Introduction

Proposal

Test the internal network of a home by testing different aspects of connectivity between two devices and comparing them. The tests will be usable by ISP services and network owners. This project is a proof of concept.

Motivation

Our sponsor, CableLabs, gave us the challenge of creating a mobile and web application that interfaces with your internal network to give you diagnostics on your connection. The group was split up into three groups to focus on each aspect of the code base. The first group worked with the router (which we have named Mercury), the second group worked on iOS and Android development (Hermes), and the third group worked on the web portal and database (Zeus).

Authors

Name	Focus	Email
Zacharia Anders	Web Portal, Database, Tests	Zach@nde.rs
Nicolas Broeking	Mobile Development and Tests	nbroeking@me.com
John Jones	Team Lead, Documentation	jthmjones@gmail.com
Sarah Feller	GUI Development and Documentation	sarah.e.feller@gmail.com
Joshua Rahm	Router Development and Tests	joshuarahm@gmail.com
Michael Williams	Web Portal and Documentation	mike2457@gmail.com

Target Audience

CableLabs CLIPPER Team

Installation

Mobile Phone and Tablets

Android

Load the project in Android Studio. Press the run button. This will push the binary down to the phone. *Note: The phone must be in developer mode*

iOS

Load the project into xcode via the .xcodeproj file. Then press run at the top of the GUI. This will push the binary down to the phone. For iOS Devices you need to have valid apple code signing certificates. To obtain these you must be a registered iOS developer.

Router

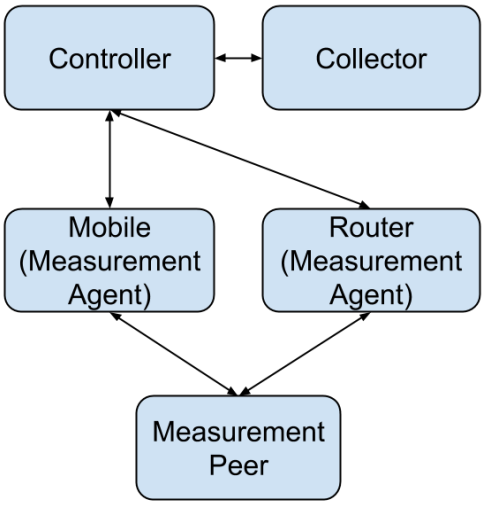
- After installing OpenWRT, the instructions of which are on the public OpenWRT wiki, copy the tarball package-mips.tgz to the router (using scp or other means). Log into the router via ssh or telnet. Extract the tarball with tar -xzvf package-mips.tgz. Cd into the package_mips directory and finally run ./startoplabs.sh. This will start the process.
- To install the process to a standard directory make sure the libraries in lib are located somewhere on LD_LIBRARY_PATH (i.e. /lib or /usr/lib)
- make sure that the port 8639 is not blocked by the firewall

Controller

- All of the Python dependencies for the web service are listed inside the zeus/requirements.txt file. You can use pip to install all the necessary packages and versions from this file
- This project is designed to be deployed on mod_wsgi. Inside zeus there is a wsgi file that can be modified to suit the local install.

System Architecture

Figure 1. Basic Overall System Architecture



Mobile Devices (Hermes)

- The mobile devices are what the user interacts with in order to start a performance test.
- This allows an interface to login to the controller, start a performance test, and view results.

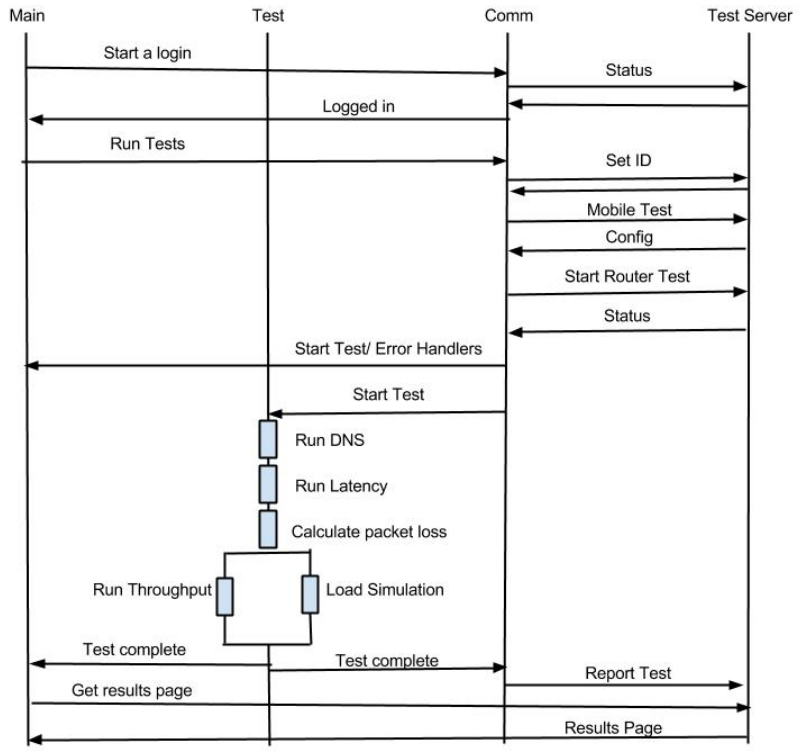
Router (Mercury)

- The router runs a performance test at the same time as the mobile device. We use this to be able to compare where bottlenecks are in the network.

Controller (Zeus)

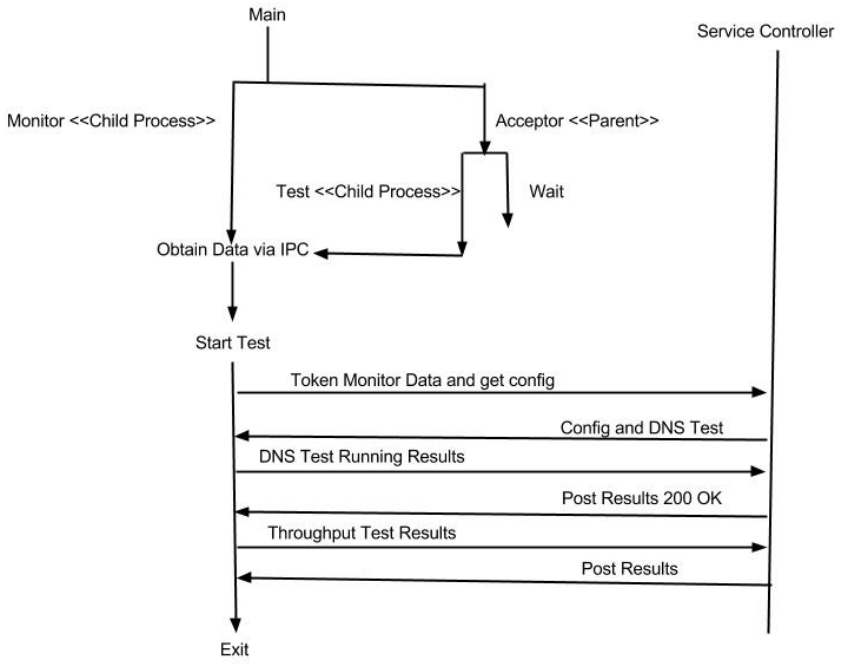
- The controller acts as a bridge between all moving parts in the system. The mobile devices and routers talk directly to the controller to collect data and for message passing.

Figure 2. Mobile Application Sequence Diagram



Our mobile application contains three subsystems, the Main, the Communication, and the Tester. When the application is started all three subsystems are set up. If the application is running in the background, the main subsystem stops, but the Communication and Tester subsystems continue running. The GUI is updated at every stage change throughout the system. We provide error handling in the form of pop ups that tell the user what happened and allows them to move to a screen that corrects a the problem.

Figure 3. Router Sequence Diagram



Our router application contains four subsystems: a Throughput subsystem, a DNS subsystem, a Logging subsystem, and a Monitor subsystem. The Monitor subsystem gets forked once from the main.cpp file and is set to store data to look at the ambient network usage. The Throughput and DNS subsystems are threads that run to perform network analysis on the system. The logging subsystem is used to multiplex logs for users.

Website Documentation

REST API

Authentication

Overview
There are three types of Authentication in the OpLabs Web Application. To make it easier to keep track of which type is which (and where it applies), I will attempt to color code them.
Table 1. Types of Authentication in the Web Application

Authentication Type	Target Platforms	Notes
Client Session	All Major Browsers	Client sessions are stored using Flask's authenticated cookies. These are persisted by the browser, and enable stateful browsing. (I.E. Being logged in) **Multiplicity**: Many client session can exist per user.
User Token	Mobile (iOS, Android)	There is a REST API for generating tokens. The client application is responsible for storing and forwarding their authentication token with every request manually. **Multiplicity**: Only one user token can exist per user.
Router Token	Router (OpenWRT)	Router tokens are generated when the **web server** decides to initiate a test (whether by consequence of user action or automatically). This token is pushed to the router on first contact, and must be used on subsequent communications. **Multiplicity**: Every router token points to a TestResult, which is owned by exactly one user.

Initial Authentication

Each of the three types of authentication are initialized in their own manner. Each is described in greater detail below.

1. Initialization of the Client Session is performed when the user logs into the website. This is done via the web interface, where a user can submit their email and password to log in. If the server accepts the (email, password) pair, it stores an authentication cookie on the client's browser. This cookie is stored in plaintext, but signed by the server's randomly generated key to prevent tampering. No sensitive information is stored in the user's client-side session.
2. Initialization of the User Token is performed when a RESTful web request is `POST /api/auth/login` with valid parameters in the two expected fields, `'username'`, and `'password'`. If the server accepts the given (email, password) pair, it will generate a JSON object containing an `'auth_token'` field. This is the User Token, and should be stored by the caller in order to make future authenticated calls. Any RESTful web requests missing this token will immediately fail.
3. Initialization of the Router Token is performed immediately prior to contacting a router to begin a test. When starting a test, the server will generate a unique token for the router. This token links the router to a specific test result record, which links the router to other tests and the user's account.

Running Tests

Beginning Mobile Tests

When beginning new tests from the mobile application, there are a couple steps that need to occur. It is assumed that every POST or GET request mentioned from this point onward is accompanied with a valid User Token.

1. A new test set needs to be created. This is what all of the individual test results are tied to. This can be done by POST /api/test_set/create. This will return a JSON object which includes a set_id.
2. The next two steps can be done in either order
    a. The mobile test:
        i. Before beginning the mobile speed test, the phone should POST /api/start_test/mobile. This POST should include the set_id, as well any additional information (Mobile OS, IP, Network type, etc.)
        ii. This request will return a configuration JSON blob, including a couple lists of 'ookla_ips', 'dns_ips','ping_ips', as well as a result_id. These are subject to change as the tests themselves are implemented. The Phone should then begin the test.
        iii. After the phone collects its results, it can send a JSON map of results to POST /api/test_result/<result_id>/edit. This map is expected to use key names which match columns in the database.
    b. The router test:

i. In order to begin a router test, the mobile application should POST /api/start_test/router. This will return a JSON blob which includes a result_id.
   - This API takes an optional 'address' parameter, consisting of an IP (and optional port) on which to contact the router (e.g. '127.0.0.1', '127.0.0.1:1234')

ii. Asynchronously, the web server will attempt to reach out to the router and 'poke' it, initiating the router testing process.

iii. Because the mobile application needs to wait for the router test to finish, it can use this result_id to periodically poll GET /api/test_result/<result_id>/status. This will return the status of the current router test, or some sort of error code/error message.

3. At this point, you would retrieve all of the data for your set_id and display it to the user. (See section [Retrieving Results])

Retrieving Results

Retrieving Mobile/Router Test Results

1. Make a request to /api/test_result/<id>
2. The API will return JSON that looks like this:

```
{u'connection_type': u'wired',
u'device_ip': u'0.1.2.3',
u'device_name': u'Test harness',
u'dns_response_avg': 30.0,
u'dns_response_sdev': 1.22,
u'download_latencies': [10,
                        15,
                        20,
                        25],
u'download_latencies_avg': 17,
u'download_throughputs': [100.0],
u'download_throughputs_avg': 100.0,
u'interface_stats': [{u'intf': u'eth0',
                  u'rx_bytes': 200,
                  u'timestamp': 0,
                  u'tx_bytes': 100},
                  {u'intf': u'eth1',
                  u'rx_bytes': 1073741824,
                  u'timestamp': 1,
                  u'tx_bytes': 16777216},
                  {u'intf': u'eth1',
```

```
                    u'rx_bytes': 200,
                    u'timestamp': 2,
                    u'tx_bytes': 100}],
    u'latency_avg': 20.0,
    u'latency_sdev': 1.25,
    u'message': u'This is my example message.',
    u'network_type': u'IPv4',
    u'packet_loss': 0.01,
    u'packet_loss_under_load': 0.05,
    u'state': u'running',
    u'status': u'success',
    u'upload_latencies': [10,
                    15,
                    20,
                    25],
    u'upload_latencies_avg': 17,
    u'upload_throughputs': [0,
                            1,
                            2,
                            3,
                            4],
    u'upload_throughputs_avg': 2}
```

Editing Results

Posting Mobile Results

1. Collect Data
2. Post data to /api/test_result/<id>/edit
3. Expected mobile parameters:

```
    {
      'dns_response_avg' : <dns>,
      'packet_loss' : <packet_loss>,
      'latency_avg' : <latency>,
      'upload_throughputs' : <throughputUpload>,
      'download_throughputs' : <throughputDownload>,
      'packet_loss_under_load' : <packetlossUnderLoad>,
      'throughput_latency' : latencyUnderLoad,
```

```
        }
```

Throughput

Client testing Procedure

1. Retrieve ‘server_ip’ key from config[throughput_config]
   Format: “<ip_address>:<port>”
2. Start background UDP Latency tests
   a. Record round-trip latency
   b. 200ms delay between tests (5 tests per second)
3. Connect to server_ip (Possibly with multiple connections, TBD)
4. Receive bytes from server
   a. The server will attempt to continuously send bytes for 10 seconds
5. Timeout after 10 seconds
6. Begin sending bytes to the server
   a. The server will attempt to continue reading until you close the connection
7. Timeout after 10 seconds
8. Close connection
9. Stop background UDP Latency Tests
10. Post results to Server

Extend the amount of dns requests and latency tests we send to 100

Use Cases

Login

In order to run a test a user first logs on by:

1.) Pressing login button
   a.) The Main subsystem talks to the Communication subsystem which checks the server for its status
   b.) The Communication subsystem tells the Main subsystem if the login has passed
      i.) The Main subsystem handles any errors thrown
2.) The user is brought to the main screen

Run Test

To run a test:

1.) User presses “Run Tests” button
    a.) Main subsystem tells the Communication subsystem to start running the tests
    b.) The Communication subsystem sends POST requests to the server to set and return an ID
    c.) The mobile test is then sent to the server and a config is sent back
    d.) The router test starts and the server sends a status back
        i.) When the router is started the file main.cpp is run and forks two child processes, the monitor and the main
            (1) The monitor is set to store data to look at the ambient network usage data.
            (2) The main process is blocked and waits for the server which generates a router token and spawns a new process, Test Runner
            (3) The parent process, main.cpp, waits and isolates crashing from main and gets results from monitor
                (a) These results are the number of bytes sent and received from each network interface every 10 seconds. It keeps 1000 samples which is reasonable for the router and weeks’ worth of data.
            (4) The main fork then starts a test by kicking off multiple threads of the state machine
                (a) The state machine messages two threads, DNS Tester and Throughput Tester
                    (i) The state machine keeps track of what is currently running and if it times out it exits, gets cleaned up, and the test is a failure
                    (ii) The state machine is idle until a request is sent
                    (iii) It then sends the data and router token to the server which receives config at this time
                        1. If a 200 status is received the DNS Tester and the Throughput Tester sends requests
                    (iv) DNS Tester and Throughput Tester are asynchronous and the Test Runner exits and rejoins the main.cpp parent
                    (v) When done the state machine exits and goes back to idle
    e.) When a test is running, the Main subsystem starts test handle errors and moves into the animation
    f.) While the animation is running, the Communication subsystem tells the Tester subsystem to start running multiple test: run DNS, run latency, calculate packet loss, and run throughput and load results simultaneously
    g.) The Tester subsystem tells the Communication subsystem and the Main subsystem that the test is completed
    h.) The Communication subsystem reports the test to the server and the Main subsystem changes to the web view and requests the server for a results page which is returned back to the Main subsystem
2.) Test results are displayed when the test finishes. *Note: Android Upload Throughput is an overestimate*

Conclusion

Challenges

Setting up a system that uses multiple devices is difficult. A restful API had to be build for the server side so that this project could be added to later. Each of these projects had to communicate well with each other. JSON was used to make the connection between each service. Security also had to be taken into consideration. Authentication of the service was built into this project from the start.

Results

IOS and Android devices can install this application. A version of OpenWRT can now run a test in a home. Actual measurements are aggregated and displayed on the web service. A user or ISP can view these results from their device or web service.

Appendix

Glossary of Terms

CLIPPER	Name of the project at CableLabs. The Hermes project is a subset of the Clipper project that deals with home network monitoring as a proof of concept.
Hermes	The phone and tablet applications that start the tests and show the data to the consumer of a service. The device will also run a test that is compared to the router tests.
Mercury	The router side code that communicates with the testing service to run tests on the in home network.
Zeus	The web application and database that keeps track of user accounts and manages and stores results.
ISP	Internet Service Provider
MSO	Multiple System Operator