueing discipline.

Importantly, the resource is held until it is explicitly returned to the pool. This is the (only) difference between a resource pool node and a conventional queueing node. Once acquired the resource is held at the `ResourcePool` node for a time that is determined by a specified delay. When the customer leaves the node the resource is still held, however. For example,

```
Delay d = new Delay( new Deterministic( 0 ) ) ;
ResourcePool resPool = new ResourcePool( "Pool", d, 5 ) ;
```

In this example, customers leave the node as soon as the resource is acquired, as the local resource holding time (`d`) is zero. All resources at a `ResourcePool` node are identical in respect of the holding time distribution.

A unit of resource may be returned at any point by invoking the resource node's `releaseResource` method, e.g.:

```
resPool.releaseResource() ;
```

This will cause any waiting customer at the head of the queue to acquire the resource. Note that `ResourcePool` is a subclass of `InfiniteServer`.

The queues at a resource node have infinite capacity by default, although a finite-capacity queue can be built by supplying an additional integer capacity parameter to any queue constructor—see Section 3.13. For example, a finite-capacity FIFO queue can be built by suitably parameterising the `FIFOQueue` constructor (see below) and by passing the queue explicitly to the `ResourcePool` constructor, e.g.

```
FIFOQueue fq = new FIFOQueue( 6 ) ;
ResourcePool resPool = ResourcePool( "Res", d, 5, fq ) ;
```

With this parameterisation, customers that enter the node when the queue is full are lost and are a sent to a special `nullNode`. They can instead be routed to a nominated "loss node" by invoking the `setLossNode` method in class `ResourcePool`, e.g.

```
Node lostCustomerSink = new Node( "Losses" ) ;
ResourcePool resPool
  = new ResourcePool( "Res", 5, d, fq ) ;
resPool.setLossNode( lostCustomerSink ) ;
```

The current population of the queue associated with a `ResourcePool` node can be obtained via the method `queueLength`, e.g.

```
int pop = resPool.queueLength() ;
```

This can be used to implement queue-length-dependent routing, for example.

In addition to FIFO queues, the `network` package also supports LIFO queues (`LIFOqueue`), queues with random insertion and removal from the head (`RandomQueue`), priority queues (`PriorityQueue`) and ordered queues, which insert customers according to a specified ordering, with removal from the head (`OrderedQueue`). These are described in more detail in Section 3.13.

3.8 Queueing Nodes

Queueing nodes comprise a specified integer number of servers and a queue of waiting customers. They are identical to resource nodes except that the "resource" (i.e. a server) is released when the customer leaves the node. A `QueueingNode` queues waiting customers in FIFO order by default, e.g.:

```
Delay serviceTime = new Delay( new Exp( 2.0 ) ) ;
QueueingNode qNode
  = new QueueingNode( "QNode", serviceTime, 3 ) ;
```

which models a 3-server queueing node with exponentially-distributed service times with rate 2 per server.

Again, finite-capacity FIFO queues and non-FIFO queues can be supported as for a resource node, e.g.:

```
QueueingNode qNode2
  = new QueueingNode( "QNode", 3, serviceTime, fq ) ;
```

Note that class `QueueingNode` is a subclass of `ResourcePool`.

3.9 Processor Sharing Nodes

A processor sharing node is a special node type that apportions processing capacity evenly among a set of "queued" customers. Actually, the customers are best thought of as sitting in a pool, each receiving the same amount of service per unit time. For example,

```
ProcessorSharingNode psn
  = new ProcessorSharingNode( "PS Node", serviceTime ) ;
```

with `serviceTime` as above. It is not possible to build a processor sharing node with more than one server.

Note that class `ProcessorSharingNode` is a subclass of `ResourcePool`. However, an attempt to invoke `releaseResource` on a `ProcessorSharingNode` will induce an error as all resource (and queue) handling is managed internally.

3.10 Preemptive Queueing Nodes

Preemptive-resume and preemptive-restart mechanisms are built in as subclasses of `QueueingNode`. These both assume that the queue associated with the node has infinite capacity and that there is a single server.

Customers are removed from the queue when they are placed into service and returned to the head of the queue when they are preempted. The default queue in both cases is LIFO but, as above, this can be replaced with any queue type. However, note that LIFO, FIFO, random and ordered queues all behave identically here, as customers are removed from the head. Priority queues will behave differently. For example,

```
PreemptiveResumeNode resumeNode
  = new PreemptiveResumeNode( serviceTime ) ;
PreemptiveRestartNode restartNode
  = new PreemptiveRestartNode( serviceTime, pq ) ;
```

with `pq` as above.

3.11 Boxed Queues

A `BoxedQueue` node is a node containing just a queue. Customers entering the node are placed in the queue. Because the queue is passive an additional method `dequeue` is provided to remove the customer at the head of the queue, e.g.

```
BoxedQueue bq = new BoxedQueue( "BQ", new PriorityQueue( 4 ) ) ;
bq.enter( new Customer() ) ;
bq.enter( new Customer() ) ;
Customer c = bq.dequeue() ;
```

3.12 Adding Custom Functionality

3.12.1 Subclassing `Node`

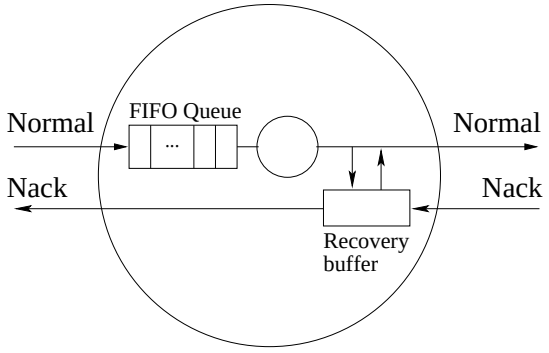
The `Node` superclass contains protected methods `accept` and `forward` that can usefully be overridden in subclasses, in order to extend a node's functionality. `accept` is invoked when a customer enters a node and `forward` is invoked when a customer is about to leave a node. In the root `Node` class, for example, `accept` invokes `forward` immediately. This sends the customer to the node's output link and subsequently on to some successor node in the network, depending on the nature of the node's link.

Implementation details: The various subclasses of `Node` work by redefining the `accept` and `forward` methods. For example, class `Sink` redefines `accept` so that the customer is registered as having completed:

```
Network.completions++ ;
Network.registerCompletion( Sim.now() - c.getArrivalTime() ) ;
```

The `ResourcePool` class redefines the `accept` method of the parent class (in this case `InfiniteServerNode`) so that incoming customers are queued (or rejected) if no resources are available. In class `QueueingNode`, the `forward` method is overridden from its definition in `ResourcePool` so that the acquired resource is released immediately when a customer departs. Finally, note that a `Source` node injects customers by invoking its own `forward` method for each new customer that is generated. This passes the customer into the network via the source node's output link.

Example As an example, the following sketches the code for a router, a subclass of a `FIFOQueueingNode`, that has an additional "recovery queue" in the event of a packet being corrupted, having earlier been forwarded toward its destination. The recovery queue is a circular buffer (not supported in the library described here but assumed to be implemented elsewhere in the model code).



Packets forwarded are added to the buffer. If the forwarded packet is corrupted down the line, a Nack is returned (here a different customer class) containing the identifier of the corrupted packet. If the original packet is still in the buffer, it is re-sent. If not, the Nack is passed back (presumably to the previous router) in the hope that it can be recovered there and re-sent. Circular buffer lookups are assumed to be based on packet identifier. The buffer has a finite capacity, which is a parameter of the buffer contructor. The example does not show the wiring of the network although it can be assumed that the outgoing `Link` routes normal packets, Nacks and other packet types differently — they are different classes of `Packet`, which is presumably a subclass of `Customer`.

```
class Router extends QueueingNode {
  CircularBuffer recoveryQueue ;

  public Router( String id, Delay d, int qCap, int bufferCap ) {
    super( id, d, 1, new FIFOQueue( qcap ) ) ;
    recoveryQueue = new CircularBuffer( bufferCap ) ;
  }

  protected void accept( Customer c ) {
    if ( c.getclass() == normalPacket ) {
      super.accept( c ) ;
    } else if ( c.getclass() == NackPacket ) {
      if ( recoveryQueue.contains( NackPacket.getID() ) ) ;
        Packet packet = recoveryQueue.lookUp( NackPacket.getID() ) ;
        forward( packet ) ;
      } else {
        // Send Nack back down the line...
        forward( c ) ;
      }
    } else {
      // All other packets are ignored
    }
  }

  protected void forward( Customer c ) {
    Packet packet = (Packet)c ;
    recoveryQueue.addToBuffer( packet ) ;
    super.forward( packet ) ;
  }
}
```

3.12.2 Subclassing `InfiniteServer`

The method `invokeService( Customer c )` is defined in class `InfiniteServer` where it schedules an event which implements the end of the service delay for customer `c` (see below for how to manage events in general). This method can be called by any subclass of `InfiniteServer` in order to add functionality prior to the delay; however, it cannot be overridden.

3.12.3 Subclassing `Link`

In some situations it may be appropriate to induce application-specific actions within a link, rather than at a node. As an example, a customer that departs a specified node may need to release a resource acquired elsewhere in the network, or it may need to change its class and/or priority. These effects can be

achieved either by subclassing the node and redefining the `forward` method, or by redefining the associated `Link` object at the node's output.

To implement the latter we extend the `Link` class, or any of its subclasses, and over-ride the method `move` which moves a given customer to a destination node. The required "special action" can be added to the `move` method thus:

```
class SpecialBranch extends ClassBranch {
  public SpecialBranch() {
    super( new int[] { 1, 2, 3 },
           new Node[] { exit, exit, node } ) ;
  }
  protected void move( Customer c ) {
    // Do special action here
    super.move( c ) ;
  }
}
```

3.13 More on Queues

All queues store objects of type `Customer`. An important invariant of most queues is that a customer is deemed *not* to be queueing whilst it is receiving service. This greatly simplifies the implementation of multi-server nodes and arguably better reflects reality. However, some care must be taken when interpreting various measures computed during a simulation. For example, in the case of preeemptive disciplines the measured mean queueing time will not be the same as the mean total time spent at a queue by a customer, as a customer may rejoin the queue several times during its visit to a node. In a processor sharing node a customer is demmed to be in the "queue" for its entire sojourn at the node. The mean time in the queue is then the same as the mean waiting time.

LIFO queues and random queues are parameterised as per FIFO queues, e.g.

```
Delay d = new Delay( new Deterministic( 100 ) ) ;
LIFOQueue lq = new LIFOQueue() ;
RandomQueue rq = new RandomQueue( 10 ) ;
ResourcePool resPool = new ResourcePool( "Res", d, 5, rq ) ;
```

The last line builds a resource pool with 5 resources and a random queue with capacity 10 for queueing waiting customers; customers hold a resource locally for 100 time units before leaving the node.

A priority queue comprises a specified number of sub-queues, each of which is FIFO by default. The i^{th} queue stores customers of priority i and the highest priority is 0. The constructor is parameterised by the number of priority classes (number of sub-queues), e.g.

```
PriorityQueue pq = new PriorityQueue( 6 ) ;
```

The sub-queues are FIFO by default but this can be modified by overriding a protected method `buildOneQueue()` in the `PriorityQueue` class that builds one such queue, e.g.

```
protected Queue buildOneQueue() {
  return new RandomQueue( 4 ) ;
}
```

The combined effect of the above is to build a priority queue supporting customer priorities 0-5, where the queue for each priority class has capacity 4 and implements a random queueing discpline.

Ordered queues make use of the customer ordering (see Section 3.3 above). The default ordering uses the customer class (0 is the minimum element in this ordering). A different ordering can be used by redefining `smallerThan` in any subclass of `Customer`, e.g.

```
class Patient extends Customer {
  int age ;
  public Patient( int patientClass, int age ) {
    super( patientClass ) ;
    this.age = age ;
  }
  public int getAge() {
    return age ;
  }
  public boolean smallerThan( Customer e ) {
    return this.getAge() <= ((Patient)e).getAge() ;
  }
}
```

In this example, patients will be inserted in age order, with the youngest appearing first. For example

```
Node source    = new PatientSource( "Source", new Exp( 1 ) ) ;
Delay d        = new Delay( new Exp( 2 ) ;
OrderedQueue q = new OrderedQueue() ;
Node qn        = new QueueingNode( "MM1", d, 1, q ) ;
Node sink      = new Sink( "Sink" ) ;
source.setLink( new Link( qn ) ) ;
qn.setLink( new Link( sink ) ) ;
```

where `PatientSource` is as in Section 3.4. Each patient injected by the source will have an ordering based on age. When these hit the queueing node (`qn`) they will be queued in age order.

Implementation Note: Processor sharing nodes are implemented by storing customers in order of their service completion time, with respect to a local clock. This clock ticks slower as customers are added to the queue. A processor

sharing "queue" is implemented internally using an ordered queue, where the queue entries comprise customers and their finish times. The ordering is defined by implementing the `Ordered` interface in a class `OrderedQueueEntry`:

```
public class OrderedQueueEntry extends Customer
                                implements Ordered {
  double time ;
  public Customer entry ;

  public OrderedQueueEntry( Customer c, double t ) {
    time = t ;
    entry = c ;
  }
  public boolean smallerThan( Customer e ) {
    return ((OrderedQueueEntry)this).time <=
           ((OrderedQueueEntry)e).time ;
  }
}
```

3.14 Output

Each node in a queueing network maintains measurement variables internally. At the end of a simulation (or at any point during a simulation, e.g. during a 'batched means' run) a summary of all the measures from a node can be logged by invoking the method `logResults` on a that node, e.g.

```
source.logResults() ;
```

The measurements for an entire network, which comprises global network measures and local measures for each node, can be generated by invoking the static method `logResults` in class `Network`, viz.

```
Network.logResults() ;
```

The `logResults` method can be invoked any number of times, up to some specified limit.

Once the results have been logged they can be displayed by invoking the static method `Network.displayResults`. An optional parameter is the confidence level. The current values supported via encoded tables of the Student's 't' distribution are 0.01, 0.025, 0.05 and 0.1. If no confidence level is supplied a value of 0.05 is assumed.

The accuracy of the confidence interval will depend on the properties of independence and normality. If the logged measures are dependent and/or are not normally distributed, the computed confidence interval will be inaccurate. With independent replications, mean values will be approximately normal, but variances and other measures may not. Some care is therefore needed in the

interpretation of confidence intervals reported in the model output. The point estimates in each case will, however, be reliable.

All node types have a `resetMeasures` method that resets the node's internal measures (see also Section 4.4). This is typically used to "delete" measurements taken during initialisation transients, when observing a system in its steady state. There is no built-in mechanism for detecting approximate steady state but, once the approximate length of the warm-up transient has been established, there is a built-in mechanism for resetting measures after the warm-up has expired; this is described in Section 4.4.1.

3.15 Tracing and Debugging

A trace of customer flow through a network can be generated by turning on a debug trace facility via a static method in class `Debug`, thus:

```
Debug.setDebugOn() ;
```

This can be toggled (`On/Off`) dynamically if necessary. The default is `Off`.

Additional trace messages can be added via the static method `Debug.trace` which displays a given string only if the debug toggle is on, e.g.

```
Debug.setDebugOn() ;
Debug.trace( "Debug is now on" ) ;
```

4 The Simulation `tools` Package

The classes `Event` and `Sim` can be used to construct an event-driven simulation. The `network` package is built on top of these.

4.1 Events

All events are subclasses of `Event` and must implement a method `invoke` which defines what the event does. The `Event` constructor is parameterised by the time that the event is to be invoked, e.g.:

```
class NewEvent extends Event {
  public NewEvent( double t ) {
    super( t ) ;
  }
  public void invoke() {
    System.out.println( "NewEvent has been invoked" ) ;
  }
}
```

4.2 Distribution Samplers

In addition to `Exp( double r )` and Weibull(double alpha, double beta) introduced earlier, there are other distribution samplers. Here is the complete list:

```
Cauchy( double a, double b )
Deterministic( double t )
Erlang( int k, double theta )
Exp( double r )
Gamma( double theta, int beta )
Geometric( double p )
Normal( double mu, double sigma )
Pareto( double k, double a, double b )
Uniform( double a, double b )
Weibull( double alpha, double beta )
```

There are also empirical samplers:

```
DiscEmpirical( double xs[], double fs[] )
```

which samples an empirical discrete distribution defined by a set of observed values and their frequencies, and

```
ContEmpirical( double xs[], double fs[] )
```

similarly, except that the `xs` delimit contiguous intervals on the real line.

With the exception of the Gamma sampler and two empirical samplers, there are equivalent static methods for sampling each distribution within the associated class. For example `Exp.exp( 2.0 )` generates a sample from the exponential distribution with parameter 2 (mean 0.5).

4.3 Simulation

An event-driven simulation involves scheduling instances of the `Event` class in a hidden time line and then processing them in time order.

A simulation is built by subclassing `Sim`. The `Sim` class contains entirely static methods so that the various methods can be invoked from anywhere without having to pass around a `Sim` object explicitly.

`Sim` is an abstract class which requires a termination function `stop` to be defined (a call-back). This enables the application to define arbitrary termination criteria, e.g. based on the model state, virtual time etc.

The preferred approach is for the `Sim` subclass to contain the complete model state as instance variables and the set of events as inner classes; this also avoids the need to prefix `Sim` method names with "`Sim.`". This will be assumed in the following.

The passage of time is modelled by an internal clock that can be inspected via the method call `now()`, e.g.:

```
double currentTime = now() ;
```

This is a static method, so the call `Sim.now()` works fine in all contexts. Within a `Sim` subclass, the prefix can be dropped.

The `Sim` class also defines a static method `schedule` for adding a new event instance to the time line and a method `simulate` which begins the processing of the time line. Here is a simple example of a complete simulation model:

```
import tools.* ;

class TickerSim extends Sim {

  // Example state variable
  int n = 0 ;

  // Example event
  class Tick extends Event {
    public Tick( double t ) {
      super( t ) ;
    }
    public void invoke() {
      n++ ;
      System.out.println( "Tick " + n + " at time " + now() ) ;
      schedule( new Tick( now() + 10.0 ) ) ;
    }
  }

  // Example termination function; abstract in superclass
  public boolean stop() {
    return now() > 100 ;
  }

  // Here, the constructor starts the simulation.
  public TickerSim() {
    schedule( new Tick( 0.0 ) ) ;
    simulate() ;
  }

  // Main method simply invokes the Sim constructor
  public static void main( String args[] ) {
    new TickerSim() ;
  }
}
```

This schedules a `Tick` event every 10.0 time units. Each event invocation schedules the next one. A single integer state variable tracks the number of tick events to date. When executed, it produces the output:

```
Tick 1 at time 0.0
Tick 2 at time 10.0
Tick 3 at time 20.0
Tick 4 at time 30.0
Tick 5 at time 40.0
Tick 6 at time 50.0
Tick 7 at time 60.0
Tick 8 at time 70.0
Tick 9 at time 80.0
Tick 10 at time 90.0
Tick 11 at time 100.0
```

Implementation detail: The `InfiniteServer` class implements a customer delay by redefining the `accept` method to insert an event representing the end of the service delay. The `invokeService` method introduced earlier uses the static `schedule` method in class `Sim` to do this. Note that the "`Sim`" prefix is required in this case as the call is being made from outside a `Sim` subclass.

4.4 Measures

`Resource` objects and the various types of `Queue` object all have built-in measures. The tools used to maintain these measures can be used elsewhere. A `CustomerMeasure` maintains customer-oriented measures, such as mean waiting time. A `SystemMeasure` maintains system-oriented measures such as mean population. Both are subclasses of the abstract class `Measure`. The difference is that the latter accumulates the integral under a step function, whilst the former accumulates a sum.

By default, the first and second moments are maintained by a `Measure`. Up to 10 moments can be computed by appropriately parameterising the `Measure` constructor. The sample n^{th} moment and the the mean and variance of a measure can be obtained through the API. For example,

```
CustomerMeasure cm = new CustomerMeasure() ;
cm.add( 1.0 ) ;
cm.add( 2.0 ) ;
cm.add( 3.0 ) ;
double m = cm.mean() ; // The sample mean will be 2
SystemMeasure sm = new SystemMeasure( 3 ) ;
sm.add( 1.0 ) ;
sm.add( 2.0 ) ;
sm.add( 3.0 ) ;
m = sm.mean() ;
double v = sm.variance() ;
double m3 = sm.moment( 3 ) ;
```

Note that the mean in the case of `sm` will be undefined in this example, as the three calls to `add` occur at the same (virtual) time. In practice `add` will be called

as a simulation progresses, i.e. over a passage of virtual time.

4.4.1 Measure Resets

All measures have a `resetMeasures` method that can be used to reset the measure's internal variables. This is necessary when "deleting" observations during the transient warm-up phase of a simulation, when the user is interested in modelling the steady state.

There is no built-in mechanism for detecting when a simulation is close to a steady state. The warm-up period must be specified by the user and will typically be based on observations of pilot runs of the model. When the approximate warm-up period is known, the `simulate` method described above can be optionally parameterised by this warm-up period. Internally this will set up an "end of warm-up" event whose sole action is to invoke a method `resetMeasures()` defined in the `Sim` class. This can be overridden to reset any or all measures defined in the model. For example,

```
class SimSubclass extends Sim {
  Resource server = new Resource() ;

  // Rest of Sim subclass code

  public void resetMeasures() {
    server.resetMeasures() ;
  }

  public SimSubclass() {
    simulate( 10000.0 ) ;
  }
}
```

which resets the measures in a `Resource` object after the first 10000 time units of the simulation. A warning will be issued if the `resetMeasures` method is not overridden when a warm-up period is specified.

Implementation detail: All `Queue` objects in the network package maintain the mean and variance of the queue length and time spent in the queue using built-in measures. A `Resource` object maintains the resource utilisation using a built-in `SystemMeasure`. Access to these measures is provided through methods with obvious meaning, for example:

```
double mql = fq.meanQueueLength() ;
double vql = fq.varQueueLength() ;
double mtq = fq.meanTimeInQueue() ;
double u = server.utilisation() ;
```

These internal measures can be reset, for example to allow measurements taken during initialisation transients to be ignored. `Queue` and `Resource` objects provide their own methods for resetting internal measures:

```
fq.resetMeasures() ;
server.resetMeasures() ;
```

4.5 Independent Replications

4.6 Result Logging

A simulation can straightforwardly be repeated many times by re-invoking the appropriate `Sim` subclass constructor. At the end of each run it will be normal to log a summary of the measurements obtained during the run. The `Logger` class contains static methods for logging a single result and for displaying a summary of all logged results, possibly over many runs. Each result is logged by passing a `String` identifying the result, and the result value, to the static method `logResult`, e.g.

```
Logger.logResult( "PS node delay", psn.meanTimeInQueue() ) ;
```

with `psn` as in Section 3.9.

If `logResult` is called repeatedly with the same string identifier, each new result will be added to those already logged. A set of logged results can be displayed by invoking the static method `Logger.displayResults` directly. An optional parameter specifies the confidence level, which functions as described in Section 3.14.

The static method `Network.displayResults` described in Section 3.14 works by invoking `Logger.displayResults`. For details of how confidence intervals are produced from the logs, refer back to this section.

4.7 Replication Management

Using the above approach a simulation can be executed several times simply by reinvoking the corresponding `Sim` subclass constructor. Automatic control over the replication process is provided by an abstract class `ReplicatedSim`. This contains an abstract method `runSimulation` which should be instantiated so that it invokes a given simulation by invoking its constructor. The `ReplicatedSim` has two parameters: the number of replications and a confidence level. The `runSimulation` method is invoked the specified number of times and the `Logger.displayResults` is then used to display a summary of all the results logged during the runs. For example:

```
public class ManySims extends ReplicatedSim {
  public ManySims( int n, double a ) {
    super( n, a ) ;
  }
  public void runSimulation() {
    new OneSim() ;
  }
  public static void main( String args[] ) {
    new ManySims( 3, 0.05 ) ;
  }
}
```

which invokes the `Sim` subclass construtor `OneSim` three times and reports the results using a confidence level of 0.05.

5 Examples

To illustrate how the various classes can be put together, some simple examples are now provided. The source code for these can be found in the distribution.

5.1 MM1 Queue - Hand-coded Version

The following details an M/M/1 queue simulation build without using the `network` package. Arrival and departure events are modelled explicitly and the population of the queue is maintained by an integer instance variable. The service rate is hard coded to 4 and the arrival rate to 2, although these are easily generalised to be parameters. Only the server utilisation is measured (via a `Resource`) and logged. The main method runs the model three times and then displays a summary of the results.

```
import tools.* ;

class SSQSim extends Sim {
  Resource server = new Resource() ;
  int pop = 0 ;

  class Arrival extends Event {
    public Arrival( double t ) {
      super( t ) ;
    }
    public void invoke() {
      schedule( new Arrival( now() + Exp.exp( 2 ) ) ) ;
      pop++ ;
      if ( server.resourceIsAvailable() ) {
        server.claim() ;
        schedule( new Departure( now() + Exp.exp( 4 ) ) ) ;
      }
    }
  }
  class Departure extends Event {
    public Departure( double t ) {
      super( t ) ;
    }
    public void invoke() {
      pop-- ;
      if ( pop > 0 )
        schedule( new Departure( now() + Exp.exp( 4 ) ) ) ;
      else
        server.release() ;
    }
  }
  public boolean stop() {
    return now() > 1000000 ;
  }
  public SSQSim() {
    schedule( new Arrival( now() + Exp.exp( 2 ) ) ) ;
    simulate() ;
    Logger.logResult( "Utilisation", server.utilisation() ) ;
  }
  public static void main( String args[] ) {
    new SSQSim() ;
    new SSQSim() ;
    new SSQSim() ;
    Logger.displayResults( 0.01 ) ;
  }
}
```

Here is some sample outout:

```
SUMMARY OF STATISTICS

Confidence level: 1.0%

Utilisation
   Point estimate:  0.5002281258375239
   Degrees of freedom: 2
   C.I. half width: 8.417644702812562E-4
```

5.2 MM1 Queue - Queueing Network Version

The next example uses the `network` package to build the same simulation. The system comprises a source node a queueing node and a sink. The simulation maintains a large array of internal measures as detailed earlier. Here, however, only the server utilisation is extracted for logging.

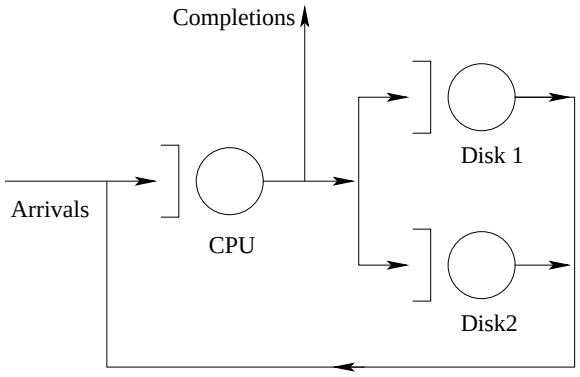
```
import network.* ;
import tools.* ;

class MM1Sim extends Sim {

  public MM1Sim() {
    Network.initialise() ;
    Delay serveTime = new Delay( new Exp( 4 ) ) ;

    Source source    = new Source( "Source", new Exp( 2 ) ) ;
    QueueingNode mm1 = new QueueingNode( "MM1", serveTime, 1 ) ;
    Sink sink        = new Sink( "Sink" ) ;

    source.setLink( new Link( mm1 ) ) ;
    mm1.setLink( new Link( sink ) ) ;

    simulate() ;

    Network.logResult( "Utilisation", mm1.serverUtilisation() ) ;
  }

  public boolean stop() {
    return Network.completions == 1000000 ;
  }

  public static void main( String args[] ) {
    new MM1Sim() ;
    new MM1Sim() ;
    new MM1Sim() ;
    Network.displayResults( 0.01 ) ;
  }
}
```

This version executes approximately four times slower than the hand-crafted original, but is very much easier to follow. Also, a full complement of measures is maintained behind the scenes, even though only the utilisation is logged in this case.

Note that the sink node can be omitted, but then no completions will be logged and the `stop` method will always return `false`. The termination criteria could be modified, however, e.g. so that it is based on time.

5.3 CPU Model

This example models a queueing network comprising a CPU and two disks as shown below.



Jobs arrive according to a Poisson process and join the CPU. The mean CPU visit time is 5ms. At the end of a visit the job either leaves (probability 1 in 121 visits) or visits one of two disks (probabilities 70/121 and 50/121 respectively). The disk service times are 30ms and 27ms respectively. All time delays are exponential. The code here shows one run of the model over 10000 completed jobs.

The time delay distributions here are all exponential, so this model can be analysed analytically. It is easy, however, to change the distributions.

```
import network.* ;
import tools.* ;

class CPUSim extends Sim {

  public CPUSim() {
    Network.initialise() ;
    Delay cpuTime   = new Delay( new Exp( 1/0.005 ) ) ;
    Delay disk1Time = new Delay( new Exp( 1/0.03 ) ) ;
    Delay disk2Time = new Delay( new Exp( 1/0.027 ) ) ;

    Source source      = new Source( "Source", new Exp( 0.1 ) ) ;
    QueueingNode cpu   = new QueueingNode( "CPU", cpuTime, 1 ) ;
    QueueingNode disk1 = new QueueingNode( "Disk 1", disk1Time, 1 ) ;
    QueueingNode disk2 = new QueueingNode( "Disk 2", disk2Time, 1 ) ;
    Sink sink          = new Sink( "Sink" ) ;

    double[] routingProbs = { 1.0/121.0, 70.0/121.0, 50.0/121.0 } ;
    ProbabilisticBranch cpuOutputLink
      = new ProbabilisticBranch( routingProbs,
                                 new Node[] { sink, disk1, disk2 } ) ;
    source.setLink( new Link( cpu ) ) ;
    cpu.setLink( cpuOutputLink ) ;
    disk1.setLink( new Link( cpu ) ) ;
    disk2.setLink( new Link( cpu ) ) ;

    simulate() ;

    Network.logResults() ;
  }

  public boolean stop() {
    return Network.completions == 10000 ;
  }

  public static void main( String args[] ) {
    new CPUSim() ;
    Network.displayResults() ;
  }
}
```