|  |
| --- |
| Final ReportonOpportunisticMobile WirelessNetworks(O-MWNet) |
|  Professor Jiang Li |
|  Howard University, USA |

Table of Content

1. Introduction and Organization
2. An Overview of Pigeon Networks
3. Building an Emulation Framework for Pigeon Networks using Virtual Machines
4. Tweaking the Emulation Framework for efficient Code Migration to ORBIT Lab
5. Using the ORBIT Lab Nodes to Emulate a small Pigeon Network
6. Using the OMF Interface and the OML Library of NORBIT Lab to Emulate a small Pigeon Network
7. Using the OMF interface and the OML library of ORBIT lab to Emulate a small Pigeon Network
8. Emulating a large Pigeon Network on the ORBIT lab
9. On running an Experiment with multiple instances on the ORBIT lab
10. Our Comments/Feedback on ORBIT Lab
11. Our Comments/Feedback on NORBIT Lab
12. Summary and More Comments
13. Introduction and Organization

Our research project in Howard University deals with a special type of opportunistic mobile wireless networks known as *Pigeon Networks*. Our main goal is to experiment with this type of opportunistic mobile wireless networks (O-MWNet) on ORBIT using its GENI interface. We aim to measure the performance metrics (e.g. delay, throughput, and loss ratio) of a pigeon network in the experiments. In addition, we investigate the ease of using the GENI interface of ORBIT to do these experiments to uncover its strengths and limitations.

During the project we had to traverse through the several milestones. Note that due to lack of sufficient information on the ORBIT wiki, we did some of our intermediate experiments on NORBIT lab in past. We aimed to gain related expertise from NORBIT so that we could replicate our experiments thereafter. In most part, we were successful in fulfilling this goal.

Below we list the major milestones with the date of completion, and for each such milestone, we state the main elements therein. Later in this document, we discuss in detail our achievements, limitations, and problems encountered in meeting each milestone.

1. Built an emulation framework for Pigeon Networks using virtual machines (completed by 10-11-2010): Analyzed the role of different nodes in a pigeon network and designed the modules of each type of nodes. Emulated these nodes using multiple virtual machines.
2. Tweaked the emulation framework so that most of the code can be reused on the ORBIT lab (completed by 10-31-2010): Modified the design of the modules in a node so that we can easily migrate most of the code to ORBIT lab.
3. Used the ORBIT lab nodes to emulate a small pigeon network (completed by 12-15-2010): The nodes communicate with each other via wireless connection. A patched version of the Madwifi driver was used to control the connectivity among nodes.
4. Used the omf interface and the oml library of NORBIT lab to emulate a small pigeon network (completed by 1-31-2011): An ED (experiment description) script was executed via omf (version 5.3) to run the whole experiment. The nodes communicate with each other via an Ethernet. A packet filtering technique such as iptables was used to control the connectivity among nodes. The performance metrics were collected in the database server using the oml2 library, which are analyzed using a query language, sqlite3.
5. Used the omf interface and the oml library of ORBIT lab to emulate a small pigeon network (completed by 3-23-2011): An ED (experiment description) script was executed via omf (version 5.2) to run the experiment. The nodes communicate with each other via wireless or Ethernet connection. A packet filtering technique such as iptables was used to control the connectivity among nodes. The performance metrics were collected in the database server using the oml2 library.
6. Extended the previous code (e.g. programs running on ORBIT nodes) as well as the ED (experiment description script) to emulate a large pigeon network (completed by 2-28-2012): The large network consists of a home host node, a pigeon node, and multiple (in order of tens) foreign hosts. The ED script was extended in multiple ways, e.g. by making it flexible so that it can emulate varying number of foreign hosts as supplied by the user. At the beginning of the experiment the ED also loads the driver module in each node, configures the NIC as required. The ED is executed via omf (version 5.3), and the nodes communicate via wireless connection. In contrast to our earlier work, in the current ED we add one application called topology manager which runs in the pigeon node in addition to the regular pigeon program. The topology manager uses a packet filtering technique such as iptables to control the connectivity of the pigeon. The performance results were collected in the database server using the oml2 library. Note that in our previous experiments we were emulating only one foreign host while now we are able to run an experiment emulating tens of foreign hosts. As an example, we successfully tested our code with up to 31 foreign hosts.
7. (completed by 3-31-2012)We explored how to extend the ED to emulate an experiment with multiple instances. The motivation is that in real life the experimenter often needs to run multiple iterations of an experiment with some varying parameters to collect statistically significant performance results. We tried to design the ED and the other code including input data files in such a way that we move towards automating the whole experiment. Unfortunately, we found that with the current infrastructure available to the experimenter, it is not possible to fully automate an experiment with multiple runs with varying parameters. We suggested a couple of new features to be introduced on the ORBIT lab, which could help us achieve full automation in future.
8. (completed by 6-30-2012) We built a visualization module which shows the topology (location of the nodes) of the pigeon network on a browser window. It displays the real-time movement of the mobile nodes (e.g. the pigeon) in addition to the static nodes of the network as emulated on the ORBIT grid. We use d3.js module on the browser side which is a Javascript library for the visualization task.

Along with this document we also provide the **source code** of our implementation. For each of the above milestones we provide the corresponding version of code, which could be found in the corresponding directories. Find attached the five sub-directories of the source code under the directory, ‘code’. Inside each sub-directory we include a README file which specifically describes how to use the associated code.

We are aware of running late in terms of timely completion of this project as promised in our earlier progress report. The main reasons which contributed to this delay are as follows: (a) Lack of detailed/consistent information on ORBIT wiki, (b) unavailability of ED features (to the best of our knowledge) specialized for running a long duration experiment with multiple instances, (c) our experiment involving tens of nodes while nodes on the ORBIT grid are not very reliable, e.g. some node on the ORBIT grid may become unresponsive in the middle of the experiment.

We are grateful to the administrators of ORBIT and NORBIT lab for their kind help, detailed instructions, and suggestions through numerous emails. However, acquiring specific information about the test bed via emails is a slow process, as expected. It could have been way faster had ORBIT wiki delivered updated and consistent information. It would be even better had the wiki possessed a tool to search for specific information via key words. As part of this project, we investigated the ease of using ORBIT and NORBIT lab. This report includes two sections where we provide our feedback and comments about these two test beds.

The milestones which remain to be achieved by us are (a) Add visualization components, and (d) Develop a demo and the final report.

Before we report on our achievements till date, we present a brief introduction to Pigeon Networks.

1. An Overview of Pigeon Networks

Figure : A tiny pigeon network - the Pigeon traverses between the Home Host and the Foreign Host.

A pigeon network is a special type of Disruption Tolerant Network (DTN). The unique feature of this special type is that each message carrier, also called a *pigeon*, is owned by a host node. There can be multiple host nodes present in the network, which are interested in communicating with each other. The owner host node of a pigeon is called the *home host* of the pigeon, and other host nodes are known as the *foreign hosts* with respect to that pigeon. A pigeon has dedication to its owner in terms of selecting whose commands to follow and which messages to carry. The fundamental unit of a pigeon network is a home host coupled with its pigeon, which is known as a *cell*. The whole network can be considered as the interaction among multiple cells.

In the emulation work, we focus on one *cell* (involving one home host and one pigeon)and its interaction with one or more foreign hosts. For the ease of presentation, we first discuss a tiny pigeon network with single foreign host, and then we extend it to a larger pigeon network involving multiple foreign hosts. In a tiny pigeon network there are three nodes, namely (a) the pigeon P, (b) the home host H, and (c) the foreign host F, as illustrated in Figure 1. These nodes have the following characteristics. The pigeon node P is mobile and moves back and forth between the home host H and foreign host F whereas H and F are static. Either of H and F generates messages in course of time, whose generation time is modeled by a statistical distribution e.g. the uniform distribution or the Poisson distribution. Each message needs to be delivered to the other node than the node where it was generated, i.e. H’s messages need to reach F and vice versa. The pigeon node P acts as the message carrier.



Figure : How the messages traverse through the 3 nodes which are the Home host (H), the Pigeon (P), and the Foreign Host (F) --- each node has two queues.

Each node possesses a wireless communication system and can connect to another node within its communication range while we assume that all nodes’ communication radii are same, say c. However, the distance *dHF* between H and F is way longer than their communication range (i.e. *dHF > c*) and they cannot communicate directly to each other. After messages are generated in a node (H or F), they are stored in a queue, called the *message generation queue*, at the generating node before being picked up by the pigeon node. Pigeon P, being a mobile node, carries H’s messages to F, and P also carries F’s messages to H.

We model the message transfer within the whole network as illustrated in Figure 2. We assume that either of H and F has two queues: node N = H or F has two queues, a message generation queue, QNg and a message sink QNs. We also consider that P has two queues: QPH stores messages destined to H and QPF is there to store messages destined to F. When node P visits node H, P picks up H’s messages from QHg and stores them in QPF and next time when P visits F, P delivers messages in QPF to F’s sink, QFs. Similar message transfer occurs in the other direction, i.e. P picks up F’s messages from QFg when P visits F and stores them in QPH and delivers them to H’s QHs when P is within the communication range of H.

Figure : A pigeon network with multiple foreign hosts Fi – the pigeon P visits the seven foreign hosts Fi following a scheduling algorithm. P can visit the home host H with higher frequency, i.e. there can be multiple loops in P’s trajectory. In this example, P’s trajectory consists of two loops: (a) H-F7-F5-F3-H, and (b) H-F1-F4-F2-F6-H.

We assume that all messages are of the same size. We also assume that the length of each queue in the pigeon is configurable, i.e., the queue lengths are calculated depending on the user’s input before start-up. The time to leave the currently visiting node (H or F) for P is decided by a scheduling algorithm, *S*. P has a mobility model, *M* which determines its instantaneous location at time *t* given the speed. Note that locations H and F are fixed and they are determined before the emulation starts. Another deciding factor of the simulation program is the message generation algorithm, *G* which determines the message generation time. In future, we aim to allow the user to incorporate his/her own version of *S*, *M,* and *G* and for the time being, we provide a simple default version of *S*, *M,* and *G* along with the emulation framework.

We can easily extend the aforementioned tiny network to a large pigeon network involving multiple foreign hosts. Figure 3 illustrates an example of such a network where the pigeon P starts from the home host H and visits the foreign hosts (Fi) in some sequence while it delivers/receives messages. The pigeon can return to the home host H before visiting all of the foreign hosts, and P can start again from H to make more visits to foreign hosts. The whole trajectory of the pigeon P is determined by the scheduling algorithm. We focus on the scenario where H has messages to be delivered to any foreign host Fi, and Fi can also have messages to be delivered to H. Of course, P is the message carrier for all of these messages. We extend the design of message queues as illustrated in Figure 2 for the scenario with n foreign hosts as follows: Instead of only one queue in P to store messages for H we introduce n queues (one queue corresponds to messages coming from one particular Fi), and instead of only one queue in P to store messages for foreign host we introduce n queues (one queue corresponds to messages destined to one particular Fi). Similarly, H now uses 2n queues in total instead of only 2 queues. However, the design of a foreign host Fi remains same as shown in Figure 2.

1. Building an emulation framework for Pigeon Networks using virtual machines

Before experimenting with ORBIT lab, we build a framework to emulate a tiny pigeon network as the first step. We use C programming language and the socket API to build this framework.

We build the simulation framework as modular as possible so that the user can replace any module later with his/her own version without worrying about tweaking other modules. We envision each node N to possess several modules as described below. Depending on the specific characteristic of a particular node, some modules are present or absent. As an example, when N is the Home Host (H) or the Foreign Host (F), its location is static, so it does not need any mobility feature.

The following modules are present in either of H and F (i.e. N = H or F).

1. Message Generation Module (MGM): It uses N’s message generation algorithm *GN* to generate messages and store the generated messages in the message generation queue QNg.
2. Message Transfer Module (MTM): This module in N interacts with the message pickup module (MPM) of the Pigeon (P). Message transfer is initiated only when P initiates the pickup process. After P picks up messages from QNg, MTM does the bookkeeping job for QNg.
3. Message Reception Module (MRM): This module of N interacts with the message delivery module (MDM) of P. This module receives the messages delivered by the pigeon and stores them in the sink QNs. Message reception is initiated only when P initiates the delivery process.
4. Message Consumption Module (MCM): We assume that MCM consumes messages stored in the sink QNs. This module can compute the protocol performance statistics such as average message delay or message loss ratio.
5. Connectivity Response Module (CRM): This module helps the pigeon discover the host node N. CRM receives the ping messages from the pigeon and sends back a ping reply message.

Figure 4: VM4 (Controller R) has 3 network interfaces r1, r2 and r3. VM4 is set up as a real router. The mobility module M of P updates its current location to R and accordingly R enables or disables interfaces r2 and r3. Note that R is transparent to all other modules of P. Also, R is transparent to all modules in H or F.

We assume that P tries to connect to N once P reaches N, and if the connection is established, the messages are picked up (or delivered) from (or to) N by P. The pickup (or delivery) start time of P is decided by its scheduling module, *S .* The following modules are present in P.

1. Scheduling Module (*S*): Depending on the current scenario (e.g. number of the messages waiting at H or F), this module determines when P should make trips to other nodes. *S* makes one choice out of three options: (a) P should remain at the current location, (b) P should move towards H, (c) P should move towards F. Currently, we are using the Gated Scheduling which is a popular scheduling algorithm.
2. Node Mobility Module (*M*): It updates the current location of P and manages the mobility feature. Note that the Scheduling Module, *S* gives input to *M*. We assume that P does not move during the pickup and delivery.
3. Connectivity Module (C): This modules checks P’s current connection status with H and F. This module communicates with CRM module of H or F. Basically, C sends a UDP ping message which is received by CRM.
4. Message Pick-up Module (MPM): This module picks up the messages from the message generation queue of N (N = H or F) and stores them in the corresponding local queue, QPN. Message pick up is possible only if connectivity module, C decides positively. The time to visit a node by P is decided by the scheduling algorithm, *S*
5. Message Delivery Module (MDM): This module delivers the messages from P’s local queue, QPN (N = H or F) to node N. We assume that as soon as the connection is established the message is delivered.

Figure 5: The functional view of the system --- each dotted box represents a process which realizes a node. Each process runs one or multiple threads. Each thread realizes one or more modules. Green boxes represent the message queues. Thread 2 in node N runs a TCP server, Thread 3 runs a TCP server whereas Thread 3 runs an UDP server. On the other hand, Thread 2 in P has a TCP client and an UDP client.

We emulate the connectivity of the nodes with the help of a global **router**, R which acts as an **arbitrator** as illustrated in Figure 4. We assume that R periodically receives the information about the current location of P and so R can decide P’s connectivity with any host N at any point of time. In case R determines that P and N should be connected or disconnected, R enables or disables the connection between them by making its relevant network interface up or down.

Note that by definition some of the modules act concurrently, e.g. in a host N (N= H or F), the message generation module (MGM) and the message transfer module (MTM) or the message reception module (MRM) are simultaneously active. To manage this concurrency while ensuring the modularity in the system design, we envision each node (which is realized by one unique process) to execute multiple threads. Each thread realizes one or more modules. We already discussed the modules which are present in host N or Pigeon P. The threads present in the Pigeon P and a host N (N =H or F) are illustrated in Figure 5. Host N has a TCP server running on Thread 2, a TCP server running on Thread 3, and an UDP server running on Thread 4. On the other hand, P has a TCP client and a UDP client running on its 2nd thread. This TCP connection is used for message transfer and UDP pings are used for node discovery. In Figure 4, we do not show the router, R which is transparent to other nodes.

Finally, we stress that the emulator framework is built by compiling four .c files: arbitrator.c (which represents the router), home.c, pigeon.c, and foreign.c. Each executable realizes the representative process of the corresponding node on one virtual machine (VM). We realize each of these 4 nodes as a VM (Ubuntu 10.4). We use VMware Workstation 7.1 on Windows as the virtualization technology.

1. Tweaking the emulation framework for reuse on the ORBIT lab

Thanks to the fact that (a) each node in our emulation framework being a separate VM, and (b) each node in ORBIT is a physical machine, the code migration is expected to be straightforward. We consider the thread-based architecture of the modules of P, H and F to remain same in the most part. Note that we could not realize the router node R in ORBIT because all nodes in ORBIT setup can listen to one another. Yet, the router in our emulation framework being transparent, we are able to directly migrate to ORBIT most of the code of H and F.

* 1. **Extending the design of the Pigeon:**

In the absence of the arbitrator router, we need to modify the design of the Pigeon (P) so that we can realize P’s dynamic connectivity with host nodes, which varies with time due to P’s mobility. We extend the mobility module M of P, and incorporate a connection control mechanism inside M. The main idea is that at any point of time the mobility module can measure the distance of P from any other host node (H or F), and thus M can decide P’s connectivity with other hosts. The assumption is the host nodes are static, so P’s current location is enough to decide the connectivity.

For controlling the dynamic connectivity of P, we add (to M) a packet filtering mechanism which can be realized via well known tools such as iptables, ebtables. These tools are readily available to be installed on any standard Linux OS. An example of invoking such a packet filter in our emulator setup is as follows, where 10.0.2.2 is the source of the host node whose packets the pigeon wants to drop:

**sudo iptables -A INPUT -s 10.0.2.2 -j DROP**

In our emulator setup, the communication between the pigeon and home/foreign host is via a router. So, only IP layer filtering will work, but not the MAC-layer filtering. In the ORBIT lab, any other MAC layer filtering mechanism such as orbitfilter can also be used which is available for the wireless communication setup.

In this modified design, the pigeon does not need to report its location to anybody else such as the arbitrator. So, from the mobility module, M we take out some functionality which was responsible to report the current location to the arbitrator.

 **4.2. Considering mobile host nodes**:

We need to further extend the design if we want to generalize the feature of a host node and consider the possibility of a mobile host. In the modified design, the mobility of a host node needs to be emulated. We have one proposal for this extension, where each mobile host intermittently sends its current location information to each pigeon. We also implemented and successfully tested this idea in a small scale network. However, some points like scalability remains to be addressed. In the rest of this document, we do not further consider this extension and stick to the design presented in Section 2.1, which assumes that the host nodes are static.

1. Using the ORBIT lab nodes to emulate a small pigeon network

One of our tasks is to find out how to realize the mobile node i.e. the pigeon. We saw in some documents on ORBIT wiki (http://www.orbit-lab.org/wiki/HowTo/virtualMobilitySS) that ORBIT has a relevant tool which is known as “Spatial Switching”. However, we understand that this tool can only support discrete mobility. We want to explore how to emulate the continuous mobility of the pigeon or at least we want to control the granularity of the discrete mobility.

The orbit administrators admit that it is extremely hard to emulate continuous mobility in the true sense. There are two broad approaches for emulating mobility---(a)Noise injection: Details of which can be found here - http://orbit-lab.org/wiki/Documentation/ArbInterference, (b) Random packet loss injection - this can be done with the use of tools like ebtables, iptables, or other packet filtering schemes. We explored the second path.

All nodes of the orbit lab setup can hear one another in terms of both Ethernet and wireless connection. So, we cannot directly borrow the concept of the arbitrator from our basic emulation framework as described in Section 1. Instead, we use a modified design of the pigeon, as presented in Section 2.1, to emulate the dynamic connectivity of the pigeon.

The nodes communicate with each other via wireless connection. The mobility module M of the pigeon P breaks and makes the connection between P and the host nodes, depending on the current location of the pigeon. A patched version of the Madwifi driver (known as orbitfilter) was used to control the connectivity.

The patch orbitfilter serves as a packet filter. When the pigeon decides to break the connection with a host its mobility module invokes the following command:

 **iwpriv ath0 orbitfilter MAC-address-of-a-host 0 0**

On the other hand, when the pigeon decides to re-establish the connection to the host, it invokes the following command:

**iwpriv ath0 orbitfilter MAC-address-of-a-host 100 100**

Apart from the on/off status of the connection, orbitfilter can also emulate different loss rates.

We note that we could use other packet filters such as iptables and ebtables as an alternative to orbitfilter, however iptables or ebtables can emulate only on or off status of the connection. The example usage of iptables and ebtables are as follows.

**iptables -A INPUT -m mac --mac-source 00:0f:ea:84:c8:01 -j DROP**

The above command makes packets from the mac-source to be dropped, i.e. the connection is off.

**iptables -D INPUT -m mac --mac-source 00:0f:ea:84:c8:01 -j DROP**

The above command deletes the particular existing IPTABLES rule. i.e. the connection is on.

**ebtables -A INPUT -s 00:0f:ea:84:c8:01 -j DROP**

**Steps to follow to run the experiment:**

(a)After logging to the console machine, invoke the following command.

**omf-5.2 load all image.ndz**

It will write the disk image with the name 'image.ndz' on all the nodes on the 'grid' or sandbox. As we use the orbitfilter patch of Madwifi, we need to load such an image (from the repository) which has this patch installed.

(b)We need to power-on the nodes via “**omf-5.2 tell on**”. Then, after waiting for some time, we log into the nodes (as root) on which we want to run our application i.e. pigeon, home and foreign.

(c) Copy one source file (home.c, pigeon.c or foreign.c) along with the error handling component 'DieWithError.c' to the corresponding node. Log in as the root to each of the three nodes and compile the source code to generate the binary.

(d) We need to configure the wireless interfaces of the nodes. Here are some examples of useful commands.

**node1-1:~# iwconfig ath0 mode ad-hoc essid test**

It sets the essid and mode as ad-hoc of interface ath0.

**node1-1:~# ifconfig ath0 192.168.1.1**

It sets the IP address of interface ath0.

**node1-1:~# ifconfig –a**

It lists the all interfaces and we can check that the wireless interface is properly configured.

(e) Run the executable in each of the three machines. The log files are collected which are analyzed later to evaluate the performance.

1. Using the omf interface and the oml library of NORBIT lab to emulate a small pigeon network

ORBIT measurement framework and library (oml) is a distributed framework which enables real-time collection of data while the experiment runs. Application programmers can use oml APIs to enable the application to inject measurements data to a central repository (oml server). More information about installing and using oml is available in the following links on the wiki.

1. http://mytestbed.net/projects/omf/wiki/UsingOML
2. http://mytestbed.net/projects/oml/wiki/Client\_Programming
3. http://oml.mytestbed.net/projects/oml/wiki/Installing\_OML\_packages

For the last couple of years, NICTA and Winlab have been co-developing the **OMF** **Framework** for networking test beds. The NICTA test bed is also known as the NORBIT lab whereas Winlab test bed is referred to as ORBIT.

**Preparatory Steps to install oml2 client**

1. log on to the console machine via “ssh Username@norbit.npc.nicta.com.au”
2. Install a baseline disk image on the node, which we have reserved. Here we assume that this node is: **omf.nicta.node2**.

omf-5.3 load -t omf.nicta.node2 -i baseline.ndz

1. Log on to node2 where we want to install oml2 library: ssh root@node2
2. Before we can install the OML Ubuntu packages, we must tell Ubuntu (or Debian) where to get them. To do this, we append the following line to /etc/apt/sources.list file via “deb [http://pkg.mytestbed.net/ubuntu lucid/](http://pkg.mytestbed.net/ubuntu%20lucid/)”
3. Install oml2 dev library as follows.

 sudo apt-get update

sudo apt-get install liboml2-dev

1. Save the node image.

omf-5.3 save -n omf.nicta.node2

INFO omf.nicta.node2: - Saving image of '/dev/sda' on node 'omf.nicta.node2'. INFO omf.nicta.node2: to the file **'sroy-node-omf.nicta.node2-2010-12-16-16-58-22.ndz**' on host '10.0.0.200'

So, whenever oml2 client is required in future, we can get it by loading the saved image **'sroy-node-omf.nicta.node2-2010-12-16-16-58-22.ndz’** on other nodes .

**How to call the oml API from the application**

We note that to add the oml functionality to our applications (pigeon.c, home.c , and foreign.c) we need to embed a few API calls into the source of the applications. The instructions are available on the wiki at http://mytestbed.net/projects/oml/wiki/Client\_Programming . The main elements are defining one or more measurement points (MP), and calling omlc\_init(), omlc\_start(), and omlc\_inject() from the right places in the source. Refer to the source code provided in the associated folder.

When an event happens (like message generation, pickup or delivery) we inject time stamp from the corresponding place in the code. We inject the time stamps both from the previous line and next line of the code to check later if there is any difference in time. We also inject the location information of the injecting node. The pigeon periodically inject its location information even if there is not event regarding message pickup or delivery. This will enable us to verify the pigeon path after the experiment.

**Writing an ED (experiment description)**

We wrote an ED (experiment description) script which represents the whole experiment. We provide the ruby file (ED\_complete\_experiment) in the associated folder. Once the ED is written, the experimenter can run it at the console machine, and nothing else needs to done from the experimenter’s end. The script was executed via omf (version 5.3) to run the experiment. During the experiment each node injects relevant data into the oml server.

We also provide other two ED files, present in the associated folder, which are useful to test out the proper working of oml while running one or two nodes.

**Running an ED (experiment description)**

We run the ED script using the latest version of omf which is 5.3. The nodes communicate with each other via an Ethernet. A packet filtering technique such as iptables was used to control the connectivity among nodes. We set the message size as big as 1MB so that we can observe the latency incurred in the process of message receipt and delivery.

**How to retrieve the database dump from the oml server**

Currently the collected measurements are stored in a SQLite database, which is located on the server that runs the oml Measurement Collection Server. For each new experiment execution, a measurement database is created with the same name as the Experiment ID. The performance metrics were collected in the database server using the oml2 library. The wget utility is used to retrieve the data if we know the experiment ID. As an example,

**$ wget "http://localhost:5053/result/dumpDatabase?expID=testing\_slice-2011-01-06t12.50.14%2B11.00" -O myDatabase**

Then, the database dump “myDatabase” are analyzed using a query language, sqlite3 which is available in the NORBIT console. A list of queries which we used to measure the performance metrics are available in a file called “DB\_queries” in the associated folder.

1. Using the omf interface and the oml library of ORBIT lab to emulate a small pigeon network

We faced some confusion regarding the current version of oml in ORBIT. Some of the documents available on the wiki (or the mailing list) discuss the older version of the oml. Here are some examples.

<http://www.orbit-lab.org/wiki/Documentation/OML/FAQ>

<http://www.orbit-lab.org/wiki/Tutorial/CollectingMeasurements>

<http://orbit-lab.org/pipermail/orbit-user/2007-March/001367.html>

Finally, via communicating with the ORBIT administrators and the mailing lists, we came know that we can use oml2 on ORBIT. As in NORBIT lab, we need to install oml2 client (dev version) in ORBIT. We did this installation on node1-2 of sandbox 1, which had the current baseline image. Then, we saved the node image as sroy-node-1-2-2011-02-18-00-15-31.ndz.

We note that to add the oml functionality to our applications (pigeon.c, home.c , and foreign.c) we need to embed a few API calls into the source of the applications. The instructions are available on the wiki at http://mytestbed.net/projects/oml/wiki/Client\_Programming . Refer to the source code provided in the associated folder.

When an event happens (like message generation, pickup or delivery) we inject time stamp from the corresponding place in the code. We inject the time stamps both from the previous line and next line of the code to check later if there is any difference in time. We also inject the location information of the injecting node. The pigeon periodically inject its location information even if there is not event regarding message pickup or delivery. This will enable us to verify the pigeon path after the experiment.

We wrote the ED (experiment description) scripts each of which represents the whole experiment. We provide the ruby file (ED\_grid\_wireless) in the associated folder, which represents the experiment using the wireless communication. In addition, we provide another script (ED\_grid\_ethernet), which represents a similar experiment where the nodes communicate via wired connection. Once the ED is written, the experimenter can run it at the console machine, and nothing else needs to done from the experimenter’s end. The script was executed via omf (version 5.2) to run the experiment. During the experiment each node injects relevant data into the oml server.

The nodes communicate with each other via an Ethernet. A packet filtering technique such as iptables was used to control the connectivity among nodes. We set the message size as big as 1MB so that we can observe the latency incurred in the process of message receipt and delivery.

For each new experiment execution, a measurement database is created with the same name as the Experiment ID. The performance metrics were collected in the database server using the oml2 library. The wget utility is used to retrieve the data if we know the experiment ID. As an example,

**wget -O myDBdump.txt** [**http://oml:5052/result/dumpDatabase?expID=sb1.orbit-lab.org\_2011\_02\_23\_18\_20\_18**](http://oml:5052/result/dumpDatabase?expID=sb1.orbit-lab.org_2011_02_23_18_20_18)

We also provide one simple ED file (ED-test-wireless-connection), present in the associated folder, which tests the proper working of the wireless connections. Further, we include another ED file (ED-test-oml-to-file) which is useful to test out the proper working of oml while oml clients write to a file instead of the oml server.

1. Emulating a large Pigeon Network on the ORBIT lab

We stress that one of our goals is to compare the emulation results with the simulation results which we obtain independently. To do the above, we first simulate the pigeon scheduling algorithm for a large pigeon network on the local machine; in particular, the simulation is written as a C Program which considers a large DTN involving one home host, one pigeon, and multiple (in order of tens) foreign hosts. As an example, the output of the simulation is the average message delay. Then, on Orbit lab we build an equivalent DTN involving same number of real host nodes (e.g. one orbit grid node acts as one real host node), and emulate a mobile pigeon node (on a orbit grid node) which communicates with other nodes via real wireless interface. We finally compare the simulation results (obtained in simulation experiments described above) with the similar results obtained from the implementation on the Orbit lab. This is a challenging task because to make a reliable comparison, we have to replicate multiple parameters as used in the simulation experiments during the emulation experiment on the Orbit lab. Note that there are usually multiple instances (i.e. trials) of the simulation experiment necessary with some varying parameter (e.g. nodes’ emulated locations) to produce the (statistically reliable) average value of the intended metric, e.g. average message delay. For each of the instance of simulation experiment, we have to recreate similar scenarios in the Orbit lab experiment. The challenge sounds bigger when we take into account the possibility of some nodes behaving unexpectedly, e.g. the node being suddenly not responding. We note that the experimenter should be able to do everything from the grid console without manually logging into the grid nodes. In our opinion, with the current Orbit setting, the experimenter has to take the following steps for completing the above task. Currently, as an experimenter we are also doing the same to carry out our experiment.

1. From the grid console, load the image on the nodes (possibly a subset or all nodes of the grid, depending on the particular experiment’s requirement) of the orbit grid. How many extra nodes to start with (to be on the safe side) will depend on the general reliability of orbit lab nodes as observed in past.
2. After the imaging is done, get the list of active nodes from the system file “/tmp/grid.orbit-lab.org\_timeStamp\_topo\_active.rb” and copy the list to a test ED script (e.g. our test0.rb). This script basically attempts to check if the nodes meet the resource criteria of the experiment. As an example, test0.rb attempts to load ath5k module on each node and then configures and sets up a network interface. If on some node this module or corresponding hardware is unavailable, then the setup process will fail i.e. the network interface will not be active in the test. As a result, that node will be of no use to the experiment, which we detect in step 3.
3. (a) Copy a shell script (which detects the unresponsive nodes) e.g. “pingTest.sh” to pigeon node's experiment directory e.g. /BigPigNet .

 **scp pingTest.sh root@node12-16:BigPigNet**

(b) Run the pingTest to detect the unresponsive nodes by pinging the wireless interface of all nodes

 **ssh root@node12-16 "BigPigNet/pingTest.sh"**

(c) From the above output, we can figure out which nodes are active and which are not.

1. Delete the unresponsive nodes from the node list of the test ED (e.g. test0.rb) and then copy the remaining node list to the main experiment ED file, (eg. our test3Num31.rb). If we need less number of nodes in the main experiment, at this point we select the final node list for the experiment. On the other side, if the number of remaining nodes is less than what is required, then we have to redo Step 1 to 4 above, or the experimenter has to give up and return later. We note that above are basically steps to select the final set of nodes called ‘experiment nodes’.
2. Now it is time to prepare the selected nodes with required software/programs for the particular experiment. We may need to copy the program files (which will be compiled to produce executables) and datafiles to the experiment nodes as necessary. The nodes may use these datafiles during the experiment to get the information about the experiment topology. Further, we understand that we may need to run multiple instances of a single experiment as we discussed before. Across these instances, the value of some parameters can be different (e.g. host nodes’ emulated locations) while the value of other parameters can remain same (e.g. the number of home hosts and foreign hosts). So, the experimenter has two options: (a) copy one or more datafiles to the experiment nodes, which contain the information about all the instances of the experiment, or (b) before we start one instance, we copy the corresponding datafile(s) to the experiment nodes. In our case, we took the second option as discussed below.

Of course, we need to copy (e.g. via scp) the program files “Pigeon.c”, “topo.c” to pigeon node, file “Home.c” to home host and file “Foreign.c” to the multiple foreign hosts on the grid console. Then, we compile these programs on the above nodes via ssh on the grid console. Someone can argue that the above copy and compile operation is not needed if these executables are already present as part of the node image. However, we think that copying is necessary just before we start the experiment because we may need to change some parameters (e.g. size of the messages transferred among nodes) in those programs across several experiments. In addition, we scp datafile “coordData” to home host node where the home program only uses the hostCount information present at the first line. Also, before each instance of the experiment (while we run a ***for* loop** through step 5-6), we scp datafiles (file “coordData” has node coordinates info, and file “visitSequence” has pigeon’s node visit sequence info) to the pigeon node. In the pigeon node, the pigeon program uses the coordinates and visit sequence info while the topology manager program (topo.c) uses only the coordinates file.

1. Run the experiment ED (e.g. our test3Num31.rb) from the grid console. We stress that one run in Step 6 corresponds to only one instance of the simulation experiment.

Note that we can perform some of the above steps (5-6) in semi-automated fashion via one or more scripts (refer our prepare.sh). However, sometimes we may observe that a few nodes fail to properly communicate with others during a run, e.g. some node may go down or UDP ping timeout can occur unexpectedly. Probably, there is no way we can rule out the possibility of this kind of problems occurring. The effect of this problem is equivalent to the node being unavailable for the experiment. The other nodes, however, may run and produce results which may or may not be useful depending on the particular need of the experiment. For some experiment, we may have to guarantee that all of the experiment nodes being active. In that case, we may need to go back to Step 1 to grab new active nodes to replace the unresponsive nodes. Later in this document we suggest that introducing some new features in Orbit infrastructure can remove the above burden from the experimenter’s shoulder, and we can make an automated as well as robust run of the whole experiment possible.

1. At this point the experiments are done, and data collection as well as analysis phase begins. During the experiment the nodes can inject data into the database via the OML framework. In particular, while the experiment ED runs (e.g. test3Num31.rb), the nodes can insert data to the oml2 database. After the experiment is over, we run sql query to extract necessary results from the database. Now the remaining challenge is that for each experiment run we may need to modify some part of the database query strings according to the used parameters. As an example, in our experiment home host H enters a logical fID (foreign host ID) to H’s database tables while each foreign host node F enters its own nodeID (e.g. node13-7 of the grid) in F’s table. Note that F does not know its own fID as viewed by the home host. As a solution to the above problem, we can generate a map from fID to nodeID according to the ED (which has the used nodeID list) so that we can properly join multiple tables inside the query. As an example, one query computes the average message delay, whose computation relies on such table join operations. Another solution is at the first encounter of H and F during the experiment H can send the fID to F. We will employ this solution in near future.
2. Compare the Orbit results as obtained through the database queries with the previously collected simulation results.

From the above experience, we come to the conclusion that an experimenter will be greatly benefitted if a few new features are introduced as part of an Experiment Description (ED), which we discuss in the next section (i.e. Section 9).

**Building a Visualization Module**

We remind the reader that ORBIT lab does not provide an off-the-shelf visualization module which we could readily use to display the network topology. We built a visualization module which shows the location of the nodes of the pigeon network on a browser window. It displays the real-time movement of the mobile nodes (e.g. the pigeon) in addition to the static nodes of the network as emulated on the ORBIT grid. We used d3.js module on the browser side which is a Javascript library for the visualization task.

In particular, we built a *proxy server* in the local machine which intermittently retrieves over the Internet the real-time experiment data from the OML server through the ORBIT **result service**. The data is stored in a local file and the browser keeps on receiving the data from the proxy server via asynchronous read operation. Figure 6 illustrates the whole system and shows the data flow path.



Figure 6: The real-time data flow in the visualization system– the experiment data are injected by the ORBIT grid nodes into the OML server, and then the wget program of the Proxy Server downloads the data in a local file. Finally, the Browser retrieves the data asynchronously and displays the emulated network.

1. On running an experiment with multiple instances on the ORBIT lab

From our experience of running an experiment with multiple instances involving a large pigeon network, we come to the conclusion that an experimenter will be greatly benefitted if a few new features are introduced as part of an Experiment Description (ED). We note that ED is basically a ruby script. We believe that these new features as discussed below will move us towards the possibility of running the whole experiment with multiple instances in an automated fashion.

1. Via the ED the experimenter should be allowed to submit a ***generic*** ***resource*** ***request*** to the experiment controller (EC) for a set of nodes (say n nodes) with required specifications on which he/she would like to run an experiment across multiple instances. The EC can internally choose any n nodes from the available pool satisfying the user specifications. Currently, the experimenter defines a ***topology*** by specifying particular node IDs after he/she came to know from other sources (e.g. orbit web portal) that these particular nodes satisfy his/her requirement. As an example, some of the grid nodes do not have ath5k driver which may be needed for a particular experiment. This is a cumbersome process. Furthermore, if any of the specified nodes becomes unresponsive during the experiment, it is the experimenter’s responsibility to detect the problem and in that case redo the whole experiment. It will be very helpful if EC keeps an eye open on the status of the nodes during the runs, and whenever some of the nodes are detected displaying some problem, EC can internally (transparent to the experimenter) replace the faulty nodes with new active nodes and redo the run only from where the problem has occurred. Of course, we take an assumption that change in the physical location of node’s hardware (e.g. wireless antenna) does not make any significant difference in the experiment results. This assumption holds at least in our particular experiment methodology.
2. To the best of our knowledge, currently, an experimenter defines **a group** consisting of a set of nodes (say m nodes) in ED by explicitly specifying the node Ids. It will be beneficial to the experimenter if he/she can define a group with ***generic set of m nodes.*** That means,EC can choose any m nodes from the available pool and made them available to the experimenter. As an example, before our experiment we may need to prepare the nodes (e.g. copying the program from the console to all the nodes of a particular group) differently depending on the group they belong to. Currently, we need to perform this preparation step separately (outside of the ED) using explicit node Ids, which makes it impossible to do this step in an automated fashion. It will be helpful to the experimenter if he/she can define this preparation step within the ED using group Ids, e.g. copying a program from the console to all the nodes in group Gi and compile it on the nodes.
3. A special ***for loop*** construct can be made available as an ED feature, in which each iteration might correspond to one instance of the experiment. Before an iteration starts, the ED script can take actions to **prepare** the nodes accordingly. An iteration should start only if all nodes are healthy, e.g. *onEvent (:ALL\_UP\_AND\_INSTALLED),* which is related with Feature 1 above. There can be another sub-feature introduced in ED via which one or more data files can be copied from the grid console to the experiment nodes before an iteration starts. We may need to copy different data files from the console to the nodes depending on their group Id. The nodes will get the necessary information about the parameters of a particular run by accessing these data files.

**Discussion:** The above features will be particularly useful when the experimenter needs to run an experiment for long time involving multiple instances across which some parameters of the setup vary. One of the main ideas is to relieve the experimenter from the pain of rechecking if some of the nodes have become unresponsive in the middle of the experiment. With the current version of ED features, to the best of our knowledge, the experimenter has to restart the whole experiment from the scratch, which results to annoyance of the experimenter and huge wastage of time and resource. Another idea is to give a handle to the experimenter so that ED can automatically manage multiple iterations with dynamic settings.

One can argue that with the current available features of an ED, an experimenter can use the basic ***for loop*** in the ED (refer test4.rb) to perform multiple runs. We tested this usage and observe that this is not very helpful while we face the following issues. Note that currently we start (and run) each iteration of the experiment (refer test3Num31.rb) manually (as discussed in Section 8), and we do not use any ***for loop***, so we did not face the following issues.

1. After an iteration we sometimes get the following error which says that some node was unable to gracefully *stop* the application (through group.stop-application construct of ED).

ERROR NodeHandler: The resource 'node12-9.grid.orbit-lab.org' reports an unknown error while ERROR NodeHandler: executing a command. Error type is 'FAILED\_EXIT'.

ERROR NodeHandler: The error message is 'Failed to terminate application: 'Foreign#21' - Error: 'undefined method `stdin' for nil:NilClass''.

1. Before an iteration starts we sometimes get the following error which says that some node was unable to start some service (e.g. after invoking group.start-application in ED) because it was unable to stop the application in the last iteration.

ERROR NodeHandler: while running the application 'Foreign#23'; ERROR NodeHandler: The error message is 'bind() failed: Address already in use.

(c) We further observe that when we set large waiting time between two successive iterations, the number of errors as reported above is reduced (e.g. when waiting time = 30 sec) or there occurs no error (e.g. when waiting time = 60 sec). The remaining challenge is how we can know a safe value of the waiting time beforehand.

(d)We note that the data inserted in the oml database correspond to the whole experiment. That means the data related to each run instance is not stored in a separate table. What can we do to address this problem? As an example, in our experiment setup the schema of table PigeonOmlClient\_locPigeon is as follows.

oml\_sender\_id INTEGER, oml\_seq INTEGER, oml\_ts\_client REAL, oml\_ts\_server REAL, "loc1" REAL, "loc2" REAL, "tag" TEXT;

In the output of the query “select \* from PigeonOmlClient\_locPigeon;” we see the following rows among others.

32|563|314.516374588013|314.518243|123.463584899902|151.061218261719|PigeonReportsLocation

32|564|315.056346893311|315.058216|117.995872497559|159.437210083008|PigeonReportsLocation

32|1|401.580090999603|401.582017|209.286468505859|203.718460083008|PigeonReportsLocation

32|2|402.120632998645|402.173602|218.573059082031|207.436965942383|PigeonReportsLocation

The above rows validate that the oml keeps the data of all the iterations in a single table while fortunately the oml\_seq is reset after each iteration. **This can give us a clue on how we can generate separate tables corresponding to different iterations from the original table.** Furthermore, we see that there is a break of about 86 secs ( = 401 - 315) between two successive iterations which corresponds to the waiting time mentioned in the ED script. However, it will be more helpful and robust if the system by itself generates separate tables for the experiment instances. It will be beneficial to the experimenter if the system gives an ID to each instance of an experiment run in addition to giving an ID to a whole experiment. Then, the oml database tables can be also kept separately for each run which will make data collection and analysis far easier.

(e) It is not easy to copy the parameter values to the nodes in an automated fashion before each iteration starts. We believe an experimenter will find it very helpful if there is a specialized preparation clause in ED. The experimenter can provide the parameters values as written in a single file separated by the iteration Ids and specify the name of this data file in the ED as part of the preparation clause. During the experiment, before each run instance starts, the system can copy only the relevant part of the data file to the corresponding experiment nodes so that those values of parameters are used. This feature will relieve the current burden on the experimenter to manually copy the parameter values to the experiment nodes before each run instance.

1. Comments/Feedback on ORBIT Lab

We have faced several difficulties while doing the experiments on ORBIT. Some of them are contributed by some inconsistent information being present on the wiki or some tool’s version mismatch, and so on. We reported the most recent challenges which we faced on ORBIT lab along with some suggestions for advancement in the previous section.

Below we report the problems which we experienced (before 03/23/2011).

1. The ORBIT wiki page, <http://www.orbit-lab.org/wiki/SandboxMap> : Information about the nodes is not complete or out-of-date. As an example, to the best of our knowledge, some sandboxes have more than two nodes, which is not clearly stated here.
2. We searched for non-trivial example codes for how to simulate mobility of a node on ORBIT. We could not find some mobility emulation examples on the wiki. <http://orbit-lab.org/wiki/Documentation/ArbInterference> does not have any example which illustrates mobility.
3. Normally an experiment is described in a script file, e.g. "my\_experiment.rb", which is run by using the command "omf exec my\_experiment". However, the tutorial script in “omf exec --tutorial“ is already stored in a repository known by the omf exec application. Thus the "--tutorial" option instructs the omf exec application to fetch that script and execute it.

Got this message:

 INFO Experiment: load system:exp:stdlib

 INFO Experiment: load test:exp:tutorial

It is not mentioned on the wiki where these directories are.

1. omf stat system:topo:circle

 # will query the status of the nodes in the topology file 'repository/system/topo/circle.rb'

It is not clear where this directory is.

1. It should be stated that the experimenter should wait some time after ‘omf load’ image and before ‘omf exec ‘. We have to make the node ON via “tell on” and wait for some time. There is some confusion. omf stat [[1,1],[1,2]] says the nodes are up, but the first few messages in ‘omf exec –tutorial’ says that the nodes are down.
2. In wiki, it’s not clear if some commands to write experiment description (ED) are old and unavailable. E.g. ‘orbit’, ‘nodehandler’. Further, we can get warning messages if we do not follow specific omf version features. As an example, we may get the following warning.

**'whenAllUp' is deprecated! Please use 'defEvent' and 'onEvent' commands**

 **WARN exp: Deprecated commands will be removed in future OMF versions**

 **WARN exp: Moreover, they may not allow the use of some features in this version**

1. On OMF website (http://mytestbed.net/wiki/omf/OMF\_User\_Guide) it is said that omf-5.3 is the latest and omf-5.2 is deprecated. However, when I login to nodes on orbit lab, I see only omf-5.2 is available. This is confusing.

I copied/created an ED file (experiment description) hello\_world.rb following omf-5.3 instructions. However, it does not compile. Then, I went back to the deprecated website of omf-5.2 and followed their format of creating ED and then it worked.

1. After we load an image it says : “INFO NodeHandler: Web interface available at: <http://10.10.0.10:4000>”. How to get this web interface?
2. We were unable to download source of madwifi on a node following the instructions available on the wiki. The errors are as follows.

node1-2:~# svn co http://svn.orbit-lab.org/svn/orbit/madwifi/trunk/ madwifi

Authentication realm: <http://svn.orbit-lab.org:80> ORBIT

Password for 'root':

Authentication realm: <http://svn.orbit-lab.org:80> ORBIT

Username: sroy

Password for 'sroy':

Authentication realm: <http://svn.orbit-lab.org:80> ORBIT

Username: sroy

Password for 'sroy':

svn: PROPFIND request failed on '/svn/orbit/madwifi/trunk'

svn: PROPFIND of '/svn/orbit/madwifi/trunk': authorization failed (http://svn.orbit-lab.org)

1. This OML page on orbit is invalid now, but not removed from wiki

http://www.orbit-lab.org/wiki/Tutorial/CollectMeasurements

Same problem applies to the following link.

http://www.orbit-lab.org/wiki/Tutorial/AnalyzeResults

1. We tested that the query tool, sqlite3 does not exist on the console of ORBIT lab. Then, it will be difficult to analyze the data stored by the oml server.
2. Wiki does not state where the log of HelloWorld or other experiment is stored. We can just see an xml file in /tmp on node1-1, which has the following content.

<omlc id='Sender' exp\_id='sb2.orbit-lab.org\_2011\_03\_15\_18\_21\_54'>

<collect url='tcp:oml:3003'><mp name='udp\_out' interval='3'/>

</collect></omlc>

1. On OML wiki, it is written that For WINLAB, the OML data retrieval services are running on the server at the host & port: "oml:5053". It implicitly assumes that we use omf-5.3. However, if we use omf-5.2 which is the standard for the Winlab, this does not work. Instead, the host & port: "oml:5052” are to be used.

Below we report the problems which we experienced after 03/23/2011.

1. We are currently making the pigeon program “sleep” at the beginning for 5 seconds to ensure that Topology Manager program is ready by then. The above sleep is not an ideal solution because it hampers the experiment time calculation. The ideal solution would have been if we start the pigeon program after some time via the ED. But we could not do it because we do not know how the ED (ruby script) can start two applications (of the same group) one after the other with some delay in-between.
2. In near future, we need to add visualization component to our project. We plan to observe the experiment data while our experiment is running. From the ORBIT admin we came to know that the EC provides a web page, which we could use. If we augment the ED to have some graphs of the measured metrics, they can be displayed live on that web page. However, now we are confused about if the above is a good idea: We see on

(<http://www.mytestbed.net/projects/omf/wiki/BasicTutorialStage0-5-3>) that EC Web feature will be deprecated in OMF version 5.4. Some tutorial pages with full description have been already removed from the above wiki page.

1. We get the following error when we try to use offh flag to power off nodes through omf-5.3. sroy@console:~$ omf-5.3 tell -a offh -t system:topo:all

INFO NodeHandler: OMF Experiment Controller 5.3 (git a8a4e16)

/usr/share/omf-expctl-5.3/omf-expctl/tellnode.rb:93: undefined method `loadTestbedConfiguration' for OConfig:Module (NoMethodError)

1. We sometimes see the difference of our experiment running on sb4 compared to the same experiment on the grid. We report that some experiment script does not properly run on the grid, however, we see that same script (with those obvious configuration changes) runs on sb4 perfectly for more than 2400 seconds. The ORBIT admin believes that we are having problems with ad-hoc mode and cell splitting (assuming that we are using ath5k as a driver which in turn is having issues with ad-hoc mode and leader election). We should check which nodes belong to the same cell and we should be able to find out if that is the case. In addition, sb4 is special in the sense that the nodes are in the RF enclosures and are connected with wires to the RF Matrix and have a "perfect" connectivity (unless someone leaves the matrix in inconsistent state which happens every once in a while); also, they are "mono-culture" nodes - i.e. the same, whereas in the grid we have nodes of varying CPU capabilities which might impact timing performance.
2. Typically, the "omf-5.2 load image" takes more than 15 minutes which wastes a lot of time. We would like to know if there is any option to apply to accelerate this step. The output texts from the screen during this step show that if some nodes are down, then "resetting", wait, "giving up" take lot of time. We wonder if there is any reliable way to skip these steps.

From the ORBIT admin we came to know the following: Under the hood, the Disk Imaging process is just an OMF experiment similar to other experiments that one can run. The "omf-5.2 load" command is just a wrapper around that. Thus, one can run it directly with "omf-5.2 exec" and by doing so, one can overwrite the default properties of that experiment. If our goal is to "speed" the imaging, there are 3 properties that we may want to change. If we would like to change these defaults, we can run the imaging experiment directly using the following command:

omf-5.2 exec -s system:exp:imageNode -- --nodes NodeDescriptorHere --image ImageNameHere --timeout TimeoutValue --resetDelay ResetDelay --resetTries ResetCount

It will be very helpful to a beginner if there are some examples with the above options on the wiki. Now the next important thing is what options are available for omf-5.3.

1. Sometimes (seldom) we observed a weird behavior of ORBIT nodes. Say we run an experiment which should stop after 600 seconds, but it may not stop even after 30 minutes. We use normal "wait 600" in our ED ruby script. How to make it perfectly reliable?
2. Sometimes we observed that one of our experiments involving tcp data transfer among nodes halts unexpectedly. We stress that we don't inject any noise in the experiment. For the debugging purpose, we have disabled the ipfilter (which is used only in one node) so that there is no filtering. Note that our experiment runs successfully where only small messages (around 14 bytes each) are sent. The problem occurs when an experiment involves bigger messages (around 1000bytes) --- the message transfer via tcp halts and the interesting thing is it halts at a random point of time. We have further investigated to find out the root of the problem. Basically, while the experiment halts, we manually logged in to the nodes and tried to ping each other to see the connectivity. The summary of our finding is that during the problem the pigeon is unable to ping any other host. Note that the problem happens only after pigeon's some successful data transfer to every other node in the previous rounds.

After discussing with ORBIT admin, we guessed that the source of the problem can be using madwifi driver. Should we use ath5k instead of madwifi? This can be an important question to a beginner. The ORBIT admin added that supposedly ath5k is a bit more mature but it all depends on what features we are looking for. Arguably we should make sure that only one of them is used (i.e. just check that they are both blacklisted in /etc/modprobe.d/ ). We guess these issues could be discussed in detail on the wiki to help the ORBIT users.

1. Comments/Feedback on NORBIT Lab

We stress that one of the goals of GENI is to hide the experiment setup/installation details from the experimenter (user). GENI aims to provide an interface to the experimenter via which the experimenter can submit the specification of the resources which he/she needs to acquire to run an experiment. Then, it is up to the GENI to provide the user the matching resource. It does not matter to the user if the resource is located in one geographical location or distributed over multiple locations as long as he/she can successfully perform the experiment.

To achieve this goal, GENI needs to ensure that the test beds, which are available in different sites, have sufficient commonalities so that they can be tied together to be used in the same experiment. Although NORBIT and ORBIT test beds are designed with the above vision, they still have some differences which need to be addressed before we can meet the above goal. During our experience with these two test beds, we observed that in some cases there are striking differences, even in terms of available software, software version, wiki instructions/information, and so on.

Below we report some of the problems we faced during experiments on NORBIT lab (before 03/23/2011).

1. NORBIT lab has oml with version 2 and NORBIT wiki discusses the usage of oml2. However, ORBIT wiki talks about prior oml version, while ORBIT lab recent images are installed with oml2 clients. This creates lot of confusion to the new user/experimentor. The wiki pages corresponding to old version of oml should be deleted or deprecated.
2. We successfully ran an ED which executes some oml-enabled application in multiple nodes with no error reported. However, initially we did not find any the database entry injected in the oml server. ) NORBIT administrators investigated the problem, and it turns out that there was a version mismatch between the oml client (v 2.5) and the OML server (v 2.4) on NORBIT. Then, they have upgraded NORBIT to the latest oml server version and the problem disappeared.
3. On 01/13/2011, when I was doing experiment on node5 in norbit lab, some other user hijacked my node and loaded a new image. My data was lost. We need to take caution in future. I communicated with the NORBIT admin. After he investigated, I understood that the problem came from my ill-handling of the google calendar service which is used in the reservation process.
4. My application (home, pigeon, or foreign) has multiple threads. Sometimes, I was not able to make my application inject data to the oml server without having any apparent reason. The OML client library is not yet thread-safe, and this might be the reason what was causing the server to reject measurements.  A symptom of the library's lack of thread safety would be garbled network stream.

One work around is to protect all calls to omlc\_inject with a mutex.  That's probably the simplest.  This may interfere with the running of an experiment, but there isn't much else that can be currently done. The administrators said they did some work on making the library thread safe, but it is currently in the middle of being integrated into the main development tree and is currently broken.  It will be released with version 2.6 of OML, but unfortunately there is no firm release date for OML 2.6 yet.

1. Summary and More Comments

We have reported our achievements in emulating a pigeon network on ORBIT and NORBIT testbeds. We have recently completed the visualization component which shows the basic topology of the experiment. In the GEC (GENI Engineering Conference) 14, we have given a demo of our emulation work including the topology visualization. We think in future a user would like to move towards more automation of the experimentation process in the context of running an experiment with multiple iterations (i.e. trials) with dynamic parameters. The automated process should also include the resource request step and the preparation step. In this report, we have made some suggestions to ORBIT on addressing some of the above challenges.