Proceedings of LISA '99: 13th Systems Administration Conference

Seattle, Washington, USA, November 7–12, 1999

GTRACE—A GRAPHICAL TRACEROUTE TOOL

Ram Periakaruppan and Evi Nemeth

GTrace – A Graphical Traceroute Tool

Ram Periakaruppan and Evi Nemeth – University of Colorado at Boulder and Cooperative Association for Internet Data Analysis

ABSTRACT

Traceroute [Jacobson88], originally written by Van Jacobson in 1988, has become a classic tool for determining the routes that packets take from a source host to a destination host. It does not provide any information regarding the physical location of each node along the route, which makes it difficult to effectively identify geographically circuitous unicast routing. Indeed, there are examples of paths between hosts just a few miles apart that cross the entire United States and back, phenomena not immediately evident from the textual output of traceroute. While such path information may not be of much interest to many end users, it can provide valuable insight to system administrators, network engineers, operators and analysts. We present a tool that depicts geographically the IP path information that traceroute provides, drawing the nodes on a world map according to their latitude/longitude coordinates.

Introduction

Today's Internet has evolved into a large and complex aggregation of network hardware scattered across the globe, with resources accessed transparently with respect to their location, be it in the next room or on another continent. As the Internet becomes increasingly commercialized among many different corporate administrative entities, it is more difficult to ascertain the geographical routes that packets actually travel across the network. Knowledge of these geographical paths can provide useful insight to system administrators, network engineers, operators and analysts.

It is challenging to obtain the location for a given node of a path since there is no existing database that accurately maps hostnames or IP addresses to physical locations. Although RFC 1876 [RFC1876] defined a DNS resource record to carry such location information (the LOC record) for hosts, networks and subnets, very few sites maintain LOC records. Hence there is no straightforward way to determine the physical location of hosts.

GTrace is a graphical front end to traceroute that uses a number of heuristics to determine the location of a node. Often the name of a node in the path contains geographical information such as a city name/abbreviation [Wood98] or airport code [Halsey98]. GTrace operates on the assumption that these codes and names indicate the physical location of the node. The locations obtained are connected together on a world map to show the geographical path that packets take from the source to destination host. GTrace also tries to verify the validity of each location obtained, eliminating ones that are incorrect.

The following sections review the traceroute tool and describe the design and implementation of GTrace. We also show example output from GTrace.

Traceroute

Traceroute is a tool that discovers the route an IP datagram takes through the Internet from a source host to a destination host. It works by exploiting the TTL (Time To Live) field of the IP Header. Each router that handles an IP datagram decrements the TTL field. When the TTL reaches zero, a router must discard the packet and send an error message to the originator of the datagram.

Traceroute uses this feature, initially sending a datagram with the TTL set to one. The first router along the path, upon receiving the datagram decrements the TTL, discards the datagram and sends back an ICMP error message. Traceroute records this first IP address (source address of the error message packet) and then sends the next datagram with the TTL set to two. This process continues until the datagram finally reaches the target host, or until the maximum TTL threshold is reached.

Design and Implementation of GTrace

Recognizing that it is not possible to obtain precise physical location information for all existing IP addresses, our main design criteria for GTrace was that it be sufficiently flexible to support the addition of new databases and heuristics. We chose to implement GTrace in Java, for both its portability and its new Swing [Swing] user interface toolkit. GTrace operates in two phases. In the first phase GTrace executes traceroute to the destination host and tries to determine locations for each node along the path. During the second phase, GTrace verifies whether the locations obtained in the previous phase are reasonably correct.

GTrace is composed of the following seven key components: Graphical User Interface, Dispatcher Thread, Hop Threads, Lookup Client, NetGeo Server, Lookup Server and Location Verifier. Figure 1 illustrates the overall architecture of the tool. The function of each component is described below.

Graphical User Interface

The Main Thread handles all features of the Graphical User Interface and is responsible for spawning the dispatcher thread when a destination host is specified. Figure 2 shows a snapshot of GTrace on startup. The GUI has two sections, with a map on the top and traditional traceroute output below. The tool supports zooming in or out of particular regions of the maps. Twenty-three maps are available courtesy of VisualRoute [VisualRoute] and users can also add their own. We later provide an example that highlights some of the features of the GUI.

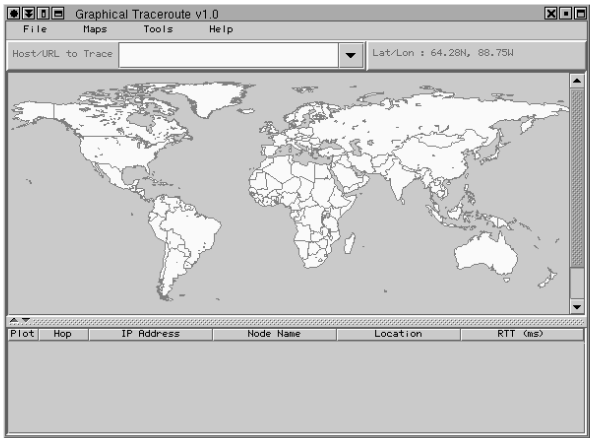


Figure 2: GTrace's startup screen.

Dispatcher Thread

The function of the dispatcher thread is to execute traceroute to the destination host. It then reads the output of traceroute, creating a new thread for each line of output. These threads are referred to as hop threads. The dispatcher thread can also read traceroute output from a file, which allows users to visualize traceroutes performed using third-party traceroute servers.

Hop Threads

Each hop thread parses its line of traceroute output and immediately notifies the main thread so that it can update the display with relevant traceroute fields for the corresponding hop. It then creates an instance of the Lookup Client, which tries to determine the location of the node and return the resulting information to the main thread before exiting.

Lookup Client

The Lookup Client tries to determine the location of a node by using a set of search heuristics. Many of the nodes in a typical traceroute path are in the '.net' domain. Often the names of these nodes have some

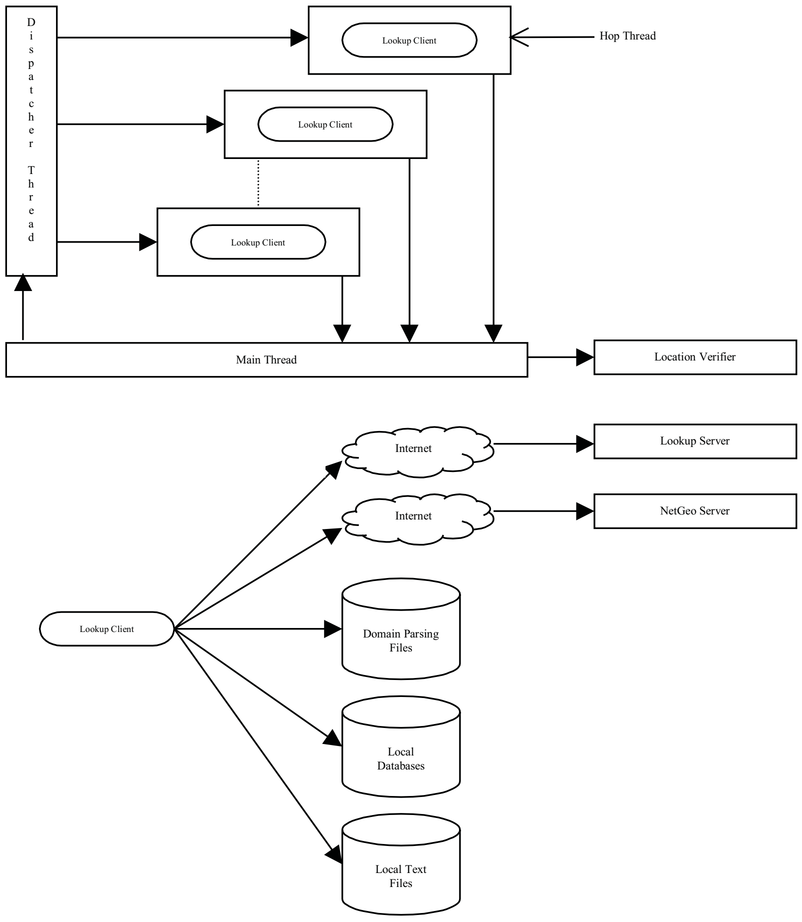


Figure 1: GTrace architecture.

geographical hint in them. The Lookup Client uses customized domain parsing files that specify rules for extracting these geographic hints. We have such files for several '.net' domains that use internally consistent naming conventions within their domain.

However this technique does not solve the problem of locating nodes that do not have embedded geographical hints. GTrace also utilizes databases from CAIDA [DBCAIDA] and NDG Software [DBNDG] that map hostnames and IP addresses to latitude/longitude coordinates. For nodes with no information in these databases, the Lookup Client uses the domain's registered address (unfortunately often only the headquarters for a geographically distributed infrastructure) obtained through a whois lookup to determine the location. Nodes for which the Lookup Client is unable to determine a location are listed in the text portion, but skipped in the geographical display.

The search algorithm is described below. We try each heuristic in turn, stopping as soon as one yields a location. The Lookup Client also makes a note of the search step that produced the location, providing this information to the user as well as the Location Verifier.

Search Algorithm

1. Check the cache to see if the location for the IP address has already been determined from a previous trace.
2. Check if the host has a DNS LOC record. If not, reduce the hostname to the next higher level domain (i.e., remove the first component of the name) and check again for a LOC record. Continue until we have reached the last meaningful component of the name (for example foo.com in xxx.foo.com or bar.com.au in xxx.yyy.bar.com.au). Note that if a site has a LOC record for the whole domain, but machines are located outside the scope of that LOC record, GTrace would end up using incorrect data. If the Location Verifier detects such a situation, GTrace will notify the user and optionally can be configured to notify GTrace's author, who will contact the DNS administrator at the corresponding site to correct their LOC records.
3. Search for a complete match of the hostname/IP address in the databases and files specified in the GTrace configuration file.
4. If the hostname has a corresponding domain parsing file, use the rules defined in the file to extract geographical hints and proceed as indicated in the file.
5. Reduce the hostname to the next higher level domain as in step two and search for a match as in step three. The process is repeated until we have reached the last meaningful component of the name.
6. Query the NetGeo [NetGeo] server with the IP address. NetGeo determines the location based on whois registrant information.
7. If still no match occurs and the last two letters of the hostname end in a two-letter country code, map it to the geographic center of that country.

The search algorithm is ordered in decreasing level of location reliability. Locations obtained from steps 2 and 3 are taken as authoritative, while those from step 4 onward are considered a guess. Cache entries will indicate whether the location was authoritatively determined or was a guess; this status determines the color of the lines connecting the nodes on the map.

The Lookup Client does not determine locations for IP addresses that fall in the ranges 10.0.0.0-10.255.255.255, 172.16.0.0-172.31.255.255 or 192.168.0.0-192.168.255.255, as these blocks are reserved for private internet use [RFC 1918]. Unfortunately some addresses in these blocks do occur in traces since some ISPs use this address space for internal router interfaces. These nodes are shown in the text portion of the display with the location marked as private internet use.

The Lookup Client queries the Lookup Server if one is defined in the GTrace configuration file and if location information has not been obtained through step 1, 2 or 3 of the search algorithm. GTrace compares the reply from the Lookup Server with any obtained previously from local lookups, with preference given to the location obtained through a lower numbered search step. Based on the GTrace configuration file, the Lookup Client also uses databases, text files and domain parsing files as follows.

Databases

The Lookup Client may need to perform lookups in many databases before determining a location. GTrace's database support is provided by the BerkeleyDB [BerkeleyDB] embedded database system, which supports a Java API that the Lookup Client uses to query the databases. The database interface allows multiple thread reads on the same database at the same time. Locking is not an issue, since Lookup Clients only read, do not write.

Five databases are packaged with the GTrace distribution:

Machine.db [DBCAIDA]	Maps machine names to their latitude/longitude values.
Organization.db [DBCAIDA]	Maps organizations to their latitude/longitude values.
Hosts.db [DBNDG]	Maps IP addresses to their latitude/longitude values.
Cities.db [DBCAIDA]	Maps cities around the world to their latitude/longitude values.
Airport.db [AirportCodes]	Maps airport codes to their latitude/longitude values.

One can add a new database in BerkeleyDB format to GTrace with GTraceCreateDB and by adding an entry to the GTrace configuration file. The contents of the database, i.e., whether it maps hostnames, IP addresses, or both to latitude/longitude values, also have to be indicated in the configuration file. The user can also add records to existing databases using GTraceAddRec. GTraceCreateDB and GTraceAddRec are Java classes packaged with the GTrace distribution.

Text Files

Users may also specify new locations for nodes in text files, though it is more efficient to create a database for large data sets. New files have to be listed in the GTrace configuration file in order for the search algorithm to have access to them.

Domain Parsing Files

Files describing properties of each domain are used to ferret out geographical hints embedded in hostnames. These files define parsing rules using Perl5 compatible regular expressions. GTrace uses the regular expression library from ORO Inc. [ORO-Matcher] for parsing. New files can be added and existing ones modified without requiring any changes to GTrace.

For example, ALTER.NET (a domain name used by UUNET, a part of MCI/WorldCom) names some of their router interfaces with three letter airport codes as shown below:

```
193.ATM8-0-0.GW2.EWR1.ALTER.NET
(EWR -> Newark, NJ)
190.ATM8-0-0.GW3.BOS1.ALTER.NET
(BOS -> Boston, MA)
198.ATM6-0.XR2.SCL1.ALTER.NET
(Exception)
199.ATM6-0.XR1.ATL1.ALTER.NET
(ATL -> Atlanta, GA)
```

Figure 3 shows an example of a GTrace domain parsing file that would work for ALTER.NET hosts. The file first defines the regular expressions, followed by any domain specific exceptions. The exceptions are strings that match the result of the regular expressions. The user may identify the exception's location either by city or by latitude/longitude value using the format shown below:

```
exception=city,state,country
          city,country
          L: latitude, longitude
```

In the former case, the user should also use GTraceQueryDB to ensure that the cities database has a latitude/longitude entry for the city specified. The first line in Figure 3 defines a substitution operation, which when matched against 193.ATM8-0-0.GW2.EWR1.ALTER.NET, would return 'EWR'. The contents following the last '/' of the first line indicate what to do with a successful match, namely in this case to instruct the program to first check for a match in the data specified in the current file and then for a match in the airport database.

The reason for checking the domain parsing file first is that sometimes the naming scheme for a given domain is not consistent. For example, a search for SCL obtained from 198.ATM6-0.XR2.SCL1.ALTER.NET in the airport database would return a location for Santiago de Chile. In the case of ALTER.NET, they also use three letter codes that are not airport codes but abbreviations for US cities (Figure 3 illustrates three such abbreviations). Note that if this exception list were not present and SCL did get mapped to Chile, the Location Verifier would likely have eliminated it using the Round Trip Time (RTT) heuristic described later, which would have recognized the RTT as much too small to get a packet to Chile and back.

Sometimes ISPs name their hosts with more than one geographical hint in them. For example VERIO.NET names some of their hosts in the following format: den0.sjc0.verio.net, which typically suggests source and destination of the interface. If there is no rule on whether the convention is to use the source or destination label first in the hostname, the rule could be defined to extract both and GTrace could use the Location Verifier's heuristics to guess.

The advantage of this technique is that one can describe an entire domain as a set of rules without needing database entries for every host in the domain. The limitation of the technique is that it will fail for domains that do not use internally consistent naming schemes.

NetGeo Server

The original design of the Lookup Client performed and parsed results of whois lookups directly, which required storage of a prohibitively large number of mappings of world locations to latitude/longitude values. Distributing such a large database with GTrace was not ideal. CAIDA's NetGeo [NetGeo] tool, with its ability to determine geographical locations based on the data available in whois records, provided a vital resource.

```
s/.*?\.([^\.]+)\d\.ALTER\.NET/$1/this,airport.db
scl=santaclara, ca, us
tco=tysonscorner, va, us
nol=neworleans, la, us
```

Figure 3: Example of a domain parsing file for ALTER.NET.

NetGeo is a database and collection of Perl scripts used to map IP addresses to geographical locations. Given an IP address, NetGeo will first search its own local database. If a record for the target address is found in the database, NetGeo will return the requested location information, e.g., latitude and longitude. If NetGeo finds no matching record in its database, it will perform one or more whois lookups until it finds a whois record for the appropriate network. The NetGeo Perl scripts will then parse the whois record and extract location information, which NetGeo both returns to the client and stores in its local database for future use.

The NetGeo database contains tables for mapping world location names (city, state/province/district, country) or US zip codes to latitude/longitude values. Most whois records provide enough address information for NetGeo to be able to associate some latitude/longitude value with the IP address. Occasionally the whois record only suggests a country or state, in which case NetGeo returns a generic latitude/longitude for that country or state. In preliminary testing, NetGeo has been able to parse addresses and find (albeit sometimes imprecise) latitude/longitude information for 89% of 17,000 RIPE whois records, 76% of 700 APNIC whois records and for more than 95% of 30,000 ARIN whois records.

Lookup Server

The Lookup Server handles requests from Lookup Clients and tries to determine the location of a host or IP address by executing steps three, four and five of the search algorithm. This information is sent back to the client, which then decides whether to use the location information or not depending on the locations it might have received from other Lookup Servers or lookups it performed locally. The Lookup Client selects the location that was obtained from the lowest numbered search step. The Lookup Server can also be requested by the Lookup Client to execute step two of the search algorithm. This is because not all versions of nslookup support queries for LOC records. GTrace tests the version of nslookup on the machine it is running on to determine if such a request is necessary.

Location Verifier

The Main Thread invokes the Location Verifier once all the hop threads have died and the trace is complete. The task of the Location Verifier is to check whether the locations obtained for nodes along the path are reasonable. The verifier does not determine new locations for nodes, it only indicates to the user why an existing location might be wrong and where the node could possibly be located.

The verifier algorithm is based on the fact that IP packets can not travel faster than the speed of light. Light travels across different mediums at different speeds: 3.0 x 10^8 m/s in vacuum, 2.3 x 10^8 m/s in copper and 2.0 x 10^8 m/s in fiber [Peterson]. GTrace uses the speed of light in copper for all of its calculations.

For each successive pair of hops that have locations, the verifier algorithm uses the deltas of the round-trip times (RTT) returned by traceroute to rule out locations that are physically not possible. Traceroute measures RTT rather than one way latency, as this would require control over both end nodes and delays are often not symmetric. Also, one must be cautious

Hop	Node Name	IP Address	Search Step	Location	RTT (ms)
1	pinot-fe2-0-0	(192.172.226.65)	6	San Diego	0.917 ms
2	medusa.sdsc.edu	(198.17.46.10)	3	San Diego	0.881 ms
3	sdsc-gw.san-bb1.cerf.net	(192.12.207.9)	4	San Diego	1.944 ms
4	pos0-0-155M.san-bb6.cerf.net	(134.24.29.130)	4	San Diego	4.640 ms
5	atm6-0-1-622M.lax-bb4.cerf.net	(134.24.29.142)	4	Los Angeles	9.598 ms
6	pos6-0-622M.sfo-bb3.cerf.net	(134.24.29.233)	4	San Francisco	415.317 ms
7	pos10-0-0-155M.sfo-bb1.cerf.net	(134.24.32.86)	4	San Francisco	16.813 ms
8	192.205.31.29	(192.205.31.29)	6	New Jersey	16.917 ms
9	att-gw.sf.cw.net	(192.205.31.78)	4	San Francisco	81.281 ms
10	corerouter2.SanFrancisco.cw.net	(204.70.9.132)	4	San Francisco	81.254 ms
11	core1.Washington.cw.net	(204.70.4.129)	3	Washington	89.727 ms
12	mix1-fddi-0.Washington.cw.net	(204.70.2.14)	4	Washington	89.708 ms
13	**vsnlpoone.Washington.cw.net**	**(204.189.152.134)**	**4**	**Poone**	**706.301 ms**
14	202.54.6.17	(202.54.6.17)	6	Madras	697.946 ms
15	202.54.6.254	(202.54.6.254)	6	Madras	702.893 ms
16	giasmda.vsnl.net.in	(202.54.6.161)	4	Madras	704.856 ms

Figure 4: A sample traceroute output produced by the first phase of GTrace.

with the RTT values since they incorporate several components of delay. The RTT between two nodes has four components: the speed-of-light propagation delay, the amount of time it takes to transmit the unit of data, queuing delays inside the network and the processing time at the destination node to generate the ICMP time exceeded message. Traceroute typically sends 40-byte UDP datagrams, so it is safe to assume negligible transmit time. Ideally, for the verifier algorithm one would like the RTT to represent only the propagation delay, but this is not the case due to variable queuing and processing delays, hence it is not possible to set the upper bound on the RTT to a hop. Accordingly the verifier algorithm uses the minimum RTT returned by traceroute, as this would represent the best approximation of the propagation delay. Things are further complicated by the fact that the RTT delta between hops k and k+1 can be biased because the return path the ICMP packet takes from hop k can be totally different from the return path it takes from hop k+1. The Location Verifier tries to re-determine RTT values for hops it thinks are biased using ping.

By default, traceroute sends three datagrams each time it increments the TTL to search for the next hop. Changing the value of the q parameter in the GTrace configuration file will modify this behavior. The larger the value of q, the more accurate the estimate of the propagation delay, but large values of q also slow down GTrace as traceroute has to send q packets for each hop.

Knowing the geographical distance between two nodes, GTrace can calculate the time-of-flight RTT (the propagation delay at the speed-of-light in copper), compare it against traceroute's value and flag a problem if the RTT is smaller than physically possible. In such a case either the location of the source or of the destination or both is incorrect. The details of the verification algorithm are as shown in the next section.

Verifier Algorithm

1. Ideally, the RTT to hop k in a path should always be less than the RTT to hop k+1 or k+2, but this is not always true due to queuing delays, asymmetric paths and other delays. We allow a 1ms fudge factor to cover such discrepancies. Thus the RTTs between hops k and k+1 should be such that $\mathrm{RTT}(k) \leq \mathrm{RTT}(k+1) + 1$ ms. If this condition does not hold true then the RTT to each of the out-of-order hops preceding hop k is estimated again with ping, i.e., till the first hop j preceding k such that $\mathrm{RTT}(j) \leq \mathrm{RTT}(k+1) + 1$ ms. If the RTT estimates obtained using ping still do not satisfy the condition $\mathrm{RTT}(k) \leq \mathrm{RTT}(k+1) + 1$ms, then hop k is not used in the later stages of the verifier algorithm.
2. Cluster the traceroute path into regions having similar RTT values. This is based on the assumption that nodes with similar RTTs will tend to be in the same geographic region.
3. For each region identified in the previous step, calculate the time-of-flight RTT for pairs of hops that have locations. If the RTT delta reported by traceroute for that pair of hops is smaller than the time-of-flight RTT, flag the pair of hops so that it is corrected in step 5.
4. Repeat step three for hops falling on the edges of adjacent regions.
5. Try to 'correct' unreasonable location values that were identified in steps three and four using the reliability of the search step that produced the location match. Adjacent nodes between regions are corrected first because they represent larger and probably more inaccurate locations. Correcting the nodes identified in step three follows this. By correct, we mean trying different alternatives for the incorrect location based on the cluster in which it falls, flagging it to the user and not plotting it in the display.

Example

Consider the trace shown Figure 4, where locations are expressed as city names for ease of illustration. The Search Step column indicates which step of the search algorithm produced the location for that hop. Step one of the verifier algorithm would mark hop 13 as unusable since its RTT is greater than its subsequent hops. In this case it is probably due to the return path from hop 13 being longer than that from hop 14. Next, step two of the algorithm would cluster the traceroute path into the following regions: 1-4, 5, 6-8, 9-10, 11-12 and 14-16. Step three would flag that there is a problem between hops 7 and 8 since it is not possible for a packet to travel from San Francisco to New Jersey in less than a millisecond. Likewise, step four would flag a problem between hops 10 and 11. Step five would first try to correct hops 10 and 11 since they fall in different regions. Seeing that the location for hop 11 was obtained through step three of the search algorithm and hop 10 was from a higher step, the Location Verifier would change hop 10's location to that of hop 11's, in this example to Washington and rerun the algorithm from step 3. This process is repeated until all locations from one hop to the next are physically realistic. In the end the Location Verifier would have indicated to the user that hop 8 is incorrect and is most probably located somewhere near San Francisco. Hops 9 and 10 are also incorrect and may be in Washington with their interfaces labeled San Francisco to identify the other end of that link.

Configuration Files

The configuration options in GTrace are quite flexible. How it functions and executes the search algorithm depends on the contents of two configuration files: GTrace.conf and GTraceMaps.conf.

GTrace.conf

GTrace.conf specifies the location of the commands GTrace uses and lists databases, text files, Lookup Servers if any, to use in the search algorithm. Figure 5 shows an example configuration file. This file is automatically generated by the configure scripts while installing GTrace.

Green	Both endpoints are authoritative locations.
Yellow	One endpoint is authoritative and the other is a guess whose location is not a country center, state center or obtained from a whois record.
Blue	Both endpoints are guesses and the locations of both the endpoints are not a country center, state center or obtained from a whois record.
Red	One endpoint is a location that is a country center, state center or obtained from a whois record.

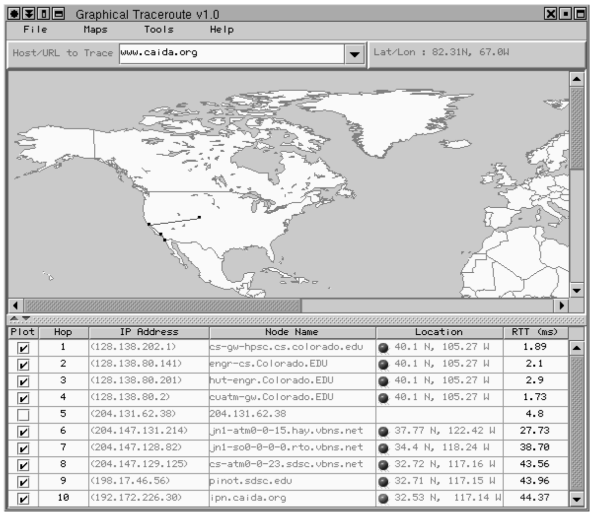
Table 1: Reliability representation colors.

GTraceMaps.conf

The GTraceMaps.conf configuration file specifies attributes of the maps that GTrace uses in displays. Users can add their own maps as part of or independent from the existing world hierarchy. Independent maps allow users to describe their own intranet topology and then use GTrace as a graphical debugging tool within their network.

GTrace Features

Figure 6 shows an example of a trace that was executed from University of Colorado, Boulder to CAIDA in San Diego. On the display, the colors of the lines on the map indicate the reliability of the location obtained for the endpoints. The colors are decided based on the criteria in Table 1.



Plot	Hop	IP Address	Node Name	Location	RTT (ms)
✓	1	(128.138.202.1)	cs-gw-hpsc.cs.colorado.edu	● 40.1 N, 105.27 W	1.89
✓	2	(128.138.80.141)	engr-cs.Colorado.EDU	● 40.1 N, 105.27 W	2.1
✓	3	(128.138.80.201)	hut-engr.Colorado.EDU	● 40.1 N, 105.27 W	2.9
✓	4	(128.138.80.2)	cuatm-gw.Colorado.EDU	● 40.1 N, 105.27 W	1.73
☐	5	(204.131.62.38)	204.131.62.38		4.8
✓	6	(204.147.131.214)	jn1-atm0-0-15.hay.vbns.net	● 37.77 N, 122.42 W	27.73
✓	7	(204.147.128.82)	jn1-so0-0-0-0.rto.vbns.net	● 34.4 N, 118.24 W	38.70
✓	8	(204.147.129.125)	cs-atm0-0-23.sdsc.vbns.net	● 32.72 N, 117.16 W	43.56
✓	9	(198.17.46.56)	pinot.sdsc.edu	● 32.71 N, 117.15 W	43.96
✓	10	(192.172.226.30)	ipn.caida.org	● 32.53 N, 117.14 W	44.37

Figure 6: Example of a trace produced by GTrace.

The table in the lower section of the display consists of six columns. The first column provides the user with a checkbox that is enabled for each location plotted on the map. The user can disable a checkbox and the corresponding location will be skipped.

```
#GTrace configuration file

#Paths
TRACEROUTE=/usr/sbin/traceroute -q 3
WHOIS=/usr/bin/whois
PING= /usr/sbin/ping
NSLOOKUP=/usr/sbin/nslookup
DOMAINFILES=/home/ram/gtrace/data
DATABASES=/home/ram/gtrace/db

#Names of databases and text files to be used
#for location lookups. Order is important, list
#them in the order they should be searched.
CITIES=cities.db
AIRPORTS=airport.db

HOSTSLOC=Machine.db,hostnames/ipaddr;
         Hosts.db,ipaddr;
         Organization.db,hostnames/ipaddr;

TEXTFILES=England.txt,hostnames/ipaddr;

#Location of Lookup Servers if any
LOOKUPSRVS=
```

Figure 5: Sample GTrace.conf file.

Locations that are flagged as unreasonable by the Location Verifier are not plotted by default.

The second, third and fourth columns display the hop number, IP address and host name respectively. Clicking on columns three and four will bring up whois information for the node.

Column five provides the latitudes and longitudes obtained for each hop. Clicking on this column will provide an explanation of how the location was determined and whether the Location Verifier detected any problems. A small colored ball in front of the latitude and longitude value indicates which search step produced the location. The colors and the search step they represent are given below:

Green	Step 2 LOC record.
Yellow	Step 3 Complete match
Blue	Step 4 Domain parsing file
Cyan	Step 5 Hostname reduction match
Red	Step 6 whois record
Gray	Step 7 Country code

The last column shows the smallest of the round trip times returned by traceroute. The color of the value indicates how many packets timed out: black implies that no packets timed out, blue implies that one packet timed out, and a value in red indicates that two or more packets timed out.

Using GTrace in the Local Environment

System Administrators often use traceroute as a debugging tool to identify problems in their network. GTrace provides a visual representation that can facilitate understanding and debugging of their network. It can be used to discover routing loops as well as for deciding routes. For example in a large campus if a path from host A to host B (located in the same building) goes across campus and back, the routing could be fixed to avoid such inefficient paths. GTrace can also be useful from an end user perspective. Students can use the tool to work out the topology of their campus network.

Conclusion

GTrace is a handy tool for identifying network topology and routing problems as well as gaining more macroscopic insight into the Internet infrastructure. While GTrace uses several heuristics to determine locations and its approach does not guarantee accuracy, it is robust and extensible. New databases, new Lookup Servers and learned insights into ISP's naming conventions can easily be added to GTrace. We hope that users and system administrators will find GTrace useful and contribute their own domain parsing files, or even run their own Lookup Servers for community use.

The practical success of GTrace lies in the rules defined for the '.net' domains, since these comprise the majority of hops in many traceroutes. Looking up a '.net' name in the whois database is only useful for small localized ISPs. Relying on whois heuristics would result in backbone providers' '.net' nodes to all uselessly map to a single corporate headquarters for that provider.

The accuracy of this tool would be much improved if the Internet community maintained LOC records in the DNS. Unfortunately since LOC records are optional, non-trivial in effort to support and without any clear payoff to ISPs, pervasive use of them will probably never occur and geographic visualization of arbitrary Internet infrastructure will continue to require heuristics to determine physical location of nodes.

Acknowledgments

We would like to thank kc claffy at CAIDA for suggesting the idea to develop this tool. We would also like to mention a special word of thanks to the following people and institutions: VisualRoute for permission to use their maps and labels, Sleepycat Software for the BerkeleyDB Package, Jim Donohoe for developing NetGeo and to the entire research team at CAIDA who helped with many aspects during the development of GTrace.

Several students (Colorado: Robert Cooksey, Brent Halsey, Jamey Wood, Jeremy Bargen and UCSD: Jim Anderson) wrote graphical traceroute tools as class projects in Evi Nemeth's Network System's class. Many good ideas from these students' projects were incorporated into GTrace.

Availability and Support

GTrace-1.0 is the current release and it can be downloaded from the GTrace home page at http://www.caida.org/Tools/GTrace. The source code comes with the GTrace distribution. Further information on using the tool or how you can contribute domain parsing files can be found on the GTrace home page.

Author Information

Ram Periakaruppan is pursuing his Master's degree in Computer Science at the University of Colorado, Boulder. He can be reached at <ramanath@cs.colorado.edu>.

Evi Nemeth has been a computer science faculty member at the University of Colorado for years. Currently she is on leave doing the IEC (Internet Engineering Curriculum) project at CAIDA (Cooperative Association for Internet Data Analysis) on the UCSD campus and working furiously to make the publisher's deadline for the third edition of the UNIX System Administration Handbook. She can be reached at <evi@cs.colorado.edu>.

References

[AirportCodes] Listing of Airport Codes, http://www.mapping.com/airportcodes.html .

[BerkeleyDB] BerkeleyDB Package Distribution, http://www.sleepycat.com .

[DBCAIDA] Database files compiled by CAIDA, http://www.caida.org/NetGeo/NetGeo/ .

[DBNDG] Database file compiled by NDG Software, http://www.dtek.chalmers.se/~d3augus t/xt/dl/ .

[Halsey98] Brent Halsey, Visual Traceroute, Project Report submitted in Evi Nemeth's Networking class at University of Colorado, Boulder, 1998, http://www.caida.org/Tools/GTrace/paper/halsey.pdf .

[Jacobson88] Van Jacobson, Traceroute source code and documentation. Available from: ftp://ftp.ee.lbl.gov/traceroute.tar.Z .

[NetGeo] The Internet Geographic Database, http://www.caida.org/Tools/NetGeo .

[OROMatcher] OROMatcher – Regular Expression Package for Java, http://www.savarese.org .

[Peterson] Peterson, Larry L., & Davie, Bruce S., Computer Networks – A Systems Approach, Morgan Kaufmann, (1996).

[RFC1876] RFC 1876, Davis, C., Vixie, P., Goodwin, T., and Dickinson I., A means for Expressing Location Information in the Domain Name System, January (1996).

[RFC1918] RFC 1918, Rekhter, Y., Moskowitz, B., Karrenberg, D., Groot, G. J., Lear E., Address Allocation for Private Internets, February (1996).

[Swing] Java Foundation Classes – Swing http://java.sun.com/products/jfc/ .

[VisualRoute] Maps from VisualRoute, http://www.visualroute.com .

[Wood98] Jamey Wood, *Graphroute*, Project Report submitted in Evi Nemeth's Networking class at University of Colorado, Boulder, 1998, http://www.caida.org/Tools/GTrace/paper/wood.pdf .