Yet Another Network Analyser

Yavar Naddaf . Iulian Radu . Matthew Sniatynski

CPSC 417 Project
June 16, 2005

Prof: Alan Wagner

Table of Contents

1. Introduction

- 1.1 Overview
- 1.2 System Requirements
- 1.3 Usage

2. Python Component

- 2.1 Overview Diagram
- 2.2 GraphBuilder
  - 2.2.1 Module YaNA_buildGraph
  - 2.2.2 Module YaNA_netGraph and Graph Format
- 2.3 Link Analysis
  - 2.3.1 Bandwidth
  - 2.3.2 Latency
- 2.4 Dot Conversion
- 2.5 NS Generation

3. Web Component

1. Introduction

1.1 Overview

YaNA (Yet another Network Analyzer) is a web-enabled network discovery and analysis tool capable of mapping and analyzing the routes on which data packets flow from a source machine to a multiple network-connected destination host machines.

Target ip addresses are provided as input via the web interface, and the program generates a graph of the network path as seen by the source machine. Output is generated in PNG and NS script file format, allowing further analysis and simulation using the Network Simulator / Animator utilities.

The utility is composed of two main components. At the core lies the **YaNA Python script**, responsible for building routes to the destination hosts, analysis of the routes, and generation of PNG and NS output files representing the network topology. To accomplish this, the component makes use of the *traceroute* and *graphvis* tools. The second portion of the utility is the **YaNA HTTP server**, responsible for collecting IP addresses, periodically running and managing the YaNA python script. Both these components are detailed in this document.

1.2 System Requirements

Python 2.3.3+ (http://www.python.org)
Graphviz 2.2.1+ (http://www.graphviz.org/download_source.php)
Standard C libraries: socket, pthread
Console utility: traceroute
Network Simulator & Animator (http://www.isi.edu/nsnam/ns/) (optional)

1.3 Usage

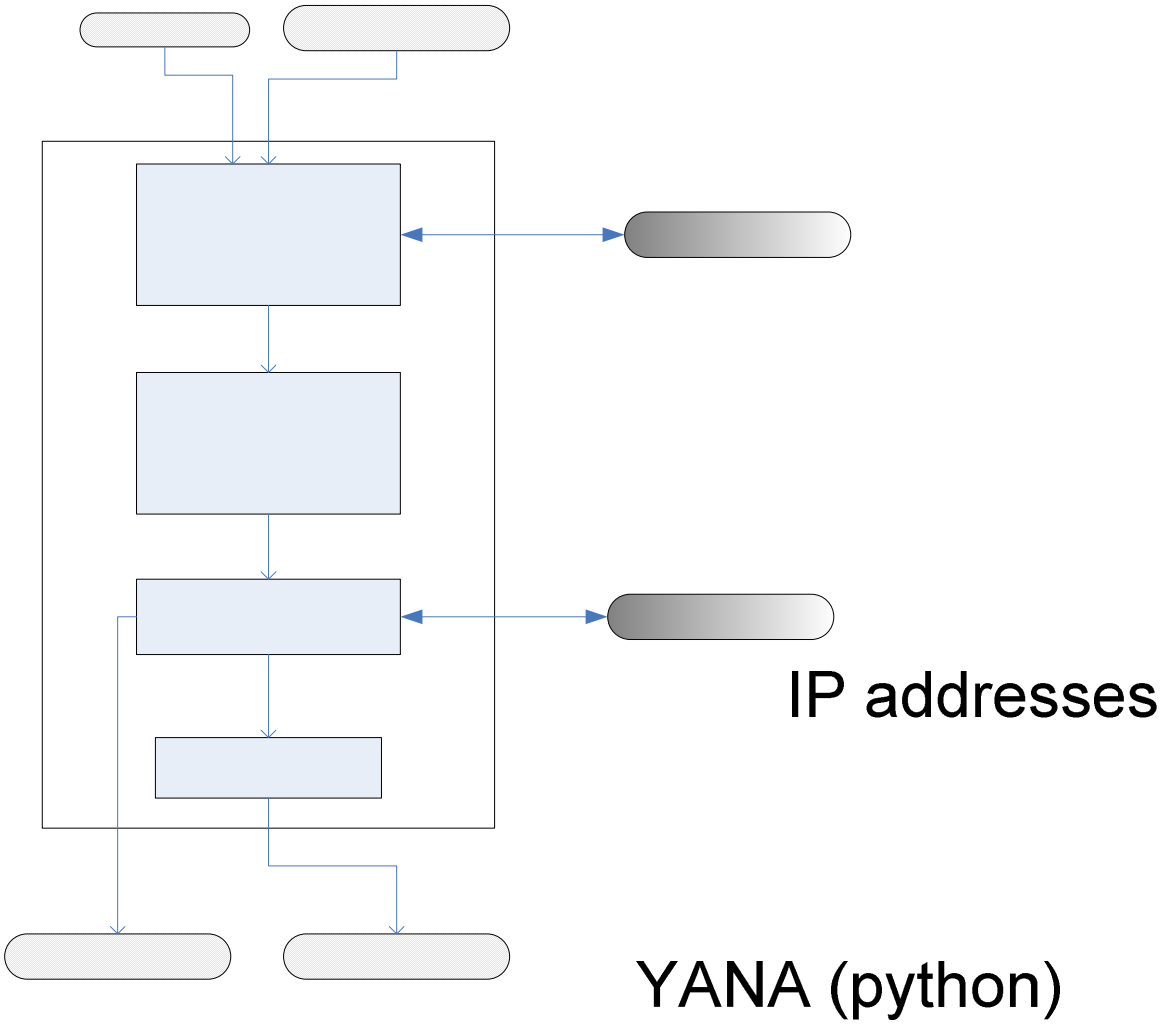
The following assumes that YaNA files have been unpacked into a directory, HTTP server compiled, and Python and Graphvis paths setup.

Start the HTTP server by executing “httpsrv”. The server port will be output, and the YaNA status and configuration page will be accessible on the host machine at URL path “/yana/”. The server should run on a remote machine, not locally browsable.

The server will periodically call python to execute the “YaNA.py” script, updating the network graph with newly acquired client and user specified IP addresses, and providing new PNG/NS files. The Statistics page provides various execution statistics and listing of server processed IP addresses, while the Configuration page allows the user to add target IP addresses, reset the network graph and change periodic update.

2. Python Components

2.1 Overview Diagram



2.2. Graph Builder

This component repeatedly calls the “traceroute” utility to find routes and packet round-trip times to the selected target IPs. Each host is traced with multiple small and large packets, and the resulting network graph is stored in a dictionary structure. Extensive parsing of the traceroute output is done to ensure a maximum amount of path information is extracted from the variable output lines. Program structure is detailed in the following sections.

2.2.1 Module YaNA_buildgraph.py

Four key elements in this module are the four regular expression (regexp) objects used to parse information from the output of traceroute. These are named wellFormed, wellFormedUnknown, latTimes, and ipOnly. We will describe their functionality briefly one by one.

wellFormed: This regexp matches a properly formatted line of traceroute output, of the following format…
<number> <verbose-address> (<ip-address>) <float-time> ms <float-time> ms <float-time> ms

wellFormedUnknown: This regexp matches a properly formatted line of traceroute output when no information at all is obtained from the host/router corresponding to that line. It looks like… <number> * * *

latTimes: This regexp matches anything resembling a millisecond latency time. Calling it like this "latTimes.findall('input string') will return a list of all of the millisecond latency times found in the input string.

ipOnly: This regexp matches anything resembling a numerical ip-address of the standard dotted-quad format (dotted decimal notation). Calling ipOnly.findall('input string')[0] will return the first occurrence of an ip-looking object in the input string.

One large function does the majority of the work in this module – traceRT. This function is called with three input arguments: a YaNA_netgraph instance, a destination host ip-address, and a single character denoting the packet size to traceroute with. The netgraph instance can be empty (a new netgraph object) or can already have stored information. In either case, new information discovered by the iteration of traceRT will be added to the netgraph. The basic idea of the traceRT function is quite simple – call the standard unix 'traceroute' program using the destination host as the argument (along with a few options defined in the YaNA_config.py file), parse in the output, and add the parsed output to the netgraph. The output will consist of numerous lines, with each line detailing the address of a router somewhere along the path from the source host (where this program is being run) to the destination host, and the latency (ms) delay between the source host and this intermediate router.

The regular expressions detailed above are used to extract the ip-addresses and latency times from these lines of traceroute output. Combinations of these regular expressions are used to robustly extract this information from lines that may not resemble perfect lines of traceroute output (since traceroute is inherently 'hacky', sometimes weirdness appears in the output). In the worst case, as long as an ip-like thing can be found, and at least one millisecond delay time, the line will be used, and that intermediate router's information will be added to the netgraph. Otherwise that line will be skipped. When everything has worked well, the netgraph will contain nodes which represent the source host, the destination host, and each intermediate router, and edges connecting these nodes which represent links between these hosts and routers. The edges have two properties associated – the small packet and big packet round trip times which are calculated from the millisecond delay times in the traceroute output. These are described in more detail in the documentation for YaNA_netgraph.

The traceRT function returns a numerical code == 0 for success, and != 0 if some type of failure has occurred. Two situations cause a value other than 0 to be returned – the first is if traceroute returned no output at all. The second is if more than a certain number of the lines of traceroute output (number can be changed in the YaNA_config.py file) were matched by the 'wellFormedUnknown' regexp. This can indicate various types of messy situations, and it may be desirable to include these traceroute runs in the netgraph.

2.2.2 Module YaNA_netgraph and Graph Format

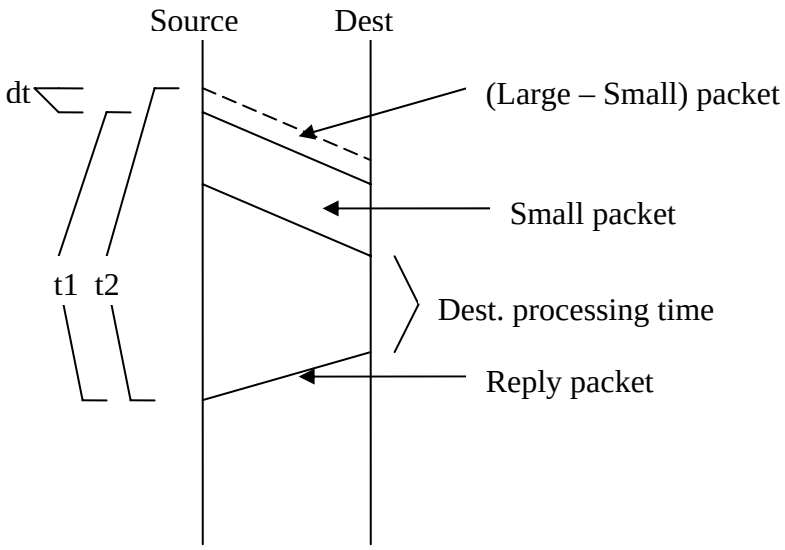
The YaNA netgraph module encapsulates a python dictionary-based directed graph structure, and provides some convenience functions for manipulating this graph. The two main function are addNode and addEdge. AddNode is called with the ip-address of the node to be added, this ip-address is then added to the graph (as a node – surprise!). AddEdge takes four arguments: the source IP, destination IP, the small packet return time, and the big packet return time. These return times must be calculated from the millisecond latency times returned by traceroute. If spRT or bpRT are None, no time information will be added to the edge. This edge information is required for the calculation of the link bandwidth and latency. The use of nested python dictionaries provide an unparalleled level of flexibility, as the netgraph data structure can be easily expanded in the future – more link and intermediate router information can be added – and no code needs to be changed.

2.3. Link Analysis

After the graph building component terminates, each link contains an average for round-trip time of small and big packets, as reported by the traceroute utility. The analysis component is responsible for using these times to determine the bandwidth and latency of each link.

2.3.1 Bandwidth

Bandwidths are calculated using the time difference between sending big and small packets on a link.



At the base of calculation lies the formula bandwidth = datasize / transfertime

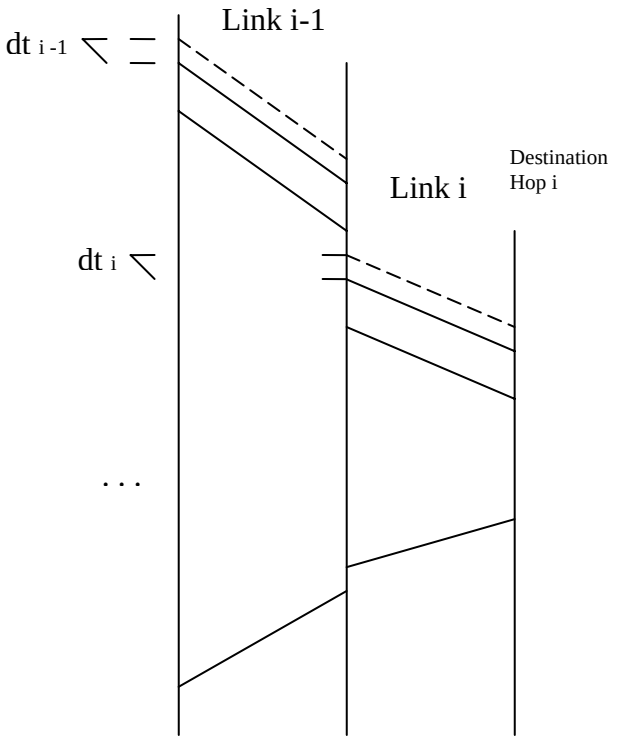
As can be seen from the traffic flow diagram, the bandwidth for one hop is:

$$\text{Bandwidth} = \frac{(\text{big\_packet\_size} - \text{small\_packet\_size})}{(\text{big\_packet\_reply\_time} - \text{small\_packet\_reply\_time})}$$

For cumulative hops, the difference in reply times to a specific destination hop i is:

$$dTotal = dt1 + dt2 + \ldots + dt_i + dt_{i-1}$$

Thus Bandwidth of hop i = $\frac{(\text{big\_packet\_size} - \text{small\_packet\_size})}{(\text{big\_packet\_reply\_time} - \text{small\_packet\_reply\_time}) - \text{sum(transfer times of the packet difference over previous links)}}$



Because of this dependency on previous links, the link bandwidths are calculated in a depth-first on the network graph.

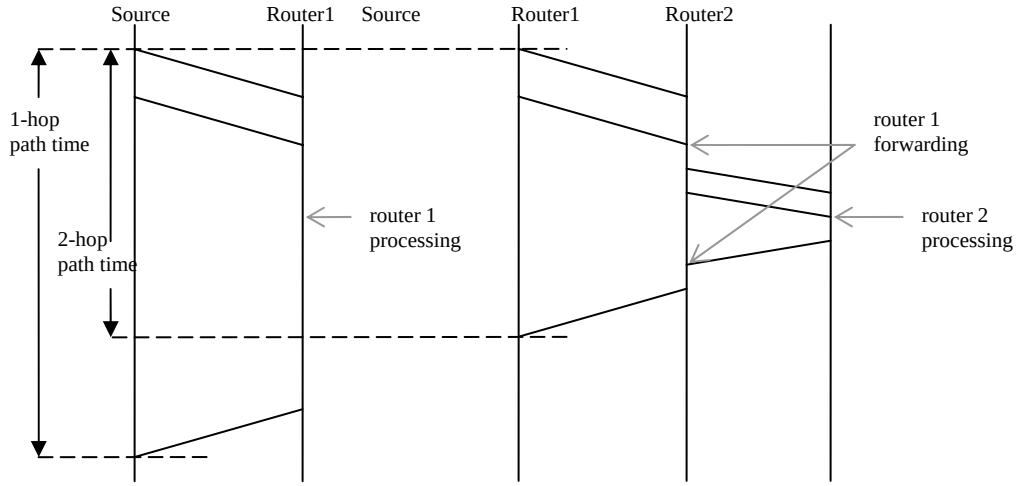
2.3.2 Latency

Latencies follow a very simple calculation, largely due to a limitation encountered in practice.

Limitation:

It was assumed that routers have a processing delay which is identical for forwarding and replying to packets. Thus, knowing the trip time and size of the packet sent and received, we could determine a latency time for the link which includes a router processing delay. However, in practice routers forward packets in a much faster (forwarding) time than replying (processing).

Because of different such processing delays between routers, we could get to a situation where the reply time for a 1-hop trip is *slower* than that of a 2-hop trip, as depicted:



Simplified calculation:

For our model, we simply assume that all hops have equal latencies to a host, and attempt to use the transit times calculated to keep a minimum latency on each link. We determine how much time elapses in transit for each path from source host to each leaf of the graph, calculate an average time for each link passed, and propagate this value upwards on from the leaves up to the source, storing a minimum latency on each link along the path.

2.4. Dot Conversion

Once bandwidths and latencies have been calculated, the graph is then converted to a dot file. The Graphviz utility is used to generate a visual layout of the graph. This format is necessary for visualization of the graph and for aiding production of the NS script.

A PNG image of the topology is generated at this stage, containing nodes identified by IP addresses, and appropriate links.

2.5. NS generation

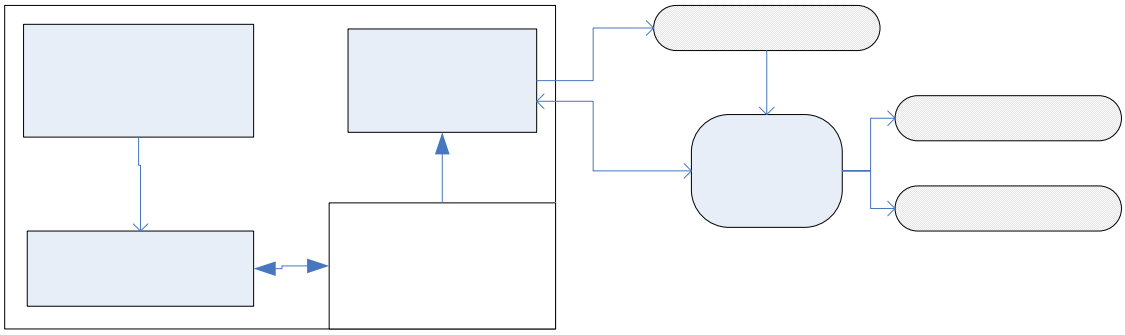
The dot output graph is parsed by this component, generating a properly laid out NS representation of the network. The NS file supplies the appropriate information for all nodes and links in the graph, storing link bandwidth, latency and queue size for use in NS and NAM analysis.

3. Web Component

This application is observed and controlled through a web interface supplied by a modified HTTP server.

The server is responsible for periodically executing the program and providing clients with YANA output and statistics. IP addresses are collected from served clients, or manually added through the web interface. The program output PNG and NS files are accessible, along with various execution statistics.

A regular HTTP server was modified to accomplish this:



The server code was extended to allow passing of arguments via URL. A set of YANA variables have also been added for tracking newly added IP addresses and various statistics; these are modified from the web interface, and read when executing the YANA script. Finally, the server contains a specialized "Pulse" thread which periodically executes the script if any IPs are to be added to the graph.