Jiazzi Network Simulator

Douglas Creager Todd Nightingale

16 May 2002

Abstract

JNS utilizes the modularization and separate compilation features of the Jiazzi extensions to Java. By doing so, the major components of a simulated network can be defined completely independently of each other. Futhermore, by redefining the core components of the simulator itself, Jiazzi's modularity can allow different simulation engines to perform the analysis, without having to redefine or recompile any other portion of the simulator. Most importantly, the mixin capabilities of Jiazzi allow component designers to extend existing components piecewise, without having to worry about the details of the underlying component implementation. These features make JNS an extremely extensible and pluggable network simulation tool.

1 Motivations

An important tool for the network protocol designer is the network simulator. It allows the designer to test their protocols in a contained, well-defined environment, providing feedback on the protocol's behavior in various circumstances. Several tools already exist to perform and analyze such simulations. However, they have several drawbacks. For instance, the standard tool used in academia, NS, suffers from an incrementally-designed system that leaves several features implemented in an extremely ad hoc manner. For instance, routing, one of the most important features of any collection of network protocols, has to be implemented as a special case; it is not a simple extension of the base system. Because of this, all connections between two computers must contain in their representation hard references to their target. (See the dotted lines in figure 1.) These references do not represent any physical connections between the protocols; ideally, this means that the simulated system should be able to work like the real-world system, without these links.

The Jiazzi Network Simulator (JNS) is designed from the ground up to be extremely extensible and pluggable. By using the units framework provided by Jiazzi, an extension to the Java programming language, these goals are accomplished. New protocols can be implemented and compiled completely separately from the rest of the system. Existing protocols can be extended piecewise using the Jiazzi's mixin support. Core components of the system can be reimplemented for speed or efficiency, and linked into the rest of the code base without requiring any extra recompiling. Finally, network topologies are defined in terms of which general type of protocol is used (*e.g.*, TCP, UDP, ICMP, etc.), rather than any specific implementation. The necessary pieces are not linked together until simulation time, and all of these links are specified in one, compact place.

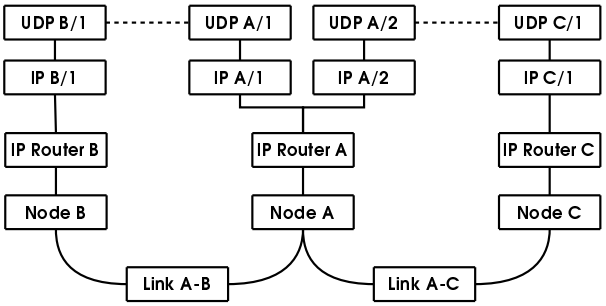

Figure 1: Network topology

2 Standard interfaces

To prevent unnecessary module dependencies from arising, all network components in JNS implement one of the interfaces defined in the *interfaces.base* signature (see table 1). An alternative design to this would be to declare all components in terms of Jiazzi class signatures. However, doing so would limit the connectivity between the JNS components. For instance, the scheduler, as defined, can handle any implementation of the Event interface, without introducing any dependencies between the scheduler and the module that defines the event.

Event	Scheduler events
OutputHandler	Output handlers
Packet	Data packets
PipelineElement	Packet pipeline elements
Queue	Packet queues

Table 1: Standard interfaces

An Event represents anything that be scheduled to happen at a particular simulation time. Links, for instance, will define an implementation of Event to represent a packet arriving at its destination. OutputHandlers can be defined to handle any debug or trace output that the network components generate. Packets represent the pieces of data being sent around the network; typically, each network protocol will also define its own type of Packet. As such, Packets are defined with a default encapsulation behavior; each Packet implementation must define a getInnerPacket() method. A base Packet, which would just contain a sequence of arbitrary application data, would define this method to return null. The Queue interface defines the methods necessary for a packet queue, which is used by links and protocols alike to allow different queuing strategies to be investigated in a similarly pluggable fashion.

The PipelineElement interface requires slightly more explanation. The simulated network topology and protocol is abstracted out to a slightly more general view, that of a packet pipeline. In this view, all nodes and protocols are merely connections in large collection of packet plumbing. Each element of this plumbing determines the method by which packets are transported around. To model this, these elements of the simulation all implement the PipelineElement interface, which specifies the methods for sending packets in both the upstream and downstream directions.

Since new protocols and new packet types are most often defined together, a *module.def* signature is provided to supply protocols and packets to the simulator as a pair. New modules will implement signatures that extend *module.def* to provide the appropriate constructors and accessors to their dependent units.

The core components around which the entire simulator is defined are not defined in terms of a Java interface like the individual components. Rather, they are defined strictly as Jiazzi units, with the appropriate package signatures. Since, in the current model, every component in a given network must refer to a single, specific simulation engine, these are specified as module dependencies and linked in as Jiazzi units.

One of these core components is the event scheduler, defined by the *scheduler.base* signature. The event scheduler publishes methods to deal with queuing new events into the scheduler and sending output to the user. Various output handlers can be registered with the scheduler; when any network component subsequently calls one of the scheduler's output functions, this call is propagated to all registered output handlers.

To further eliminate module dependencies between the scheduler core and the various network components, a second core unit is defined. Prior to passing control of the simulation to the scheduler, the simulation engine delegates the initialization of a network topology to an initializer unit, which implements the *initializer.base* signature. Doing so confines to a very specific, limited portion of the code both the description of the topology itself, and the specification of which component modules are needed for a given simulation.

The last core component in the system is the *program.base* signature, which defines the entry point to the simulation engine. The provided implementation provides the glue between the initializer unit and the scheduler unit.

3 Base classes and units

In addition to the interfaces and signatures described above, the base JNS system also includes some abstract classes to aid in the development of new components, as well as a default implementation of base data packets and the IP, UDP, and TCP protocols (defined by the *module.base*, *module.ip*, *module.udp*, and *module.tcp* signatures).

3.1 General plumbers and protocols

As described above, most elements of a simulated network deal with the transport of packets through a pipeline, as defined by the PipelineElement interface. A default implementation of this interface is provided, where pipeline elements can be connected in a tree structure. Each element has a single parent downstream, and multiple children upstream. The GeneralPlumber class, defined by the *plumber.base* signature, provides methods for registering and deregistering children, and implements the default transport behavior in both directions. Typically, a network node will be the parent of a tree of protocols, providing the standard end-to-end layout of functionality.

A futher generalization comes from the fact that most network protocols behave in the same way, very similar in function to the generalized plumber described in the previous paragraph. Like plumbers, they exist in a tree of pipeline elements; however, they almost always perform the additional task of encapsulating a packet within another before passing it further downstream. The GeneralProtocol class, defined by the *protocol.base* signature, extends the GeneralPlumber class to provide this behavior.

Note that this tree of pipeline elements exists at a single network node, where there is assumed to be instantaneous communication between the layers of the protocol stack. Because of this, no new scheduled events are needed to completely process a packet traversing the tree.

3.2 Network topology

The actual topology of a network is constructed with the Node and Link classes, defined in the *graph.base* signature. An example topology can be found in figure 1. As mentioned in the introduction, the dotted lines indicate extra links that would appear in an NS representation of the topology; in JNS, only the solid-line connections are needed.

Nodes in a network are implemented as PipelineElements, with a null downstream parent and a stack of protocols upstream. For instance, in a representation of a standard TCP/IP network, each node would be connected to an IP protocol object for each open connection on the node. Each of these IP protocols would be connected to an appropriate TCP or UDP protocol. Finally, each of the TCP and UDP protocols would be attached to the appropriate application protocols (HTTP, FTP, etc.).

In the opposite direction of the protocol stack is the network itself, where Nodes are connected to each other via Links. These Links are not defined as PipelineElements, since there is no inherent notion of upstream or downstream. It is true that some network layouts impose an upstream/downstream view onto the topology; however, this is a property of the specific topology in use, not of topologies in general. Futhermore, in the case of a router, a Node would have more than one Link in the "downstream" direction, violating the tree structure of the PipelineElement design.

Instead of defining Links in terms of the PipelineElement interface, they are defined as unidirectional connections between two nodes, with physical delay and bandwidth parameters, respectively expressed in seconds and bytes per second. When a source Node receives a downstream-traveling packet, it sends it to the specified Link. This Link sends it across the simulated physical link to its destination by scheduling a packet arrival Event for the appropriate time in the future. When the appropriate simulation time occurs, the Link processes the Event by sending the packet upstream to the destination Node.

Every Link also has an associated Queue, which handles the Link's behavior when its source Node is sending data faster than the physical link can transport. The Queue can also handle prioritization of the Packets in the queue to simulate bandwidth-sharing protocols. However, "smart" queues like this will usually not be inserted into the topology at the Link level; rather, they will most likely be used with a more complicated protocol further upstream at the appropriate Nodes.

4 Mixins

One of the features of JNS most useful to the component designer is Jiazzi's mixin capabilities. By using this language feature, the component designer can create new implementations of an existing protocol by redefining a few small aspects of the component's behavior. Furthermore, these behavioral changes are defined independent of the underlying implementation.

For instance, there are several extra features that can be turned on or off in TCP, yielding different performances based on the circumstances of the individual network. By specifying each of these features as a separate mixin, they can be successively applied to a base implementation of TCP, yielding a fully functional protocol with the desired behavior, all specified at simulation time.

5 Simulation execution

Assuming that the necessary protocol units are available, the simulation user must currently write two Jiazzi units, one atomic and one compound, in order to perform a simulation. The atomic unit is an instance of the *initializer.base* signature, which is responsible for setting up the network topology and providing any initial Events to the scheduler. Since this initializer is defined as an atomic unit, its dependencies are still expressed as package signatures, not concrete package implementations. This flexibility allows the same topology initializer to be used with different implementations of the desired protocols in the network.

The second unit written by the user is a compound

net.tnight.jns.interfaces	Standard interfaces
net.tnight.jns.scheduler	Event scheduler
net.tnight.jns.core	Component glue
net.tnight.jns.graph	Network topology
net.tnight.jns.plumber	GeneralPlumber
net.tnight.jns.protocol	GeneralProtocol

Table 2: Necessary units

net.tnight.jns.stderr	Standard error output handler
net.tnight.jns.base	Base data packages
net.tnight.jns.ip	IP protocol
net.tnight.jns.udp	UDP protocol
net.tnight.jns.tcp	TCP protocol
net.tnight.jns.taildrop	Naïve tail-drop packet queue

Table 3: Other predefined units

unit which glues together concrete implementations of all the units necessary for the simulation. In general, all of the units listed in table 2 will be required, along with units implementing necessary protocols (see table 3), and the initializer unit written by the user. Because of the number of units necessary in the linking stage, an effort has been made to name unit dependencies in a consistent manner. This allows the user to use the compact `bind to *@package` syntax, reducing the size of the `bind package` clause of the unit specification.

6 Conclusions

Once the proper underlying framework was designed, the Jiazzi units allowed the base protocols to be designed extremely quickly. True, there is nothing cutting-edge about the provided protocols; however, these simple versions of the protocols were surprisingly quick and easy to implement. Jiazzi's separate compilation features really helped facilitate rapid design and implementation.

If designed properly, however, the modularity and pluggability features of JNS could easily have been accomplished using standard Java. While the Jiazzi units allow the modularity of the design to be expressed explicitly, they are by no means necessary. The scheduler is defined to work strictly on instances of standard Java interfaces; there is no inherent reliance on Jiazzi units. By using a bit of the Reflection API, we can even use a Java-based initializer class to eliminate any name-dependency of the scheduler on the specific protocols, just as in the Jiazzi version.

The place where Jiazzi really benefits the JNS, and where Java cannot help, is in the use of mixins to help extend existing protocols piecewise. Especially in a field where networking research involves incremental changes and tweaking of existing algorithms, this form of piecewise subclassing can be extremely invaluable.

In conclusion, the extensions to the Java language provided by Jiazzi helped JNS achieve a level of modularity, pluggability, and quick, piecewise extensibility that existing network simulators do not exhibit.

7 Appendix: Code

Attached are illustrative excerpts from the JNS code. Presented first are the signatures of several key components of the system: the standard interfaces unit, the default network topology unit, the scheduler unit, the packet/protocol module prototype, the output handler prototype, and the initializer prototype (figures 2 through 7).

Following the signatures is the code to the Graph unit's Link class (figures 8 and 9), showing an example of scheduling Events to handle events that take place over a defined period of time.

Finally, the implementation of UDP is presented as an example of extending the *module.def* signature to add a protocol and packet to the simulator (figures 10 through 12).

```
signature interfaces.base =
{
    package interfaces;

    public interface Packet
    {
        public int getSize();
        public interfaces.Packet getInnerPacket();
        public int getID();
        //public String toString();
    }

    public interface PipelineElement
    {
        public boolean canAcceptPacket(interfaces.Packet p);
        public void send(interfaces.Packet p, int path);
        public void recv(interfaces.Packet p);
    }

    public interface Event
    {
        public void process();
    }

    public interface OutputHandler
    {
        public static final int  ERROR;
        public static final int  WARNING;
        public static final int  DEBUG;

        public void log(int level, String message);
    }

    public interface Queue
    {
        public boolean isEmpty();
        public void enqueue(interfaces.Packet packet);
        public interfaces.Packet dequeue();
    }
}
```

Figure 2: *interfaces.base*

```
signature graph.base = {
    package interfaces, graph, plumber, scheduler;

    public class Node
    extends plumber.GeneralPlumber
    {
        public Node(String name);
        public void addLink(graph.Link link);
        public void removeLink(graph.Link link);
    }

    public class Link
    {
        public Link(graph.Node from,
                    graph.Node to,
                    double     delay,
                    double     bandwidth,
                    interfaces.Queue queue,
                    String     name);

        public void enqueuePacket(interfaces.Packet packet);
    }
}
```

Figure 3: *graph.base*

```
signature scheduler.base = {
    package interfaces, scheduler;

    public class Scheduler
    {
        public static void schedule(interfaces.Event event, double time);
        public static double now();
        public static void run();
        public static void run(double timeout);

        public static void registerOutputHandler(interfaces.OutputHandler handler);
        public static void log(int level, String message);
    }
}
```

Figure 4: *scheduler.base*

```
signature module.def = {
    package interfaces;

    public class Packet implements interfaces.Packet {}
    public class Protocol implements interfaces.PipelineElement
    {
        public interfaces.Packet encapsulate(interfaces.Packet);
        public interfaces.Packet decapsulate(interfaces.Packet);
    }
}
```

Figure 5: *module.def*

```
signature output.base =
{
    package interfaces;

    public class OutputHandler implements interfaces.OutputHandler {}
}
```

Figure 6: *output.base*

```
signature initializer.base =
{
    public class Initializer
    {
        public static void setup(String[] args);
        public static void cleanup();
    }
}
```

Figure 7: *initializer.base*

```
package out;

import interfaces.*;
import scheduler.*;
import java.util.*;

public class Link
{
    private Node    from, to;
    private double  delay, bandwidth;
    private String  name;
    private Queue   waiting;
    private List    pipe;
    private boolean idle = true;

    public Link(Node from, Node to, double delay,
                double bandwidth, Queue queue, String name)
    {
        this.from = from;
        from.addLink(this);
        this.to = to;
        this.delay = delay;
        this.bandwidth = bandwidth;
        this.name = name;
        this.waiting = queue;
        this.pipe = new ArrayList();
    }

    private class PacketArrivalEvent implements Event
    {
        Packet  packet;

        private PacketArrivalEvent(Packet packet) { this.packet = packet; }
        public String toString() { return "Packet arrival on "+name; }
        public void process() { packetArrived(packet); }
    }

    private class LinkIdleEvent implements Event
    {
        public String toString() { return "Link "+name+" idle"; }
        public void process() { linkIdle(); }
    }

    public void enqueuePacket(Packet packet)
    {
        Scheduler.log(25,"enqueue "+packet);
        waiting.enqueue(packet);
        processWaitingQueue();
    }
```

Figure 8: Link class

```
    private void processWaitingQueue()
    {
        if (idle && !waiting.isEmpty())
        {
            Packet  packet = waiting.dequeue();
            Scheduler.log(25,"piping "+packet);
            pipe.add(packet);
            Scheduler.schedule(new PacketArrivalEvent(packet),delay);
            Scheduler.schedule(new LinkIdleEvent(),packet.getSize()/bandwidth);
            idle = false;
        }
    }

    private void packetArrived(Packet expected)
    {
        Packet  received = (Packet) pipe.get(0);
        if (!expected.equals(received))
            Scheduler.log(5,"Packet arrival out of order on link "+name);
        pipe.remove(received);
        to.recv(received);
    }

    private void linkIdle()
    {
        idle = true;
        processWaitingQueue();
    }
}
```

Figure 9: Link class (cont.)

```
signature module.udp = z: module.def + {
    package interfaces, protocol;
    bind package interfaces to z@interfaces;

    public class Packet implements interfaces.Packet
    {
        public Packet(interfaces.Packet inside,
                      int sourcePort,
                      int destinationPort);
        public int getSourcePort();
        public int getDestinationPort();
    }

    public class Protocol
    extends protocol.GeneralProtocol
    implements interfaces.PipelineElement
    {
        public Protocol(String name,
                        interfaces.PipelineElement target,
                        int source, int dest);
    }
}
```

Figure 10: *module.udp*

```
package out;

public class Packet
    implements interfaces.Packet
{
    private interfaces.Packet inside;
    private int sourcePort, destinationPort;

    public Packet(interfaces.Packet inside,
                  int sourcePort,
                  int destinationPort)
    {
        this.inside = inside;
        this.sourcePort = sourcePort;
        this.destinationPort = destinationPort;
    }

    public int getSourcePort() { return sourcePort; }
    public int getDestinationPort() { return destinationPort; }

    public interfaces.Packet getInnerPacket() { return inside; }
    public int getSize() { return 8+inside.getSize(); }
    public String toString() { return "UDP["+inside+"]"; }
    public int getID() { return inside.getID(); }
}
```

Figure 11: UDP Packet class

```
package out;

import interfaces.PipelineElement;
import protocol.GeneralProtocol;

public class Protocol
    extends GeneralProtocol
    implements PipelineElement
{
    private int source, dest;

    public Protocol(String name, PipelineElement target, int source, int dest)
    {
        super(name,target);
        this.source = source;
        this.dest = dest;
    }

    public boolean canAcceptPacket(interfaces.Packet p)
    {
        if (p instanceof Packet)
        {
            Packet udp = (Packet) p;
            return (udp.getSourcePort() == source) && (udp.getDestinationPort() == dest);
        }
        return false;
    }

    public interfaces.Packet encapsulate(interfaces.Packet p)
    {
        return new Packet(p,source,dest);
    }

    public interfaces.Packet decapsulate(interfaces.Packet p)
    {
        if (p instanceof Packet)
        {
            Packet udp = (Packet) p;
            return p.getInnerPacket();
        }
        scheduler.Scheduler.log(0,"Decapsulating non-UDP packet");
        return null;
    }

    public void send(interfaces.Packet p, int path) { super.send(p,path); }
    public void recv(interfaces.Packet p) { super.recv(p); }
}
```

Figure 12: UDP Protocol class