Center for Information Technology Integration
Project Report
June 30, 2003

This report discusses progress on the tasks outlined in a Memorandum of Understanding dated October 2, 2002, which constitutes the joint understanding of CITI and ITCom in pursuing a research and development partnership.

Statement of Work

The primary goal of the partnership for FY 2002-2003 is to identify and enhance network test and performance tools that can help ITCom staff build and maintain a more secure and functional network at the University of Michigan. The first focus is on Iperf, an open source network measurement tool from the University of Illinois, with the goal of developing an authenticated Iperf service running on multihomed servers placed strategically in a VLANed network.

Task 1: Globus integration

In this task, CITI will make a Globus service that provides remote network test and measurement to authorized users. ITCom is currently deploying measurement stations throughout the network. This task will enhance the test bed by providing a web interface, authorized access, and strong authentication. This task leverages CITI's prior efforts in Authenticated Quality of Service and KX.509.

The initial implementation will build on Iperf; later enhancements that allow extended functionality within the Globus framework are anticipated. Authorization can be based on environmental and Iperf parameters such as user identity and group membership, CPU load on the test harness, time-of-day, TOS, TTL, TCP or UDP, bandwidth (UDP only), etc.

We extended our Authenticated Quality of Service work based on Globus 1.3 GARA to run Iperf between two performance monitoring platforms (PMP), and demonstrated it in November 2002. Users with kx509 credentials run Iperf via an https web interface. The web server runs kx509 proxy software and the GARA client. Authorization based on Iperf input parameter values, environmental parameter values, and group membership was demonstrated. This version of the code was not flexible; it was difficult to add new network testing tools, output was stored only on the node's local file system, and scheduling of network testing was not implemented.

We rebased to Globus 2.4, added a PHP module to the web server, and rewrote portions of the GARA client and service. The PHP module creates a web page based on a configuration file. After a PHP module reads in user's values, it encodes the request in Globus's RSL format and uses it as an input to the GARA client module also running on the web server. We rewrote the GARA client and service to pass the RSL all the way through the Globus gatekeeper to the GARA service, which adds the RSL along with other necessary information to the slot manager. The slot manager is the portion of the GARA service which schedules jobs in the future. We extended the slot manager to be

able to schedule arbitrary jobs (not just bandwidth brokering). The slot manager reads a configuration file to locate the program to run for a scheduled job. The infrastructure handles synchronous and asynchronous requests. If the user places a synchronous reservation, output is returned to the web server and displayed to the user, as well as being stored on the network test node local filesystem. If the user places asynchronous request (ie., a reservation in a future), the results of the execution are stored on the network test node filesystem. Please see Appendix A for more details about PHP module and the extended GARA infrastructure.

The result of this work is the ability to add a new network testing tool to the Globus PMP service by constructing a web page via a PHP configuration file, and adding the test executable location to a configuration file on the server. The service does not need to be stopped or compiled to add a new test tool. Changes in authorization policies still require a compilation.

We demonstrated this new Globus service to Roy Hockett – we ran traceroute, ping, and Iperf by scheduling a short time into the future.

Task 2: Multihomed Iperf stations

An Iperf platform that supports multiple interfaces can be situated at a nexus of networks and used to test them all. For example, such a platform could be used to send test data out one interface, into the Internet, and collect it on another. However, ordinary UNIX routing specifies a default route per host, not per interface. Consequently, test data arriving on one interface may be returned on another, which interferes with the goal of multihomed Iperf stations. This task will extend the Linux kernel routing framework so that interfaces can support routing behaviors that are more complex.

Using a Linux-based policy routing strategy via the `iproute2` package, we implement one routing table per VLAN attached to an Iperf station, and a routing policy that selects the correct routing table based on the source address of the outgoing packet. This effectively emulates a default route per virtual interface.

Correct operation has been verified on a CITI testbed consisting of two multihomed Iperf stations and two Linux boxes set up as routers.

We include in the project deliverables a sample configuration file for defining the correct policy routes and a script, `ntap-config`, for installing them into an Iperf station, as well as scripts for post-processing `tcpdump` output for verifying correct operation and for listing the policy routes in a compact format.

Please see Appendix B for a detailed description of our testbed architecture, sample packet flow, and a description and usage guide to `ntap-config`.

Task 3: Performance of multihomed test stations

Network tool performance is tied to a number of parameters, such as aggregate bandwidth and CPU speed. In this task, we will model and measure the multihomed network test station to determine the limits of its performance as interfaces are added. With this understanding in hand, we will determine the feasibility of using a multihomed network test station as both source and sink of test data.

Our performance tests have determined that a PMP can produce a maximal Iperf throughput of 2.5 Gbps using the loopback interface on a single host, and 940 Mbps between two hosts connected by a private 1 Gbps Ethernet network. These results were obtained for TCP using standard frame sizes.

Thse two results together suggest that bottlenecks exist in one or more of the system buses, memory system, DMA system, driver, or network card of a PMP. Further testing of simultaneous data sources and sinks over multiple physical interfaces can take place once we have three PMPs in place in the ITCom lab.

Please see Appendix C for detailed performance results.

Task 4: Remote testing

This task develops a user-friendly application that authorized users can run on their desktops to test and measure network performance, both end-to-end and segment-by-segment. This application will take parameters such as TCP or UDP port number, TOS/Diffserv code point, bandwidth, destination, etc. The application will find test stations along the path to the destination and run end-to-end and segment-by-segment tests on them.

The service described in Task 1 is web based, and allows segment-by-segment testing between PMPs, leaving the sending host to first PMP segment and the last PMP to receiving host segment untested.

Shawn McKee (smckee@umich.edu) is the chairman of the Internet2 end-to-end technical advisory group (TAG) which is working with developers from CalTech to write a Java applet to run network tests between hosts. As a start, they have implemented a Java applet that runs Iperf against a server that has the Web100 kernel patch which exposes details of the network stack to the OS. By building our PMPs with the Web100 patch, we can test the host to PMP network segments.

Our current design for the discovery of the network segments and therefore the addresses of the PMPs for the segment-by-segment tests uses `traceroute` for path discovery and an LDAP database to map subnets to network node addresses. The LDAP schema used is an initial primitive implementation, and communication is done via SASL/GSSAPI/ Kerberos V5 if required.

Using `traceroute` for segment discovery is problematic; while `traceroute` identifies the correct router, it returns the address of the wrong interface. Further conversation with Roy Hockett has determined that this issue will not occur for Internet2 routers and has produced an algorithm for resolving this issue in the end-system routers.

Milestones and Deliverables

September 2002

Task 1 begin

Task 2 begin

November 2002

Task 1 deliverable: Globus service that provides remote Iperf network test and measurement to authorized users.

Task 1 continue

Task 4 begin

January 2003

Task 2 deliverable: Extend the Linux kernel routing framework so that complex routing behavior can be set for each interface.

Task 2 end

Task 3 begin

May 2003

Task 1 deliverable: Enhanced Globus service that provides remote network test and measurement to authorized users.

Task 3 deliverable: Report on performance of multihomed network test station performance when used as both source and sink of test data.

Task 3 end

June 2003

Task 4 deliverable: User-oriented Linux application that authorized users can run on their desktops to test and measure end-to-end and segment-by-segment network performance.

Task 4 end

Appendix A

PHP module

The web interface is managed by a PHP script that builds the parameter page and processes the input data. The script is driven by a configuration file that details the options to be passed to the new program. The script outputs a string containing the parameters in a form suitable for input to the GARA module operations described in the next section.

GARA module

The GARA web module is constructed from four simple operations provided by the GARA infrastructure: `reservation_create`, `reservation_cancel`, `reservation_status`, and `reservation_callback`.

`The reservation_create`() function places a reservation by encoding the user's request in the RSL format. An RSL request is a collection of attribute and value pairs with certain syntax restrictions. A typical RSL for a `traceroute` request contains all available command-line options and looks like:

"`&(reservation-type=network) (endpoint-a=141.211.133.59) (endpoint-b=141.211.133.121) (packet_length=--EMPTY) (first_ttl=--EMPTY--) (nofrag=--EMPTY--) (sockdebug=--EMPTY--) (gateway=--EMPTY--) (iface=--EMPTY--) (useicmp=--EMPTY--) (max_ttl=30) (usenumaddr=--EMPTY--) (port=--EMPTY--) (norouting=--EMPTY--) (n_queries=3) (src_addr=--EMPTY--) (tos=--EMPTY--) (verbose=--EMPTY--) (wait_time=5) (checksums=--EMPTY--) (pause_msecs=--EMPTY--) (start-time=now) (bandwidth=0) (duration=2) (protocol=tcp) (request-type=1) (option-string-a= 141.211.133.121) (exec-name=traceroute) (client_name=aglo)`"

Some of the parameters are required inputs to the GARA infrastructure such as `endpoint` fields that are used to contact appropriate PMP nodes, a `start-time` field used to determine when to perform the requested operation, and a `duration` field used to stop the request (eg., stop a started `iperf` server process). Other parameters, specific to the requested remote operation (eg., `traceroute`), are not interpreted by the GARA infrastructure but might be used by the KeyNote security policy engine before accepting a reservation.

The `reservation_cancel`() function allows for cancelling a previously scheduled reservation. It also serves a cleanup purpose, letting the GARA service know that the user is no longer interested in this reservation and thus allowing the GARA service to remove all the stored information about the request.

The `reservation_callback`() function provides the functionality to register a callback for a previously placed reservation and listen for the GARA reservation events `operation_started` and `operation_finished`. This functionality is used in case the user is interested in the output of his or her previously-created reservation. After

placing the `reservation_callback`() request, the caller can wait until the `operation_finished` event is received before proceeding with its execution. To prevent the GARA client module from waiting indefinitely in case of some sort of failure, there is a timeout mechanism in place that will wake up the waiting thread after a (currently hardcoded) maximum allowed period of time and will inform the user that the request has failed.

The `reservation_status`() function allows for checking on a status of the reservation. If the operation is finished, then the result data, stored at each of the PMP nodes involved in the request, is sent back to the GARA client module and displayed on the web page.

Different combinations of these four operations provide the functionality for performing different types of requests: one node synchronous (or asynchronous) requests (eg., `traceroute` or `ping`), where the GARA client module places a reservation to one PMP node; two node synchronous (or asynchronous) requests (eg., `iperf` or any other client/server tools); or more complex requests such as one-node request to run `traceroute`, followed by several two-node requests to `iperf` based on the discovered path.

The GARA service part has four components: main server, KeyNote security policy engine, scheduler, and application executioner.

The main server listens and accepts incoming request of the types listed above.

Before any type of the request is handled, a KeyNote security policy engine is consulted. The current implementation authorizes requests based on the user's group membership. Group membership can be retrieved either from an AFS PTS server or from a file stored in a local filesystem. A security policy engine can support different policies for different executables (eg., separate policies for `traceroute` and `iperf`). It's implemented by adding an appropriate suffix to a default policy prefix (eg., policy.traceroute and policy.iperf).

The scheduler is responsible for checking, scheduling and stopping reservations based on their requested start times and durations. Once a reservation is ready, a scheduler looks up the corresponding RSL for the request, creates a unique filename where the output of the execution is redirected, and forks the application executioner.

The application executioner is responsible for doing any application specific setup and the actual starting (or stopping). The current implementation does not do any application specific setup but only looks up the correct path to the executable (eg., traceroute) for the PMP machine (eg., /usr/sbin/traceroute), since the executable is not necessarily in the default path and it is not assumed that the placement of the executable is known to the user who is placing the request through a web page. After an application executioner starts an application, it record the process's pid. When the application executioner is called again to stop the application, it looks up the appropriate pid and stops its execution.

Appendix B

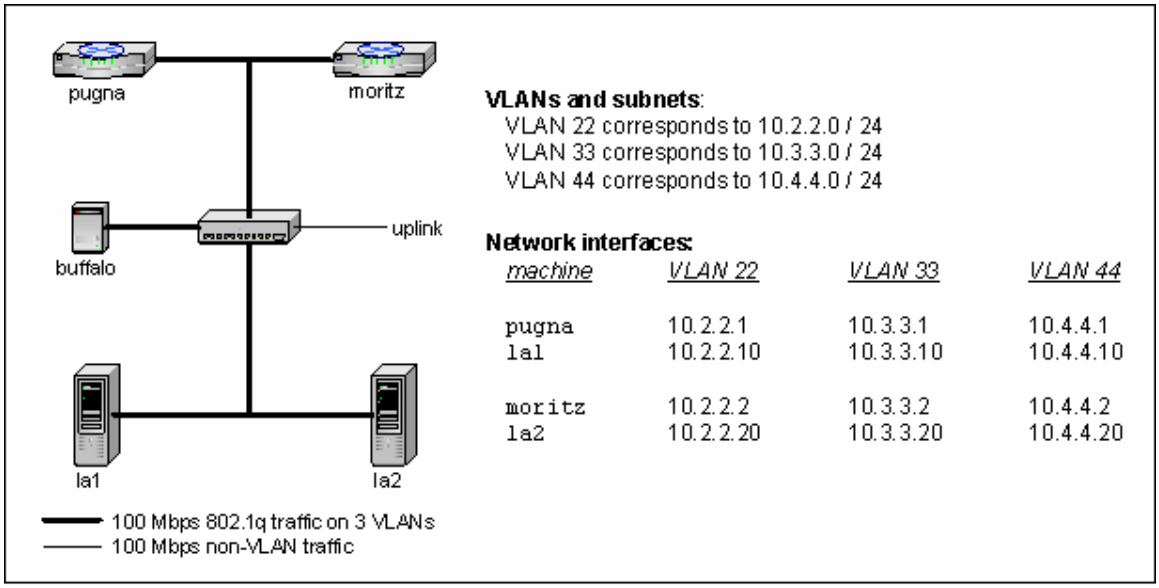

Figure 1. NTAP testbed network at CITI. Two routers (`pugna` and `moritz`) are each associated with a PMP (`la1` and `la2`, respectively). `buffalo` snoops all VLAN traffic through a central hub.

Our demonstration network (see **Figure 1**) is a testbed consisting of two routers (`pugna` and `moritz`), two PMPs (`la1` and `la2`), and a machine (`buffalo`) that monitors the traffic between the other four. `la1` is associated with `pugna`, just as `la2` is associated with `moritz`. `pugna` and `moritz` serve as an example hop along a path whose performance is to be measured.

By running the `iperf` server on `la1`, for instance, and binding it to `la1`'s VLAN 33 interface, 10.3.3.10, and then running the `iperf` client on `la2`'s VLAN 22 interface, 10.2.2.20, we can watch the inter-VLAN traffic and verify that our policy routing is functioning as expected. A packet traveling from 10.2.2.20 to 10.3.3.10 would be sent out `la2`'s default route on VLAN 22, which is `moritz`' interface at 10.2.2.2. `moritz` routes the traffic over to VLAN 33 and sends it along to `pugna` at 10.3.3.1, who then forwards it to `la1` at 10.3.3.10. Note that `la2` first selects a routing table based on the outgoing source address and thereafter makes a routing decision based on the destination address; this is why, despite having a presence on VLAN 33, `la2` nevertheless sends the packet out of its VLAN 22 interface.

Similarly, when sending a reply, `la1` selects a routing table based on the outgoing source address. Each source-based routing table is configured with the appropriate default rule, effectively emulating a default route per virtual interface, forcing packets to be reflected out of the same interface on which the incoming packet should have arrived. Policy routing thus implements the desired “packet reflection” for the multihomed `iperf` tests.

la1's routing tables are shown in the following partial output from our lsrt tool:

```
[cja@la1 ~]$ lsrt
1000: from 10.2.2.0/24 lookup vlan22
  10.2.2.0/24 dev vlan22  scope link
  default via 10.2.2.1 dev vlan22
1001: from 10.3.3.0/24 lookup vlan33
  10.3.3.0/24 dev vlan33  scope link
  default via 10.3.3.1 dev vlan33
1002: from 10.4.4.0/24 lookup vlan44
  10.4.4.0/24 dev vlan44  scope link
  default via 10.4.4.1 dev vlan44
```

The first output line indicates that packets with source addresses in the 10.2.2 subnet are to use the vlan22 routing table. This table consists of the next two lines and performs conventional destination-based routing: packets whose destinations are in the 10.2.2 subnet are routed locally back to vlan22; all other packets are forwarded to 10.2.2.1, which is one of pugna's addresses, using vlan22. This routing table thus ensures incoming vlan22 traffic goes back out on vlan22. The rest of the tables function similarly.

Despite the fact that the PMPs are interconnected through a hub, the process of having them communicate through their respective routers simulates the behavior of a PMP in the field. The hub merely allows buffalo to examine inter-VLAN traffic with tcpdump. Note that the inclusion of a hub precludes the demonstration of intra-VLAN performance testing, since the traffic would bypass the routers entirely. Nevertheless, intra-VLAN traffic is, in practice, a simplification of inter-VLAN traffic. Replacing the hub with a switch would get closer to a real-life scenario, but snooping would not be as effective.

ntap-config

We have designed a tool, ntap-config, which works on the PMPs and aids in the creation and configuration of 802.1q virtual interfaces and their corresponding policy routing data in the iproute2 Linux package. Given properly installed NICs, an 802.1q-savvy kernel, and access to the vconfig VLAN configuration utility, ntap-config will read a configuration file and bring up the virtual interfaces specified therein. The configuration file, usually /etc/ntap.conf, can be modified and ntap-config immediately re-run to reconfigure the devices on-the-fly.

ntap-config has the following options:

```
-q          be quiet; produce no output on success
-v          be verbose; print routing tables and rules after setup
-c <file>   use <file> as the conf file, instead of `/etc/ntap.conf'
```

```
-u          bring up all devices in the conf file and configure them
-d          bring down all devices in the conf file
-f          flush the routing caches of all devices in the conf file
```

The configuration file (e.g. `/etc/ntap.conf`) contains some subset of the following options (dev, vid, and ip are required; mask and brd will become class A/B/C defaults, based on ip):

```
dev:        host-device       # the host interface on which to add the
                              #   virtual interface; e.g., eth1
vid:        <number>          # the vid of the VLAN; e.g., 22
ip:         dotted-quad       # the dotted-decimal IPv4 address
                              #   associated with the interface
mask:       dotted-quad       # the dotted-decimal representation of
                              #   the netmask
brd:        dotted-quad       # the dotted-decimal representation of
                              #   the network broadcast address
mac:        MAC addr          # a full MAC address for the interface,
                              #   or a * followed by a suffix
ro_table:   table-name        # the name of the iproute2 table used in
                              #   rule and routing entries
ro_entries: entry [; entry]*  # an iproute2-style routing table entry,
                              #   sans table and device names
ru_entries: entry [; entry]*  # an iproute2-style policy routing rule
                              #   entry or entries, sans table name
```

For example, configuring a VLAN 33 virtual interface on host interface eth1, with IP 141.211.133.43, netmask 255.255.255.0, broadcast address 141.211.133.255, a rule to send all traffic destined for 141.211.92.0/24 to lookup routes in the routing table "net-92", and then to send everything to the default route 141.211.133.1, would look like this:

```
dev:        eth1
vid:        33
ip:         141.211.133.43
mask:       255.255.255.0
brd:        141.211.133.255
ro_table:   "net-92"
ro_entries: "default via 141.211.133.1"
ru_entries: "to 141.211.92.0/24 table net-92"
```

Optionally, perhaps for testing purposes it would be convenient to replace the last octet of the host interface's MAC address with "33". If the original MAC were 11:11:11:22:22:22, adding either of the following two lines:

```
mac:        *:33
mac:        *33
```

```
would achieve the desired result, a MAC address of 11:11:11:22:22:33.
A full MAC address need not be prefixed with an asterisk; currently
only prefixes are matched with the asterisk.
```

Appendix C

Our initial performance tests have concentrated on our loaned PMPs that we have named `la1` and `la2`. Both hosts contain four 2.2 Ghz Intel Xeon processors with 2 GB RAM and dual Tigon3 PCIX 133MHz 64-bit 10/100/1000BaseT Ethernet cards and run 2.4.18-3smp Red Hat 7.3 Linux. We have created a separate VLAN over a directly connected Ethernet cable between the two hosts for these tests; this removes the interference caused by our routers and our 100 Mbps hub.

Our first test measures performance over the loopback interface on `la1`. One Iperf server is started; successive instances of the client are then started in parallel. The results show that all processors are fully occupied with four Iperf clients and yield an aggregate TCP throughput of approximately 2.5 Gbps. We use an MSS of 1456, 4 shy of the usual value, to work around a bug in the Ethernet drivers for our router machines, in which the additional 802.1q fields cause maximally-sized Ethernet frames to appear to be too large. This gives an approximate lower bound of the maximum throughput that can be expected from a PMP.

With an Iperf server running on `la2`, and successive instances of the client started on `la1`, we measure a maximum aggregate throughput of 940 Mbps for two to eight Iperf clients, the highest number tested. Aggregate processor idle time is about 80% on both `la1` and `la2`, even with eight clients.