ORBIT Testbed Software Architecture: Supporting Experiments as a Service

Maximilian Ott, Ivan Seskar, Robert Siraccusa, Manpreet Singh
WINLAB, Rutgers U.
{max, seskar, rjs, singh}@winlab.rutgers.edu
www.orbit-lab.org

Abstract

This paper presents the software architecture of the ORBIT radio grid testbed[1]. We describe the requirements for supporting the lifecycle of an experiment and how they influenced the overall design of the architecture. We specifically highlight those components and services which will be visible to a user of the ORBIT testbed.

1. Introduction

The ORBIT project [1] provides a flexible wireless network testbed that is open to the experimental research community. It was started to address limitations in understanding real world wireless networks caused by the community's reliance on simulations based on simplifying assumptions, or simple experiments with a small number of devices.

While in most fields of science an experimental result will only become accepted when it has been repeated by peers, this practice has so far been an elusive goal for the networking community. Even repeating results based on simulations is difficult due to dependencies on various software and hardware configurations that are hard to capture.

The ORBIT project is attempting to address these issues by:

- Providing a large set of resources to conduct interestingly large experiments in realistic settings.
- Capture all dependencies and most environmental conditions (especially complex radio link layer issues) to facilitate repeatable experiments.

To that end, we are building two different testbeds. The first is an indoor grid consisting of 400 nodes arranged in a 20 by 20 grid separated by about 1 meter between adjacent nodes. Each node is built on a standard PC platform with multiple wirelesses and wired network interfaces. For the second testbed, we will deploy similar nodes over an area of approx. 1.5 square miles. Some of these nodes will be placed on campus shuttle buses to provide mobility along fixed routes and a fixed schedule. The second testbed will also include a programmable UMTS basestation with respective network interfaces on a subset of mobile and stationary nodes. In addition, we will deploy various devices to measure traffic and interference across the relevant portion of the radio spectrum. We will also provide signal generators to allow the experimenter to create controlled interference.

Repeatability of wireless experiments is clearly a challenge, but it is a basic architecture principle throughout the design of every aspect of the testbed. While we cannot repeat the outdoor radio channel between two shuttlebus-based radios during a summer thunderstorm, we will capture all necessary information to re-run an experiment by any authorized user at a later time. This way an experimenter can run multiple experiments back-to-back using the same equipment, software, and property settings during similar environmental conditions. In fact, one of the primary goals of the indoor grid-based testbed is to control the radio environment as much as possible to allow for repeatability independent of time. It is located in a place with a relatively quiet ambient spectrum, with the majority of interference produced by controllable instruments in a repeatable fashion.

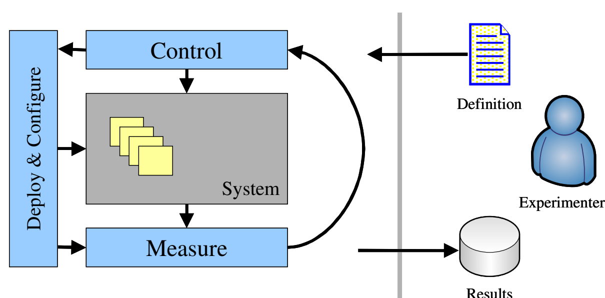

Figure 1: Model of experiment

[1] Research supported by NSF NRT Grant #AN10335244.

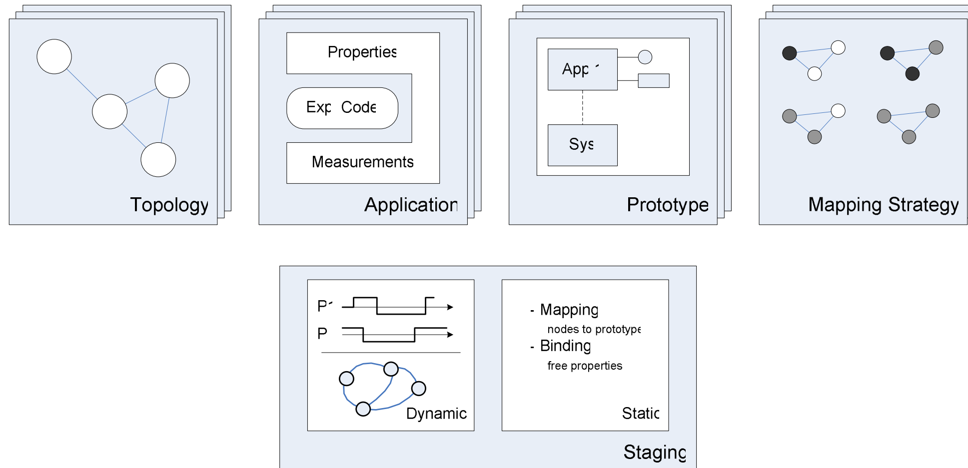

Figure 2: Components of experiment definition

However, building a wireless testbed for the community that can be reliably used by experimenters remotely poses an interesting design challenge and is more than just an assembly of interesting hardware. In order to make it useful for a wider audience it must be easy to use. In fact, our core mantra is:

- Make simple experiments easy
- Make complex ones possible

In other words, we need to provide various levels of abstractions and support and let the user pick the right one suitable for their experiments. For instance, most experiments on mobile applications will be happy with a "known-to-work" node configuration, while those working on low-power sensor networks may prefer to work "on the bare metal".

What remains constant is the focus on performing experiments. We want to make it as easy as possible to design, schedule, perform, and analyze an experiment. Only a clean and structured approach will ensure that we capture all the intrinsic information to facilitate repeatability. As an important added benefit, we can automate many tasks and in turn reduce the time an experimenter requires access to the shared resources.

The remaining sections are organized as follows: We first introduce a model for experiments and describe the various parts which define them. We then describe OML, a service for collecting measurements; and how to port applications to ORBIT. Finally, we provide a brief tour of the remaining, supporting services.

2. Defining an Experiment

As mentioned in the introduction, the ORBIT testbed is a shared resource allowing its users to perform experiments. But what is an experiment in this context? Simply put, an experiment can be defined by a collection of resources assembled into a system. Some of these resources will expose properties which can be set and possibly dynamically changed during the execution of the experiment. Finally, and most importantly for scientific research, measurements will be taken from various parts of the experiment setup to allow for analysis either during or after the experiment run.

A more detailed model is shown in Figure 1. The experimenter on the right provides a description of the experiment, its static and dynamic behavior and the set of results which should be collected. The description of the experiment allows the automatic deployment and configuration of various resources to construct the experiment. It also describes the value of experiment properties, either as constants, or as functions of time, or events. And finally, the experiment description defines what to measure and how often to measure. It should be noted, that this model also supports interactive experiments where the experimenter interactively changes properties, most likely in response to observing some of the measurements.

In this model, the experiment description alone should hold all the information required to build and run the experiment, as well as re-run it at a later time. Specifically, experiments on the ORBIT testbed are defined by the topology of the network, and the configurations of the individual nodes in this network. The sheer number of possible combinations requires a more systematic approach which is shown in Figure 2. The description of an experiment consists of the following sections:

- Topology

- Applications
- Prototype
- Mapping
- Staging

2.1. Topology

The topology section defines the nodes involved in the experiment and the communication pattern between them. While the choices will be limited for the outdoor testbed and primarily depend on the bus schedule, the design of the grid testbed allows for greater flexibility. We are using controlled interference from some of the nodes and signal generators to "shape" the channels between the remaining nodes and with it the topology. However, realizing an arbitrary topology remains a very challenging research topic. For the foreseeable future, the experimenter will be required to choose from a library of tested topologies such as linear chain, grid, star, etc. We also expect the research community to select a small number of topologies to benchmark results for scenarios, such as "class room", "office", "hallway", "chain of sensors", "uniformly distributed sensor grids", etc.

2.2. Applications

The applications section will list all the applications used by this experiment, such as traffic generators, traffic sinks, etc. This list can only contain already registered applications. We are treating applications as resources as they can be used by many different experiments and we also expect that there will be far fewer application developers than experimenters. Similar to an experiment an application is defined by an application description (AD). The AD describes the application's dependencies on other resources, such as libraries, operating systems, devices and their drivers. It also describes all properties, constraints on their value, and if they can dynamically be changed during an experiment. In addition, the AD lists all the measurements an application provides and which can be collected during an experiment. Appendix C lists a shortened sample of an AD.

The application itself is packaged appropriately and uploaded to the ORBIT repository. From there it can be loaded on the desired nodes involved in the experiment prior to running the experiment.

2.3. Prototype

The grid testbed provide the users access to 400 nodes. However, we assume that an experimenter will create a much smaller number of roles which will be assigned to the nodes with only small "localizations". We have formalized that by allowing for the definition of prototypes and mapping strategies, which describes the assignment of one or more prototypes to each node used in the experiment.

A prototype description contains a list of property declarations (e.g. packet size, packet rate) and a list of applications which should be installed on the node. In addition, it defines for each application bindings for all relevant application properties, as well as a listing of all the statistics to be collected and how often to collect them.

The prototype properties provide a mechanism to bind application properties to node specific properties, such as the node's location.

Prototypes can be defined directly in the ED, or uploaded as a reusable component. The latter case allows and ED to simply refer to a prototype ensuring consistency across multiple experiments. Externally defined prototypes have a version number like applications. This allows the independent refinement of a prototype definition while capturing the version used for a particular experiment maximizes repeatability.

2.4. Mapping

As mentioned before, a mapping strategy defines what prototype(s) to assign to each node in the experiment. The simplest strategy is a list explicitly assigning a specific prototype to a specific node. Other strategies may define probability distributions, or algorithms to better capture certain scenarios, or remove bias. Mapping strategies can also define properties which in this case will be bound to values specific to an experiment. Similarly to prototypes, mapping strategies can be defined inside and outside the ED, including versioning.

2.5. Staging

The sections of the ED described so far have dealt with describing the resources needed for the experiment and how to link them to each other. They also provide a mechanism to bind properties of a specific resource or class of resources to "experiment" properties. Next, we address the handling of the dynamic component of the experiment that involves changing properties (such as packet size, frequency of operation, power settings, packet rate, etc.) during experimentation. For these dynamic properties, the respective values may change with time or as a reaction to a particular event.

As a simple example, we consider an experiment to measure the correlation between received throughput and packet size for a particular offered load. Let us also assume that the ED has defined a property "generator/packetSize" bound to the traffic generator on all sender nodes. The following script will increase the packet size by 256 every 2 seconds until it reaches 1280:

```
for {set s 256} {$s <= 1280} {incr s 256} {
  setParam /e/generators/packetSize $s
  sleep 2000
}
```

In contrast, we could take advantage of the real-time measurement support in Orbit (which we describe later) and design an experiment which will increase the packet size until the packet error observed by the receiver exceeds a certain value:

```
set s 0
do {
  incr s 256
  setParam /e/generators/packetSize $s
  sleep 2000
  set pktError [oml::runQuery "pktError"]
} while { $pktError < 0.1 }
```

The experiment specific properties are also accessible to the experimenter through a web-based management console, or programmatically through a web service. The latter allows the experimenter to implement, or use more advanced experiment management algorithms, or applications.

However, one of the major challenges in controlling a large number of distributed resources is dealing with error conditions and the latency with which they can be discovered. For instance, in the above example, the setParam function may set the respective property on many nodes. In a conventional staging script we would need to add many additional lines of code to ensure that indeed all generators have successfully performed the change. In short, error checking and resolving them will quickly overwhelm the actual experiment code. This will not only increase the effort necessary to define a robust experiment, it will also make it much harder to extract the essence of an experiment from the script. What is necessary is a clean separation of actual experiment related instructions and robustness related functionality.

To achieve this we are mapping all properties and their meta information into a tree structure. For instance /e/generators/packetSize defines the path from the root of the tree to a tree element which represents the packet size of all "generators" used in the experiment, while /n/n2-3/proc/generator/packetSize represents the same property of a specific generator running on node "n2-3". In addition, the state of the tree is monitored by rules. These rules fire, either periodically, or due to specific changes to the state of the tree. For instance, the mapping stage in our previous example added a rule to synchronize the two mentioned elements. In addition, the "/n/n2-3/proc/generator/packetSize" element has multiple children nodes. The "current" element contains the value confirmed by the application. The "requested" element contains the value requested by the staging script. In fact, it is this tree node which is set by the above mention synchronization rule. We can now add a rule to monitor discrepancies between the values of related "requested" and "current" elements. If these values remain different over a certain period, the rule fires and the associated script can either raise an alarm, or try to re-issue the respective command.

The same mechanism can also be used to implement a type of barrier synchronization. The following rule fires when all nodes have booted up. It then configures the first wireless interface (w0) to operate in ad-hoc mode. In addition, the ESSID of w0 on all nodes in the "sender" group is set to "ORBIT1".

```
whenAll /n/*/*/system/status are "UP" do {
     configure /*/* /net/w0/mode ad-hoc
     configure /sender/* /net/w0/essid ORBIT1
     ...
}
```

These settings do not really need to be synchronized, but this way the Node Handler will only multicast two messages to the entire testbed. The first argument of "configure" is a pattern defining the recipients of the respective command.

However, we can easily change the rule to fire as soon as an individual node comes up, resulting in one or two messages per node:

```
on /n/*/*/system/ is "UP" do {
  configure $node /net/w0/mode ad-hoc
}
on /n/sender/*/system/ is "UP" do {
  # this one only fires for "sender" nodes
  configure $node /net/w0/essid ORBIT1
}
```

This approach is very powerful. We have been building up a library of rules to monitor the health of the system and fix small problems autonomously. These rules can easily be incorporated in experiments allowing the experimenter to concentrate on what should be done when everything works fine. However, we acknowledge the difficulties of debugging a rule-based system where the interactions between rules is

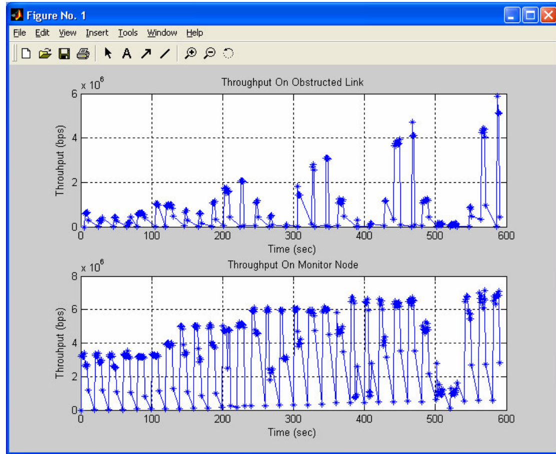

Figure 3: Sample experiment result

not always obvious. Only time and feedback from the experimenter community will tell if we are on the right track.

3. Measurement Framework (OML)

We have repeatedly stressed the importance of collecting measurements during the course of running an experiment. Collecting measurements is often an afterthought and done in an ad-hoc fashion. A standard method is writing lines of tab separated numbers and symbols into various log files. It is not uncommon that the process of collecting all log files from all the nodes in a distributed experiment may take longer than the experiment itself. In addition, each application tends to define its own format which is rarely documented. It is also common that crucial information, such as node ID, date, or parameter settings are missing from theses logs.

For all these reasons we have decided that ORBIT should provide a service to support the collection and to a certain degree, the analysis of measurements. We therefore, designed and built the ORBIT Measurement Framework (OML). OML defines a Measurement Point as a point in a program where the program collects a tuple of measurements. Or more specifically, where the program calls a function of the OML client library, or an automatically generated wrapper functions with a type-safe signature. The tuple will be serialized by OML client library, and sent to a Collection Server where it will be inserted into a relational database. At the end of the experiment, all the information generated by this experiment should reside in a single database which will be available to the experimenter for further analysis. As many commonly used tools, such as Matlab, or MS Excel, as well as scripting languages, such as Perl and TCL, provide SQL database adapters, analyzing an experiment will become much simpler. Especially when compared to collecting, parsing, and merging hundreds of log file.

Figure 3 shows the result of a simple Matlab script which is listed in Appendix B.

All measurement points are described in the Application Definition. This allows various tools to

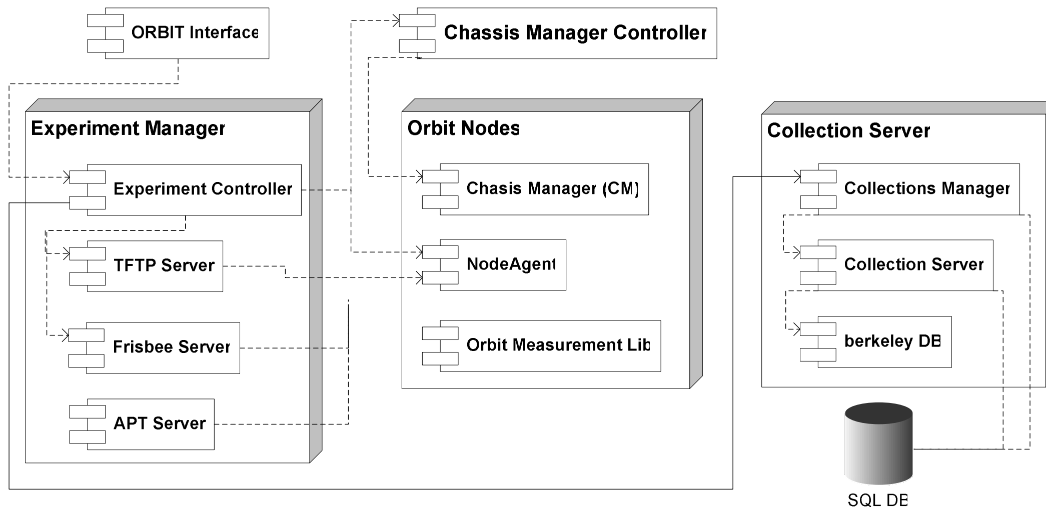

Figure 4: Testbed Services

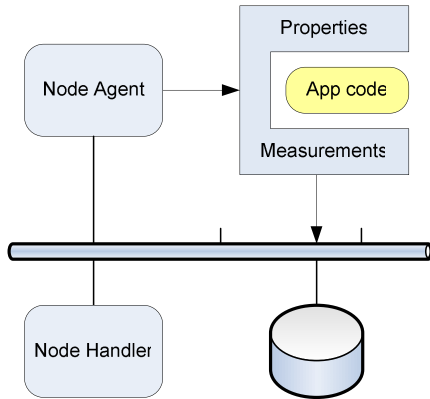

Figure 5: Staging experiments

automatically generate the above mentioned wrapper functions, as well as the schema for the collection database.

The OML client library also allows the dynamic insertion of filters on a per-experiment basis. These filters pre-process the emitted measurements on the client side and help to control the amount of measurements actually collected. For instance, we have implemented filters which average over a certain time frame, or only report substantial changes of a measurement value. OML also provides an API to allow experimenters to customize OML for their specific purposes.

It should be noted that we differentiate between the definition of an application's Measurement Points and the filter settings in the OML client library for a particular instance of the application. The MPs are defined by the application developer in the AD, while the filter settings are specified by the experimenter in the EP, as they will depend on what subset of potential measurements an experenter is interested.

More details on the internal design and how OML can scale to hundreds of nodes can be found in [2]. This document also contains early-stage performance measurements.

4. Applications on Orbit

For most users, the core of the experiment, such as a routing algorithm, will be encapsulated in an application. The experiment itself is then a collection of such applications running on various nodes with specific property settings. While the Orbit infrastructure will take care of configuring the internals of the nodes, we need to provide a framework for application developers to easily adapt their applications to the Orbit experiment model as outlined in a previous section.

Figure 4 shows how an experiment is staged on Orbit. A central Node Handler (NH) communicates with Node Agents (NA), one on each active node. Applications will be loaded and executed by the NA on instructions from the NH. These instructions will also include the initial property settings for the applications. However, to allow the experiment to change properties dynamically and tie an application into the OML framework, the application needs to be integrated into what we call the Orbit Application Harness (OAH). This process starts with the definition of the Application Definition (AD). We have developed a set of tools which automatically generate the harness code which is visualized as the C-clamp in the top right corner of Figure 4. The application only needs to provide an entry function and an update function if it supports dynamic properties. The harness code includes a data structure for all properties and type-safe function declarations for every measurement point defined in the AD. We will also provide a set of "make" and "ant" targets to automate the tasks of compiling the application; packaging it, together with the AD, into an "apt" package; and uploading the result to the Orbit repository.

We believe that a subset of the OAH will be useful even outside of Orbit and we are planning on releasing a toolkit, possibly including an appropriate subset of OML, for general use in the future.

5. Behind the Scene

So far we have concentrated on describing the parts of Orbit which are visible and of importance to the experimenter. In the reminder of the paper we will describe the various components which facilitate the execution of experiments. Figure 5 depicts an overview of all the services currently deployed.

We mentioned in the Introduction that we want to support experimenters at various "comfort" levels: from the "cushy" user-space with pre-configured devices, to the bare metal. However, to allow experimenters full access to all resources also creates big operational challenges. For instance, how do we reclaim a node when an experimental device driver locks up the entire node?

For many experiments it will be necessary to perform operations which normally require administrator or root privileges. How can we ensure that these changes do not affect the next experimenter?

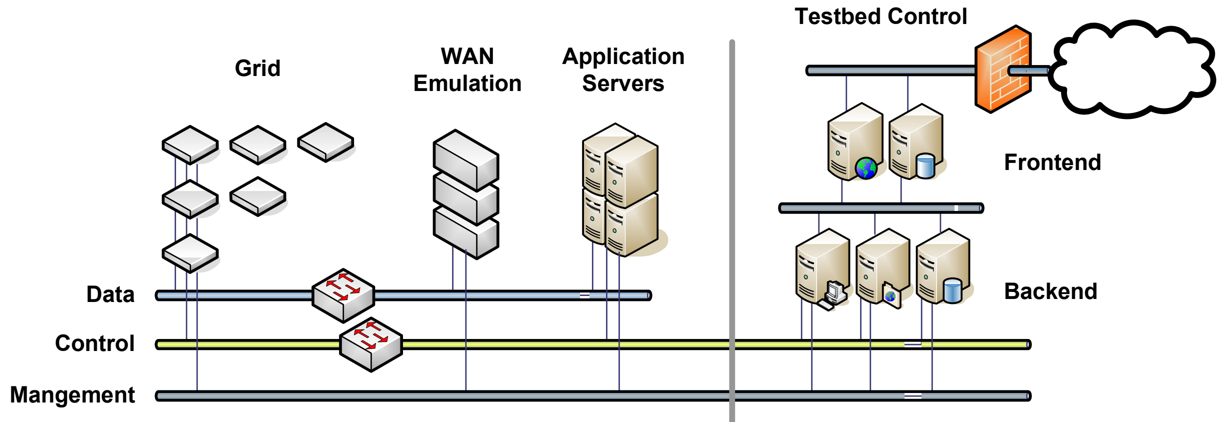

Figure 6: Network architecture

We have repeatedly mentioned the importance of collecting measurements. Ideally, these measurements should be available to the experimenters almost instantaneously to allow for steerable experiments to maximize the time slice given to the experimenter, or allow for the earliest termination of the experiment if something goes wrong. How can we collect measurements without interfering with the experiment itself?

The following sections will try to answer theses questions.

5.1. Orbit Nodes

The nodes serve as the primary platform for the experiments. They are based on an off-the-shelf PC platform with some modifications:

- 1 GHz VIA C3 processor with 512 MB RAM and a 20 GB local disk
- Two mini-PCI based 802 a/b/g interfaces
- USB ports and one PCI slot for further expansion
- Two 100BaseT Ethernet ports
- Integrated chassis manager

The chassis manager (CM) provides an additional Ethernet port which allows us to remotely monitor the status of a node independent of the node's CPU and the network interfaces under its control. The CM can independently power the node on or off, it reports supply voltage levels as well as temperature, and it provides remote access to the node's serial console. The experimenter will only have restricted access to these capabilities as the CM provides us with the crucial safety net which allows us to reclaim a node no matter what state it is in.

5.2. Testbed Architecture

Figure 6 shows the different components of the grid testbed and the various networks connecting it. As mention above, each node has two 100BaseT Ethernet ports. The "Data" port can be exclusively used for experiments. All control traffic, such as communication between the Node Handler and all its Node Agents, as well as all measurements will use the Control port. In fact, the default settings leave the Data port unconfigured and require the experimenter to specifically configure it through the NodeHandler. At the same time, the Control port comes up at boot time with an IP reflecting its location in the grid. It can also not be changed by an experimenter. While we cannot easily enforce it, we encourage all experimenters to simply forget about the Control network. If an experiment needs a fixed network connection for a node, as would be the case for an access point, it should use the Data port.

The testbed also includes a few WAN emulation nodes based on NISTNet[3]. We can emulate a WAN connection between two nodes by assigning their respective Data ports to separate VLANs. The two ports of the WAN emulator will be connected to the same VLANs and forward packets according to the characteristics of the desired WAN connection.

A set of generic application servers provide support for experiments such as mobile terminal access to Internet based services (e.g. web pages, multimedia streams, etc.). In fact, the WAN emulation will most likely be used between nodes representing wireless access points and the application servers.

5.3. Mobility support

One of the most challenging goals we set ourselves is to support mobility without physical movement as it constitutes an even bigger maintenance challenge. We are currently experimenting with an approach where the mobile application will reside in a server off the grid using a virtual wireless device. The virtual driver will connect through a tunnel to a node with a real wireless interface which simply forwards all received packets into the tunnel and transmits all packets coming from the tunnel. Coarse mobility can be accomplished by simply redirecting the tunnel to a node at a different location according to a specific mobility pattern.

5.4. Utility Services

The testbed also contains a set of utility servers to provide standard services, such as NTP to synchronize timestamps across all nodes. These servers also host the OML backend as well as the Node Handler.

One service we want to specifically mention is Frisbee [4] which was developed by the Emulab [5] team. Frisbee implements a clever, secure multicast protocol to image the disk of many nodes simultaneously. As mentioned before we want to ensure that an experiment cannot affect a future one. As we give user complete access to the nodes, the safest method to ensure a clean node is to fully install a new image on every node at the beginning of every experiment. Obviously, this has to be done as quickly as possible to minimize “retooling” time between experiments. Our goal is to complete this task on all 400 nodes in less than five minutes. We have not been able to verify that yet, but our experiences on smaller set-ups confirm the results presented in [4] which also includes measurements for even larger setups, all in line with our goal.

5.5. User Portal

The user portal is the interface between the experimenter and the testbed. It supports the full life-cycle of an experiment: define, schedule, run, and analyze. As many of these tasks lend themselves to automation we are providing all exported functionality primarily as a web service and restrict the user interface component to those services. This way we will ensure that anything a user can do through the web interface, can also be accomplished by program executing in the user’s domain. In fact, we hope that this approach will seed various tools we would not have thought of, or the resources to realize them.

6. Conclusion

In this paper, we presented the software architecture design for a novel radio grid emulator testbed. We have introduced a model for defining experiments consisting of various re-usable sections to facilitate systematic as well as repeatable experiments. We also described many of the services we developed to assist testbed users. Finally, we explained the “safety nets” which allows us to provide the users with full access to almost all resources while maintaining 24/7 operation in an (almost) lights-out facility.

7. References

[1] I. S. D. Raychaudhuri, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, M. Singh, "Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols," submitted to the IEEE Wireless Communications and Networking Conference, New Orleans.

[2] M. Singh, "ORBIT Measurements Framework and Library (OML): Motivations, Design, Implemantation, and Feature," submitted to Tridentcom, 2005.

[3] D. S. M. Carson, "NIST Net A Linux-based Network Emulation Tool," Computer Communication Review, vol. 33, 2003.

[4] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, Chad Barb, "Fast Scalable Disk Imaging with Frisbee," presented at USENIX Annual Technical Conference, 2003.

[5] Emulab Homepage, http://www.emulab.net.

A. Experiment Definition

```
<?xml version="1.0" encoding="UTF-8" ?>
<orbit xmlns="schema.orbit-lab.org/062804">
  <experiment id="winlab:wp3:05022004">
    <description>
A simple 2x2 grid with diagonal pairs of sender/receivers on separate channels
    </description>
    <project refid="winlab:wp3" />
    <topology refid="public:topology:grid:1.0" />
    <prototypes>
      <prototype pid="sender" extends="winlab:wp3:prototype:sender">
        <!-- bind parameter to dynamic value in state space "/e/senders/..." -->
        <set-param name="channelA" binding="senders/channelA" />
        <set-param name="channelB" value="1" />
      </prototype>
      <alias pid="receiver" refid="winlab:wp3:prototype:receiver" />
    </prototypes>
    <mapping refid="winlab:wp3:mapping:diagonal-2-2" xOffset="0" yOffset="0"/>
    <stagging refid="winlab:wp3:mapping:ramp:1">
      <set-param name="maxPacketSize" value="1280" />
    </stagging>
    <experimenters>
      <experimenter refid="max" />
    </experimenters>
  </experiment>
</orbit>
```

B. Sample Matlab Script for Analyzing Results

The following script was used to create Figure 3:

```
function nsf(dbServer, dbUser, dbPW, database);

% Part where we retrieve data from the database;
mysql('open',dbServer, dbUser, dbPW);
mysql('use', database);
output = struct('time',[],'thr_all',[],'node',[]);
[output.time, output.thr_all, output.node]
   = mysql('select timestamp, throughput, node_id from group2');
[thru1_4, time1_4, thru3_1, time3_1] = sort_mysql(output);

% Finally, the plotting part
subplot(2,1,1);
plot(time1_4, thru1_4, '-*');
title('Throughput On Obstructed Link');
xlabel('Time (sec)'); ylabel('Throuhput (bps)'); grid on;
subplot(2,1,2);
plot(time3_1, thru3_1, '-*');
title('Throughput On Monitor Node'); xlabel('Time (sec)');

ylabel('Throuhput (bps)'); grid on;
```

C. Application Definition

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<orbit xmlns="schema.orbit-lab.org/062804">
  <application id="orbit:winlab:sensorNets:ap">
    <name>AccessPoint</name>
    <version major="0" minor="1" revision="0" />
    <organization>
      <name>WINLAB, Rutgers University</name>
      <url>http://www.winlab.rutgers.edu/</url>
    </organization>
    <shortDescription>Simulate an access point in a sensor network</shortDescription>
    <description>
An access point which periodically sends out a beacon advertising its capabilities and records topology
and routing information.
    </description>
    <url>http://apps.orbit-lab.org/sensorNets/ap/</url>
    <properties>
      <property>
        <name>sensornet_interface</name>
        <mnemonic>s</mnemonic>
        <type>xsd:string</type>
        <dynamic>yes</dynamic>
        <description>Device name for sending beacons</description>
      </property>
      …
    </properties>
    <measurements>
      <measurement id="topology">
        <metric id="node_id" type="xsd:int">
          <description>ID of reporting access node as set in property "sensornet_node_id"</description>
        </metric>
        …
      </measurement>
      …
    </measurements>
    <!--  Admin & Developer -->
    <issueTrackingUrl>http://apps.orbit-lab.org/issues/winlab/sensorNets</issueTrackingUrl>
    <repository>
      <development>scm:cvs:pserver:anoncvs@cvs.orbit-lab.org:/winlab/sensorNets/ap</development>
      <binary>apt:repository.orbit-lab.org/orbit/binary:???</binary>
    </repository>
    <developers>
      <developer>
        …
      </developer>
    </developers>
    <dependencies>
      <dependency>
        <id>libmac</id>
        <version>>= 0.4</version>
        <url>apt:repository.orbit-lab.org/debian/binary:libmac</url>
      </dependency>
    </dependencies>
    </application>
  </orbit>
```