NATTT Design Specification

September 17, 2009

1 Introduction

This document describes the high-level architecture and design of the NAT Traversal Through Tunneling (NATTT) project. The goal of NATTT is to provide a simple tool that can (initially) be run on users' computers (servers, desktops, or laptops) that can tunnel through NAT boxes and provide end-to-end reachability. This goal is motivated by the observation that NATs effectively eliminate the ability of parties in the Internet to make arbitrary connections to any machine, as the target machine may lie behind a NAT box and thus doesn't have a routable address. Nevertheless, services that accept inbound connections (such as HTTP, SMTP, etc) may be run on these hosts and must be accessible from the public Internet. NATTT has 2 main components to address this need: i) DNS naming component that allows any client to locate and construct a routable packet to clients behind NATs, and ii) a component that manages NATTT tunnels and exports local IP addresses to clients so that 3^{rd} party software (such as web browsers) does not need to be modified *at all* to use NATTT.

There are many constraints and pitfalls that complicate this setting. NAT boxes vary quite widely in their implementations and behaviors. It is unfortunate that very little common behavior can be expected across them. For this reason, the NATTT design treats NAT boxes like black boxes and requires only one behavior from them (and attempts to use one other): port forwarding and universal plug and play (uPnP). The details of this are discussed in the architecture section.

The rest of this document is structured as follows: Section 2 describes the motivation of this project. Next, Section 3 details the requirements. After this, Section 4 describes the architecture and Section 5 outlines the component decomposition.

2 Motivation

3 Requirements

1. Must be able to detect NXDOMAINs on outbound A queries to local caching resolver, intercept, and insert queries for NAT3 RR at same name. If response exists, must create an NAT3 tunnel and return to original querier a locally routable IP address that NAT3 will be listening on,

2. Must be able to accept ALL IP traffic on the local IP subnet: 127.1.0.0/16 (henceforth referred to as Net_{nat3}).

3. Must handle traffic on Net_{nat3} by encapsulating original IP traffic into NAT3 tunnel specified by the destination address in Net_{nat3}

4. Must listen on a well known port (100) for incoming NAT3 connections and be able to de-encapsulate them and deliver them locally. When delivering them, an IP in Net_{nat3} must be used as the source, so NAT3 can receive any responses.

5. Must only require port forwarding of a single port on NAT boxes

   (a) May use Universal Plug and Play (uPnP) instead of manual port forwarding, but only when that is available.

6. Must compile and run properly on Linux (kernel 2.6.x) and OS X 10.4.x and OS 10.5.x.

4 Architecture

The architecture of the NATTT client is broken into the following main components:

- DNS Resolver
- Tunnel Manager

4.1 DNS Resolver

The DNS Resolver component intercepts all DNS requests issued by the user and will forward them to the network's local caching resolver. When that resolver gets an NXDOMAIN back for an A record query, the NATTT resolver re-issues the same query, but specifies the NAT3 RR type. This allows NAT3 to see if there is a fallback connection to a destination.

If the NAT3 request finds RRs, then the other end must be behind a NAT box. At this point the resolver communicates with the Tunnel Manger in order to create a tunnel through the remote NAT box. This is depicted in Figure 1

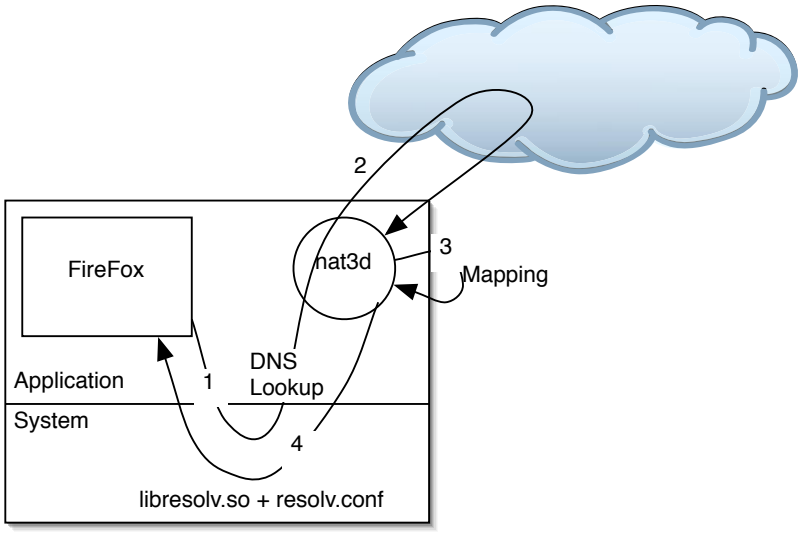

Figure 1: DNS control flow

4.2 Tunnel Manager

The Tunnel Manager is responsible for the network-layer communications between a local application (such as FireFox) and a remote destination that may be behind a NAT box. The tunnel manager creates and maintains a cache of tunnels that are in use.

Outbound Traffic: All traffic that is to be sent through the tunnel is captured by the Tunnel Manger and encapsulated. Specifically, all packets (TCP and UDP) are encapsulated into UDP datagrams. The IP header of the datagram contains the information about the public address of the remote NAT. The datagram then has a second (encapsulated) IP header that specifies the *internal* address of the destination. The local (source) information in each header is the local (internal) IP address of the user's host. As the packet traverses the local NAT box (if there is one), the outer header will have its address re-written. When the Tunnel Manager needs to create a new tunnel, it allocates a new IP address inside of the 127.1.0.0/16 subnet and creates a mapping for this new IP and the outbound port used to actually send the packet. This is a subnet that the Tunnel Manager accepts all local traffic from (on the individual) machine. By using this network, the Tunnel Manager can assign its own locally routable addresses without worrying about potential address conflicts (because 127.0.0.0/8 is defined to be a non-routable prefix). These IP addresses can then be used by 3rd party applications as a tunnel ingress in place of the remote destination. Thus, applications like FireFox are given an IP address that is unique to a local tunnel, and the tunnel manager intercepts all traffic and encapsulates

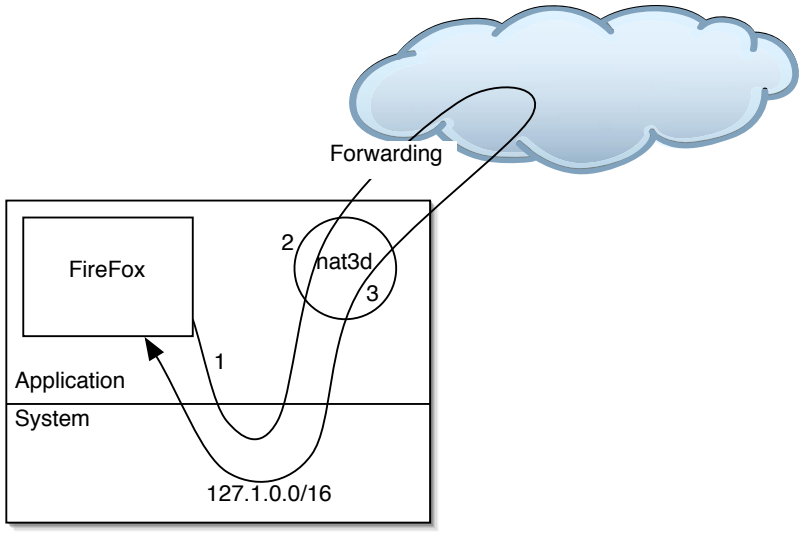

Figure 2: Tunnel Manager control flow

it.

Inbound Traffic: All inbound traffic is received by the tunnel manager on a specific port that it listens on. When traffic is received for the Tunnel Manager, its external and internal headers are used to identify if there is an existing tunnel from the remote source, and if no tunnel mapping exists, a new one is created. The tunnel mapping indicates a specific local IP address (in the 127.1.0.0/16 subnet) to use as the source address when re-writing the inbound packet. This way, the client (such as FireFox) will respond to a local IP address that the Tunnel Manager will intercept and can then rewrite the packet for.

This is depicted in Figure 2.

5 Component Decomposition

The architecture of NATTT has 2 major components: the DNS Resolver and the Tunnel Manager. This section breaks these 2 high level components down into their C++ classes and specifies their interfaces and interactions.

The overall design and interactions of these classes is shown by Figure 3. The individual classes are described below, and this section concludes with some use-cases.

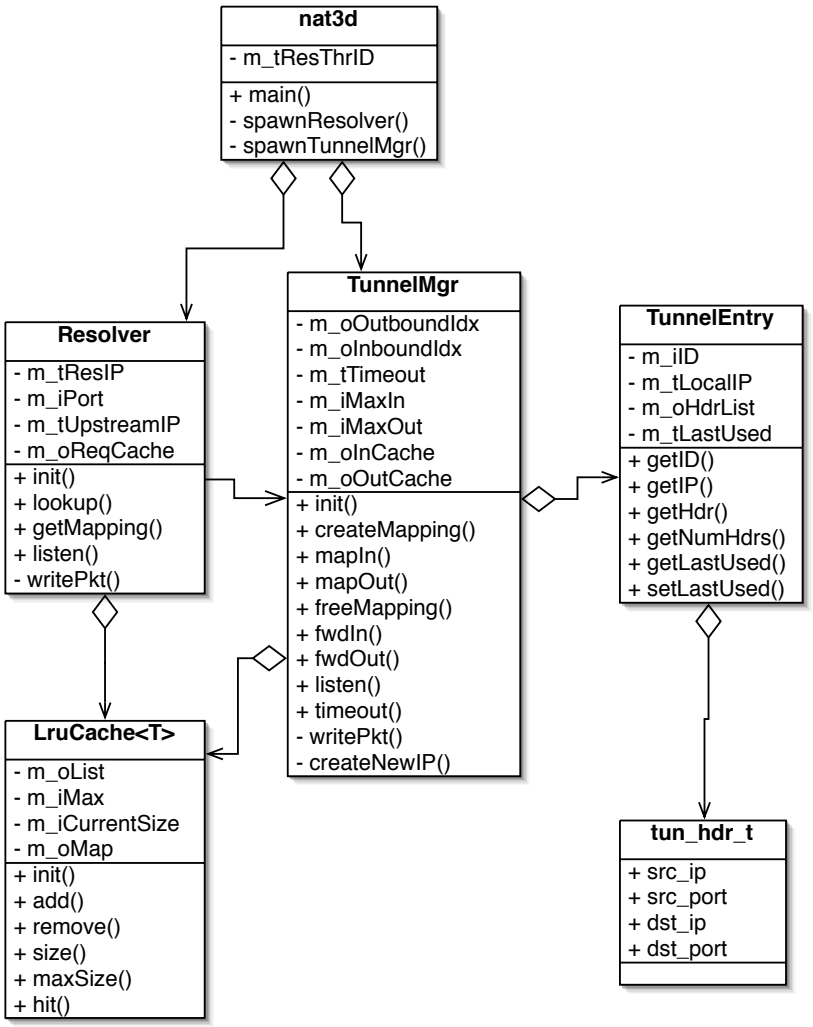

Figure 3: A UML diagram of the NATTT daemon.

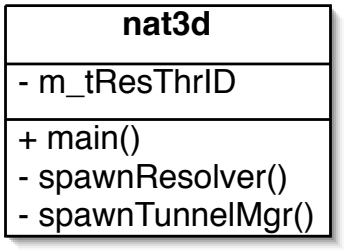

Figure 4:

6 nat3d

This section describes the nat3d daemon component, which is described by Figure 4. This daemon is the actual NATTT executable. It runs 2 threads: one for the DNS resolver logic, and a second one that manages the IP tunnels. This component is very lightweight and is not responsible for much logic.

6.1 Methods

Public Methods

- main(): This is the main method of the daemon. It spawns a worker thread for the resolver and the directly calls the tunnel manager's logic.
- spawnResolver(): This method is uses as a callback to pthread_create() and initializes and drives the DNS resolver logic.
- spawnTunnelMgr(): This method initializes and drives the IP tunnel manager logic.

6.2 Member Variables

- m_tResThrID: thread_t This is the ID of the thread that runs the DNS logic.
- m_tTunThrID: thread_t This type is reserved for the future and currently does not hold any information.

7 LruCache

This section describes the LruCache component, which is described by Figure 5. The LRU cache is a simple template class that implements a Least Recently Used eviction policy for its entries. It is configurable to have a maximum size and has simple accessors and an init function. It uses an STL map¡¿ class to to store its data and an custom list class that supports random access.

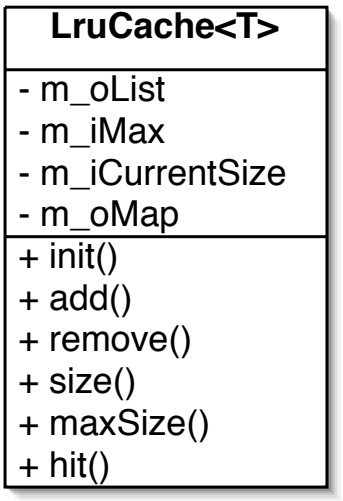

Figure 5:

7.1 Methods

Public Methods

- init(): This method initializes all of the internal state. This method takes an optional parameter that specifies the maximum size that the cache may grow.
- add(): This method takes a template type ¡T¿ and adds it to the internal structures.
- remove(): This method removes an object from the cache.
- size(): This method returns the current size of the cache.
- maxSize(): This method returns the upper limit on how large the cache may grow.
- hit(): This method is used to lookup an entry in the cache and move it to the most recently used if it exists.

7.2 Member Variables

- m_oList
- m_iMax
- m_iCurrentSize
- m_oMap

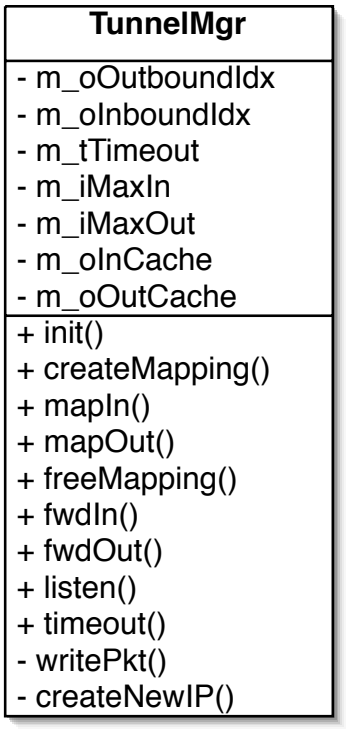

Figure 6:

8 TunnelMgr

This section describes the TunnelMgr component, which is described by Figure 6. This component maintains a cache of inbound and outbound connections. These caches are instantiations of the LRU Cache class (described in Section 7). The connections are placed in caches so that if connections are not properly cleaned up, the manager will not continue to maintain stale connection mappings.

As outbound connections are made new mappings are created via the createMapping() method. The results are then placed in the outbound connection cache, and referenced by the outbound index (which is an STL map). Similarly, inbound connections are managed by the inbound analogs. These structures are managed separately so that audit trails can be maintained about active inbound vs. outbound connections, and so that logic can (potentially) be different when handling them. Caches are filled with TunnelEntry objects that fully describe the headers needed to encapsulate outbound packets, and the local IP address needed to forward inbound packets.

The main thread of the manager blocks on the listen() method which awaits

incoming connections and outgoing packets (via a select() call). Outbound packets are passed the the fwdOut() method which creates the proper headers and encapsulates the packet. Inbound connections are handed to the fwdIn() method which tries to interpret the inbound headers and look them up in the inbound cache.

8.1 Methods

Public Methods

- init(): Initialize the internal state of the manager and clean up all data structures.
- createMapping(): Create a new tunnel. Map a new IP address to a set of TunnelEntry objects that specify the headers needed to create the tunnel. Place this mapping in the appropriate IP lookup cache and index.
- mapIn(): Take an inbound packet and determine which internal IP address it belongs to and retrieve all state needed.
- mapOut(): Take an outbound packet and determine what headers are needed to deliver it to the proper destination and retrieve all state needed.
- freeMapping(): Clean up state information from index and cache for a specific mapping (IP and tunnel headers).
- fwdIn(): Take an inbound packet, lookup its mapping info, de-encapsulate, and format the packet for local delivery.
- fwdOut(): Take an outbound packet, lookup its mapping info, encapsulate it for tunneling to a remote destination.
- listen(): Await inbound connections on a network port.
- timeout(): Check to see if any tunnels have been inactive for long enough that we can reclaim and free their state.

Private Methods

- writePkt(): Given a set of headers or a new IP address encapsulate of de-encapsulate and return a routable packet.
- createNewIP(): Perform the platform specific operations of allocating a new IP address in the local subnet.

8.2 Member Variables

- m_oOutboundIdx: ¡placeholder¿
- m_oInboundIdx: ¡placeholder¿

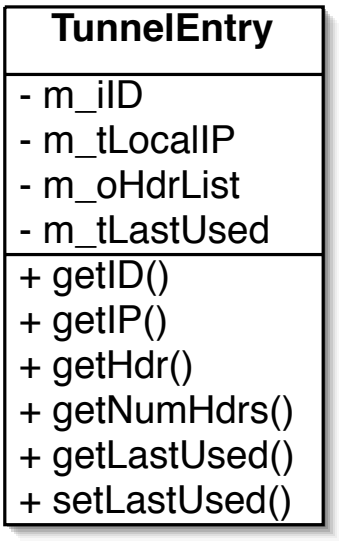

Figure 7:

- m_tTimeout: time_t - This member is the amount of time an entry is allowed to be idle before it can be considered for removal.
- m_iMaxIn: int - This member is the maximum number of inbound mappings that may be kept.
- m_iMaxOut: int - This is the maximum number of outbound mappings that may be kept.
- m_oInCache: LruCache - This member maintains a lookup of TunnelEntry objects based on the IP address the inbound packet came from.
- m_oOutCache: LruCache - This member maintains a lookup of TunnelEntry objects based on the IP address the outbound packet came from.

9 TunnelEntry

This section describes the TunnelEntry component, which is described by Figure 7. This is a very simple abstract data type (ADT). It encapsulates the set of headers needed to encapsulate an outbound IP packet, and the local address used to deliver traffic to a local client. The goal of this class is to simple wrap the logic that determines how many headers (i.e. levels of NAT) are needed and to actually encapsulate and de-encapsulate packets.

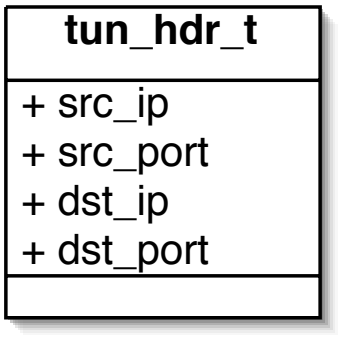

Figure 8:

9.1 Methods

Public Methods

- getID(): Simple accessor that returns the ID of this entry.
- getIP(): Simple accessor that returns the local IP address that corresponds to this tunnel.
- getHdr(): Takes an ordinal postion and returns the header there.
- getNumHdrs(): Returns the number of headers.
- getLastUsed(): Returns the time of the last access.
- setLastUsed(): Sets the last access time.

9.2 Member Variables

- m_iID: int - Internal ID of this entry.
- m_uLocalIP: unit32_t - Local IP address of this tunnel.
- m_oHdrList: list¡¿ - List of headers to enapsulate packets for this tunnel.
- m_tLastUsed: time_t - Time of last use.

10 tun_hdr_t

This section describes the tun_hdr_t struct, which is described by Figure 8. This component is a very simple data structure that defines a single header needed to encapsulate a single layer of a NAT3 tunnel. For example, with a single NAT box, 2 tun_hdr_t structures are needed (one for the outermost IP header, and one for the internal header).

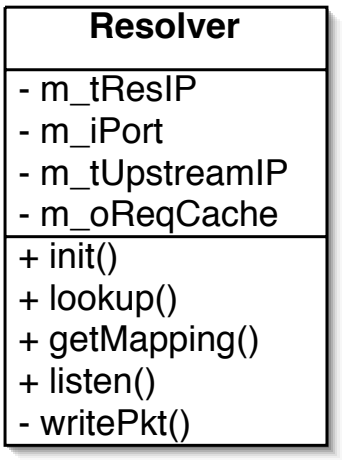

Figure 9:

10.1 Member Variables

- m_uSrcIP: uint32_t
- m_uSrcPort: uint16_t
- m_uDstIP: uint32_t
- m_uDstPort: uint16_t

11 DnsResolver

This section describes the DnsResolver component, which is described by Figure 9.

The resolver is a very loose wrapper around the DNS library and the DnsQuery class. The resolver's main task is to read bytes from the wire to feed into the DNS library which returns fully-parsed objects representing those DNS queries. It then feeds those queries into objects of the DnsQuery class, which will be described in a later section. That class will have the resolver send further packets as required by the NAT3-specific DNS logic.

11.1 Methods

Public Mehtods

- init(): This method reads the system resolv.conf file and prepares the object to accept DNS queries.

- listen(): This method edits the resolv.conf file to interpose the NAT3 resolver between the system resolver libraries and the local caching resolver. It causes the daemon to bind to the DNS socket and calls read_loop() to accept DNS queries.

Private Methods

- select_loop(): This method is a basic select() loop, which calls read_loop() and write_loop() depending on the read/write state of the file descriptor.
- read_loop(): When the DNS socket is ready to be read, this method continues to read packets from it until the recvfrom system call would block. Each time it reads a packet, it calls the DNS library to parse it and hands the parsed packet to read_packet() to handle it.
- write_loop(): If there is a queue of DNS packets waiting to be sent and the UDP socket becomes ready to write, this method will dump as many packets from the queue as it can before the socket would block.
- read_packet(): Once a properly parsed packet is received, this method handles it. This is the method acts as a marshaller between DnsQuery objects and the DNS packets from the wire.
- try_send(): This method will attempt to send a DNS packet if there is no packet queue. It will enqueue it if there is any problem sending.
- enqueue(): Add packet to queue for sending.
- writePkt(): Once the socket becomes ready to write, this method is used to write enqueued data.

11.2 Member Variables

- m_uResIP: uint32_t - IP address of IP to listen on
- m_uPort: uint16_t - Port to listen on
- m_iFD: int - File descriptor used for network.
- m_uUpstreamIP: uit32_t - IP of upstream caching resolver.
- m_oReqCache: LruCache - Cache of outstanding queries.

12 DnsQuery

This section describes the DnsQuery component, which is described by Figure 10.

The DnsQuery component encapsulates the NAT3-specific DNS logic needed by this application. It responds to error replies to questions for A resource

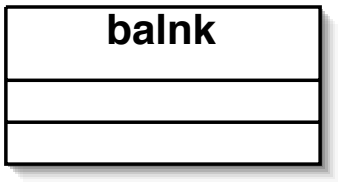

Figure 10:

records by generating questions for NAT3 records. If those are properly received, it asks the TunnelManager to set up a mapping. It will then construct a DNS response with an A record for this local mapping. It returns both this DNS response and the mapping itself to the resolver.

12.1 Methods

Public Methods

- constructor: The constructor takes a socket address and a question packet and encapsulates them.
- add_response(): This method takes a parsed DNS packet that represents a response to a previous packet. The DnsQuery object then uses the data in this packet to create a "next packet" to send, encapsulating the high-level NAT3-specific DNS logic.
- next_packet(): This method returns a wire-encoded DNS message and a destination socket address representing the next packet to be sent.
- get_mapping(): This methods returns the mapping given by the Tunnel-Manager to the DNS resolver if one has been made.
- done(): True when a query has been completed.

Private Methods

- get_A_RR(): Returns the first A resource record from the question section of the input packet, if one exists.

12.2 Member Variables

- m_src: Socket address representing the source of the original packet.
- m_packet: Original DNS reply packet that was received from the caching resolver.
- m_fwd_packet: Packet to be forwarded to the caching resolver or original questioner.

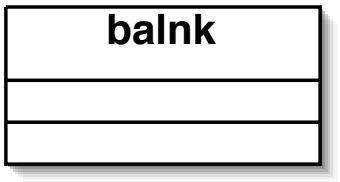

Figure 11:

- m_orig_error: Original error packet (i.e., NXDOMAIN) in case no NAT3 RR could be found.
- m_done: Flag used by done() method.

13 DnsPacket

This section describes the component, which is described by Figure 11.

This component represents a high-level view of a DNS packet. It encapsulates all the header information as well as the RRs that were received from the wire.

13.1 Methods

Public Methods

- add_bytes(): This method is called by the resolver when bytes have been received from the network. It can be called as many times as necessary and will continue to add bytes to the buffer. For a UDP packet, this will only be called once. However, this can be called arbitrarily many times for a TCP packet.
- parse(): Once all the data has been received from the wire, calling this method parses the packet and returns a boolean value depending on the success or failure of the parse.
- to_wire(): This method converts the message from high-level objects into the canonical wire representation of a DNS packet.
- header(): This method returns the header section of the DNS packet.
- questions(): This method returns a list of the question RRs from the packet.

Private Methods

- clear_vectors(): This method frees the memory associated with the RRs and is used by the destructor.

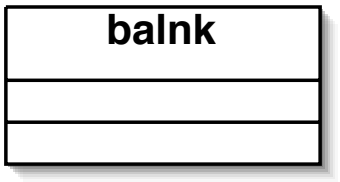

Figure 12:

13.2 Member Variables

- m_header: A DnsHeader object representing the parsed headers.
- m_qd: Question RRs from the original DNS packet.
- m_an: Answer RRs from the original DNS packet.
- m_ns: Authoritative NS RRs from the original DNS packet.
- m_ar: Additional RRs from the original DNS packet.
- m_bytes: Buffer of bytes from the network.

14 DnsHeader

This section describes the DnsHeader component, which is described by Figure 12.

This class holds the header information for a DNS packet, providing convenience methods for various data.

14.1 Methods

Public Methods

- init(): Loads the header from a wire representation.
- id(): Get the ID of the query.
- qd_count(): Get the query count.
- an_count(): Get the answer count.
- ns_count(): Get the authoritative NS count.
- ar_count(): Get the additional count.
- response(): Is this a response?

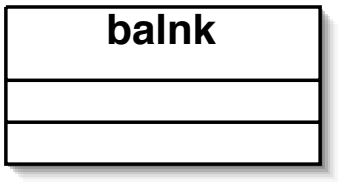

Figure 13:

- rcode(): Get the RCODE.
- to_wire(): Convert to the wire format.

Private Methods

- pack_flags(): Pack flags into wire representation.
- unpack_flags(): Unpack flags into a common C-style structure.

14.2 Member Variables

- m_init: True if packet has been initialized.
- query_id: Query ID of the packet.
- flags: Header flags.
- qdcount: Question count.
- ancount: Answer count.
- nscount: NS count.
- arcount: Additional count.

15 DnsName

This section describes the DnsName component, which is described by Figure 13.

This class implements the semantics of a DNS name. Due to the use of DNS-style compression, handling of DNS names is non-trivial.

15.1 Methods

Public Methods

- from_wire(): A factory method which parses a name from the wire and returns an object of the DnsName class on successful parse.

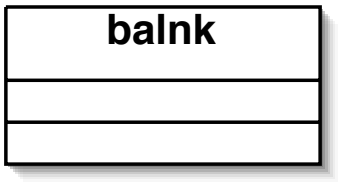

Figure 14:

- to_wire(): Takes a name length and an optional pointer and converts the name into a compressed wire representation.

Private Methods

- empty_list(): Frees memory associated with the object.
- read_name(): Reads a name from a packet, handling compression as needed.

15.2 Member Variables

- m_parts: List of the period-delimited parts of the DNS name.
- m_length: Length of the uncompressed name.

16 DnsRR

This section describes the component, which is described by Figure 14.

This represents an RR of any type for which we do not have a specific implementation. It handles the high-level functions of a DNS RR packet.

16.1 Methods

Public Methods

- parse(): Takes a wire representation of a DNS packet and parses it into an RR. This is a factory method, and will return an object inheriting from DnsRR depending on the type of the RR.
- type(): Returns the numeric type of the RR. A list of symbolic constants for common types is included.
- rdata_valid(): Returns true if the RDATA is of a valid format (which depends on the underlying RR type).
- set_rdata(): Set the RDATA section to a bag of bytes.

Private Methods

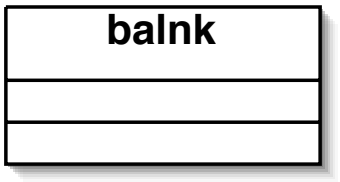

Figure 15:

- factory(): Static factory method that returns a new RR of a numeric type, used by parse_question_rr() and parse_normal_rr().
- parse_question_rr(): Parse an RR from the question section, returns an RR of the proper type if no error occurs.
- parse_normal_rr(): Parse an RR from any section other than the questions section. Returns RR of the proper type if no error occurs.

16.2 Member Variables

- m_init: True if the object has been init.
- m_name: DnsName object representing the name of the packet.
- m_type: Numeric type of the RR.
- m_class: RR class (as defined by RFC1035).
- m_ttl: TTL of RR (-1 if question RR).
- m_rdata: Binary RDATA.
- m_rdlen: Length of RDATA.

17 DnsA

This section describes the DnsA component, which is described by Figure 15.

The DnsA is a subclass of DnsRR which implements a common A resource record.

17.1 Methods

Public Methods

- rdata_valid(): Check if the data is valid
- ip(): Get the IP
- set_ip(): Set the IP.

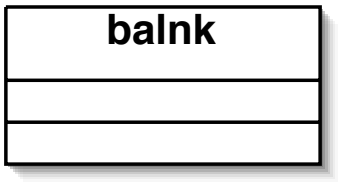

Figure 16:

18 DnsCompression

This section describes the DnsCompression component, which is described by Figure **??**.

DNS name compression is tricky business, and this class aims to simplify it by providing a compression context.

18.1 Methods

Public Methods

- add_name(): This method takes a name and returns the number of leading name parts that remain after compression. It also returns a pointer if any compression occurs. It records metadata for use in future add_name() method calls.

18.2 Member Variables

- m_names: A mapping of names and sub-names to DNS pointers. For the purposes of compression, the names are actually stored backwards.