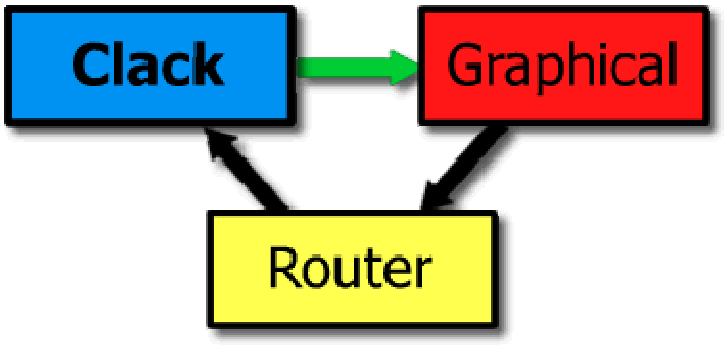


Clack Developer Documentation

Contact: info@clackrouter.net
Document Version: 2.0 (3/22/2006)
Associated Clack Version: 1.0

The Clack Graphical Router Toolkit is an extensible framework for graphical education about networking infrastructure and protocols. Clack is a tool for educators looking to give students a better avenue for learning about low-level Internet networking in an intuitive and hands-on way.

This document is for those who wish to extend Clack to provide additional functionality. The two main ways to extend Clack are:

- Adding new "router components" to implement new router functionality or specialized traffic inspection and analysis capabilities.

- Adding new "Clack applications" to provide different traffic loads and types within a Clack virtual network.

Clack was designed specifically so that adding new functionality would be extremely simple. Section 1 of this document provides an overview of the Clack architecture, which will be useful in providing an understanding of the overall system and defining Clack specific vocabulary. Section 2 describes everything that is needed to run Clack locally for development.

The remainder of the document then relies heavily on code examples to demonstrate how to extend Clack. Section 3 describes building Clack router components, starting with simple examples and then describing how increasingly sophisticated types of components can be constructed. Section 4 focuses on building Clack applications and can be read largely independent from Section 3.

Main sources for help with Clack development:

- This developer guide and the corresponding *net.clackrouter.example.** source files available within the Clack source code download.

- The Javadoc for the entire Clack source code is online at http://www.clackrouter.net/javadoc/ and is also available for download on the site.

- Viewing existing router components and applications as examples is a great way to become familiar with the Clack paradigm. Download the source and explore the *net.clackrouter.component.*, net.clackrouter.net.application.**, and other packages.

- Clack was detailed an extensive undergraduate thesis documents available at: http://www.clackroute.net/clack_thesis.pdf . While this document is somewhat academic it remains the most detailed description of Clack's design and capabilities and skimming it can provide lots of good information.

- Current Clack components, along with detailed descriptions are available via the Clack Component Index at http://www.clackrouter.net/componentindex.html . These are the same HTML descriptions available in the component's property views.

- Questions not handled in the above documentation can be directed to info@clackrouter.net , we're happy to help!

Section 1: Introduction to the architecture of a Clack Router

Clack routers exist in virtual networks within the Virtual Network System (VNS), which is operated at Stanford University. While the goal of Clack is to make sure that only very few details related to VNS are ever exposed to you, an understanding of how Clack interfaces with the system is important.

The Virtual Network System (VNS)

The Virtual Network System (VNS) provides the base of network virtualization upon which the Clack toolkit is built. A VNS server can host hundreds of virtual network topologies, which are made up of both physical and virtual hosts connected by virtual links. The VNS infrastructure, including servers and physical hosts, is operated and administered out of Stanford University for use by both local and remote courses. Physical hosts are real machines residing within the VNS lab that are virtualized into each topology. Virtual hosts are not physical machines but rather represent generic network processing entities with interfaces connected to other virtual or physical hosts in the topology.

After connecting to the VNS server, a client like a Clack router can take on the role of a virtual host by requesting that the server forward the client traffic seen by a particular

virtual host within a virtual topology. The client may then inspect, discard, modify, or inject new traffic into the network using any of the virtual host's interfaces, giving it the full capabilities of a real network host. Packets extracted from or injected into the network are full Ethernet frames, with the virtual host performing all packet-level processing.

The most significant contribution of VNS is that hosts on these topologies are given routable IPs and are connected to the Internet. Thus, the traffic processed by virtual and physical hosts represents real Internet protocols, allowing a student to open a command prompt, ping her router, and watch the router respond.

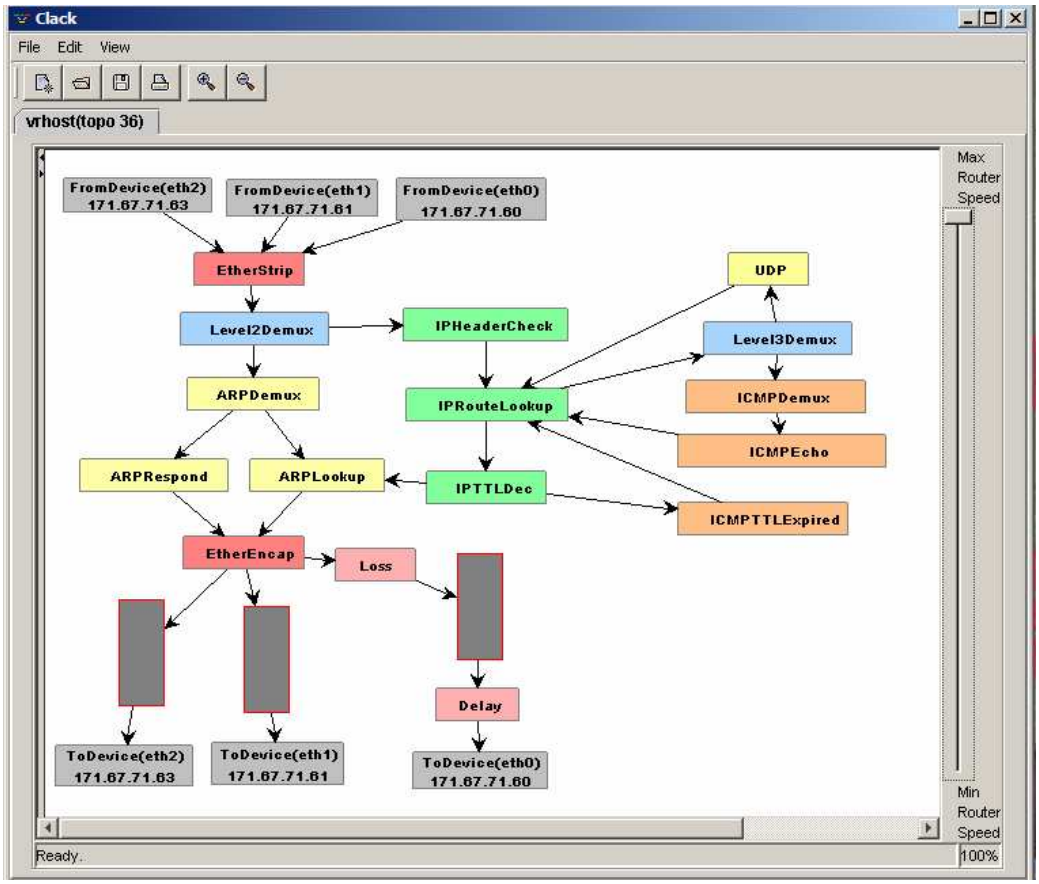
For more information about VNS visit http://yuba.stanford.edu/vns/.

Clack Basics

The Clack Router Toolkit is a Java library and accompanying graphical application that provides a platform for using VNS to achieve a variety of network educational goals. Clack can be run as a standalone Java application, or as a trusted Applet.

A Clack router is a collection of interconnected packet processing blocks that we refer to as "components." This design logic borrows heavily from the Click Router, a high-performance modular router for Linux created at MIT. As in Click, an individual component generally implements only a small portion of router functionality, keeping component design simple and easy to understand. Components perform packet processing and can keep state as configured manually by the user or automatically via a protocol. For example, the IPRouteLookup component contains a simple routing table and tags packets with an identifier based on the packet's correct out-bound interface. Components have one or more "ports," that provide channels for passing packets between different components. Each port has a direction, "In" or "Out," and semantics that are specified by the component creator. For example, the IPRouteLookup component has an "In" port for accepting packets that require a routing table lookup and an "Out" port for packets that have been tagged with the outgoing interface. Other IPRouteLookup "Out" ports carry packets sent to local IP addresses or packets for which no routing entry could be found.

Another distinction is that ports may be classified as either "Push" or "Pull." With a "Push" port, data can be sent to or from a component immediately, but in the case of "Pull" ports, the recipient only receives a packet when it asks its neighboring port for a packet. This "Pull" behavior is designed to allow for various types of packet buffering within the router.



A simple configuration for a router with three interfaces. Packets enter at FromDevice components at the top and exit the router at ToDevice components near the bottom. Components processing at layers higher in the standard OSI model are positioned further to the right and components are logically grouped by color.

Component ports are connected via unidirectional connectors called “wires.” Wires begin at an ”Out” port and end at an ”In” port and only ports of the same “Push” or “Pull” classification can be connected. Clack routers are simply a set of components connected by wires that implement the desired packet processing functionality.

Router Packet Processing

A router component initiates packet processing because of one of two possible events. First, a Clack client may recognize that the VNS server has sent it a packet destined to one of its virtual interfaces within the topology. In this case the main router processing loop pushes the packet to an input port on a FromDevice component representing the correct input interface. The packet may continue to be processed by being pushed out ports of successive components until it is discarded or reaches a queuing component. The second method for initiating packet processing allows components to register a call-back function with the router, which will be called once per processing loop. This “polling” behavior allows internal components to initiate packet processing by removing buffered packets from queues or generating new traffic.

Component and Router Implementation

Packets themselves are Java objects, inheriting from the generic VNSPacket class. Packet processing is simplified by the use of specialized protocol-specific packet classes with accessor functions for header information and data. Each component is implemented as a single Java class that inherits from the abstract ClackComponent class, which provides the basic mechanism for interoperability among all components. Ports are implemented within the ClackPort class that handles the actual transfer of packets between components. Each component object has a collection of ClackPort objects representing the description and connectivity of each logical port.

All components with push input ports implement the void acceptPacket(VNSPacket packet, int port) function to act as the starting point for packet processing. Since each port number has semantics defined by the component writer, the method knows how to handle packets passed to any port. Packets are pushed out of a component using the void pushOut(VNSPacket packet) on a ClackPort object, which sends the packet to the connected component for further processing.

For a pull input, components call the VNSPacket pullIn() method of the ClackPort class, which queries the neighboring component connected on that port to see if it has a packet to pass on. Components with output pull ports provide the VNSPacket acceptPullRequest() method to answer these queries. In most cases, a pullIn() call is initiated by the call-back function registered with the router. For call-backs, components must register with the router and implement the void poll() method of the ClackComponent interface.

Current Router Functionality

We have already implemented a base set of components, comprising a simple router. This includes ARP functionality, basic IP forwarding and ICMP handling for generating echo, TTL-expired, and port-unreachable messages. In our implementation, queues are explicit components to provide greater transparency and configurability. In addition, we have developed simplified but inter-operable TCP and UDP stacks, including a supporting socket interface that supports the creation of networking applications to run on top of these network layers.

Section 2: Developing With Clack

This section covers the basics you will need to know in order to set up your environment for developing with Clack. The remainder of this document assumes the use of Clack version 1.0.0, though later versions should work just as well.

Running Clack

While Clack is most often deployed as a Java applet, development is greatly simplified by running Clack locally.

In order to develop new components or applications for Clack, you DO NOT NEED TO DOWNLOAD THE SOURCE. Downloading the JAR file (labeled as "Binary" in the downloads section) and including it in your classpath will be sufficient.

To run Clack as seen in the website demo, simply unzip the jar file to the current directory and run (all as one line):

```
java –cp clack-1.00.jar net.clackrouter.gui.ClackLoader –u
http://www.clackrouter.net/demo/main_demo.topo
```

Clack Configuration Files

When running Clack above, we used the default Clack config file from the online demo. Clack configuration files define the following:

- Connection parameters used to contact the VNS server.
- The range of VNS virtual network topologies that the Clack client will attempt to connect to. Clack will attempt to use successive topologies in a range until it finds one that is available.
- Configuration information for each router within the topology. This includes what components are inside the router, how these components are connected by wires, and where components are located graphically within Clack's "router view".
- The graphical layout of different hosts for the "topology view" of Clack that is displayed immediately after a topology is loaded.

For very simple development, using this default file is sufficient, but if you want to either need to connect to a set of topologies separate from the public pool, or you want to automatically load a different default router configuration (ie: one with your new component automatically included), you will want to have your configuration as a local file.

While running Clack, you can modify your router to your liking, and then choose **File->Save** to save the configuration to the local file system. In this example, we use a locally saved config file called myclack.topo saved in the current directory.

We can now run:

```
java –cp clack-1.00.jar net.clackrouter.gui.ClackLoader –f myclack.topo
```

While most of the configuration file should not be edited manually, it can be useful to open the XML file in a text editor in order to change either the connection settings or the range of topologies. In either case, only the top line must be modified, which looks like:

```
<topology number="36-40" server="vns-2.stanford.edu" port="12345">
```

This line specifies that Clack will connect to the topology range 36-40 and will use the VNS server running on port 12345 of vns-2.stanford.edu.

Running Extended Versions of Clack

The default class for running Clack is the ClackLoader, which can run Clack both as an applet and as a Java application. It simple passes command line arguments to a ClackFramework object which handles everything else.

```
public class ClackLoader extends Applet {

	//When launched as an Applet
	public void init() {
		String[] params = {};
		if(getParameter(ClackFramework.PARAM_STRING) != null)
			params =
		getParameter(ClackFramework.PARAM_STRING).split(" ");
		load(params, this);
	}

	// When launched as an application
	public static void main(String[] args) {
		load(args, null);
	}

	public static void load(String[] args, Applet parent){
		// create main framework handle
		ClackFramework framework = new ClackFramework(parent);
		// configure and show the Clack application
		ClackFrameworkHelper.configureClackFramework(args, framework);
	}
}
```

Extending Clack to support new components and applications is simply a matter of letting the framework know about these new resources. To do this, create a class of your own identical to the ClackLoader class and use the addAdditionalComponent() and addClackApplication() methods of the ClackFramework class.

When compiling these new classes, you must include the Clack JAR file in your classpath. For example, if you have all of your files in the local directory, type:

javac –cp clack-1.0.0.jar *.java

If you implemented your loader class as in MyLoader.java (using the default package) and have this and any other required class files in the current directory, you can run Clack using:

java –cp .:clack-1.0.0.jar MyLoader –f myclack.topo

Note that the classpath includes two elements: the clack JAR files and the current directory.

Building from Source

While adding Clack components or applications does not require building Clack from source, you may want to build Clack from source to either make substantial modifications or to make debugging easier.

To do this, you must have Apache Ant (http://ant.apache.org) installed in addition to a java complier. Grab the source zip file from the downloads section and unzip it. From the root of this directory structure run ant on the provided build.xml file. The main Clack JAR file can then be found in the build/lib subdirectory.

Section 3: Creating Clack Components

You now know enough of the Clack background to start the fun stuff: building new router functionality that can be integrated into Clack.

EvenOdd Component

Well, here it is, your first Clack component:

```
public class EvenOdd extends ClackComponent {

        public static int PORT_IN = 0, PORT_ODD_OUT = 1, PORT_EVEN_OUT = 2;
        public static int NUM_PORTS = 3;

        public EvenOdd(Router r, String name){
                super(r, name);
                setupPorts(NUM_PORTS);
        }

   protected void setupPorts(int numPorts){
        super.setupPorts(numPorts);
        createInputPushPort(PORT_IN,"input", VNSIPPacket.class);
        createOutputPushPort(PORT_ODD_OUT, "odd packet output",
VNSIPPacket.class);
```

```
        createOutputPushPort(PORT_EVEN_OUT, "even packet output",
VNSIPPacket.class);
    }

        public void acceptPacket(VNSPacket packet, int port_number){
                if(port_number == PORT_IN){
                        VNSIPPacket ippacket = (VNSIPPacket)packet;
                        byte[] dst_bytes =
ippacket.getHeader().getDestinationAddress().array();
                        if((dst_bytes[3] & 0x1) == 0){
                                log("Even!");
                                m_ports[PORT_EVEN_OUT].pushOut(packet);
                        }else {
                                log("Odd!");
                                m_ports[PORT_ODD_OUT].pushOut(packet);
                        }
                }else {
                        /* impossible */
                }
        }

}
```

As you can tell, we've skipped the classic "hello world" example and have gone right for a serious example. This component receives IP packets and sends the packet out one of two output ports depending on whether the final byte in the IP address is even or odd.

The first thing to notice is that this class, like ALL Clack components, extends the base ClackComponent class. This base class provides a significant amount of common functionality used in all components.

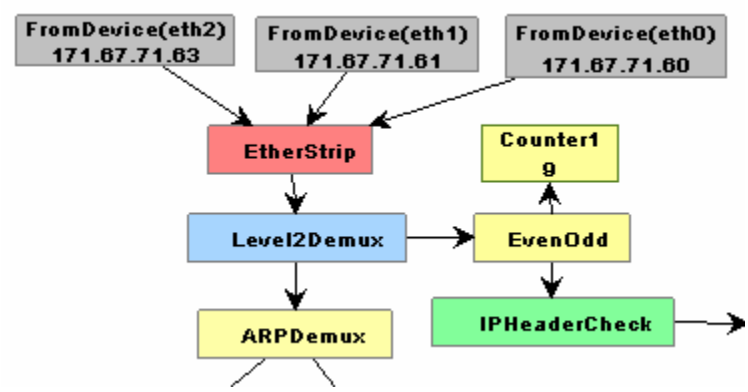
The type and meaning of the component's ports are defined in the setupPorts() method. The call to the superclass method allocates three ports for the ports, and then we create three individual ports, specifying the port number, a text description, and typing information to describe what kind of packets may be passed through the ports. All of these ports are typed to accept IP packets.

The other main method, acceptPacket() implements packet processing when a packet is received via the input port. For packets that have arrived via our input port, we cast the packet as an IP packet, and then get the bytes representing the destination address from the packet header. Then depending on the even/odd value of the address, we push a packet out one of two output ports. We also log a simple statement to the console for each packet.

Now let's run this component! Take the loader class that you created in Section 2, and add just a single line:

```
framework.addAdditionalComponent("EvenOdd",
"net.clackrouter.example.EvenOdd");
```

Now the “Additional” submenu of the Clack Add component menu will give you the option of adding an EvenOdd component. We want to insert this component into our router. This requires finding a place were we will be processing IP packets. Delete the connection between the “Level2Demux” and the “CheckIPHeader” components. Add an “EvenOdd” component between the two connected the IP packet output of the demux to the input of the EvenOdd component. Connect the even output of the EvenOdd component to the CheckIPHeader component, which will result in packets to even addresses being forwarded correctly through the router. Now, add a “Counter” component from the “auxiliary” sub-menu, and connect the odd output of the Even odd component to the Counter input.



The integration of your “EvenOdd” component into the router.

Double-click on the EvenOdd component to bring up its property view. Click on the “Ports” subtab to verify that the correct port connections have been made.

Now you can use File -> Save to save this file as your new “myclack.topo” file. This means this configuration will be loaded automatically each time your run Clack.

So let’s test the router. First, ping an IP address on the router that has a last byte that is even (e.g. 171.67.71.60). This should work, since the even output port forwards the packets to the IPHeaderCheck component. Look at the Log in the property view to make sure your component is reporting an even packet. Now ping an odd IP address (e.g. 171.67.71.61). No ping response will be generated, since your EvenOdd component does not forward the packet to your router’s ICMPEcho component, but instead sends it to the Counter which drops the packet.

Congratulations! You now have seen all of what you have to know to create many types of basic Clack components. We encourage you to experiment with additional modifications to this example component.

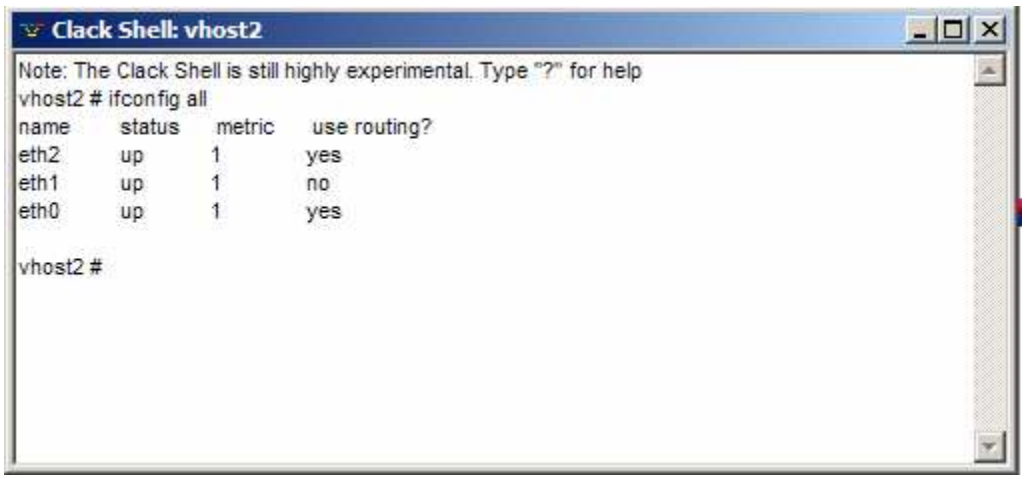
Once you are comfortable with how Clack components work, the best way to learn more and start getting involved with Clack development is to take a look at existing components in the net.clackrouter.component.* packages and explore how they work. Try modifying these components to create new ones of your own. If you have an idea for a component, but don't know how to code some functionality up, think about what other components have to do something similar and look to them for assistance.

Section 4: Clack Applications

Clack also can run applications that run in a simple "Clack Shell". These applications can be used to:

1) View or configure router state.
2) Generate interesting traffic to visualize in your router.

The Clack Shell can be launched by clicking on the shell icon in the toolbar, or using View -> SpawnClackShell . A shell is specific to an individual instance of a Clack router, and multiple shells can be launched to interact with different routers.



The Clack Shell, used for launching various Clack applications. Here a simple clone of the 'ifconfig' function displays basic information about the router interfaces.

Your First Clack Application

Applications are launched using a simple command name, and can be given command-line parameters that are parsed by the application. Applications are run as full Java threads and as a result can perform very powerful functionality. However, we of course start with a very simple example:

```
public class HelloApp extends ClackApplication {

        public void application_main(String[] args){
```

```
                print("hello...\n");
                if(args.length > 1){
                        print("You said: " + args[1]);
                }else {
                        print("You're mighty quiet today...");
                }
        }

        public String getDescription() { return "silly example application"; }

}
```

A couple of things to note: All applications subclass the ClackApplication, which provides various methods that provide a back API to the "system", such as printing to console, accessing sockets, etc. When a command is run, its command line arguments (including the command name itself) are passed to it in an arguments array, similar to any standard application you would write. Here we simply use the *print*() command to send data to the shell output. getDescription() is used to provide a brief description of the program when the user queries the shell about what applications are installed by typing "?" at the command prompt.

But how does Clack know about a new applications that you write? Simple: you inform the ClackFramework object when you are loading Clack. This is very similar to how we informed Clack about newly written components in Section 3. To use the abovedd the following line to your loading method (see net.clackrouter.example.ExampleLoader):

```
framework.getApplicationManager().addApplicationMapping("hello",
"net.clackrouter.example.HelloApp");
```

The first string argument specifies the command name that needs to be entered into the shell to run the program. The second is the class name that is run to perform the requested operations.

Accessing and Modifying Router State Using Applications

A major purpose of the Clack Shell is to provide an easy way to configure parts of the router without requiring any GUI coding on behalf or the component creator. For example, the following code snippet shows a bit of code from a simplified implementation of *Ifconfig*.

```
    public void application_main(String[] commands){
        Router router = getRouter();

        String iface_name = commands[1];
        if(commands[2].equals("up")) {
                router.updateLinkStatus(iface_name, true, true);
```

```
                print("Bringing up interface " + iface_name + "\n");
        }

    }
```

This code demonstrates the use of the *getRouter()* method, which provides a reference to the router that the shell is currently being run on. Another common paradigm for modifying are arbitrary component is to loop through all components contained by the router and modifying any that are of a certain type. For example, the Clack Shell implementation of *ping* contains the following code snippet:

```
ClackComponent[] all_comps = getRouter().getAllComponents();
for(int i = 0; i < all_comps.length; i++){
        if(all_comps[i] instanceof ICMPEcho) {
                ICMPEcho echo = (ICMPEcho)all_comps[i];
                echo.addListener(this);
                for(int j = 0; j < pingCount; j++){
                        send_times[j] = System.currentTimeMillis();
                        echo.sendEchoRequest(addr, identifier, j);
                        Thread.sleep(SLEEP_TIME_MSEC);
                }
                break;
        }
}
```

Using TCP/UDP Sockets in a Clack Application.

Finally, Clack applications can also leverage a specialize socket layer to use TCP or UDP functionality implemented within the Clack router. Because socket interfaces tend to be fairly complicated, it is best to check out the javadoc for the *TCPSocket* and *UDPSocket* classes to get the full details of how they operate. However, if you are familiar with common socket terminology, using these sockets should be a snap as the API is a simplification of common socket interfaces.

Here is the application_main() function for a Clack client application that performs a simple HTTP GET / to a provided destination:

```
public void application_main(String[] args) {
        try {
                TCPSocket socket = createTCPSocket();
                InetAddress localAddr =
                getRouter().getInputInterfaces()[0].getIPAddress();
                InetAddress remoteAddr = InetAddress.getByName(args[1]);
                if(args.length > 2)
                        localAddr = InetAddress.getByName(args[2]);
```

```
                StringBuffer sBuf = new StringBuffer(200);
                print("HTTPGetter is connecting to " + remoteAddr);
                Random rand = new Random();
                socket.bind(localAddr, rand.nextInt(5000));
                socket.connect(remoteAddr, 80);
                Thread.sleep(500);
                socket.send(ByteBuffer.wrap("GET / HTTP/1.0\r\n\r\n".getBytes()));

                while(true){
                        ByteBuffer buf_in = socket.recv(2000, 1000);
                        if(buf_in == null) break;
                        if(buf_in.capacity() > 0)
                        sBuf.append(new String(buf_in.array()));
                }
                socket.close();
                print("HTTP Get Request returned: \n" + sBuf);
        }catch (Exception e){
                e.printStackTrace();
        }
}
```

Similarly, here is a bit of code that implements a VERY basic webserver with a single static response:

```
String resp = "HTTP/1.0 200 OK\r\nContent-Type: text/html\r\n\r\n<html> <h1>
HELLO </h1> </html>\r\n\r\n";
ByteBuffer resp_buf = ByteBuffer.wrap(response.getBytes()));
socket.bind(localAddr, 80);
socket.listen();

while (true) {
        print("MiniWebServer is listening on port 80 of "     + localAddr + "\n");
        TCPSocket clientSocket = socket.accept(); // blocks until connection is received

        print("MiniWebServer received a connection! \n");
        ByteBuffer request_buf = ByteBuffer.allocate(0);
        while(request_buf.capacity() == 0)
                request_buf = clientSocket.recv(2000, 2000);

        String request_str = new String(request_buf.array());
        print("MiniWebServer received:\n'" + request_str + "'\n");

        clientSocket.send(resp_buf);
        clientSocket.close();
}
```

More examples of Clack applications can be found in the *net.clackrouter.applications* package. Note that you can also use "real" Java sockets in clack applications in order to connect to the Internet without going through Clack. This can be useful for "redirecting" traffic from elsewhere on the Internet so that it appears to be coming from within Clack. See *TCPRedirector* and *UDPRedirector* in *net.clackrouter.application* for examples of this.

Thanks for reading the "Hack Clack" developer documentation. Feel free to contact us at feedback@clackrouter.net with any questions or comments!

- The Clack Development Team