

# Denobo

Quickstart Guide

## Preface – The Purpose of this Guide

This guide is designed to get a new user working with Denobo networks as quickly as possible while giving them a brief overview of how the middleware work internally. For this reason, we get into code samples right away and do not explore the API in-depth, only touching lightly on the inner architecture of the official Denobo library. For the purposes of this document, “Denobo” is used to refer to the official Denobo middleware library, written in Java by Saul Johnson, Alex Mullen and Lee Oliver.

The Denobo Central Command testing UI is also covered lightly, with a short guide on getting started putting together and debugging networks visually using the network designer, both locally and over sockets.

## Chapter I – Exploring Entities

We can define entities in Denobo as an instance of an object that does work of some kind on its own thread. The Denobo middleware can be considered to have two main types of entities, which will be briefly discussed here.

### Agents

These are the system's atomic entities. Many uniquely-named agents are connected to form a network and when one agent wishes to send a message to another, it can do so as long as it knows the name of the recipient. The simplest form of agent in Denobo is the aptly-named Agent class. To initialise two agents and connect them together in Java, one would do the following:

```
Agent myFirstAgent = new Agent("agent1");
Agent mySecondAgent = new Agent("agent2");
myFirstAgent.connectAgent(mySecondAgent);
```

Now to attach a message handler to one agent so we can monitor any messages it receives:

```
myFirstAgent.addMessageHandler(new MessageHandler() {

    @Override
    public void messageRecieved(Agent agent, Message message) {
        System.out.println("We recieved: " + message.getData() + " from " +
            message.getOriginator() + "!");
    }

});
```

Now if we run the code:

```
mySecondAgent.sendMessage("agent1", "Hello World!");
```

We can see the following printed out on the command line:

```
We recieved: 'Hello World!' from 'agent2'!
```

### Socket Agents

In Denobo, the SocketAgent class inherits from Agent, and allows the middleware user to connect to another SocketAgent over a network using sockets if they know the IP address and port number the remote agent is advertising on. Communication across sockets takes place using the Denobo Message Transfer protocol (DMTP) for which there is a detailed specification available. The end user has no need to deal with sockets directly.

For example, to create a `SocketAgent` instance and start it running on the local machine, we would do the following:

```
SocketAgent myServerAgent = new SocketAgent("serverAgent");  
myServerAgent.startAdvertising(4757); // Start advertising on port 4757.
```

Now, to connect to that agent over a socket for demonstration purposes, we would write:

```
SocketAgent myClientAgent = new SocketAgent("clientAgent");  
  
// Connect to localhost For demonstration purposes.  
myClientAgent.addConnection("127.0.0.1", 4757);
```

It should be noted that `SocketAgent` supports encrypted sessions using RC4 encryption with a shared key established using a Diffie-Hellman key exchange, rendering it impractical for a malicious third party to intercept and read transmitted messages over a network. This security can be specified using an instance of `SocketAgentConfiguration` passed into `SocketAgent`'s constructor:

```
SocketAgentConfiguration config = new SocketAgentConfiguration();  
config.setIsSecure(true);  
  
SocketAgent mySecureSocketAgent = new SocketAgent("secureAgent", false, config);
```

`SocketAgent` also supports compression using basic semi-adaptive Huffman coding or LZW compression. This is specified in a similar manner to encryption; see the API documentation for more details.

## Cloneable Agents

If an agent is constructed to be cloneable, every callback to its connected `MessageHandler` instances on the `messageReceived` method will be on a new thread. This is useful behaviour if the end user is concerned about expensive `MessageHandler` methods blocking. To create a cloneable Agent, one would do the following:

```
// The 'true' on the following line constructs a cloneable Agent.  
Agent myCloneableAgent = new Agent("cloneableAgent", true);
```

Both `Agent` and `SocketAgent` instances can be constructed as cloneable.

## Workers/Crawlers

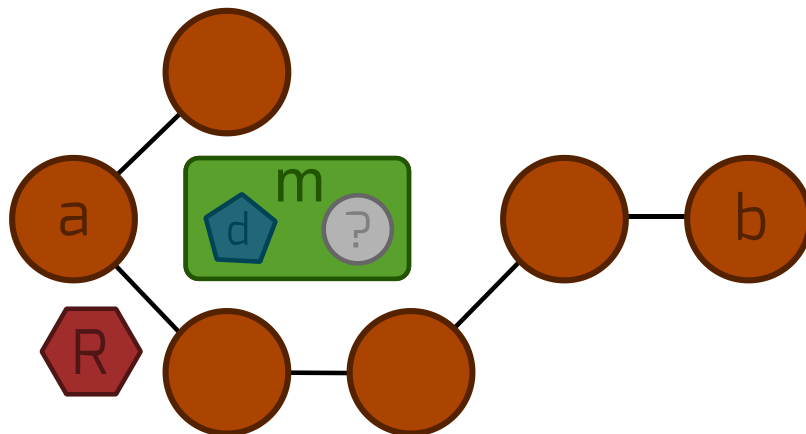
These are the system’s “maintenance robots”. When the network is changed or an agent needs to know about its structure, a worker (sometimes called a crawler) is spawned. Crawlers come in two varieties, both of which are controlled at the low level by instances of `Agent` and neither of which are of any concern to the end user who is using the API as part of their software. These two types will be discussed here.

### Routing Workers

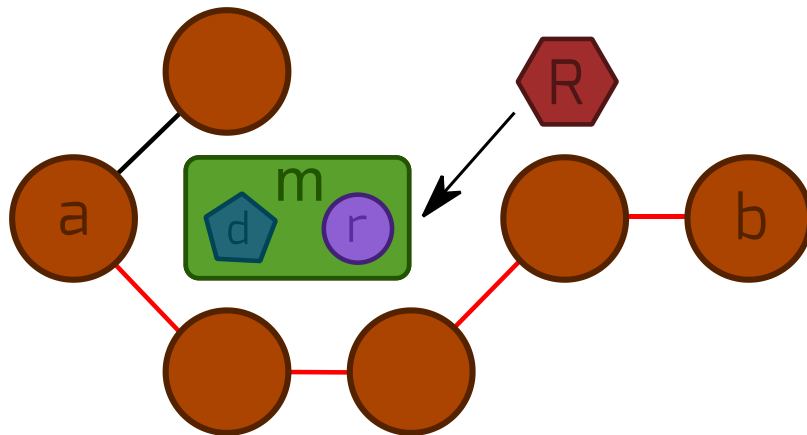
When a message is being sent from one agent to another for the first time, an instance of `RoutingWorker` is spawned to traverse the network recursively (depth-first) starting at the originator in order to plot the shortest route to the recipient. When the routing worker calls back to the agent, the route is added to the agent’s routing table and any messages awaiting that route are dispatched. If the routing worker fails to return a route within a timeout, the waiting messages are discarded and message sending fails.

Routing workers are, however, unable to cross sockets. To work around this, if the routing worker cannot find a route to a named agent on the local network, it will delegate the task of finding a route to the network’s `SocketAgents`, who find a route over DMTP by spawning routing workers at each of their remote peers. When and if a route is found, the `SocketAgents` will call back to the originating agent. An instance of `RoutingWorker` dies when it finishes its traversal of the local network and delegated the task of routing to the network’s `SocketAgents` if necessary.

For example, if agent *a* wants to send data *d* to agent *b* it needs to send a message *m*. However, in Denobo, a message must have both data and a route to travel down before it can be dispatched. Therefore, in order to get a route from itself to *b*, agent *a* spawns a routing worker *R*.



Routing worker *R* runs in its own thread while message *m* waits, traversing the network and calculating route *r* (shown in red). The message *m* now has everything it needs to reach its destination, agent *b* and is dispatched.



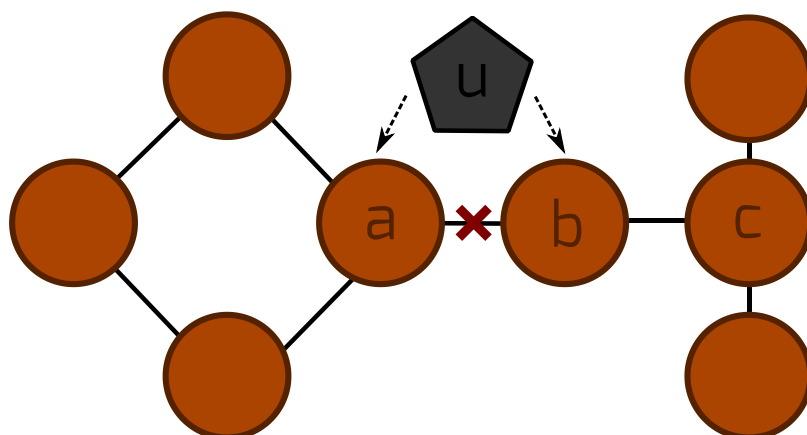
After a route from *a* to *b* has been calculated, the former stores the route in its routing table so that it doesn't need to be calculated again. In this manner, the network becomes more and more efficient as it is used.

## Undertakers

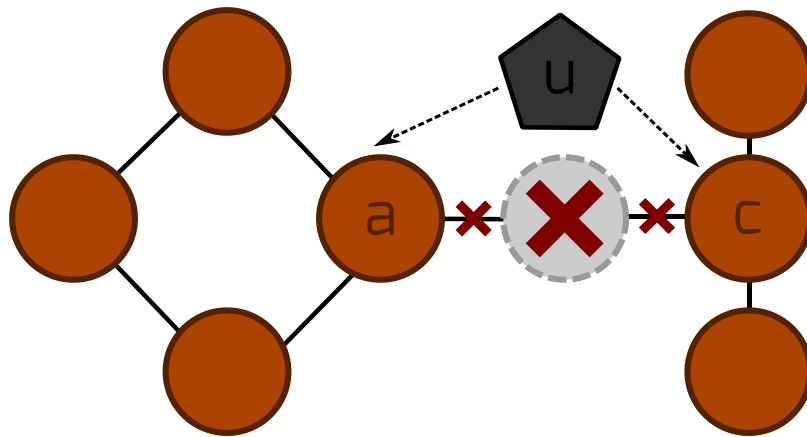
In much the same way that new routes are mapped and added to routing tables by a RoutingWorker, an Undertaker removes dead links and agents from the routing tables of agents across the network. When an agent is disconnected from another, those two agents are considered to be *invalidated* meaning that routes containing those agents can no longer be relied upon to exist. This disconnection can happen via a call to the `disconnectAgent` method or by agent shutdown and deletion.

However the disconnection occurs, an Undertaker is spawned to inform the rest of the network of the 'dead' or more accurately *invalidated* link or agent(s).

Suppose agent *a* is disconnected from agent *b* in the manner shown, and the link between *a* and *b* is severed by a call to `disconnectAgent`.



An Undertaker *u* is spawned, which starts at either end of the severed link and removes any entries containing either *a* or *b* from the routing tables of all other agents either side of the severed link. Now, suppose agent *b* was to be deleted entirely.



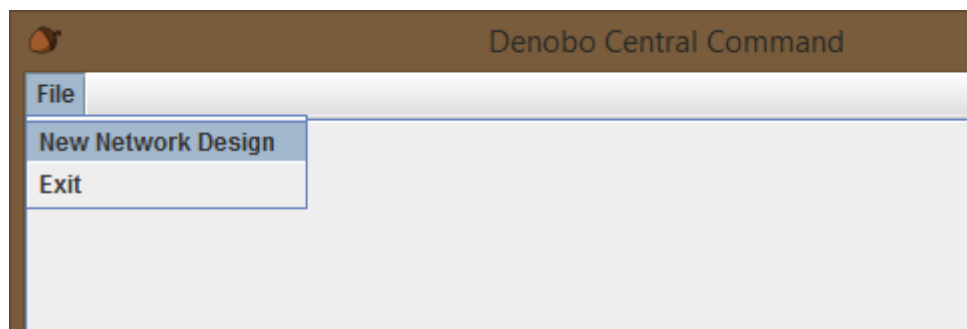
The Undertaker  $u$  would start at every agent that was connected to the deleted agent  $b$  and perform the same process for every broken link.

## Chapter II – Using the Testing UI (Denobo Central Command)

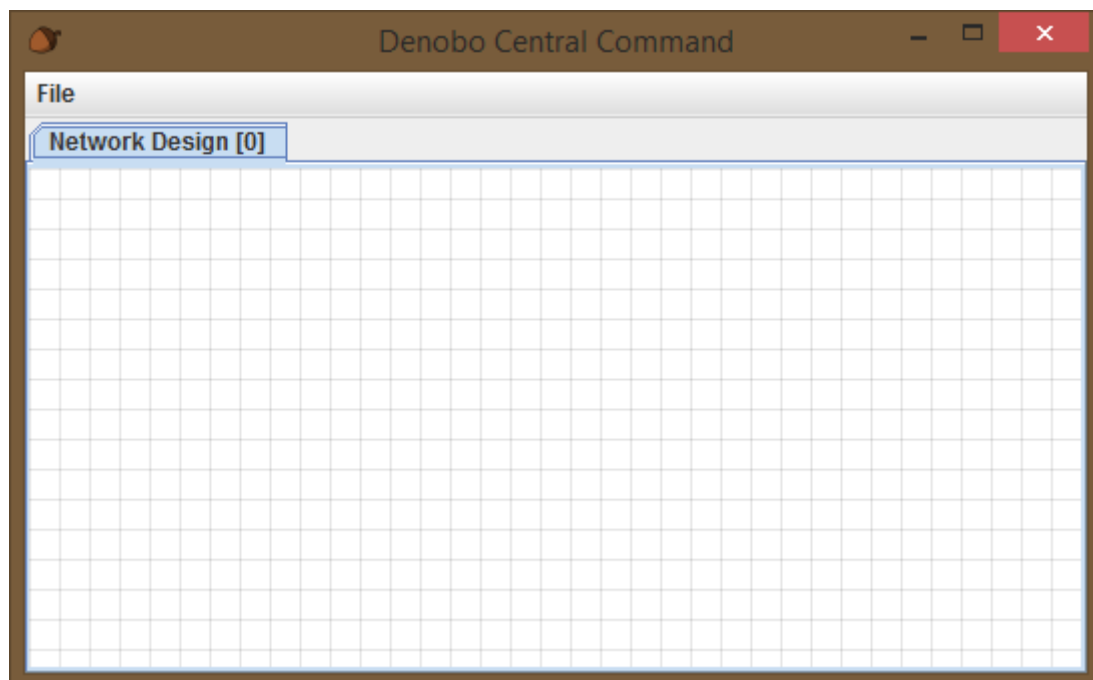
A testing UI named “Denobo Central Command” is included with Denobo for exploring and testing the various networks structures that can be created in an intuitive, visual way. As of yet, the software is feature-limited but stable.

### Creating a Simple Network

In the Denobo Central Command software, go to File -> New Network Design.

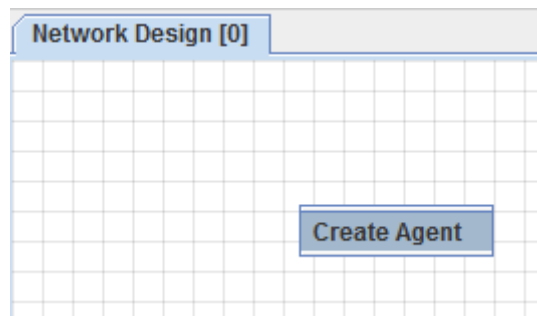


This will open a new network designer tab; your environment should now look like this:

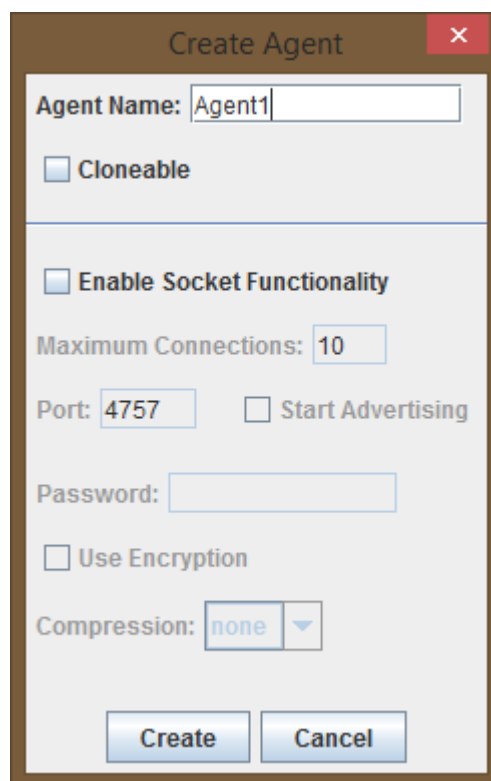




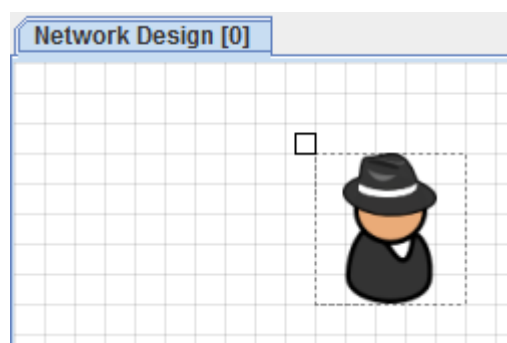
Right-click anywhere in the designer and a context menu will appear. Select “Create Agent”.



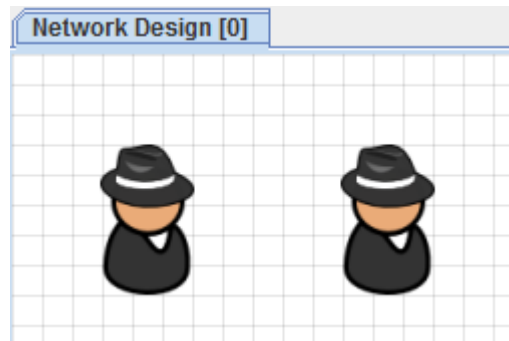
A dialog will appear. Name your agent and click “Create”. Leave the other options alone for now.



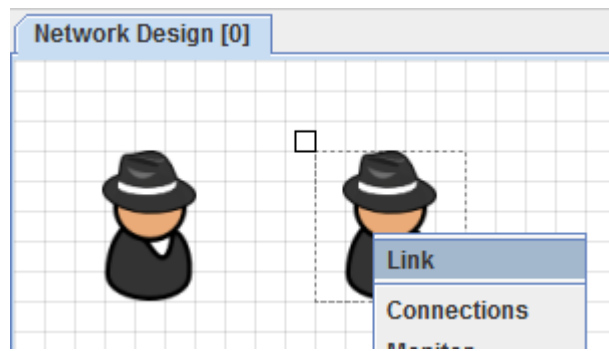
You will notice that a small icon of an ‘agent’ has appeared, and that this icon can be freely dragged around the designer.



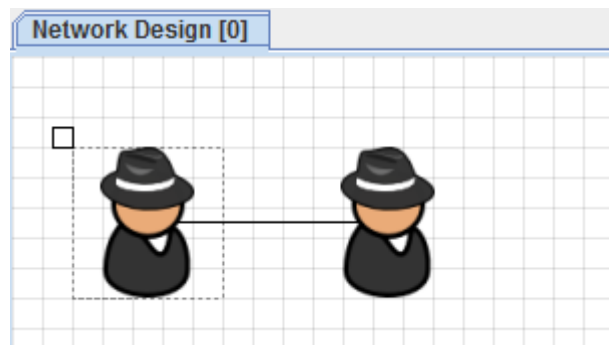
Repeat this process so that you have two agents side-by-side in the designer.



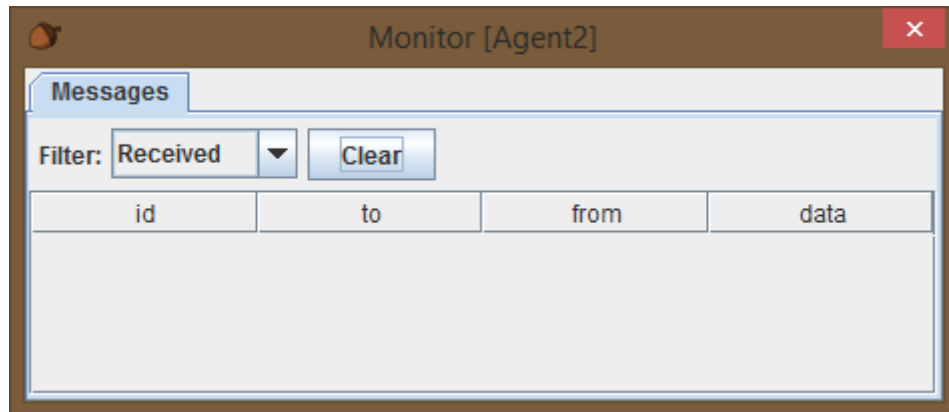
Right-click one of the agents and select 'Link', notice that a line now follows your cursor in the designer from the centre of the agent you clicked.



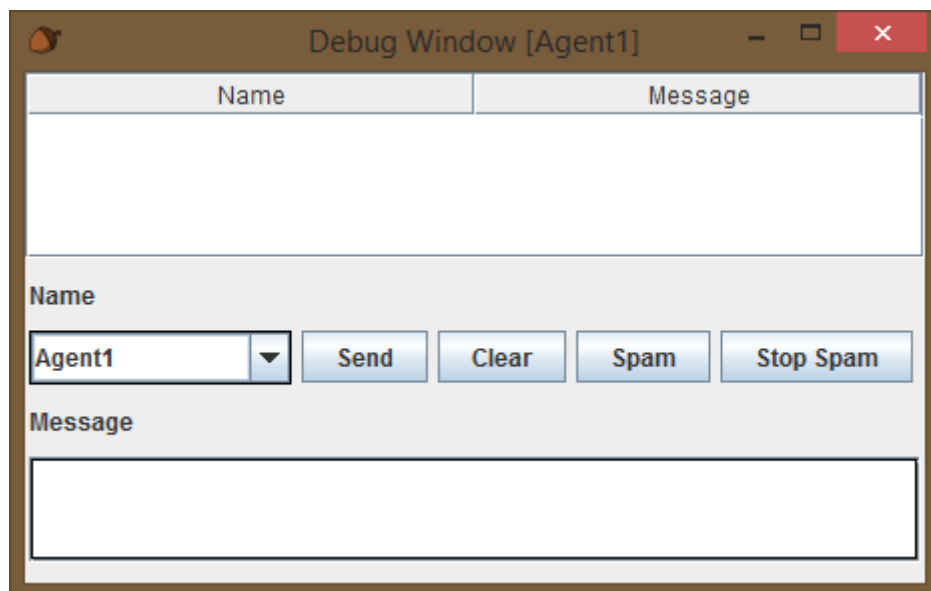
Now, click on the other agent to link the two together.



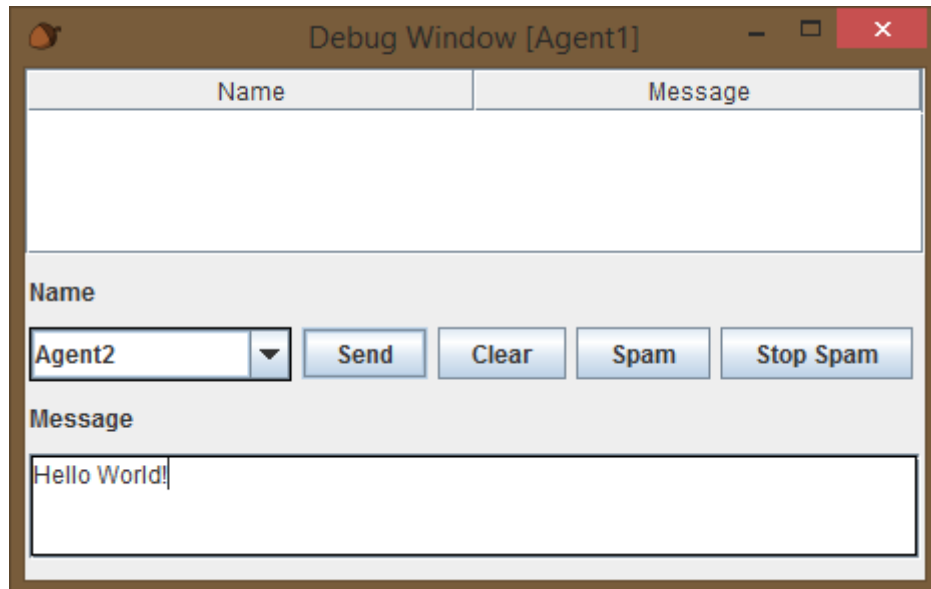
Right-click one of the agents and select “Monitor”; a dialog will appear that will display any messages received by that agent.



Now right-click the other agent and select “Debug Window”; a dialog will appear that will allow you to send messages using this agent.

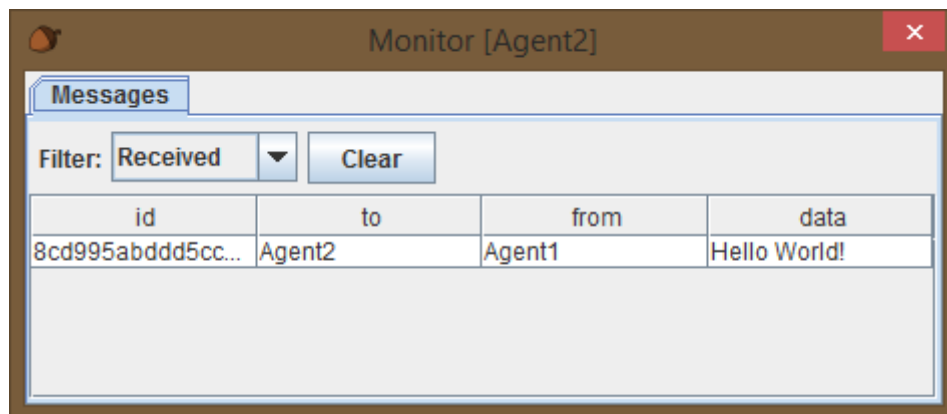


Select the name of the agent with the monitor window open (in this case Agent2) from the drop-down list in the debug window. Enter a message and click “Send”.



The screenshot shows a window titled "Debug Window [Agent1]". It contains a table with two columns: "Name" and "Message". Below the table, there is a "Name" section with a dropdown menu currently showing "Agent2". To the right of the dropdown are four buttons: "Send", "Clear", "Spam", and "Stop Spam". Below the "Name" section is a "Message" section with a text input field containing the text "Hello World!".

Check the monitor window for the other agent to see the message has been received.



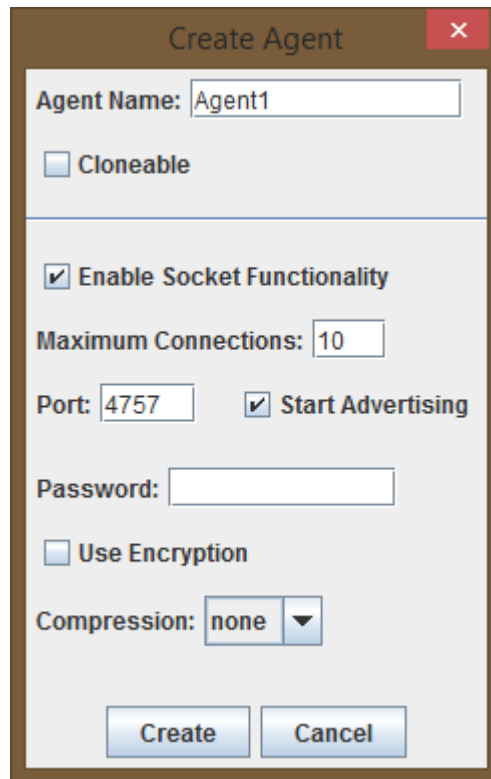
The screenshot shows a window titled "Monitor [Agent2]". It has a "Messages" tab selected. Below the tab is a "Filter:" section with a dropdown menu showing "Received" and a "Clear" button. Below the filter section is a table with four columns: "id", "to", "from", and "data". The table contains one row of data:

id	to	from	data
8cd995abddd5cc...	Agent2	Agent1	Hello World!

You can try this for any number of different network configurations of any complexity. Networks can have rings, loops and redundancy.

## Creating a Socketed Network

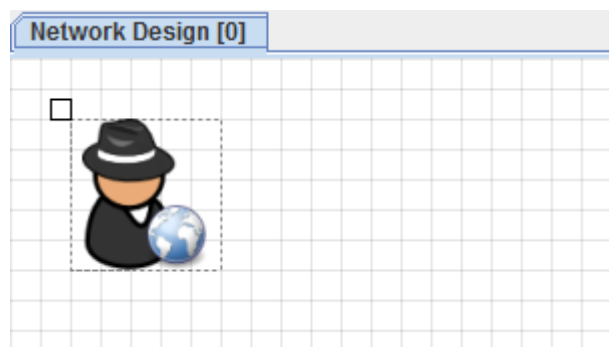
Create a new network design and bring up the “Create Agent” dialog.



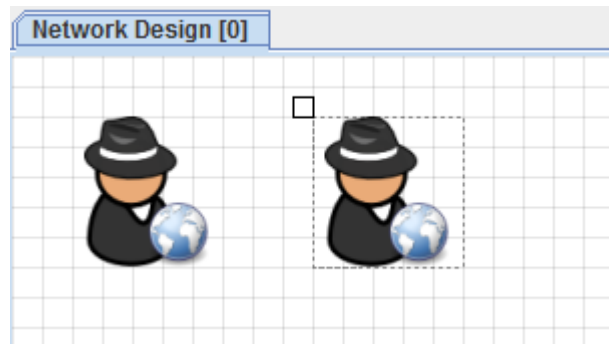
The "Create Agent" dialog box is shown with the following settings:

- Agent Name: Agent1
- ☐ Cloneable
- ☒ Enable Socket Functionality
- Maximum Connections: 10
- Port: 4757
- ☒ Start Advertising
- Password: (empty field)
- ☐ Use Encryption
- Compression: none (dropdown menu)
- Buttons: Create, Cancel

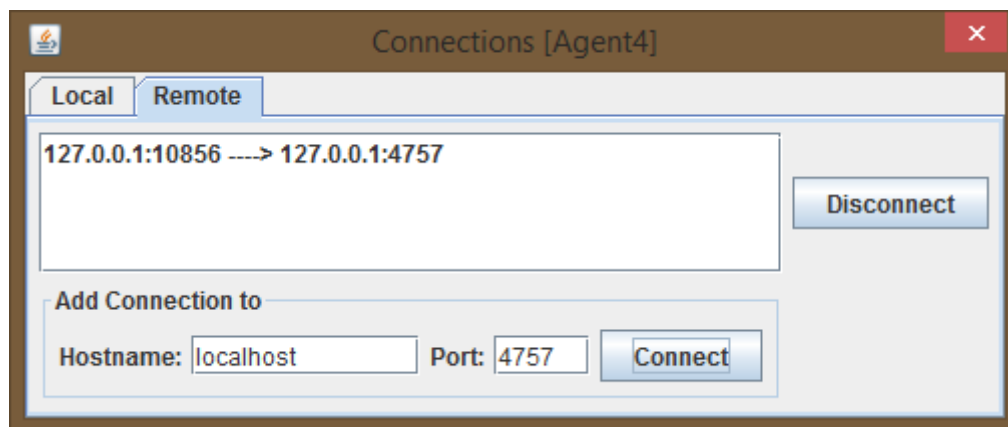
Make sure both “Enable Socket Functionality” and “Start Advertising” are checked and click “Create”. An ‘agent’ icon with a globe next to it will appear in the designer, indicating a SocketAgent. This will automatically have started advertising on port 4757.



Repeat this process to get two socket agents side-by-side in the designer, making sure that “Start Advertising” is not checked this time around.



Now, right-click the second socket agent you added and select “Connections”. Go to the “Remote” tab and click the “Connect” button leaving the default values in the “Hostname” and “Port” fields.



An entry will appear in the list box showing the connection is live. Test it with the Monitor/Debug window as for a local network to see the messages passed across the socket. It is possible to connect cross-VM and cross-machine using SocketAgents.