ForNet: A Distributed Forensics Network

Research Experience for Undergraduates, Summer 2004

Daniel Speyer
University of Maryland
College Park

Amos Wetherbee
University of Massachusetts
Amherst

Radion Khait
Polytechnic University
Brooklyn

ABSTRACT

Networks have become a ubiquitous and integral part of our nation's critical infrastructure. With the explosive growth of LAN/WAN technologies and the Internet, mitigating threats to networks has become one of the key tasks of several government and private entities. However, existing defensive mechanisms, such as firewalls and Intrusion Detection Systems (IDSs), have proven insufficient and slow in adapting to new network vulnerabilities and threats. Although it may be impossible to prevent all instances of these attacks, our research proposes that it is possible to store forensic data that can be used in identifying perpetrators after an attack is made. Towards this end we have continued developed of a production version of ForNet, a network-wide logging system. ForNet is designed to both store network traffic efficiently and to provide a querying mechanism that assists in network forensics and attack attribution. The challenge is to design a system that can store as much information about network traffic as possible, while at the same time minimizing the storage requirements of this data.

1. INTRODUCTION

This report covers undergraduate research work completed during the Summer of 2004 by Daniel Speyer, Amos Wetherbee, and Radion Khait. Their research efforts were overseen by Kulesh Shanmugasundaram and Dr. Nasir Memon. Research took place at the Information Systems and Internet Security Laboratory at Polytechnic University in Brooklyn, New York.

1.1 Structure

This report is structured into several sections. Each section describes a particular implementation that was achieved from summer research. At the end of each section is an area highlighting future work that needs to be done before ForNet will be complete.

The first three sections of this report were written by Daniel Speyer from the University of Maryland. Dan worked on a logging system for ForNet, ForNet's Query Language implementation, and a Common Interface and Combination Scheme for Data Collecting Modules, respectively. The next section describes the ForNet Network Trace Protocol and was written and developed by Amos Wetherbee from the University of Massachusetts Amherst. The last section was written by Radion Khait, a student at Polytechnic University, and represents his work in developing a graphical user interface (GUI) client for ForNet.

1.2 Background

In order to completely comprehend this report, background understanding of ForNet is required. It is recommended that the reader first reference *ForNet: A Distributed Forensics Network* by Kulesh Shanmugasundaram before continuing. This report is meant to serve as an aide in developing a formal research paper, and is only meant to be descriptive of ForNet's technical implementation. Formal publication of ForNet's original contributions to the network security field is expected before the Fall of 2005. Additionally, ForNet will soon be released as an open-source project under the GNU General Public License. For a more detailed look into the code contributions of the authors, a complete SubVersion tree of ForNet is available upon request.

2. FORNET LOGGING SYSTEM

Any large daemon requires some way to log what happens to it so that users may resolve errors, evaluate performance, and detect attacks. In Fornet's case, this problem is more complex because individual modules may need to log messages, preferably to the same place as the main log. This is solved by an implicitly object-oriented logging system which may be duplicated in a coordinated way across each module.

2.1 Output Streams and Severity

ForNet has two places it can output messages: the logfile and the standard error. If Fornet is running as a daemon, the standard error is redirected to the file "forserver.err", but this still maintains the principle of two output streams. The error stream is used for severe issues that deserve attention (it would not be unreasonable for a cron job to email the sysadmin whenever an error is recorded), whereas the logfile should record everything that happens of any significance. The logfile *includes* everything in the errorstream, but so much ordinary data that it may not be immediately apparent.

When data is logged, the code which logs it may specify a severity level. There exist two thresholds: one below which messages are ignored and one above which they are printed as errors. At present these levels are hardcoded, but in the final release, they will be set by either a configuration file or command-line arguments. They can be set differently for different modules.

The severity levels are:

- **Debug:** information which is only useful to developers
- **Info:** ordinary events
- **Warn:** problems which can be dealt with
- **Error:** problems which cause a request to fail
- **Fatal:** problems which require a restart of Fornet (logging a fatal error automatically kills the process which logs it)

2.2 Threads, Modules, and Namespaces

If all threads simply opened the same file independently and wrote to it, they would overwrite each other. This might be avoided by opening the file anew for each message, but this is highly inefficient and has race conditions. What we want is for all threads and modules to share a single file descriptor.

Ordinarily, this file descriptor could be a global variable. This is not an option here, because the modules must call logging functions, and dynamically loaded modules cannot access the namespace of the program that loaded them. The modules can, of course, be linked with the logging functions, and carry their own copy of them, but these functions will not be able to access the global variable.

The solution is to give the modules their own functions *and variables*. The file descriptor and other configuration data are stored inside static variables inside `forlog.c`. This means that when each module is loaded, its internal log must be initialized with a call to its own copy of `init_logging`. This allows each logging system to take additional information, such as the name of the module it serves.

It is probably best to think of the logging system as a class which is instantiated once in the main program and once in each module. It has a constructor (`init_logging`) which sets its private members (the static variables), it has accessors for those variables, and it has a method `forlog` which actually gets work done.

2.3 Anatomy of a Log Message

A log message is generally issued by a call to one of the log macros (`DEBUGF, INFOF, WARNF, ERRORF,` or `FATALF`). These macros are called like printf, but in fact invoke `forlog` with the appropriate parameters. These parameters include the file name and line number on which the call occurred, which is the real reason they are macros. A typical log message is shown in **Figure 1**. Log messages are in the format: `[Date]SEVERITY:module(file:line): message`

2.4 Future Work

At present, synchronization is based on the assumption that `fflush` is atomic. This is true on glibc, but may not be elsewhere. It also assumes that no log message will ever be long enough to flush itself inside of `fprintf` so only `fflush` will actually call `write`. (`fflush` is called at the end of every log message, to ensure that things don't get broken up in libc's buffers.) It might be better to make synchronization explicit.

Like most of ForNet, configuration is entirely hardcoded. This needs to be replaced with configuration files.

3. FORNET QUERY LANGUAGE

ForNet modules are required to speak XML, but are intended to be easy to write. Furthermore, the ForNet data structures (aspects and events) are not native XML structures, but are provided by ForNet's own protocol. It would not be reasonable to ask module authors to implement the protocol, so we provide a library for the purpose. The library is designed for simplicity and extensibility: so much so that a module is likely to call only two functions from it.

3.1 Events, Aspects, and Types

The fundamental unit of the query language is the *event*. An event is anything which a module might record, most often an interesting packet. Each event is for us a set of *aspects*, each of which has a *key* which is a mostly free-form string, a *type*, which is usually one of a predefined set, and a *value*, which is of that type.

Queries to modules are simply events with some aspects missing. The module then searches for events in its database that match the query and outputs them. The output is usually more specific than the input (either has more aspects, or replaces vague aspects with specific ones), but it does not have to be: a module might just *confirm* its input.

Keys are intended to be universal, and module authors are strongly urged to make them so by checking whether an aspect has already been defined in another module and using the same key. A typical key is "src_port". This is composed of to conventions, "src_" means source, and "port" means transport-layer (eg TCP) port. Aspects with this key should generally be of type "int", because TCP and UDP ports are integers.

There are 11 built in types, which should suffice for most purposes. If these are inadequate, there is also a custom type feature. The types are:

- **string**: a text string, containing only printable characters
- **bitstring**: a string of arbitrary data. In XML, it is escaped to be expressed printably. In C, it is represented as an integer length followed by an array of arbitrary characters.
- **MAC**: an ethernet MAC address
- **net**: an IPv4 network, expressed as a.b.c.d/m (eg 192.168.0.0/16)
- **netv6** the same thing for IPv6
- **ip**: an IPv4 address
- **ipv6**: an IPv6 address
- **int**: an integer

Figure 1: Typical Log Message

```
[Thu Aug  5 15:27:23 2004]
DEBUG:histogram(histogram.c:274):db_get_first_item(...)=-30991(DB_NOTFOUND:
No matching key/data pair found)
```

- **float**: an real number
- **date**: a date and time, expressed to second's precision
- **flags**: a set of boolean flags. The list of flags must be supplied with the calls into ql. The data will then be expressed as a bitvector. *This feature may not work on big-endian architectures*

3.2 The get_data() Call

When a module receives a query, what it generally wants is to extract each aspect from it. This is done with the `get_data` call. This call take several arguments:

- `xmlNodePtr query`: the event from which data is to be extracted, generally the query the module received. Note that, while it is of type xmlNodePtr, the module may treat this as opaque.
- `const char *key`: the key of the desired aspect
- `enum ql_symbol type`: the type of data to look for. This is expressed as one of ql's defined symbols. In addition to the native types, this may be AUTO, which accepts any type, or CUSTOM, which accepts a user-defined type.
- `void *out`: the buffer in which to store the data
- `int len`: the size of the output buffer
- *optional arguments*: the list of flags for a `flags` type, or the type definition for a custom type. If the custom type requires an argument, a second optional argument may be provided for it. Usually, this is omitted.

The call returns `QL_OK` (which is equal to zero) on success, or various error codes, unless it is called with type `auto`, in which case it returns the type it found on success. It returns codes and sticks data into pointers because it may be returning arbitrarily large structures, and allocation is the module's responsibility.

In order to use a custom type, a module must create a type structure with callback functions to parse, unparse and validate it, and then pass this as an extra argument.

A typical call to `get_data` might look like:

```
int min_src_port;

if( get_data( query,
              "min_src_port", QL_INT,
              &min_src_port, sizeof(int)))
   min_src_port = -1;
```

3.3 Promotions

There are several cases in which the data requested by a module is not in exactly the same form as the user's input. Generally, this is because the input is more specific than the module requires. It is best for modules to accept the broadest queries possible, and it is the job of the query processor to ensure the user never regrets this.

Let us consider the connection records module. It looks for aspects with keys "min_src_port" and "max_src_port" so that the user can request ranges. It also looks for a aspect with key "src_ip" and type network, *not ip*. This allows users to search very broadly, (eg for traffic originating from The University of Maryland on ports between 6000 and 7000), but what about more specific searches?

The module does not look for more specific aspects. This would place a burden on the module author. Instead, the query system will cast these requests into their more general form. If a module wants an aspect with key "min_foo" or "max_foo" and it isn't there, but an aspect of the same type with key "foo" is, that will be returned for both. Also, if an aspect of type `net` is requested and one of type `ip` is available (with the same key), it will up turned into a network (ie 1.2.3.4 → 1.2.3.4/32). Similarly, a `ipv6` will be turned into a `netv6` and an `int` into a `float`.

At present, this is the entire set of automatic conversions. Custom types can accept native types, but not be accepted by them.

3.4 The output_event() Call

The other side of ql is output. Once an event has been found in the database, it must be sent out a stream (or several streams, see the section on networking). This is done with an `output_event` call. Note that an entire event is output at once.

The `output_event` call takes two arguments for itself, a stream to output on (generally stdout) and a hostblock for networking (which may be passed through the module unexamined). It then takes events, which are generally three arguments: a key, a type, and a pointer to data. If the type is `flags`, it must also include flaglist. The arguments are terminated by a zero. Here is a typical (but abbreviated) usage:

```
char *flags[] = { "FIN", "SYN", "RST",
                  "PUSH", "ACK", "URG",
                  NULL};
int src_port, dst_port;
char pkt_flags;
...etc...
```

```
output_event(stdout, hb,
        "src_port",QL_INT,&src_port,
        "dst_port",QL_INT,&dst_port,
        "tcp_flags",QL_FLAGS,&pkt_flags),flags,
        0);
```

4. A COMMON INTERFACE AND COMBINATION SCHEME FOR DATA COLLECTING MODULES

In a system such as Fornet, data is collected by various modules with different purposes and possibly different authors. Each module records one particular type of event (e.g. Fast-track recognition packets). The user may then wish to query the entire set of modules, without considering which ones will give the answer. However, when the answer is sent, each event must be vouched for by a specific module. In order to find and assist the modules, there is a query processor.

4.1 Motivating Example

Foo University is determined to keep viruses off their network. They have smart routing hardware at the edge which checks all incoming packets for viruses and drops infected ones. Nonetheless, they find a virus outbreak within their network. Someone must have brought an infected laptop onto campus. Fortunately, they've been running Fornet. They want to know who brought that laptop.

4.2 The Query Model

For purposes of querying, all modules, no matter what data they record, record *events*. These events could be packets, HTTP requests, portscans or anything else the module writer can think of. Each event, is defined by a set of *aspects*. Each aspect contains a *key* a *type* and some data. Every event from every module must have a *time* aspect (of type date) but otherwise may have anything.

In general the aspects are opaque to the query processor. There are two exceptions to this. First, if the module wants minimum and maximum values for something, and only an exact value is provided, that value will be given for both (this means minimums and maximums must use non-strict comparison). Secondly, certain datatypes may be upcast at need: IPs to networks (either ipv4 or ipv6) and integers to floats.

A query is a request for events. In a query, a client says to a server, "I think there were events like this, and I want to know that about them," where this is a set of aspects with data and that is a set of aspects without data. In our example, we know the content of the virus and when it first struck – we want to know what IP it came from. We would also like to have the time and content confirmed, so we ask for those as well.

For purposes of example, we will suppose the virus contains the phrase "format c:" which does not appear in legitimate traffic (in a real virus, the defining text would be much longer, and would probably require MIME-encoding, both of which make for poor presentation). This query is shown in **Figure 2**.

4.3 A Module's Perspective

Each module does one simple task, and does so as simply as possible. This makes it relatively easy to write modules. Therefore, a module tells the query processor what sort of data it expects, and then counts on getting it.

A module first publishes its interface. This is done in XML. It specifies its input as an *inputSpec*. An inputSpec consists of either an aspect without data or a set of inputSpecs and a number which specifies how many of them are needed. This number may be anything between zero and the number of children. It specifies its output as a set of aspects without data.

The first module used in our example is the connection records module. **Figure 3** shows an example of an inputSpec.

This essentially means that any query to it *must* contain a time range and may contain various other things. The hints tag is used by the client to present a more intelligent interface, and is not used directly by fornet.

When queried, a module receives a pointer to the query (which is a parsed XML tree, but the module doesn't need to know this) and a database cursor. The module extracts the aspects which interest it from the query using the get_data function. It then iterates from the database cursor and outputs all events which match the query.

The query types should be as broad as possible. For example, IP addresses should be taken as type net if possible (if the user wishes to search for a specific IP, the processor will auto-convert it).

Figure 4 shows an example of the query that the the module receives.

Note that it contains only information describing the desired events – nothing else. The module will reply with everything it knows about all the events that meet the criteria. Also note that the query contains information which the module cannot use. This will be ignored.

The module should output exactly what it knows about each event. Returning extra data is harmless (we did the work to look it up, we may as well return it). A module should *not* echo its query if it cannot confirm it from its own data. When a module returns data, it is vouching for it. In this case, the module output resembles **Figure 5**.

Note that this output is a series of valid queries. This will become important later.

4.4 The Processor's Perspective

When the processor receives a query, the first thing it does is check which aspects are offered as input and which are desired as output. Data is not relevant here. The query processor's job is to find the module or modules which will give the desired output and get them their desired input.

In the ideal case, there exists at least one module which offers the desired output and accepts the offered input. If so, the processor simply runs this module or set of modules.

Figure 2: Example Query for a “format c:” String

```
  <command name="query" scope="server">
    <event>
      <aspect key="min_time" type="date">2004-08-05 00:00:00</aspect>
      <aspect key="max_time" type="date">2004-08-05 12:00:00</aspect>
      <aspect key="excerpt" type="string">format c:</aspect>
      <aspect key="dst_ip" type="net">128.238.0.0/16</aspect>
    </event>
    <desire>
      <aspect key="time" type="date"/>
      <aspect key="src_ip" type="ip"/>
      <aspect key="excerpt" type="string"/>
    </desire>
  </command>
```

Figure 3: An Example inputSpec

```
<inputSpec min="3">
  <aspect key="min_time" type="date"/>
  <aspect key="max_time" type="date"/>
  <inputSpec min="0">
    <aspect key="src_ip" type="ip"/>
    <aspect key="dst_ip" type="ip"/>
    <aspect key="min_src_port" type="int" hints="range"/>
    <aspect key="max_src_port" type="int" hints="range"/>
    <aspect key="min_dst_port" type="int" hints="range"/>
    <aspect key="max_dst_port" type="int" hints="range"/>
    <aspect key="tcp_flags" type="flags"  hints="SYN,FIN,ACK,RST,PUSH,URG"/>
  </inputSpec>
</inputSpec>
```

Figure 4: Example Input to a Module

```
<event>
  <aspect key="min_time" type="date">2004-08-05 00:00:00</aspect>
  <aspect key="max_time" type="date">2004-08-05 12:00:00</aspect>
  <aspect key="excerpt" type="string">format c:</aspect>
  <aspect key="dst_ip" type="net">128.238.0.0/16</aspect>
</event>
```

Figure 5: Example Module Output

```
<event>
  <aspect key="time" type="date">2004-08-05 06:01:39</aspect>
  <aspect key="src_ip" type="ip">128.238.79.61</aspect>
  <aspect key="dst_ip" type="ip">128.238.79.84</aspect>
  <aspect key="src_port" type="int">135</aspect>
  <aspect key="dst_port" type="int">3587</aspect>
  <aspect key="tcp_flags" type="flags">RST,ACK</aspect>
</event> <event>
  <aspect key="time" type="date">2004-08-05 06:01:39</aspect>
  <aspect key="src_ip" type="ip">128.238.79.84</aspect>
  <aspect key="dst_ip" type="ip">128.238.79.61</aspect>
  <aspect key="src_port" type="int">3049</aspect>
  <aspect key="dst_port" type="int">2745</aspect>
  <aspect key="tcp_flags" type="flags">SYN</aspect>
</event>
```

Each modules output is bracketed by `<module name="foo">` `</module>` tags by the processor.

If no such module exists, the processor looks for a module which gives the desired output but will not accept the offered input. If it finds one (or more), it calculates what is missing for the module, and seeks out another module that can offer that, and will accept as input what the user offered. If it finds such a module (called a "premodule"), it runs it, combines its output with the users input, and feeds that into the original module. Again, if multiple modules are fill the requirements, they are all used. In our example, this is what happens. The final output comes from a hierarchical bloom filter module (the only module whose output includes an "excerpt" bitstring). The HBF, however, insists on being given the source and destination IPs and ports, which the user does not offer. Therefore, every module which can give source and destination IPs and ports based only on time (this includes connection records, fasttracker and probably others) will be piped into the HBF along with the excerpt. The HBF will the give the only output that the user sees.

This may be continued to chains of any length using an iterative deepening search. It is probably best to set an artificial limit on chain size, because extremely long chains, if they exist, are likely to contain so many extraneous results as not to be useful. Experience suggests a limit of 2 is sufficient, and this is what our current server implements.

4.5 Future work

At present, there is no way to combine events from different modules. If two modules both record packets, but record different aspects of them, it would be ideal to have the union of their information. This is not available because two different packets could share all aspects in the intersection of the modules' fields. This could be fixed by creating unique ids for packets (32 bits would suffice, provided all modules also record times to the second), but it would leave open the more general problem of matching non-packet events.

Also, at present, the server will ignore any inputs it does not need. It might be more useful to add modules to use extra inputs and to raise warnings if inputs cannot be used. Real world testing will be necessary to determine if this is actually useful or not.

Finally, it might be useful to do more processing of data on the server. For example, it might be useful to count the outputs of a module and return only that count. Much of this can be done on the client, which is generally preferable (the client does not require as great reliability as the server, so it can more easily absorb new features), but sometimes this requires large and otherwise unnecessary data transfers. In these cases, server-side processing is appropriate.

5. NETWORK TRACE PROTOCOL

The purpose of the ForNet Network Trace Protocol ("FNNTP") is to enable forensic analysts and auditors to verify the integrity of network events after they have occurred. FNNTP is specifically designed to allow clients to verify that the source and destination of a particular network event as recorded by a particular SynApp and stored on a particular Forensic Server are consistent with the route that the event took through a ForNet-monitored network.

5.1 Motivation

One of the biggest weaknesses in the Internet today is a lack of control over the validity of IP address headers. Specifically, IP packets with forged source addresses may be easily created and sent from nearly any host on the Internet. This is analogous to mailing a letter with a fake return address. The majority of routers on the Internet ignore packet source addressing because this information is not useful in routing a packet to its destination. Indeed, it may be too much to ask a router to validate the origination of a packet in a reasonable amount of time, as this task would severely reduce the speed and effectiveness of IP networks. However, validating the source address of a packet becomes increasingly valuable in an age when worms, viruses, and malicious hackers can launch attacks and hide behind the anonymity that invalid source addressing provides. To solve this problem, we propose to use ForNet, a distributed forensics network, to deter against and halt the effectiveness of these "spoofed" attacks by providing an infrastructure for tracing any network event back to its source with a defined level of confidence. This will also allow forensic analysts to provide a "complete picture" of network events useful for evidence in court proceedings. To do this, we will add a new network trace operation to ForNet's modular querying system. This operation is implemented as the ForNet Network Trace Protocol.

5.2 Scope

The ForNet Network Trace Protocol is specifically limited in scope to provide the functions necessary to receive, transmit, and process XML documents over TCP/IP networks. These documents contain query requests and query reports as specified in the ForNet Communications Language (ForComm). These documents are validated against ForComm's XML Schema for correctness.

5.3 Background

A Forensic Server acts as a Database Management System (DBMS). It provides functionality for SynApps to store forensic data centrally, and for clients to retrieve this data by performing queries. FNNTP is concerned with the need to combine server-side query processing and query routing to answer queries posed by clients requesting information spanning an entire network of Forensic Servers.

One of the most basic operations of a Forensic Server is to answer queries from a client. These queries are posed by a client program and are specified at a certain granularity, or scope. Module-scoped queries are designed to retrieve information from a particular module of a particular SynApp. SynApp-scoped queries are designed to retrieve information relative to all modules of a particular SynApp. Finally, Network-scoped queries are designed to provide network tracing and collaboration between multiple SynApps. The ForNet Network Trace Protocol implements the operations needed to successfully pose and answer Network-scoped queries.

All queries, regardless of scope, are targeted at a particular SynApp by providing that SynApp's IP Address. However, clients establish connections with Forensic Servers, not

```
      +-------+
      | FNNTP |
      +-------+
          |
+---------------+
| ForComm (XML) |
+---------------+
          |
+------+  |  +-----+   +-----+     +-----+
|Telnet|  |  | FTP |   | TFTP| ... | ... |
+------+  |  +-----+   +-----+     +-----+
      |   |   |           |           |
     +---------+       +-----+     +-----+
     |   TCP   |       | UDP | ... | ... |
     +---------+       +-----+     +-----+
        |                 |           |
     +-----------------------------------+
     |      Internet Protocol & ICMP     |
     +-----------------------------------+
                    |
       +---------------------------+
       |   Local Network Protocol  |
       +---------------------------+
```

Figure 6: Protocol Relationships

SynApps, to transmit XML Request Documents. These Forensic Servers host the databases that of a particular SynApp that a client is trying to retrieve information from.

5.4 Interfaces

The protocol is initiated by a client who is authorized to issue network-scope queries to a particular Forensic Server. This client constructs a valid ForComm XML Request Document and transmits this document over a plaintext TCP/IP socket connection. This socket connection is established on TCP/IP port 65521 of the Forensic Server hosting the SynApp databases that the client is interested in querying against. Presumably, the client has already selected a specific set of network events identified from a previous query (although this may not be the case). Now the client is interested in "tracing" or extracting all known information about this set of events from an entire Forensics Network.

After a client has transmitted an XML Query Request document, the client must accept local TCP/IP connections on port 65522 from any Forensic Server in a ForNet network that may elect to respond to the client. Each Forensic Server that establishes a socket connection to the client will transmit an XML Query Report document to that client.

5.5 Operation

The ForNet Network Trace Protocol involves client-to-server, server-to-client, and server-to-server communications. These communications are defined by five functions on the Forensic Servers and two functions on the querying clients:

The five basic operations of a Forensic Server are Receive, Process, Route, Report, and Forward. A Forensic Server receives XML Request Documents containing network-scoped queries, processes these requests, selects a routing schedule in order to forward the requests to other Forensic Servers, reports back to the client who sent the original query with an XML Report Document, and forwards new XML Request Documents to other Forensic Servers which have been determined by the routing schedule to be candidates for information retrieval.

The three basic operations of a client are Request, Receive, and Interpret. A client constructs and transmits an XML Request Document to a specific Forensic Server hosting the SynApp databases to query against, receives XML Report Documents from Forensic Servers in response to this request, and interprets these reports by combining the results of multiple XML Report Documents.

5.5.1 Forensic Server Operations

The first operation implemented by a Forensic Server is to Receive. First, a network connection is established on port 65521 of the Forensic Server. Next, an XML Request Document is sent to the server from a client. In the future, the server will verify the XML Request Document against ForComm's XML Schema [2]. After the XML has been verified as valid, the document will be parsed and all datatypes will be validated. If any datatypes, such as dates or numbers are found to contain invalid characters or are otherwise not in a format that is readable by the Forensic Server, an XML Invalid Request Document will be returned to the client, and any errors will be sent, enclosed in `error` tags.

After retrieval of an XML Request Document, the Forensic Server begins the Process operation. An XML Request Document is processed by extracting the commands requested by the client and executing those commands in-line with the ForNet Communications Language. FNNTP is only concerned with Network-scoped queries, so any XML Request Documents which do not contain such queries are outside the scope of the FNNTP protocol.

Processing involves treating the Network-scoped query as if it were a SynApp-scoped query. Currently, both queries are processed by the same engine. Processing a SynApp-scoped Query and a Network-scoped query is done as follows:

1. The client specifies a Forensic Server and a SynApp against which the query will be processed in the ¡to¿ tag of the XML Request Document. The client also specifies the actual network-scoped query in a ¡command¿ tag.

2. The client transmits the XML Request Document to the Forensic Server.

3. The Forensic Server creates a Hostblock which contains information about the query such as who the requesting client is, which Forensic Server the request was sent to, which SynApp the query is meant for, and possibly which server forwarded the request (this last one is not included by the original client, because the client is not a server).

4. The query is extracted from the XML Request Document by scanning it for ¡command¿ tags with a NAME attribute of "query" and a SCOPE attribute of "network" or "synapp".

5. A Query Configuration is established from the information contained in the client's query. This information may include time ranges, port numbers, or any aspect of any network event provided by any module of the SynApp being queried. The goal is for the query system to return all network events that match the client's criteria. The client can also, optionally, specify which event aspects he/she desires. That is, a client may only be interested in the time a matching network event occured, or perhaps the TCP/IP flags associated with that event.

6. Once the query has been parsed into a Query Configuration, the query is evaluated. For each module used by the SynApp, the query is matched against all available data and a list of events matching the query is generated.

7. If no events are returned by the query, the client is informed that there are no results.

So far, there is no difference between a SynApp-scoped query and a Network-scoped query. However, if resulting events are outputted in query evaluation, Network-scoped queries are treated differently than SynApp-scoped queries. A SynApp-scoped query will generate an XML Report Document and send it immediately to the client through the socket connection that the client and server have already established. The server then closes the connection and leaves the client to interpret the query results. The Network-scope query, instead, will being it's Route stage.

In the Route stage, the Forensic Server knows that a particular SynApp has information to report that is relevant to the client's query. The Forensic Server must now identify which other SynApps in the same network contain information relevant to the client's query. In the future, more sophisticated methods for determining routes will be explored. Currently, each SynApp has a list of all the SynApps adjacent to the network edge that it is monitoring. This list is used to route Network-scoped queries. The logic is that if a network event is seen by one SynApp on a particular network segment, then the same event would have to have occurred on an adjacent network segment (unless the network event is local to a particular segment, in which case no routes are generated). Since there is no current mechanism for tracking the route a specific network event took through a network, routing to all adjacent SynApps is a reasonable brute-force query routing method.

If the Forensics Network is viewed as a vertex covering of a graph, then we can see that this looks like a recursive breadth-first search, but since network events travel linearly through a network, the resulting network-wide routing for a specific query will act depth-first. Whereas one SynApp may route a query to 10 adjacent SynApps, only two of those SynApps (those monitoring the upstream and downstream movement of an event) will actually continue to route the query because no one else has any information on that specific event.

When a list of query routes has been established, a socket connection is established to the client that originally posed the query. An XML Report Document is sent to the client with the results of the Network-scoped query on the specific SynApp that has received it. For example, if the client posed a Network-scoped query about a specific event that traveled past five SynApps, the client should receive an XML Report Document from each SynApp. The client should therefore have a total of five socket connections to it's local port 65522, one from each SynApp reporting.

As the XML Report Document is generated by the SynApps and transmitted to the client, the SynApps are simultaneous generating XML Query Documents which contain new Network-scoped queries. These queries are forwarded to the SynApps in the Route list. This is called the Forward stage. One key point that has not been mentioned is that a SynApp will not forward a Network-scoped query back to the server that passed the query to it. This is done to ensure that Network-scoped queries move forwards. In the future, Network-queries will be tagged with unique identifiers that ensure a Network-query is not answered twice by the same server.

The XML Query Document that is forwarded to the adjacent servers contains a new query that is the result of the recieved query after being processed by the SynApp. That is, the output of the Network-scoped query on one SynApp is the input of the new Network-scoped queries on the adjacent SynApps. The ¡from¿ tag of the new Network-scoped query is modified to indicate the SynApp that forwarded the query, but the original client attribute of the ¡from¿ tag is maintained so that all future SynApps can connect back and report to the client that originated the Network-scoped query.

5.5.2 Client Operations

Network-scoped queries are implemented by the client in a way that is very similar to SynApp-scoped queries. The client establishes a socket connection to a Forensics Server on port 65521 and transmits an XML Request Document.

However, in a SynApp-scoped query, the results are transmitted back to the client on this same socket connection. In a Network-scoped query, the only thing that a client will receive back is an acknowledgement that the Network-scoped query was processed and forwarded. This is known as the Request operation of the client.

The Recieve operation is unique to Network-scoped queries. Once a client sends out an XML Request Document specifying a network-scoped query, the client must accept connections on local port 65522 in order to receive the results of that query. Each individual SynApp involved in answering the query will connect to the client machine on that port and transmit an XML Report document with the information that the SynApp knows about the client's query.

The final operation for a client to perform, after receiving the results of all network-queries, is to Interpret the data.

5.6 Future Work

At the moment, ForComm has not been rigorously defined yet and the ForComm XML Schema does not exist for validating communications. The authorization and security protocols also need development. In the future, we will also want to incorporate XML compression into FNNTP to increase throughput.

Additional work also needs to be done in making FNNTP work for peer-to-peer networks. At the moment, hierarchical networks are assumed. Future empirical analysis will need to be performed on different querying algorithms. ForNet also needs to implement more features of FNNTP.

There are also several conceptual hurdles that need to be overcome in future development of FNNTP. A method for correlating one network event with another network event will need to be devised if the goals of ForNet are to be met. This correlation is needed to determine causality. An algorithm must also be developed that can compute the error associated with each network query.

6. PANORAMA, A FORNET CLIENT

One of the most important features that an application can provide is the way it interfaces with the user. As of Fall 2004, the client part of ForNet was written in two languages: Python and Java with most of the effort going into the Java version, although both are still under development. Java was selected as the primary implementation language due to the functionality it provides in handling graphical user interface (GUI), networking, XML, and cross-platform compatibility. The same copy of the client can be used on any operating system that has Java installed.

Panorama enables its users to query the forensic servers and retrieve the needed data. Then this information is used to either perform more queries as needed, or to investigate the result. All of the communication between the client and servers are in XML form. The following are the primary features that the client supports in more detail.

6.0.1 Connecting to Forensic Servers

In order to retrieve any information that was collected by the many servers, the client must be able to establish a connection with them. Once this "handshake" connection is made, the server provides the general information of what synapps and modules are managed by that server. It also provides attributes of modules and synapps that can be queried for. This information will help the user make more precise queries.

6.0.2 Sending a Search Query

Once the module attributes are retrieved the user will be able to use build-in tools to create the desired query for the servers to process. For example, they can enter start and end times, origin or destination address of packets as well as the type of packets. The query can be made complex by using and/or/not operators on individual parameters. The address of another computer can be either an IP or a host names, such as google.com. The query consists of multiple aspects. Each aspect contains a name, type and desired value of the parameter.

6.0.3 Processing Query Results

The response will also be in XML form. It will contain all the data about events that are valid answers for the query. For the top example, the response will be:

6.1 Data Analysis

The large amount of textual data by itself is not very useful to the user. The client helps the user to analyze and use the data. The outcome of a query can be used to make more queries. For example, once it is found that 67.86.159.68 send a packet to 128.238.10.212 on port 25, one can find out what packets went in the opposite direction on the same port at about the same time. This feature can be used to trace a particular packet amongst multiple computers. If, for example, this packet was determined to be a virus, it can be traced back to where it originated and to what other computers did it spread. It is possible to save the data to the hard drive for future processing, or to be opened in another application such as Excel. Also search queries can be saved, and then reloaded in the future to continue, or repeat, a search.

6.2 Simultaneous Queries

Since resolving a query can be a long process, the client is able to send multiple queries at one time and process their results. Because of many results, a summary is provided for each query that lists such information as the number of events returned, the size of the results, and how long is the query taking to execute. Tabs are used to display results for each query. With multiple simultaneous queries and potentially large results (can reach gigabytes in size), it is impossible to keep all the data in memory and display all to the user. For this reason efficient memory management algorithms are used that display only parts of the data. If the user desires to see more, what was in memory previously will be discarded and replaced by new data. This method will make all the data be viewable without storing it in computer's memory.

6.3 Settings and Preferences

Similar to most user interface applications, users are able to store their preferences. These include the default internet

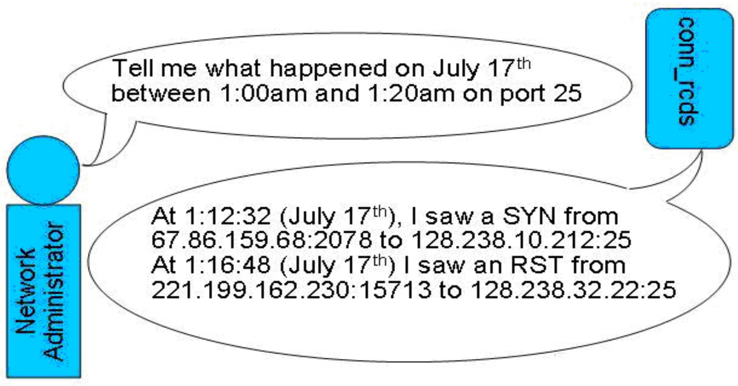

Tell me what happened on July 17th between 1:00am and 1:20am on port 25
conn_rcds
Network Administrator
At 1:12:32 (July 17th), I saw a SYN from 67.86.159.68:2078 to 128.238.10.212:25
At 1:16:48 (July 17th) I saw an RST from 221.199.162.230:15713 to 128.238.32.22:25

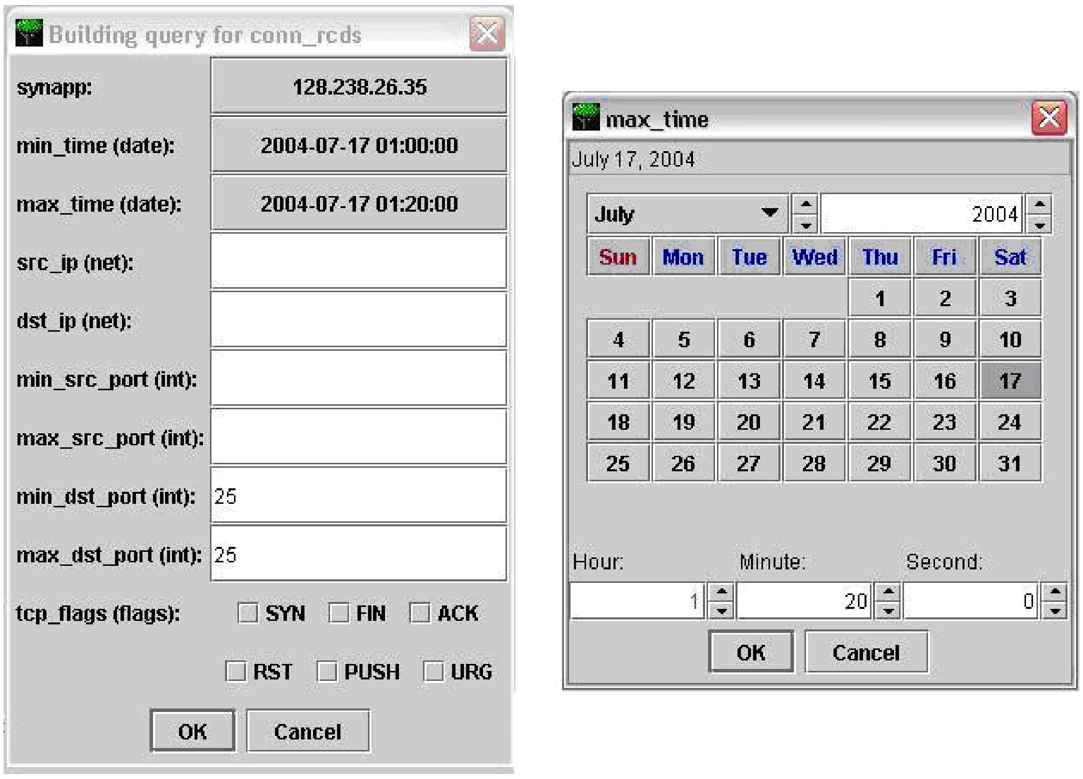
Figure 7: An Example Search Query

```
<request>
  <to server="128.238.35.91" synapp="128.238.26.35" />
  <from client="128.238.79.61" />
  <command name="query" scope="module" module="conn_rcds">
    <event>
      <aspect key="min_time" type="date">2004-07-17 01:00:00</aspect>
      <aspect key="max_time" type="date">2004-07-17 01:20:00</aspect>
      <aspect key="dst_port" type="int">25</aspect>
    </event>
  </command>
</request>
```

Figure 8: Query Request Represented in XML

```
 <module name="conn_rcds">
  <event>
    <aspect key="time" type="date">2004-07-17 01:12:32</aspect>
    <aspect key="src_ip" type="ip">67.86.159.68</aspect>
    <aspect key="dst_ip" type="ip">128.238.10.212</aspect>
    <aspect key="src_port" type="int">2078</aspect>
    <aspect key="dst_port" type="int">25</aspect>
    <aspect key="tcp_flags" type="flags">SYN</aspect>
  </event>
  <event>
    <aspect key="time" type="date">2004-07-17 01:16:48</aspect>
    <aspect key="src_ip" type="ip">221.199.162.230</aspect>
    <aspect key="dst_ip" type="ip">128.238.32.22</aspect>
    <aspect key="src_port" type="int">15713</aspect>
    <aspect key="dst_port" type="int">25</aspect>
    <aspect key="tcp_flags" type="flags">RST</aspect>
  </event>
</module>
```

Figure 9: Query Response Represented in XML



Building query for conn_rcds

synapp:	128.238.26.35
min_time (date):	2004-07-17 01:00:00
max_time (date):	2004-07-17 01:20:00
src_ip (net):	
dst_ip (net):	
min_src_port (int):	
max_src_port (int):	
min_dst_port (int):	25
max_dst_port (int):	25
tcp_flags (flags):	☐ SYN ☐ FIN ☐ ACK ☐ RST ☐ PUSH ☐ URG

max_time

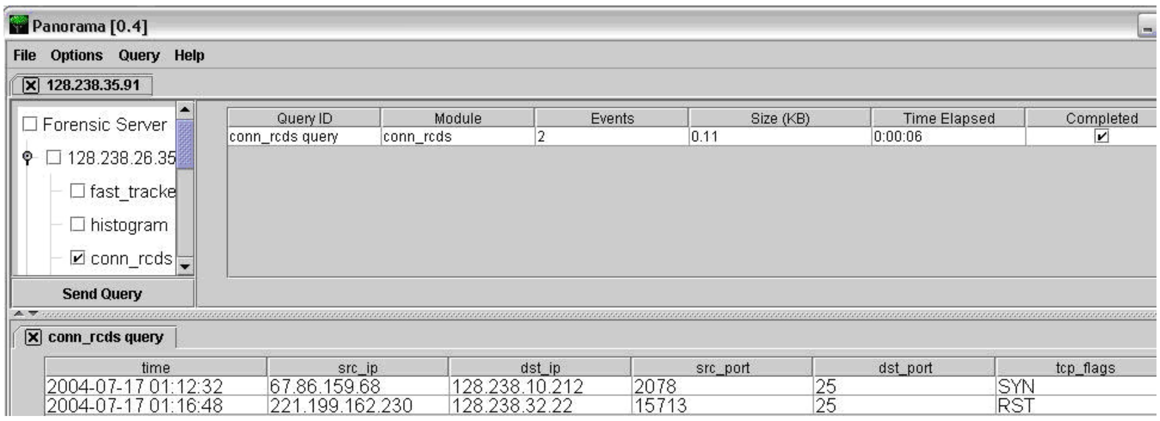
July 17, 2004

July 2004

Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Hour: 1 Minute: 20 Second: 0

Figure 10: Query Controls



Panorama [0.4]

Query ID	Module	Events	Size (KB)	Time Elapsed	Completed
conn_rcds query	conn_rcds	2	0.11	0:00:06	✓

time	src_ip	dst_ip	src_port	dst_port	tcp_flags
2004-07-17 01:12:32	67.86.159.68	128.238.10.212	2078	25	SYN
2004-07-17 01:16:48	221.199.162.230	128.238.32.22	15713	25	RST

Figure 11: Query Results Window

browser the application should use, what default server to connect to, etc.

6.4 Future Work

Panorama is in the early stages of development. The basic features to search and show results are operational, but ones more advanced are needed for easier and more efficient use of the client. These include:

- Visual representation of data to help make conclusions about network events.
- Support for visualization plug-ins. These could be third-party tools that improve visualization.
- Tools for identification of particular hosts. These include trace-back, who-is, and geographical locations.
- Batch processing: automate the sending and receiving data. For example, given a list of queries, the application will send them and save the results to a file with no user interaction required.
- Interface for managing synapps. Administrators will connect to a particular synapp and be able to make appropriate changes, view its status, etc.

7. ACKNOWLEDGMENTS

This work was supported in part through a Research Experience for Undergraduates (REU) grant from the National Science Foundation. Special thanks is extended to Dr. Nasir Memon and Kulesh Shanmugasundaram at Polytechnic University in Brooklyn, NY for their support, insight, and research training. Thanks is also extended to Dr. Frank Cassara for sponsoring the Summer Research Program and promoting undergraduate research.