A. Cerpa
USC
J. Elson
USC
H. Beheshti
Radware, Inc.
A. Chankhunthod
Network Appliance, Inc.
P. Danzig
Akamai Technologies
R. Jalan
Foundry Networks
C. Neerdaels
Inktomi Corporation
T. Schroeder
Alteon Web Systems
G. Tomlinson
Novell Inc.
March 15, 2000

NECP
the
Network Element Control Protocol
draft-cerpa-necp-03.txt

STATUS OF THIS MEMO

ABSTRACT

This document describes "NECP", a lightweight protocol for signalling between servers and the network elements that forward traffic to them. It is intended for use in a wide variety of server applications, including for origin servers, proxies, and interception proxies. The network element may also be a range of devices, including so-called L4 or content-aware switches and load-balancing routers.

NECP provides methods for network elements to learn about servers' capabilities, availability, and hints as to which flows can and can not be serviced. This allows network elements to perform load balancing across a farm of servers, redirection to interception proxies, and cut-through of flows that can not be served by the farm.

TABLE OF CONTENTS

1 Introduction

Scalable server farms with intelligent support from elements of the network infrastructure are becoming increasingly prevalent in today's Internet. The servers may be simple farms of origin servers or proxies, or more complex configurations such as interception proxies as described below. The intelligent network elements may also be any of a range of devices, including so-called L4 switches, content-aware switches, and load-balancing routers.

Before NECP, manual configuration or proprietary and mutually incompatible protocols were the only way for network elements to learn about servers' capabilities, availability, or hints as to which flows they could or could not service. NECP now provides a standard server-to-network signalling interface. This allows network elements in heterogenous environments to perform load

balancing across a server farm, redirection to interception proxies, and cut-through of flows that can not be served by the farm.

Cut-through is particularly important for interception proxies. An interception proxy is generally one that services a client's request for some service from somewhere inside the network, even though the client thinks that it is connected directly to an origin server at a network endpoint. This is often useful in that it allows network managers to realize the benefits of local proxies without requiring that all local clients configure their software to be aware of the proxy. However, it can cause problems: users have no way to go directly to origin servers, as may be required in some cases (e.g., servers that authenticate using a client's source IP address). The proxy has a high-level understanding of the application protocol; it can detect these cases and decide which flows should be cut through to origin servers. However, it is the network element that implements the forwarding policy. Thus, signalling between the proxy and the network element is required. NECP provides a standard method for this signalling.

NECP does not dictate or even address specific load balancing policies. In contrast, it provides a method for network elements to gather the data required to make logical load balancing decisions. The algorithms used to assign user requests to servers remain entirely in the purview of the network elements. We expect that this will continue to be an area of innovation among vendors and a basis for differentiation of competing products.

Section 2 of this document defines some of the terms used herein. Section 3 gives a general overview of NECP, and its relationship to other protocols is described in Section 4. Section 5 contains the canonical and complete description of NECP's semantics. Section 6 describes the motivations behind many of our design decisions; however, in contrast to the previous Section, it does not specify any part of the protocol or carry the same "force of law". We conclude in Section 7 with some implementation hints.

2 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [1].

The special terminology used in this document is defined below. The majority of these terms are taken as-is from RFC 2616 [4] and "Internet Web Replication and Caching Taxonomy" [3], and are included for here for reference. Additional standardized terminology is also given in those references.

APPLICATION PROTOCOL
A set of rules describing how to transmit information between

clients and servers. In this context, all application protocols are assumed to make use of IP [9], the Internet Protocol. A transport protocol identifier and (if applicable) a TCP or UDP port number uniquely identify an application protocol and its associated requests. Examples are HTTP (TCP port 80), FTP (TCP ports 20 and 21), DNS (TCP/UDP port 53), and RTSP (TCP/UDP port 554).

SERVER (as defined in [4])
An application program that accepts connections in order to service requests by sending back responses. Any given program may be capable of being both a client and a server; our use of these terms refers only to the role being performed by the program for a particular connection, rather than to the program's capabilities in general. Likewise, any server may act as an origin server, proxy, gateway, or tunnel, switching behavior based on the nature of each request.

CLIENT (as defined in [4])
A program that establishes connections for the purpose of sending requests.

FLOW
An IP session between two IP endpoints; often TCP connection, but possibly any other IP protocol. A single connection between a server and a client, for example, is a flow.

PROXY (as defined in [3])
An intermediary program which acts as both a server and a client for the purpose of making requests on behalf of other clients. Requests are serviced internally or by passing them on, with possible translation, to other servers. A proxy MUST implement both the client and server requirements of this specification. A "transparent proxy" is a proxy that does not modify the request or response beyond what is required for proxy authentication and identification. A "non-transparent proxy" is a proxy that modifies the request or response in order to provide some added service to the user agent, such as group annotation services, media type transformation, protocol reduction, or anonymity filtering. Except where either transparent or non-transparent behavior is explicitly stated, the HTTP proxy requirements apply to both types of proxies.

Note: The term "transparent proxy" refers to a semantically transparent proxy as described in [1], not what is commonly understood within the caching community. We recommend that the term "transparent proxy" is always prefixed to avoid confusion (e.g. "network transparent proxy"). However, see definition of "interception proxy" below.

The above condition requiring implementation of both the server and client requirements of HTTP/1.1 is only appropriate for a non-network transparent proxy.

INTERCEPTION PROXY (defined in more detail in [3])
Interception proxies receive inbound traffic flows through the process of traffic redirection. (Such proxies are deployed by network administrators to facilitate or require the use of appropriate services offered by the proxy). Problems associated with the deployment of interception proxies are described in the companion document to [3], "Known HTTP Proxy/Caching Problems". The use of interception proxies requires zero configuration of the client, and the client acts as though it is communicating with an origin server.

SERVER ELEMENT
A general term that involves any device in the network offering a service to a client. This involves servers, proxies, and interception proxies as defined above.

In this document, the acronym "SE" will be used in place of "Server Element".

NETWORK ELEMENT
An IP packet-forwarding device in the network. A Network Element can recognize flows that belong to a specific application protocols based on their transport protocol and destination port number, as described above. The network element is what does the interception of client-to-server requests on behalf of an interception proxy. Examples are so-called "L4 switches", "content-aware switches", and "load-balancing routers".

In this document, the acronym "NE" will be used in place of "network element".

Note that the term "network element" as defined by this document is narrower in scope than that described in [3].

PEER
One of a pair of cooperating network entities for which there is no clearly defined master/server, or slave/client. That is, each peer can both send requests and service requests. In NECP, server elements and network elements are peers.

CACHE (as given in [3])
A program's local store of response messages and the subsystem that controls its message storage, retrieval, and deletion. A cache stores cacheable responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. (Note that there are additional motivations for caching such as to reduce server load or increase availability.) Any client or server may include a cache, though a cache cannot be used by a server that is acting as a tunnel.

The term "cache" used alone often means "caching proxy".

3 Protocol Overview

NECP is used to exchange information and control messages between a server element (SE) and a network element (NE). The SE is configured with the IP address used for management of the NE. When the SE is initialized (for example, when it boots), it establishes a TCP connection to the NE using a well-known port number. Messages can then be sent, bi-directionally, between the SE and NE. Application level KEEPALIVE messages are exchanged so that a dead peer can be more quickly detected.

All messages consist of a fixed-length header and variable length data. The fixed-length header contains the total length of the data so that the incoming TCP stream can be quickly re-segmented into its constituent messages.

At the application layer, most messages consist of a request followed by a reply or acknowledgement. TCP handles network-layer retransmissions of messages in the case of lost packets, so the application need only transmit each request once. This protocol uses the "hard state" model; that is to say, if a positive acknowledgement is received that implies some state has been installed in the peer, that state can be assumed to remain in the peer until it expires or the peer crashes. A crashed peer can be detected using NECP KEEPALIVE messages in addition to errors that come from the TCP layer. When a node detects that its peer has crashed, it should assume that all state in that peer needs to be reinstalled after the peer revives.

Details of the packet format and the exact semantics of the messages exchanged will be described in the next section. However, in the remainder of this section, we will give a general overview of the types of messages exchanged:

"I'm here and ready for work."
: This message is sent to an NE by an SE when it comes on-line or has become ready to service requests. It lets the NE know that the network element can start forwarding requests to the SE.

"Stop sending me work."
: An SE can request that an NE stop sending it new requests from clients. This can be useful in various situations; for example, when an SE is overloaded, or when an SE wants to shut down for maintenance after completing all currently outstanding requests.

"A client needs a direct connection for future requests."
: Sometimes, an interception proxy fails in satisfying a client's request that would have succeeded if the client had been talking directly to the server. For example, a server might authenticate clients based on their source IP address; it might have allowed the client but does not allow interception proxies. In these cases, the interception proxy SE can ask the NE to allow that particular client's requests to pass directly to the Internet instead of being diverted to the SE. Then, the client

can be asked to retry its request, if the application protocol in use allows such a message. For example, in HTTP, an HTTP Redirect redirecting the client back to the same URL might be used.

4 Related Protocols

This section will describe various protocols that are related to NECP, and the nature of the relationship.

WPAD [6], the Web Proxy Auto-Discovery Protocol (work in progress), can be used by clients to learn of the existence of an appropriate proxy that is willing to service their requests. An NE that uses NECP to learn of active SEs may also give that information directly to clients using WPAD.

ICP [13], the Internet Cache Protocol, is a "de-facto standard" used by some caches to cooperate in the formation of a cache hierarchy. Caches use ICP to query sibling and parent caches to determine if objects have been cached there. This is complementary to NECP, which governs the signalling between a proxy (such as a cache) and a network element. NECP could conceivably run in parallel with ICP.

SNMP [2], the Simple Network Management Protocol, is a general protocol used for management of Internet nodes. The functionality of NECP could have been implemented as a new SNMP MIB instead of an entirely new protocol. Indeed, some of our original design efforts were directed towards an SNMP version of NECP. However, we later settled on a TCP-based protocol after we found ourselves adding features to the SNMP version that already existed in TCP. Section 6.2 explains in more detail our reasons for not using SNMP.

HTTP [4], the Hypertext Transfer Protocol, and RTSP [12], the Real Time Streaming Protocol, are two examples of widely deployed protocols used by clients to request content from servers. These protocols are not directly related to NECP, although their design does, of course, have a profound impact on the design of proxies.

5 Specification

5.1 Transport and Basic Protocol Operation

NECP uses TCP as a transport protocol. The NE listens on well-known port 3262. The SE is responsible for active opens to the NE. The exact initialization procedure is described in Section 5.4.

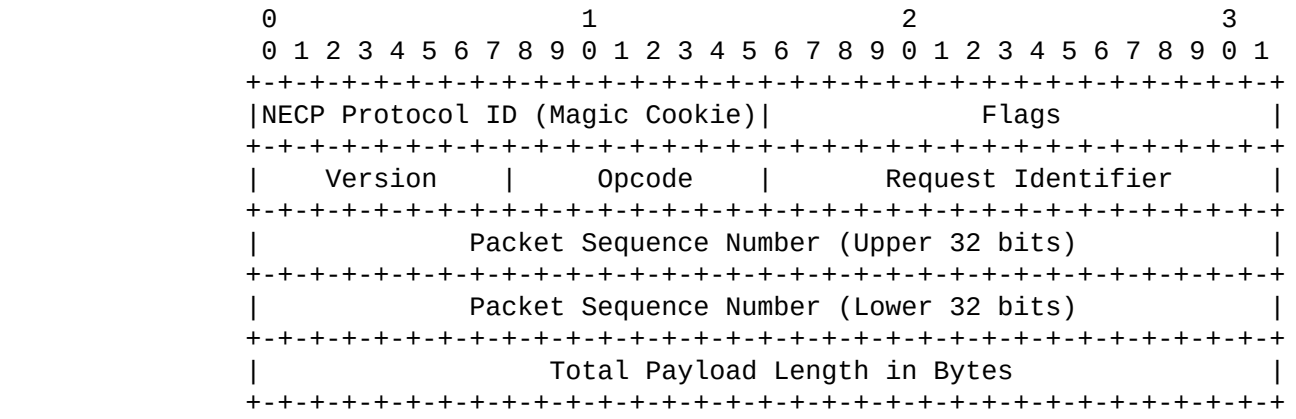
5.2 Packet Format

NECP messages always begin with a fixed-length, 20-byte header.

The payload is variable length, and can be 0. The length of the payload is indicated in the header. All data are in network byte order (big endian).

The version number of the protocol defined by this standard is 1.

5.2.1 Header Format




```
typedef struct {
   /* We'll start with an identifier so we can check for
    * correct protocol format, framing, etc. */
   U16   protocol_identifier = 0x414a /* ("AJ") */

   /* 16-bit flags field */
   U16   flags;
            0x0001    F_Basic_Payload (see Section 5.2.3)
            0x0002    F_Auth_Credential_Provided (see Section 5.8)

            0x0004    F_Error
            0x0008    F_Protocol_Version_Mismatch
                      (see Section 5.2.1)
            0x0010    F_Auth_Required (see Section 5.8)
            0x0020    F_Bad_Sequence_Number (see Section 5.9)

   /* Protocol version number -- always 0x1 */
   U8    version;

   /* Message type (see Section 5.3) */
   U8    opcode;

   /* Number used to match up requests and responses (see
    * Section 5.2.2)
   U16   request_id;

   /* Monotonically increasing sequence number; used to
    * prevent replay attacks (see Section 5.9) */
   U64   seq_num;

  /* The length of the payload only -- might be 0 */
   U32   payload_len;
} hdr_t
```

5.2.2 Header Field Descriptions

protocol_identifier: MUST contain the value 0x414a. Any incoming message whose protocol identifier field does not contain this value MUST be discarded. The protocol identifier is a "magic cookie" that is useful for detecting stream framing errors.

version: Indicates the version number of the protocol in use. All implementations that adhere to the standard specified in this document MUST set the version number to 0x1. Behavior in the case of version mismatches is described below.

opcode: Specifies the operation that this message requests. Section 5.3 contains the opcode table. Semantics of the opcodes are described in detail later in this document.

request_id: Contains a number that identifies this transaction. NECP operations typically consist of a request followed by a response. The request_id helps requesters match incoming responses to their currently outstanding requests. A node that is sending a request SHOULD select a request_id that is distinct from the IDs of all its other currently outstanding requests. The requester may use any convenient algorithm for the selection of these identifiers, and they may be reused frequently. Correspondingly, a responding node MUST include the same request_id in its response. The identifier has no meaning to the responding node and should not be treated as a sequence number.

seq_num: Contains the sequence number of the message (used only for authenticated connections); its use is described in Section 5.9.

payload_length: indicates the total length of the payload in bytes. The payload length does not include the header, which is 20 bytes. The payload length can be 0. The payload format is described in the next section.

flags: Indications of various binary conditions.

F_Basic_Payload: Indicates the payload type as described in Section 5.2.3.

F_Error: Potentially used by all response opcodes to indicate that some or all of the corresponding request was not satisfied. For some types of errors, F_Error can be used in conjunction with F_Protocol_Version_Mismatch (see below), F_Auth_Required (see Section 5.8), and F_Bad_Sequence_Number (see Section 5.9). Some reply opcodes will also send additional error indications in the message's payload, as described in detail in future sections.

F_Version_Mismatch: Used to tell a peer "I don't know how to speak your version of the protocol." The protocol version in use is advertised in each message's "version" header field, as described above. A node that receives any NECP message with a version number that it does not understand MUST send a reply with the F_Error and F_Protocol_Version_Mismatch flags set, and the "version" field set to the highest version number that the node is capable of accepting. This type of reply MUST NOT have a payload.

F_Auth_Credential_Provided, F_Auth_Required, and F_Bad_Sequence_Number are all used for authenticated connections and are described in Sections 5.8 and 5.9.

5.2.3 Payload Format

There are two payload types: a "Basic" payload (F_Basic_Payload set in the header flags), and a "Variable Length" payload (F_Basic_Payload cleared).

If the NECP connection is authenticated, each message's payload MUST be followed by a credential, as described in Section 5.8.

5.2.3.1 Basic Payloads

The message is known to have a basic payload if its header has the F_Basic_Payload flag set. A basic payload is made up of one or more "basic payload units". Each basic payload unit is a fixed size (32-byte) structure consisting of 8 4-byte integers. The number of basic payload units contained in the message can be determined by dividing the total length of payload indicated in the header by the number of bytes per basic payload unit (32). Note that authenticated connections have an extra 20 bytes after the basic payload units that must be removed before this calculation; see Section 5.8.2 for details.

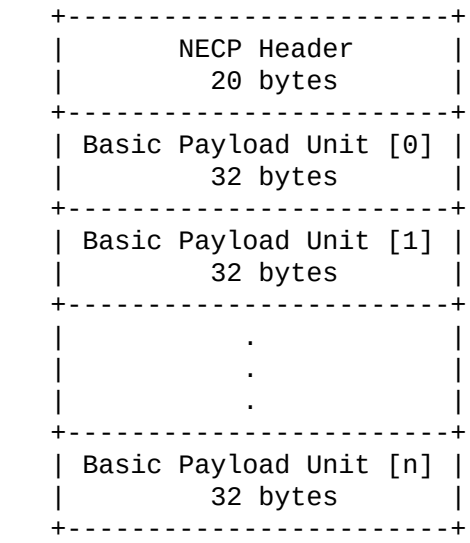
The meaning of the 8 integers in each basic payload unit depends on the message type (opcode). Not all opcodes will use all data positions; unused data positions MUST be set to 0.

Format of a basic payload unit:

```
typedef struct {
   U32     data0;
   U32     data1;
   U32     data2;
   U32     data3;
   U32     data4;
   U32     data5;
```

```
   U32     data6;
   U32     data7;
} payload_unit_t;
```

As mentioned earlier, a Basic Payload consists of one or more 32-byte Basic Payload Units (as defined above). A message containing a Basic Payload therefore has the following general form:



5.2.3.2 Variable Length Payloads

This type of payload has no structure imposed by this standard. Variable length payloads can be used by future extensions of NECP to transmit arbitrarily formatted data of any length up to 2**32-1 (the maximum size representable by the 32-bit payload length in the header). Future extensions of NECP can define opcode-specific data formats to suit their needs using the variable length binary payload option.

5.3 Control Message Identifier Table

The following is a list of message types (opcodes) used by NECP. An OPCODE is a control message identifier that is included in the message header (see Section 5.2.1) to identify the type of control message.

```
NECP_NOOP                   0x00  /* NO OPeration -- do nothing */
NECP_INIT                   0x01
NECP_INIT_ACK               0x02
NECP_KEEPALIVE              0x03
NECP_KEEPALIVE_ACK          0x04
NECP_START                  0x05
NECP_START_ACK              0x06
```

NECP_STOP	0x07
NECP_STOP_ACK	0x08

[0x09 - 0x19 Reserved for future use]

NECP_EXCEPTION_ADD	0x20
NECP_EXCEPTION_ADD_ACK	0x21
NECP_EXCEPTION_DEL	0x22
NECP_EXCEPTION_DEL_ACK	0x23
NECP_EXCEPTION_RESET	0x24
NECP_EXCEPTION_RESET_ACK	0x25
NECP_EXCEPTION_QUERY	0x26
NECP_EXCEPTION_RESP	0x27

[0x28 - 0xFF Reserved for future use]

The following sections describe in detail the semantics of the different opcodes.

NECP_NOOP is a "No Operation" opcode that can be ignored. It primarily exists for testing and to prevent accidentally zeroed-out packets from having an effect.

5.4 Initialization Procedure

When an NE becomes ready for service, it MUST passively listen for TCP connections on NECP's well-known port (see Section 5.1).

When an SE becomes ready for service, it establishes a connection to its associated NE. NECP is not used for peer discovery; an SE is expected to know the NE's address a priori (for example, through manual configuration). Once the TCP connection has been established, the SE MUST send an NECP_INIT message to the NE, and wait for an NECP_INIT_ACK message from the NE in response.

Note that NECP initialization is asymmetrical. SEs always perform active TCP opens and send NECP_INIT; NEs always passively listen for connections from SEs, and respond with NECP_INIT_ACK.

When an SE tries to establish a connection to an NE, the NE might not be available. In some cases, the TCP connection to the NE might fail (for example, due to a timeout, or an NE that is up but not listening on the NECP port). Another possibility is that the TCP connection will be successfully established, but the NECP_INIT message sent by the SE is not met with a NECP_INIT_ACK reply. In cases such as these where the NE is unavailable, the SE MAY periodically retry the connection. In other words, the SE MAY periodically retry its attempt to initiate a TCP connection; and, after a TCP connection has been established, MAY periodically send NECP_INIT messages until a response is received.

The retry interval for both of these operations SHOULD have an exponential backoff. The suggested limits of the exponential backoff are between [1 - 256] seconds, although different limits may be provided depending on users' needs. For instance, if waiting 256 seconds for the SE(s) to reconnect to the NE is too long in some environment, the administrator SHOULD be able to reduce the upper limit. An upper limit of 1 second means that the client does not want exponential backoff and the SE(s) retry every second.

This "hard state" protocol assumes that all the state in an NE needs to be refreshed after a crash. When an SE sends a NECP_INIT to an NE, the NE SHOULD delete all state associated with that SE. Similarly, the SE MUST assume that the NE has no installed state.

It is important to note that the initialization procedure opens a channel for control messages, but does not start the forwarding of client application requests from the NE to the SE. The default condition after initialization is "stop" (see Section 5.6). Redirection of packets will occur only after an SE sends a "start" flow control message to the NE (see Section 5.6). One reason for this default is that it allows the creation of a control channel from a node that will *never* receive actual work, such as a management interface. Further motivations for this decision are given in Section 6.6.

5.4.1 NECP_INIT SE->NE

An SE MUST send this message to an NE after opening a TCP connection to that NE.

The payload is a single basic payload unit of the following format:

- data0: Flags (32 bits).

  0x1: If set, all NECP messages on this connection MUST be authenticated as described in Section 5.8.

If data0 indicates that this is an authenticated connection, data1 and data2 MUST contain the SE's desired initial sequence number, as described in Section 5.9.

If no authentication is required, data1 and data2 MUST be 0.

5.4.2 NECP_INIT_ACK NE->SE

Upon receiving an NECP_INIT message from SE, an NE MUST respond with an NECP_INIT_ACK.

The payload is a single basic payload unit.

If data0 of the incoming NECP_INIT is set (i.e., indicates that this is an authenticated connection), data0 and data1 of the outgoing NECP_INIT_ACK MUST contain the NE's desired initial sequence number, as described in Section 5.9.

If data0 is not set (indicating an unauthenticated connection), but the NE does not want to use unauthenticated connections, the NE MAY return an NECP_INIT_ACK with F_Error and F_Auth_Required both set. In this case, both sides must immediately close the connection. The SE MAY retry the connection using authentication as described in the previous section.

If no authentication is required, data0 and data1 of the outgoing NECP_INIT_ACK MUST be 0.

5.5 Device Liveliness Detection and Performance Measurement

The purpose of the NECP_KEEPALIVE opcode and its reply, NECP_KEEPALIVE_ACK, is to allow either an NE or SE to determine the responsiveness and performance of its peer. The keepalive messages themselves provide the most basic, binary performance metric ("are you alive or not?"), but the messages may also carry more explicit and descriptive metrics. Nodes may use these performance metrics in any number of ways; for example, an NE may compare performance metrics from several SEs in a server farm in order to improve load balancing among them.

SEs and NEs both MUST periodically send NECP_KEEPALIVE messages to all their NECP control channels. The RECOMMENDED time interval is 5 seconds +/- a randomized interval from 0 to 1 second. NECP_KEEPALIVE queries should elicit NECP_KEEPALIVE_ACK responses, described in more detail below. If a node sends 3 consecutive NECP_KEEPALIVE queries to a peer without any responses, the node SHOULD consider its peer dead.

After a node declares a peer dead, it MUST close the NECP connection to that peer. The node MUST NOT use the "dead" connection for future control messages. If the node wishes to continue using the timed out peer, the connection to the peer MUST be reinitialized using the procedure described in Section 5.4.

5.5.1 Extended Performance Metrics

NECP_KEEPALIVE messages MAY contain one or more basic payload units, each of which contains a request for a particular performance metric. Each metric is identified by a unique 32-bit number called the Performance Query Type. These numbers will be assigned manually as new metrics are created.

An SE may simultaneously support more than one application

protocol, and its performance in servicing some of these protocols may differ from its performance for others at any given moment. Therefore, a performance query is fully specified with a tuple consisting of (IP Protocol, Port Number, Performance Query Type).

Not all nodes will be required to support all possible performance queries; thus, a method of reporting "unsupported performance query" is described below.

For reasons explained in Section 6.11, this standard only defines a single performance query type that MUST be implemented by all NECP nodes:

0x1 Health Index. A measurement of the "health" of the node in the range of 0-100 (decimal). 100 means the node is in "perfect health," meaning it is operating at its peak performance; that is to say, approaching the performance advertised in marketing literature. A Health Index of 1 means the node is on the verge of being unable to service new requests, but is still capable of doing so (albeit extremely poorly). A Health Index of 0 is a special value meaning that the node is completely unable to service requests and should not be sent any work until its Health Index increases to some value above 0.

Different implementations will likely have different implementation-specific methods of determining their health. For example, a node may choose to use the number of requests currently being serviced, available disk or memory resources, number of open file descriptors, measured quality of network connectivity, or any other means.

5.5.2 NECP_KEEPALIVE

This message is used by either an NE or SE to determine whether its peer is responsive. It can contain 0 or more basic payload units, each of which contains an extended performance query, in the following format:

- data0: Performance Query Type (32 bits)

- data1: Protocol ID (8 bits, the LSB). Identifies the next-layer protocol above IP used by the application protocol whose performance is being queried; for example, TCP (0x6) or UDP (0x11). The mapping of protocols to protocol numbers is the same as that described in [11].

- data2: Destination port number (16 bits, the 2 LSBs). Identifies the port number of the application protocol whose performance is being queried. This value has no meaning if data0 does not indicate TCP

or UDP.

5.5.3 NECP_KEEPALIVE_ACK

This message is used by the SE or NE to respond to a previous NECP_KEEPALIVE query. It can contain 0 or more basic payload units.

If the NECP_KEEPALIVE request did not contain any extended performance queries (i.e., had no basic payload units), the NECP_KEEPALIVE_ACK, correspondingly, MUST NOT contain any basic payload units.

If the NECP_KEEPALIVE request contained one or more extended performance queries, and the node supports every query in the request, the node MUST send an NECP_KEEPALIVE_ACK with the same number of basic payload units as were in the request. Each basic payload unit in the response MUST correspond to one of the queries that was requested. The basic payload units are in the following format:

- data0: Same as in 5.5.2.
- data1: Same as in 5.5.2.
- data2: Same as in 5.5.2.
- data3: Performance Query Response (32 bits). The definition of this value depends on the Performance Query Type.

If the NECP_KEEPALIVE request contained any performance queries that the node does not support, the F_Error flag in the NECP_KEEPALIVE_ACK message MUST be set. The payload MUST be a copy of all basic payload units that contained unsupported performance queries. A node that receives such an error indication SHOULD NOT send future NECP_KEEPALIVE queries containing extended performance query types that are not supported by its peer.

Responses to queries that are supported, if any, MUST NOT be included in an NECP_KEEPALIVE_ACK error message. In other words, the NECP_KEEPALIVE_ACK might contain valid performance query responses, or indications of unsupported query types, but never both. A node that receives an NECP_KEEPALIVE_ACK with an error indication MAY immediately send another NECP_KEEPALIVE query in order to retrieve the performance metrics that are supported by its peer.

5.6 Flow Control

The NECP_START and NECP_STOP opcodes that allow an SE to indicate to the NE its interest in having application-level protocol

requests directed to it. For example, NECP_START can be used to tell the NE "I am ready to have TCP port 80 requests forwarded to me." NECP_STOP, conversely, tells the NE to stop forwarding these flows.

It is very important to note that changes to the NE's list of forwarded protocols MUST only apply to future flows. Existing flows MUST NOT be affected by changes in the forwarding state. This implies that the NE MUST keep per-flow state, instead of matching packets against the current forwarding state as the packets arrive. Motivations for this requirement are described in Section 6.7.

5.6.1 NECP_START SE->NE

An SE sends this message to an NE to inform the NE that it is ready to service application-level protocol requests.

The payload consists of one or more basic payload units; each payload unit represents a request to start forwarding packets for a single application protocol. Each payload unit has the following information:

- data0: Forwarding Type (8 bits, the LSB). Indicates the type of packet forwarding requested by the SE.

  0x1: Layer 2 Forwarding. Packets are unchanged at the IP layer and above, and forwarded using the destination SE's datalink-layer address. In the common case of Ethernet, this means that the NE sends the unchanged IP datagrams using the SE's MAC address. This option MUST NOT be used if the NE and SE are not on the same datalink network; i.e., separated by a router.

  0x2: GRE Forwarding. Packets are unchanged at the IP layer and above, and forwarded from the NE to the SE using GRE encapsulation. The NE and SE may be on the same datalink network, or separated by a router.

  0x3: Layer 3 Forwarding. Packets are changed at the IP layer so that the real destination IP address of packets is the SE. (Other changes are also made as necessary; e.g. recomputation of checksums.) This option is useful for certain types of load balancing; for example, where the NE has a virtual IP address that represents a farm of SEs.

- data1: Protocol ID (8 bits, the LSB). Identifies the next-layer protocol above IP used by the application protocol; for example, TCP (0x6) or UDP (0x11). The

mapping of protocols to protocol numbers is the same as that described in [11].

- data2: Destination port number (16 bits, the 2 LSBs). Identifies the port number of the application protocol that the SE is ready to service. This value has no meaning if data1 does not indicate TCP or UDP.

5.6.2 NECP_START_ACK NE->SE

This message is used by an NE to confirm the action taken as a result of a previous NECP_START_ACK from an SE.

In the absence of errors, no payload is required and the F_Error Flag MUST be cleared. Recall that the Request Identifier field of the NECP header should allow an SE to match up its own NECP_START requests with an NE's NECP_START_ACK responses (see Section 5.2.2).

If an error occurred with one or more of the requests, the F_Error Flag in the message header MUST be set. The payload MUST be a copy of all the basic payload units that contained failed requests. Successful requests, if any, MUST NOT be included in this error message.

5.6.3 NECP_STOP SE->NE

Similar to the message described in 5.6.1, but used to stop requests instead of starting them.

5.6.4 NECP_STOP_ACK NE->SE

Similar to the message described in 5.6.2, but used to acknowledge stop requests instead of start requests.

5.7 Flow Exception List Management

NECP allows an SE to give an NE a list of (IP address, port number) tuples to indicate which flows are "exceptions." These opcodes allow the SE to maintain the NE's flow exception list; that is, add to, delete from, and query the list.

In this context, an "exception" is always a flow that should NOT be forwarded to the SE, even though it is in a class of flows that are currently being directed to SEs due to a previous NECP_START message. It is not possible to do the opposite; i.e., forward to the SE individual flows from a class of normally un-forwarded flows.

There are two types of exceptions that an SE can request:

Local Exception: "Don't forward this flow to me, but it's okay to forward it to other servers in the farm." An SE might use this if it is unable to service a flow due to (for example) a local resource shortage.

Global Exception: "Don't forward this flow to any server in the farm." An SE might use this if it believes that a flow can not be serviced by any SE in the farm; for example, if the forwarding to an interception proxy server is causing authentication problems.

The exception type is only advisory, not mandatory. An NE MAY choose to treat any exception as local even if the exception is marked as global by the SE. The motivation for this is described in Section 6.10.

It is very important to note that changes to the NE's flow exception list MUST only apply to future flows. Existing flows MUST NOT be affected by changes in the exception list. This implies that the NE MUST keep per-flow state, instead of matching packets against the current forwarding state as the packets arrive. Motivations for this requirement are described in Section 6.7.

5.7.1 NECP_EXCEPTION_ADD SE->NE

This message is used by an SE to add one or more entries to the NE's flow exception list.

The payload is basic. Each basic payload unit contains a single request, in the following format:

- data0: Scope Advisory (8 bits, the LSB):
  0x0 No Scope Indicated (NE discretion)
  0x1 Local Exception ("do not forward to me")
  0x2 Global Exception ("do not forward to farm")

- data1: TTL (32 bits). Time To Live in seconds; the length of time that the NE should honor this request before deleting it. Although this value is specified in seconds, the NE MAY round up to the next higher minute, or whatever reasonable time interval is convenient for the implementation.

  Special Value:
  0x0: Static. This item should never time out.

- data2: Source IP Address (32 bits).

  Special Value:
  0x0: Wild card. Apply this rule to all source

addresses.

- data3: Source Address Netmask (8 bits, the LSB). Gives the CIDR netmask applied to the Source Address. This value has no meaning if data2 is 0.

- data4: Destination IP Address (32 bits).

  Special Value:
  0x0: Wild card. Apply this rule to all destination addresses.

- data5: Destination Address Netmask (8 bits, the LSB). Gives the CIDR netmask applied to the Destination Address. This value has no meaning if data4 is 0.

- data6: Protocol ID (8 bits, the LSB). Identifies the next-layer protocol above IP; for example, TCP (0x6) or UDP (0x11). The mapping of protocols to protocol numbers is the same as that described in [11].

  Special Value:
  0x0: Wild card. Apply this rule to all IP protocols.

- data7: Destination port number (16 bits, the 2 LSBs). This value has no meaning if data7 does not indicate TCP or UDP.

  Special Value:
  0x0: Wild card. Apply this rule to all destination port numbers.

If an NE receives an NECP_EXCEPTION_ADD message requesting an entry in its flow exception list that already exists, the TTL of the entry in the list should be updated to reflect the value in the message just received. In other words, new entries overwrite old entries.

5.7.2 NECP_EXCEPTION_ADD_ACK NE->SE

This message is used by an NE to confirm the action taken as a result of a previous NECP_EXCEPTION_ADD from an SE.

In the absence of errors, no payload is required and the F_Error Flag MUST be cleared. Recall that the Request Identifier field of the NECP header should allow an SE to match up its own NECP_EXCEPTION_ADD requests with an NE's NECP_EXCEPTION_ADD_ACK responses (see Section 5.2.2).

If an error occurred with one or more of the requests, the F_Error Flag in the message header MUST be set. The payload MUST be a copy of all the basic payload units that contained

failed requests. Successful requests, if any, MUST NOT be included in this error message.

5.7.3 NECP_EXCEPTION_DEL SE->NE

This message is used by an SE to delete one or more entries from the NE's flow exception list. An SE can only delete its own entries from the NE's flow exception list; in other words, it can not delete exceptions that were installed by other SEs. An SE may delete any exception it has previously installed -- both local and global.

The payload and format is analogous to the one described in Section 5.7.1.

5.7.4 NECP_EXCEPTION_DEL_ACK NE->SE

This message is used by an NE to confirm the action taken as a result of a previous NECP_EXCEPTION_DEL from an SE.

In the absence of errors, no payload is required and the F_Error Flag MUST be cleared. Recall that the Request Identifier field of the NECP header should allow an SE to match up its own NECP_EXCEPTION_DEL requests with an NE's NECP_EXCEPTION_DEL_ACK responses (see Section 5.2.2).

If an error occurred with one or more of the requests, the F_Error Flag in the message header MUST be set. The payload MUST be a copy of all the basic payload units that contained failed requests. Successful requests, if any, MUST NOT be included in this error message.

5.7.5 NECP_EXCEPTION_RESET SE->NE

This message is used by the SE to reset (i.e., delete) the NE's entire flow exception list. The NE, upon receiving this opcode, MUST empty its list of flows that were excepted on behalf of the SE sending the request. No payload is required.

5.7.6 NECP_EXCEPTION_RESET_ACK NE->SE

This message is used by an NE to confirm the SE of a previous NECP_EXCEPTION_RESET. No payload is required.

5.7.7 NECP_EXCEPTION_QUERY SE->NE

This message is used by the SE to query the active flow exception list in an NE. The entire exception list is queried, i.e., all entries added by all SEs.

The payload consists of a single basic payload unit, similar (but not identical) in format to a basic payload unit described in Section 5.7.1, with the exception of data1 as

described below. This payload unit specifies a filter for the flow exception list query. The expected response from the NE will be all flow exception list entries that match the query. The fields that are valid for filtering are:

- data0: Scope advisory.
- data1: Installer's IP Address (32 bits). The IP address of the SE that installed this exception.
- data2: Source IP Address.
- data3: Source Address Netmask.
- data4: Destination IP Address.
- data5: Destination Address Netmask.
- data6: Protocol ID.
- data7: Destination Port Number.

A non-zero value in any of the valid fields indicates an attribute filter of the query. A zero value in a field indicates a wildcard that matches any flow exception list entry. This implies that query with a payload unit containing all zero-valued data fields MUST effect a query for all the entries in the NE's flow exception list.

5.7.8 NECP_EXCEPTION_RESP NE->SE

This message is used by the NE to respond to an NECP_EXCEPTION_QUERY. The payload is basic, containing 0 or more basic payload units. Each payload unit MUST contain an active IP list entry that matches the query given in the NECP_EXCEPTION_QUERY message. The response MUST contain a basic payload unit for each of the NE's active flow exception list entries. Note that if the NECP_EXCEPTION_QUERY had a zero value in the SE IP Address field (data1), the NE should send its entire flow exception list, including exceptions installed on behalf of other SEs.

The format of each basic payload unit is the same as the one described in Section 5.7.1, except that the data1 field indicates the IP address of the SE that installed the exception returned in the basic payload unit.

5.8 Authenticated Messages

NECP can use a "message authentication code" (MAC) on messages for authentication purposes. Authentication is based on a secret key shared by the NE and SE. The HMAC-SHA1 [5, 8] algorithm is used to compute a 160-bit digest over an NECP message and the shared secret. The result of the HMAC-SHA1 algorithm is the credential that authenticates the message. This document does not specify how the shared secret is distributed.

All NECP implementations MUST support the authenticated message format described in this section. However, nodes may opt to use

or not use the authenticated message format on a per-connection basis. The use of authentication is negotiated when the SE initiates the NECP connection (see Section 5.4). The NE MUST be able to support authenticated messages if the SE requests an authenticated connections. In addition, the NE MAY force the use of authentication by rejecting unauthenticated connections from the SE.

If a connection uses authentication, ALL messages in both directions MUST be authenticated; that is, they MUST contain a credential after the header and payload, and the NECP header MUST have the F_Auth_Credential_Provided flag set.

5.8.1 Generating a Credential

The credential of a message is the HMAC-SHA1 [5, 8] hash over the header, the payload, and the shared secret. We refer the reader to [5, 8] for details on how the authentication algorithm is used and the credential is computed. The output of HMAC-SHA1 is a 20-byte hash. This hash value is then appended to the header and other payload, thereby extending the payload by the size of the credential (20 bytes). The "payload_len" field of the header (Section 5.2) should be increased accordingly.

The hash MUST be computed AFTER payload_len is increased to reflect the size of the total (credential-bearing) packet, and AFTER F_Auth_Credential_Provided is set in the header.

The byte-length of the basic block of data used in the HMAC-SHA1 (parameter B) is set to 64 (B=64) [8].

In summary, the steps required to generate an authenticated message are:

1. Set F_Auth_Credential_Provided in the header flags, and increase the payload_len in the header by the size of the credential (20 bytes).
2. Calculate the HMAC-SHA1 hash over the message and shared secret.
3. Append the hash value at the end of the message.

5.8.2 Authenticating a Message

When an authenticated message is received (that is, a message that has F_Auth_Credential_Provided set in the header), the authentication is verified using the reverse of the procedure described in Section 5.8.1. Specifically:

1. Store the credential sent by the peer.
2. Calculate the HMAC-SHA1 hash over the message and the shared secret.
3. Compare the hash value with the credential sent by the peer.

4. Decrement the payload_len in the header by the size of the credential (20 bytes).

The message MUST be rejected, as described in the next section, if and only if the newly calculated hash does not match the credential sent by the peer.

In addition, if a message is received on an authenticated connection that does not have the F_Auth_Credential_Provided flag set, the message MUST be rejected as described in the next section.

5.8.3 Rejecting Messages whose Authentication Failed

If the message whose authentication failed was a request, the normal (opcode-specific) procedure for reporting a failed request MUST be followed, as described in previous sections. However, in addition to the F_Error flag that is always set on a response when reporting a failure, the F_Auth_Required flag MUST also be set.

If the message whose authentication failed was an acknowledgement of a request, the requester MUST assume that the request was not successfully executed and MAY retry the same operation again with the correct credential. After three consecutive failures, the requester SHOULD give up, and MAY report the authentication failure (e.g., via a system log). Note that due to the idempotent nature of NECP operations, retrying a request that actually succeeded the first time (e.g., if a requester receives and rejects an authentic positive acknowledgement due to a corrupted hash) should not be harmful.

5.9 Sequence Numbers

Although authenticated messages are resistant to forgery by an attacker, they are still vulnerable to replay attacks. In other words, an attacker might record a correctly authenticated messages transmitted by one of the nodes, and replay the same message back to the intended recipient some time later. Security against replay attacks is provided by monotonically increasing sequence numbers in the message header.

All NECP implementations MUST support the semantics of sequence numbers described in this section.

5.9.1 Unauthenticated Connections

Unauthenticated connections (i.e., connections that do not use the authentication; see Section 5.4.1) MUST set the seq_num field of all outgoing messages to 0, and MUST ignore the sequence number on all incoming messages. In other words, connections that do not use authenticated messages

also do not use sequence numbers.

5.9.2 Authenticated Connections

Authenticated connections use sequence numbers to avoid replay attacks. The SE and NE first learn their initial sequence numbers (see below). Then, every time a node sends a new message, it increments its sequence number. Each node remembers the last seen sequence number of valid messages from its peers. If a message is received with a sequence number smaller than the sequence number of the last authenticated message, the message MUST be rejected. A message is rejected using a procedure similar to the one described in Section 5.8.3, except that the F_Bad_Sequence_Number flag is set instead of F_Auth_Required.

When an authenticated connection starts (i.e., when a NECP_INIT and NECP_INIT_ACK are exchanged, and the NECP_INIT indicates that the connection should use authentication), the two nodes MUST exchange initial sequence numbers.

Each node tells its peer what sequence number it wants to RECEIVE. To be more explicit: the SE adds a sequence number to the data portion of its NECP_INIT message, and for the remainder of the connection, the packets emitted by the NE MUST start with that sequence number. Similarly, in the other direction, the NE sends a sequence number to the SE as part of the data in its NECP_INIT_ACK message, and the packets emitted by the SE MUST start with that sequence number.

In the following explanation, take care not to confuse the sequence number of the message itself (i.e., the sequence number in the NECP message header) with the sequence number being transmitted in the basic payload portion of the packet.

1. Each side MUST select an initial sequence number for the connection. Sequence numbers are 64 bits long. The high 32 bits SHOULD be the node's local clock, which hopefully is monotonically increasing. The low 32 bits SHOULD be 0.

2. The SE MUST send its NECP_INIT message to the NE (see Section 5.4.1), including its selected sequence number in the data1 (MSB) and data2 (LSB) fields of the NECP_INIT basic payload. The sequence number in the header of this message MUST be 0.

3. The NE receives this NECP_INIT message and notes the sequence number sent in the payload. Note that this is the only case where a message with an invalid sequence number in the header may be accepted.

4. The NE sends an NECP_INIT_ACK, sending its selected sequence number in the data0 (MSB) and data1 (LSB) fields of the NECP_INIT_ACK basic payload. The sequence number in the header of this message MUST be the sequence number that was received from the SE in step 3.

5. The SE receives the NECP_INIT_ACK and notes the sequence number sent in the payload. It MUST use this number as the initial sequence number for future messages to the NE.

6. Once the initial sequence number is in place for both the SE and the NE, they will monotonically increase these numbers in future messages send to their respective peers.

Initialization Example:

```
SE sends NECP_INIT with
   Header Sequence Number = 0 (This is always 0 on
NECP_INIT)
   Payload data0          = 0x1 (Flag that requests auth)
   Payload data1          = 0x22222222
   Payload data2          = 0x33333333

NE replies with an NECP_INIT_ACK with
   Header Sequence Number = 0x2222222233333333
   Payload data0          = 0x44444444
   Payload data1          = 0x55555555
```

Now, the messages sent from SE->NE have sequence numbers starting from 0x4444444455555555. Messages sent from NE->SE have sequence numbers starting from 0x2222222233333333.

6 Motivations and Design Alternatives

This section describes some of our design decisions in more detail, and describes the ideas and motivations behind them. This section does not define protocol requirements, but hopefully sheds light on the requirements defined in previous sections. Nothing in this section carries the "force of law" or is part of the formal protocol specification.

In general, our guiding principle was to make NECP the simplest possible protocol that would do the job, and no simpler. Many features were rejected where alternative (non-protocol-based) solutions could be found. In addition, we have intentionally left many issues at the discretion of the implementor, where we believe that doing so does not compromise interoperability.

6.1 Protocols for Configuration vs. Control

NECP was designed strictly as a control protocol, not a

configuration protocol. That is to say, NECP focuses solely on the coordination that is required between an NE and the SEs in its server farm. Data such as the current flow exception list and current load information can not be configured a priori; it must be exchanged while the network is operating. This is the space in which NECP operates.

Of course, in order for a server farm to actually work, additional configuration is likely required, possibly including:

* selection of a load-balancing policy for the NE.

* a list in the NE of IP addresses of SEs from which it should accept connections, for security purposes.

* a list in the NE of SEs that do not understand NECP, and should be included in a load-balancing group even without an NECP control channel.

* values of the various backoff timers that NECP uses.

NECP does not specify how these values are configured; we assume that some other method for configuring the SE and NE exists. This is likely to be vendor-specific. Our goal was to find the minimum information that an SE and NE need to share in order to interoperate, so as to make NECP as simple as possible. The details that do not need to be shared, such as the NE's load balancing algorithm, we feel are better left at the discretion of the implementor.

Note, however, that the above comments do not preclude the use of NECP as part of a more comprehensive management interface. Such an interface might be useful to administrators who, for example, want to manually add flows to the flow exception list.

6.2 Use of TCP as a Transport Protocol

SNMP [2], the Simple Network Management Protocol, is a general protocol used for management of Internet nodes. The functionality of NECP could have been implemented as a new SNMP MIB instead of an entirely new protocol. Indeed, some of our original design efforts were directed towards an SNMP version of NECP. However, we later settled on a TCP-based protocol after we found ourselves designing features in the SNMP version that already existed in TCP.

Similar in this sense to protocols such as BGP4 [10], NECP leverages off of TCP's retransmission policy, flow control, congestion control mechanisms, packetization of large messages, and stream demultiplexing facilities. In addition, we felt that SNMP was too heavyweight and general of a protocol for this application.

Related to our choice of transport protocol was our choice

between a "hard state" vs. "soft state" model. Soft state protocols tend to recover more gracefully from errors because they do not depend on reliable detection of failures; they periodically refresh all state as part of normal operations. However, our decision to use TCP made peer failure detection reliable, so we opted for a simpler hard state model instead.

6.3 Flow Control of NECP Control Messages

Earlier versions of NECP allowed the SE and NE to impose explicit flow control on their NECP control messages. Special cases of the NECP_START and NECP_STOP messages were defined to apply to future NECP messages themselves, instead of application protocol requests. Ultimately, we decided this feature was more trouble (complexity-wise) than it was worth; it required a large number of exceptions, clarifications, and special cases.

We decided instead to use TCP's flow control mechanisms to govern the flow rate of NECP control messages. If a node running NECP is falling behind in its message processing, it will stop reading messages from its TCP. This, in turn, will cause TCP's advertised window to close, which will signal the sender that its peer is falling behind in its message processing. We feel that this simpler (and already implemented) TCP-based mechanism is as effective as application-layer flow control.

6.4 Stream Framing

We intentionally designed NECP with a fixed-length header that contains a field indicating the length of the data to follow it. This design was meant to help implementations quickly segment streams into their constituent messages. The fact that the header contains the total length means that a node can read an entire message off of a stream before parsing the message. We feel that this design is superior to other designs, where the total length of the message can only be determined by incrementally parsing the message.

The 2-byte protocol_identifier field at the beginning is a "magic cookie" to help identify framing errors. If such an error is detected, the safest policy is to simply close the connection and reopen it according to the procedure in Section 5.4.

6.5 Message Segmentation and Chaining

Many protocols have a reasonable bound on the size of an individual message (say, 64K). In the case that a logical message exceeds the size of a network message, these protocols often contain an explicit mechanism to indicate that more data is coming.

In contrast, the size of an NECP message is, for all practical purposes, unbounded (2^32 bytes), and NECP contains no message chaining facility. Like many of our design decisions, we did this to simplify the protocol and leverage off of the facilities provided to us by TCP. By allowing such large messages, we hope that all NECP messages will "fit", and thus can do away the complexity of chaining. This approach is reasonable because the design of the protocol allows for efficient incremental handling of large messages; for implementation advice, see Section 7.1.

6.6 Initialization in STOP State

When an NE comes up (for example, after a crash), it initializes into the STOP state -- that is, it does not forward flows to SEs until it receives an NECP_START message from one of them. An alternative would be to have it start forwarding again where it left off, assuming it has some way of knowing what its forwarding state was before its crash.

Both of these alternatives are reasonable, but we made our decision because it seemed more consistent with the "hard state" model at the heart of the protocol. The SEs do not periodically refresh the state that they expect the NE is holding; they install it once and assume that it is persistent. After an NE crash, the NE state might potentially be corrupted, and the hard state model means that this corrupted state may never be corrected. A safer design seemed to be to have the NE delete all of its state and let the SEs reconstruct it.

6.7 The Per-Flow State Requirement

In Sections 5.6 and 5.7, we stressed that the burden of keeping per-flow state rested on the NE. This might seem bizarre, since an NE is typically a very high-speed device that switches at wire speed.

The requirement was actually one of three options that we considered for maintaining the forwarding of existing flows while changing the forwarding of new flows. The three options we considered were the following:

1. Since the SE is already maintaining state on connections that it is servicing, have it send that list to the NE as "exceptions" to a forwarding request.

2. Require the NE to keep per-flow state, and mandate that forwarding requests do not affect existing flows.

3. Ignore the problem, and let existing flows break when a new forwarding command is issued.

Option 1 seemed unattractive for a number of reasons. First, it would have made the protocol significantly more complex, which we wished to avoid if at all possible. Second, the solution was

not complete. An SE will only have canonical information about flows that it is currently servicing. Although it might theoretically tell the NE "cut through flows from IP address X to the Internet EXCEPT for the following flows," it would never be able to do the opposite. In other words, when asking that flows be directed from the Internet back to the SE, there is no way for the SE to know which direct-to-Internet flows would need to be excepted.

Our next choice actually was Option 3, because it seemed unreasonable to expect an NE to be able to keep per-flow state. However, after discussions with NE vendors, we learned that the leading switch implementations already keep per-flow state. We therefore settled on Option 2, and mandated that NEs keep per-flow state to ensure the consistency of all implementations that claim to be NECP compliant.

6.8 SEs and NEs that are on Different Networks

If an SE and NE are directly connected by the same datalink network, the NE can divert packets to the SE simply by putting the original (unmodified) IP packet into a datalink-layer frame addressed to the SE. When the SE receives this diverted packet, it will still bear the original packet's destination IP address -- that is, the server that the client was trying to contact.

Alternatively, in some some cases, the NE and SE are separated by several router hops. In these cases, the NE has two options for diverting the client's packet to the SE. The first option is encapsulation: the client's original packet is encapsulated in another packet that is addressed to the SE. The second option is for the NE to overwrite the destination address of the client's packet with the SE's address, and send the modified packet on its way.

In the latter case, the SE needs some way to determine what the original IP address of the packet was -- that is, the address of the server that the client was originally trying to contact. (Note that some application protocols may repeat this information anyway; for example, the Host: field in HTTP.) Earlier versions of NECP included features that would allow a SE to query the the NE for a packet's original destination IP address.

After consulting various vendors who plan to implement NECP, it was decided that encapsulation was a much better solution than destination IP address rewriting. Rewriting the destination IP address opens a can of worms, one of which was the extra features required in NECP. Therefore, the entire idea of NE address rewriting was scrapped in favor of an official recommendation that GRE [7] encapsulation be used instead.

Some vendors in fact prefer using GRE even when the SE and NE are on the same network. For this reason, a "Use GRE" flag was

added to the NECP_START opcode.

6.9 Mandatory Implementation of Security Features

We have chosen to make support for the security architecture described in Sections 5.8 and 5.9 mandatory, even though implementations may choose to use it or not use it on a per-connection basis. The reasoning behind this seeming contradiction is as follows.

On the one hand, we believe the security architecture to be an important part of the protocol, and mandating it -- rather than making its implementation optional -- will improve interoperability. In addition, assuming universal implementation allowed us to simplify the protocol because the SE and NE do not need to negotiate their "lowest common denominator" abilities.

On the other hand, we recognize that our security architecture can impose a significant computational burden on the nodes that are implementing it, so we made its use optional. Users in low-risk environments may wish to get better performance by disabling NECP's authentication features.

6.10 Denial of Service Attacks

We believe that our security architecture is sufficient to prevent attackers (who do not have the nodes' shared secret) from successfully forging NECP control messages. We believe that the use of a 64-bit sequence number should prevent replay attacks by guaranteeing that sequence numbers are never re-used. (The high 32 bits come from the clock at the beginning of the connection, and are unique for the lifetime of the device; the low bits start at 0 and are incremented for each message.)

Nevertheless, it is possible to construct a scenario where a sufficiently resourceful attacker can cause a denial of NECP message service. Specifically, an attacker might replay an old NECP_INIT message so as to trick the NE into using old sequence numbers from a previous connection. The SE will subsequently ignore messages from the NE because the NE's messages will have incorrect sequence numbers.

We considered modifications to the protocol that would prevent this type of denial of service attack. However, we decided that protection from this attack was not worth the significant complexity required to prevent it. In particular, we found the benefits to be minimal in light of the fact that TCP itself is vulnerable to denial of service attacks (for example, from forged TCP RST packets).

Another type of Denial of Service we considered, although an unlikely one, is the denial of service that one SE may be able to cause for other SEs. An SE may accidentally or maliciously

mark many flows with "global exceptions." This would cause flows to not be directed to the server farm that properly functioning SEs normally would have been able to service. To prevent this, we made the "global" vs. "local" exception flag advisory rather than mandatory. An NE that is serving mutually untrustworthy SEs can simply treat all exceptions as local.

6.11 Future Extensions

We have left a number of "hooks" in NECP that will allow future development in a number of directions. The most generic of these is the variable length payload (see Section 5.2.3.2). We found no need for unstructured messages in the current version of NECP, but nevertheless included this feature. Our hope was that future NECP features that need more data than the 128-bit basic payload will be implemented using the variable-length payload, instead of throwing the entire protocol away. For example, commands that pass URLs, or even XML-encoded commands, can use the variable length payload.

The other major area of extensibility is in the definition of performance metrics (see Section 5.5). It seems difficult now to predict which services will be used by SEs in the future, let alone what kinds of performance metrics will be meaningful in the context of those services. Instead of trying to see the future, we created a framework in which future performance metrics can be created without breaking the protocol. As new metrics are designed, they can each be assigned new numbers. Depending on interest, a block of the 32-bit query type space might even be set aside for "privately defined queries," analogous to the 192.168 netblock of IP. Another possible direction is a new protocol message that allows an NE and SE to define new performance query types at run time; for example, by associating a performance query type identifier with an SNMP MIB OID.

7 Implementation Notes

This section has advice for implementors, based on our own implementation experience.

7.1 Buffer Management

The maximum size of an NECP message is very large, which might be alarming to implementors who are expecting to need 4GB of buffer space for every NECP stream. However, the nature of the protocol allows us to get away with reasonably sized buffers, even when processing very large messages.

It should be noted that in NECP request messages, a message header and any one of the basic payload units of a message are a complete, autonomous request. Therefore, even if a very large message comes in, an NECP library might pass it to the application

one basic payload unit at a time, as it arrives. This eliminates the need to buffer the entire message, because the data read off of the network can be discarded as it is passed to the application.

In addition, it is important to observe that the lack of a chaining feature in the NECP protocol does not preclude the use of a "chaining API" between an NECP library and an application that uses it. Implementors can use any method of turning a large message into a chain of smaller ones that suits the implementation.

Authenticated connections are slightly more complex in that it is not known whether or not ANY of the message should be processed until the entire message has arrived. In this case, it is still possible for an NECP library to incrementally pass requests or replies to the application, but perhaps with some qualification that the application should not act on the data until the authentication has been verified at the end. The library might then calculate the HMAC-SHA1 hash incrementally, as the message is passed from the network to the application, and then authenticate the message at the end. We intentionally selected a hashing algorithm that requires only one pass over the data so that this would be possible.

7.2 Non-Blocking Library Design

Clearly, it is not acceptable for an NECP library to block while waiting for NECP messages to arrive, because the application using the library almost certainly has other things it would like to do. One possible solution would be for the NECP library to block with a timeout that would call an application-supplied function periodically; this is the solution used by the standard UNIX X Window System programming interface.

However, we opted for the reverse approach: let the application "run the show", and ask that the application periodically call the library, instead of having the library periodically call the application. The call that is the library's "hook" is guaranteed to be nonblocking. The function reads from readable NECP control channels (if any), and, if any complete messages are received, passes an appropriately parsed representation of the message to the application via an application-supplied callback.

We briefly considered telling the application to call the library only when there was useful work to be done (i.e., when a socket was readable) instead of periodically, but we later decided against this approach for two reasons. First, NECP has certain periodic housekeeping activities (e.g., retrying failed connections), so the library needs to be given control -- even if there is no incoming data. Second, an NE might have many potentially readable file descriptors, and it seemed too complex an API -- and too much of a burden on the application programmer

-- to pass all those FDs to the application and ask that it check to see if any of them are readable.

Yet a third option is to use a second thread that does the readability checking and periodic housekeeping; however, in our implementation, we did not want to assume that we were running on a multi-threaded operating system.

8 Security Considerations

See Sections 5.8, 5.9, 6.9, and 6.10.

9 Acknowledgments

Cerpa and Elson would like to thank Network Appliance, Inc. for support received for the development of this protocol. Funding for the RFC editor function is currently provided by the Internet Society.

10 Bibliography

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", Request for Comments 2119, Harvard University, March 1997.

[2] Case, J., et. al., "A Simple Network Management Protocol", Request for Comments 1157, SNMP Research, May 1990.

[3] Cooper, I., et. al., "Internet Web Replication and Caching Taxonomy", Internet Draft (work in progress), November 1999.

[4] Fielding, R., et. al., "Hypertext Transfer Protocol -- HTTP/1.1", Request for Comments 2068, January 1997.

[5] FIPS PUB 180-1, "Secure Hash Standard", FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION, U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, April 17, 1995.

[6] Gauthier, P., et. al., "The Web Proxy Auto-Discovery Protocol," Internet Draft (work in progress), October 1998.

[7] Hanks, S., Li, T., Farinacci, D., and Traina, P. "Generic Routing Encapsulation over IPv4 networks". RFC1702. October 1994.

[8] Krawczyk, H., Bellare, M., and Canetti, R. "HMAC: Keyed Hash Exchange for Message Authentication". RFC2104. February, 1997.

[9] Postel, J., "Internet Protocol", Request for Comments 791,

USC/Information Science Institute, Sep 01 1981.

[10] Rekhter, Y., and Li, T., "A Border Gateway Protocol 4 (BGP-4)", RFC 1654, T.J. Watson Research Center, IBM Corp., Cisco Systems, July 1994.

[11] Reynolds, J., and Postel, J., "ASSIGNED NUMBERS", STD 2, RFC 1700, October 1994.

[12] Schulzrinne, H., Rao, A., and Lanphier, R., "Real Time Streaming Protocol (RTSP)", Request for Comments 2326, Columbia University and RealNetworks, April 1998.

[13] Wessels, D., and Claffy, K., "Internet Cache Protocol (ICP), version 2", Request for Comments 2186, National Laboratory for Applied Network Research/UCSD, September 1997.

11 Authors' Addresses

Alberto Cerpa
University of Southern California
Department of Computer Science
941 W. 37th Place, SAL 300
Los Angeles, CA 90089-0781 USA
Phone: (213) 740-4496
Email: cerpa@usc.edu

Jeremy Elson
University of Southern California
Department of Computer Science
941 W. 37th Place, SAL 300
Los Angeles, CA 90089-0781 USA
Phone: (213) 740-4496
Email: jelson@usc.edu

Hooman Beheshti
Radware, Inc.
3505 Cadillac Ave., Suite G5
Costa Mesa, CA 92626 USA
Phone: (714) 436-9700
Email: hooman@radware.com

Anawat Chankhunthod
Network Appliance, Inc.
495 East Java Dr.
Sunnyvale, CA 94089 USA
Phone: (408) 822-6000
Email: anawat@netapp.com

Peter Danzig
Akamai Technologies
1400 Fashion Island Blvd
San Mateo, CA 94404 USA
Phone: (650) 372-5757
Email: danzig@akamai.com

Raj Jalan
Foundry Networks, Inc.
680 W. Maude Ave. Suite 3
Sunnyvale, CA 94086 USA
Phone: (408) 530-3317
Email: jalan@foundrynet.com

Chuck Neerdaels
Inktomi Corporation
1900 S. Norfolk St Suite 310
San Mateo, CA 94403 USA
Phone: (650) 653-2800
Email: chuckn@inktomi.com

Ted Schroeder

Alteon Web Systems
50 Great Oaks Blvd.
San Jose, CA 95119 USA
Phone: (408) 360-5500
Email: ted@alteon.com

Gary Tomlinson
Novell, Inc.
122 East 1700 South
Provo, UT 84606 USA
Phone: (801) 861-7021
Email: garyt@novell.com

FULL COPYRIGHT STATEMENT