AAA Working Group
Internet-Draft
Expires: September 2, 2005

P. Calhoun
D. Frascone
Cisco Systems, Inc.
J. Kempf
DoCoMo Labs, USA
March 2005

The Diameter API
draft-ietf-aaa-diameter-api-04

Status of this Memo

Copyright Notice

Abstract

The Diameter authentication, authorization, and accounting (AAA) protocol provides support for peering AAA transactions across the Internet. This document describes a standardized API for the Diameter protocol. The API is defined for the C language. The intent of the API is to foster source code portability across multiple programming platforms.

Table of Contents

1. Introduction

The Diameter authentication, authorization and accounting (AAA) protocol provides scale-able AAA support for peering transactions across the Internet [1]. This document describes standardized API in C for applications to access the Diameter protocol. While a standardized API is not strictly necessary for protocol interoperability, it does help to promote the use and deployment of a protocol by reducing the amount of work necessary to develop and access applications that use the protocol.

The Diameter protocol provides a basic attribute/value pair (AVP) data format, which particular application profiles extend. Processing of the extensions is handled by code specific to the application profile. Application profile customizability is reflected into the API as callback functions for C.

The callbacks implement the application profile processing for incoming messages. For outgoing calls, the C API provides an asynchronous model, leaving processing of the return message to the callbacks.

For the most part, the API hides the details of establishing peering and redirect connections, parsing and creating Diameter messages, and other work necessary to set up and maintain a redirect or peering session. The application profile code need only be concerned with processing of the AVPs defined in the application profile.

2. Binding Independent Considerations

This section discusses a number of implementation considerations for bindings.

2.1. Multithreading

The C API is expected to be thread-safe. Access to data structures shared among threads must be coordinated to avoid corruption or invalid access. In addition, API implementers are encouraged to provide the maximum amount of parallel processing within their library implementations by allowing multiple threads in the API library at once.

2.2. Error Reporting

The API reports errors resulting from client calls through language specific mechanisms. All of the functions in the API return error codes. API implementers are additionally encouraged to log errors using the appropriate platform specific error logging technique, especially for errors that result from network processing or other causes that are not directly related to an API function or method call.

2.3. String Format

C API clients are required to format strings as UTF-8 if the string contains 16 bit characters. Since the ASCII characters and the UTF-8 8 bit characters have the same codes, ASCII can be used for UTF-8 if no wide characters are in the string. All strings passed through the C API are standard null-terminated C strings. Processing to remove the null terminator for transmission on the wire is done by the library.

2.4. Handling Connections with Other Servers/Peers

The API supports making a connection with an arbitrary Diameter peer. The API allows a client to set the server address in a message (AAASetServer()). If a message is not sent to a particular server, the API library is required to infer the servers by either looking in the configuration files or dynamically determining the servers that support the extension. Dynamic determination is possible using means such as SLP [3].

2.5. Command Dictionary File

The commands that can be parsed by the local Diameter client library or server are defined in a command dictionary file containing the

command definitions including AVPs. The location and name of the command dictionary file is platform-specific. This file is read and parsed to drive creation of a command dictionary which is used by the library to parse commands. The syntax for the command dictionary file is in XML and a DTD described it is available in [4]. XML was selected as the definition language because support for XML parsing is available as an extension to the standard Java APIs [5] and as a wide variety of public-domain C libraries, simplifying implementation. Both APIs also support programmatic definition of commands, AVPs, and extensions so programs can add commands not in the dictionary for purposes of experimentation and implementing the library.

3. C API

The C language API is designed around callbacks. An application profile defines a collection of Diameter commands, and a library of callbacks for processing those commands. Each command is processed by a callback. Callbacks can also be defined that handle all commands. The API provides functions for managing callbacks, including registration and deregistration.

When an incoming Diameter command arrives, the command is parsed and passed to the appropriate callback. The callback receives as a parameter the message struct, which contains the AVPs for the command. The callback code can process the command by stepping through the AVPs.

For outgoing requests, the API provides functions for creating messages and adding AVPs. A collection of functions also provides access to the AVP dictionary.

Unless otherwise noted, parameters to API functions and callbacks are non-NULL. Some parameters may have other restrictions. If a parameter fails to satisfy the restrictions on its value, the function returns AAA_ERR_PARAMETER.

3.1. Constant Types

3.1.1. IP Address and Port

```
typedef sockaddr_storage IP_ADDR;
```

IP_ADDR provides a way of referring to an IPv4 address, IPv6 address, and IP port. The default implementation (shown here) is defined in the Basic Socket Interface Extensions for IPv6 RFC[6]

3.1.2. Command Code

```
typedef uint32_t AAACommandCode;
```

AAACommandCode provides a way of referring to the AAA command code of a command. It is used when registering callbacks, among others.

3.1.3. Vendor Identifier

```
typedef uint32_t AAAVendorId;
```

AAAVendorId provides a way of referring to the vendor identification code. It is used when registering callbacks, among others. Note that vendor id 0 is reserved and is defined by the preprocessor

constant AAA_NO_VENDOR_ID.

3.1.4. Extension Identifier

```
typedef uint32_t AAAExtensionId;
```

AAAExtensionId provides a way of referring to an application profile extension, for registering callbacks and other purposes.

3.1.5. Attribute/Value Pair Code

```
typedef uint32_t AAA_AVPCode;
```

AAA_AVPCode provides a way of referring to the code number of an AVP. It is used as a parameter to the dictionary functions, and a field in the AVP struct.

3.1.6. Value Type Identifier

```
typedef int32_t AAAValue
```

AAAValue provides a way of referring to particular dictionary-defined values. It is used in the dictionary API.

3.1.7. Value Type Identifier

```
typedef void AAAServer;
```

AAAServer is an identifier for a particular serving peer. It is used in the server access functions.

3.1.8. Session Identifier

```
typedef void AAASessionId;
```

AAASessionId is an identifier for a particular AAA session. It is used in the session APIs and when a message is created.

3.1.9. Message Identifier

```
typedef uint32_t AAAMsgIdentifier;
```

AAAMsgIdentifier is a unique identifier for an AAA message. Each individual message is marked with an identifier.

3.1.10. Callback Handle

```
typedef void AAACallbackHandle;
```

AAACallbackHandle is a type for representing the callback handle returned to the client when a callback is registered.

3.1.11. Application Identifier

```
typedef void* AAAApplicationId;
```

AAAApplicationId identifies a particular client session to the API. The application id is passed to AAAStartSession(), and is attached to incoming messages, to indicate with which client session the message is associated.

3.1.12. API Return Codes

The following status codes are returned by functions in the AAA API:

```
typedef enum {
      AAA_ERR_NOT_FOUND =       -2,
      AAA_ERR_FAILURE =         -1,
      AAA_ERR_SUCCESS =          0,
      AAA_ERR_NOMEM,
      AAA_ERR_PROTO,
      AAA_ERR_SECURITY,
      AAA_ERR_PARAMETER,
      AAA_ERR_CONFIG,
      AAA_ERR_UNKNOWN_CMD,
      AAA_ERR_MISSING_AVP,
      AAA_ERR_ALREADY_INIT,
      AAA_ERR_TIMED_OUT,
      AAA_ERR_CANNOT_SEND_MSG,
      AAA_ERR_ALREADY_REGISTERED,
      AAA_ERR_CANNOT_REGISTER,
      AAA_ERR_NOT_INITIALIZED,
      AAA_ERR_NETWORK_ERROR


} AAAReturnCode;
```

Note that these status codes are separate from the codes returned by remote AAA servers.

The following is a description of the error codes and the reasons why they can be returned:

AAA_ERR_NOT_FOUND This code is returned if a handle or id was not found.

AAA_ERR_FAILURE This code is returned if an unspecified failure occurred during an AAA operation.

AAA_ERR_SUCCESS This code is returned if the AAA operation succeeded.

AAA_ERR_NOMEM This code is returned if the operation caused memory to be exhausted.

AAA_ERR_PROTO This code is returned if a AAA protocol error occurred.

AAA_ERR_SECURITY This code is returned if a security check failed or another security error occurred.

AAA_ERR_PARAMETER This code is returned if an invalid parameter was passed to an AAA function.

AAA_ERR_CONFIG This code is returned if an error was encountered in a configuration file during library initialization.

AAA_ERR_UNKNOWN_CMD This code is returned if the library received a AAA command that is not in the set of registered AAA commands.

AAA_ERR_MISSING_AVP This code is returned if a command was received without a required AVP.

AAA_ERR_ALREADY_INIT This code is returned if an attempt is made to initialize the AAA library when it has already been initialized.

AAA_ERR_TIMED_OUT This code is returned when a network operation times out.

AAA_ERR_CANNOT_SEND_MSG This code is returned if the library can't send a message. It is also of used to application profile extensions that encounter the same error condition.

AAA_ERR_ALREADY_REGISTERED This code is returned by the command registration functions if the command was already registered.

AAA_ERR_CANNOT_REGISTER This code is returned by the command registration functions if the command could not be registered.

AAA_ERR_NOT_INITIALIZED This code is returned by any function in the API except AAAOpen() if the library hasn't been initialized.

AAA_ERR_NETWORK_ERROR This code is returned by any function if an error in networking occurs.

In addition to returning the error code, functions are required to log errors using the platform dependent logging facility.

3.1.13. Callback Location Codes

The following are codes used to indicate where a callback should be installed in callback chain for processing:

```
typedef enum {
        AAA_APP_INSTALL_FIRST,
        AAA_APP_INSTALL_ANYWHERE,
        AAA_APP_INSTALL_LAST
} AAACallbackLocation;
```

Callbacks installed with AAA_APP_INSTALL_FIRST and AAA_APP_INSTALL_LAST operate on all commands, callbacks installed with AAA_APP_INSTALL_ANYWHERE just operate on the command for which they are installed.

The codes have the following semantics:

AAA_APP_INSTALL_FIRST Install this callback as the first callback in the chain. If subsequent callbacks are installed with this code, then AAA_ERR_ALREADY_REGISTERED is returned.

AAA_APP_INSTALL_ANYWHERE Install this callback anywhere in the callback chain.

AAA_APP_INSTALL_LAST Install this callback as the last callback in the chain. If subsequent callbacks are installed with this code, then AAA_ERR_ALREADY_REGISTERED is returned.

3.1.14. AVP Data Type Codes

The following are AVP data type codes. They correspond directly to the AVP data types outline in the Diameter specification [1]:

```
typedef enum {
     AAA_AVP_DATA_TYPE,
     AAA_AVP_STRING_TYPE,
     AAA_AVP_ADDRESS_TYPE,
     AAA_AVP_INTEGER32_TYPE,
     AAA_AVP_INTEGER64_TYPE,
     AAA_AVP_TIME_TYPE,
} AAA_AVPDataType;
```

3.1.15. AVP Flags

The following are used for AVP header flags and for flags in the AVP wrapper struct and AVP dictionary definitions. The first six correspond to the AVP flags defined in the Diameter specification [1]. Some of these are also used in the wrapper struct and dictionary definitions also. The last four are used only in AAA_AVP and AAADictionaryEntry:

```
typedef enum {
    AAA_AVP_FLAG_NONE =                 0,
    AAA_AVP_FLAG_MANDATORY =            0x1,
    AAA_AVP_FLAG_RESERVED =             0x2,
    AAA_AVP_FLAG_VENDOR_SPECIFIC =      0x4,
    AAA_AVP_FLAG_END_TO_END_ENCRYPT =   0x10,
    AAA_AVP_FLAG_UNKNOWN =              0x10000,
    AAA_AVP_FLAG_ENCRYPT =              0x40000,
} AAA_AVPFlag;
```

The semantics of the flags are as follows:

AAA_AVP_FLAG_NONE ndicates that no AVP flags are set.

AAA_AVP_FLAG_MANDATORY Represents the 'M' flag in the Diameter AVP header [1], meaning the AVP is mandatory.

AAA_AVP_FLAG_RESERVED Represents the 'R' flag in the Diameter AVP header [1]. This flag is reserved and should not be set.

AAA_AVP_FLAG_VENDOR_SPECIFIC Represents the 'V' flag in the Diameter AVP header [1], meaning that the AVP is vendor specific. If this flag is set, the header will contain a vendor identifier.

AAA_AVP_FLAG_END_TO_END_ENCRYPT Represents the 'P' flag in the Diameter AVP header [1], meaning that the AVP is end-to-end encrypted [2].

AAA_AVP_FLAG_UNKNOWN Indicates that the AVP was not located in the AVP dictionary. This flag is only used in AAA_AVP.

AAA_AVP_FLAG_ENCRYPT Indicates that the AVP was either marked as AAA_AVP_FLAG_END_TO_END_ENCRYPT or that it was hop-by-hop encrypted (and thus that the AAA_AVP_FLAG_END_TO_END_ENCRYPT flag is not set on in the AVP header). If the AVP is not end-to-end encrypted, then it will be either one of the two standard hop-by-hop encrypted AVPs, Integrity-Check-Value and Encrypted-Payload [1]. This flag is only used in AAA_AVP.

3.1.16. Domain Interconnection Types

The following domain interconnection types are returned by AAAGetDomainInternconnectType(). They indicate the type of domain interconnection:

```
typedef enum {
   AAA_DOMAIN_LOCAL,
   AAA_DOMAIN_PROXY,
   AAA_DOMAIN_BROKER,
   AAA_DOMAIN_FORWARD
} AAADomainInterconnect;
```

The flags have the following semantics:

The domain name is for the local domain.

The domain name is for a proxy domain. A proxy is a server that simply forwards the request based on the user's identity or through some other means. The routing method used for a proxy is the Proxy-State method, requiring routing through a fixed chain of proxies [1].

The domain name is for a broker domain. A broker is a server that provides redirect services, allowing all servers in a roaming consortium to interact directly.

The domain name is for a forwarding domain. A forwarding domain is a proxy that uses Destination-NAI routing. With Destination-NAI routing, there is no set sequence of proxies through which messages must be routed [1].

3.1.17. Message Flags

The following type is for the AAA message flags. Currently, there are no message flags defined in the Diameter protocol [1]:

```
typedef uint8_t AAAMsgFlag;
```

3.1.18. Result Codes

The following are the result codes returned from remote servers as part of messages. They correspond directly to the result codes in the Diameter specification [1]:

```
typedef  enum {
    AAA_SUCCESS = 0,
    AAA_FAILURE,
    AAA_POOR_REQUEST,
    AAA_INVALID_AUTH,
    AAA_UNKNOWN_SESSION_ID,
    AAA_USER_UNKNOWN,
    AAA_COMMAND_UNSUPPORTED,
    AAA_TIMEOUT,
    AAA_AVP_UNSUPPORTED,
    AAA_REDIRECT_INDICATION,
    AAA_REALM_NOT_SERVED,
    AAA_UNSUPPORTED_TRANSFORM,
    AAA_AUTHENTICATION_REJECTED,
    AAA_AUTHORIZATION_REJECTED,
    AAA_INVALID_AVP_VALUE,
    AAA_MISSING_AVP,
    AAA_UNABLE_TO_DELIVER
} AAAResultCode;
```

3.1.19. Search Direction Type

The following type allows the client to specify which direction to search for an AVP in the AVP list:

```
typedef enum {
   AAA_FORWARD_SEARCH = 0,
   AAA_BACKWARD_SEARCH
} AAASearchType;
```

3.1.20. Accounting Types

The following type allows the client to specify which direction to search for an AVP in the AVP list:

```
typedef enum {
        AAA_ACCT_EVENT = 1,
        AAA_ACCT_START = 2,
        AAA_ACCT_INTERIM = 3,
        AAA_ACCT_STOP = 4
} AAAAcctMessageType;
```

3.1.21. Security Types

The following defines the possible security characteristics for a host.

```
typedef enum {
        AAA_SEC_NOT_DEFINED = -2,
        AAA_SEC_NOT_CONNECTED = -1,
        AAA_SEC_NO_SECURITY = 0,
        AAA_SEC_CMS_SECURITY = 1,
        AAA_SEC_CMS_PROXIED = 2
} AAASecurityStatus;
```

AAA_SEC_NOT_DEFINED This peer is not known to the API. It has either not connected (dynamically), or is not defined as a static client.

AAA_SEC_NOT_CONNECTED This peer is not currently connected, but it has been defined.

AAA_SEC_NO_SECURITY The peer has no security enabled. All data will be sent in the clear.

AAA_SEC_CMS_SECURITY This peer has full security enabled. All encrypted payloads will be visible only to the far end of the connection.

AAA_SEC_CMS_PROXIED This peer has a security association, but some proxies between the endpoints are decrypting/re-encrypting the data. If all proxies in the connection are trusted, then the data is assumed to be secure. If a connection has this status, and the data travels through untrusted proxies, it should be assumed that there is no more security than the data traveling in the clear.

3.2. Structure Definitions

3.2.1. Dictionary Entry Definition

The following structure is returned by the dictionary entry lookup functions. It contains information about a particular AVP in the dictionary:

```
typedef struct dictionaryEntry {
   AAA_AVPCode     avpCode;
   char*           avpName;
   AAA_AVPDataType avpType;
   AAAVendorId     vendorId;
   AAA_AVPFlag     flags;
} AAADictionaryEntry;
```

3.2.2. AVP Definition

The following structure contains a message AVP in parsed form:

```
typedef struct avp {
        enum {
                AAA_RADIUS,
                AAA_DIAMETER
        } packetType;
        AAA_AVPCode code;
        uint16_t length;
        AAA_AVPFlag flags;
        AAA_AVPDataType type;
        AAAVendorId vendorId;
        void* data;
} AAA_AVP;
```

The fields have the following definitions:

packetType Indicates whether the message is for Diameter or for Radius compatibility. If the AVP is for Radius, then the code, length, type, and data fields are the only valid fields in the structure; the other fields are all null.

code The AVP code. The type of the AVP can be determined by matching the AVP code with an AVP description from the dictionary.

length The length of the AVP's data field.

flags The AVP flags.

type The data type of the AVP's data.

vendorId The vendor id, if the AVP is vendor-specific. If the AVP is standardized, the vendorId field is set to AAA_NO_VENDOR_ID.

data The AVP data, in host byte order.

3.2.3. AVP List

The following structure is used for represnting lists of AVPs on the message:

```
typedef struct avpList{
        AAA_AVP *head;
        AAA_AVP *tail;
} AAA_AVP_LIST;
```

AVPs are kept in ordered lists. The client can use a search direction to indicated in which direction to search when trying to find an AVP.

3.2.4. Message Definition

The following structure contains the full AAA message:

```
typedef struct message {
        AAAMsgFlag              flags;
        AAACommandCode          commandCode;
        AAAVendorId             vendorId;
        AAAResultCode           resultCode;
        IP_ADDR                 originator;
        IP_ADDR                 sender;
        AAA_AVP_LIST            *avpList;
        AAA_AVP                 *proxyAVP;
        AAAMsgIdentifier        identifier;
        time_t                  secondsTillExpire;
        time_t                  startTime;
        void                    *appHandle;
} AAAMessage;
```

flags The message flags. Currently this field is always zero, since there are no flags defined for a Diameter message at this time.

commandCode The command's message code.

vendorId The vendor id of the vendor that defined the message.

resultCode Code indicating the result of the client's request. This code is sent by the peer over the wire.

originatorVersion The IP version of the originator's address.

originator The IP address of the message's originator.

senderVersion The IP version of the sender's address.

sender The IP address of the message's previous hop sender. This is only the same as originator if no proxy or broker peers are being used.

port The port on which the message was received.

avpList The list of AVPs in the message.

proxyAVP The proxy's AVP, if any. Otherwise NULL.

identifier The message's unique identifier.

secondsTillExpire Number of seconds until the message expires.

startTime The number of seconds at which the message was started.

appHandle An identifier indicating for which client session the message is.

3.3. Macros and Preprocessor Definitions

The following definition reserves the vendor id of 0:

```
#define AAA_NO_VENDOR_ID 0
```

3.4. Functions

3.4.1. Initialization and Configuration

This section contains definitions that perform initialization and configuration of the AAA library.

3.4.1.1. AAAOpen()

The following function is used to open and configure the AAA library:

```
AAAReturnCode AAAOpen(char *configFileName);
```

This function must be called before any other AAA function is called. Some of the operations that may be performed by AAAOpen() are: opening and loading the AVP and vendor dictionaries, opening connections with Diameter peers, loading Diameter extension libraries, and registering Diameter callbacks. After AAAOpen() returns, the library must be ready for the client to open a session.

Parameters are:

configFileName The global configuration file name. If NULL or the empty string, use the default for this platform. The global configuration file must contain the vendor and AVP dictionary file names, and may contain other platform-specific information needed to initialize and configure the Diameter peer.

Return values are:

AAA_ERR_SUCCESS If initialization succeeded.

AAA_ERR_ALREADY_INIT If the library is already initialized.

AAA_ERR_NETWORK_ERROR If the host name can't be determined.

AAA_ERR_NOMEM If no memory was available.

AAA_ERR_CONFIG If processing the dictionary or configuration information failed.

3.4.1.2. AAAClose()

The following function closes the AAA library:

```
AAAReturnCode AAAClose();
```

After this function is called, all other AAA functions are no longer operative.

Return values are:

AAA_ERR_SUCCESS If finalization succeeded.

AAA_ERR_NOT_INIITIALIZED If AAA was not initialized.

3.4.1.3. AAAGetDefaultConfigFileName()

The following returns the default configuration file name on the platform:

```
const char *AAAGetDefaultConfigFileName();
```

The return value is a pointer to the full pathname for the file. The pointer value should not be deallocated because it is constant and does not change.

3.4.2. Registering Commands

The functions in this section are used to register callback functions defined in a Diameter application profile extension library. The following typedef defines the interface between callback functions and the AAA library functions:

```
typedef AAAReturnCode (*func)(AAAMessage *message) AAACallback;
```

Authors of Diameter extensions must define command callback functions having this interface.

Parameters are:

message The AAAMessage to be processed.

The return value is a status code giving the operation status.

3.4.2.1. AAARegisterCommandCallback()

The following function is used to register command callbacks for processing AAA commands:

AAACallbackHandle * AAARegisterCommandCallback(AAACommandCode commandCode, AAAVendorId vendorId, char *commandName, AAAExtensionId extensionId, AAACallback callback, AAACallbackLocation position);

Parameters are:

commandCode The code of the command processed by the callback.

vendorId The vendor id of the command.

commandName A pointer to the name of the command.

extensionId The id of the extension to which this command belongs.

callback The callback function to perform processing.

position The position of the callback in the chain.

The return value is a handle used when deregistering the callback, or NULL if an error occurred while registering the callback.

3.4.2.2. AAARegisterNoncommandCallback()

The following callback registers an AAA callback to process all messages. The callback is not associated with any command, but rather will process all messages as they come down the callback chain:

AAACallbackHandle AAARegisterNoncommandCallback(AAACallback callback, AAACallbackLocation position);

Parameters are:

callback The callback function to perform processing.

position The position of the callback in the chain.

The return value is a handle used when deregistering the callback, or NULL if an error occurred while registering the callback.

3.4.2.3. AAADeregisterCommandCallback()

The following function deregisters a command callback:

```
AAAReturnCode AAADeregisterCommandCallback(AAACallbackHandle
*handle);
```

Parameters are:

handle The handle returned when the callback was registered.

The return values are:

AAA_ERR_SUCCESS Returned upon completion.

AAA_ERR_FAILURE if the callback is not registered.

3.4.2.4. AAADeregisterNoncommandCallback()

The following function is used to deregister a noncommand callback:

```
AAAReturnCode AAADeregisterNoncommandCallback(AAACallbackHandle
*handle);
```

Parameters are:

handle The handle returned when the callback was registered.

Return values are:

AAA_ERR_SUCCESS Returned upon completion.

AAA_ERR_FAILURE If the callback is not registered.

3.4.2.5. AAARegisterExtension()

The following function is used to register a Diameter extension id. This function is typically called when registering non-command specific callbacks. Extension ids for c ommand-specific callbacks are registered when the callback is registered:

```
AAAReturnCode AAARegisterExtension(AAAExtensionId extensionId);
```

The parameters are:

extensionId The extension id.

The return codes are:

AAA_ERR_SUCCESS If the registration was successful.

AAA_ERR_NOMEM if a memory allocation failure occurred.

3.4.3. Session and Server Management

The functions in this section allow the client to open, close, and register sessions, and to obtain server identifiers.

3.4.3.1. AAAStartSession()

The following function allows a client to start a session and identify it:

```
AAAReturnCode AAAStartSession(AAASessionId **sessionId,
AAAApplicationId appHandle, char *userName, AAACallback
abortCallback);
```

Parameters are:

sessionId On return, a pointer to the session id for this session.

appHandle An identifier for the client application starting the session. This identifier is attached to messages so that the client callbacks can tell which messages belong to it.

userName - The NAI of the user.

abortCallback A function to be called if this session is aborted by the server..

Return values are:

AAA_ERR_SUCCESS If the session was successfully started.

AAA_ERR_NOMEM If a memory allocation failure occurred.

3.4.3.2. AAARegisterPeerSession()

The following function allows a client to register a peer session that it has discovered through some other means, for example, by receiving an unsolicited message.

```
AAAReturnCode AAARegisterPeerSession(AAASessionId **sessionId,
AAAApplicationId *appHandle, AAAMessage *message, char *userName,
char *hostName);
```

The parameters are:

sessionId On return, a pointer to the local session id for the session.

appHandle An identifier for the client application starting the session. This identifier is attached to messages so that the client callbacks can tell which messages belong to it.

message The message from the peer containing the session id.

userName - The NAI of the user.

hostName The originator of the Diameter message

Return values are:

AAA_ERR_SUCCESS If the session was successfully registered.

AAA_ERR_NOMEM If a memory allocation failure occurred.

3.4.3.3. AAAEndSession()

The following function, sent by a client, terminates a session:

```
AAAReturnCode AAAEndSession(AAASessionId *sessionId);
```

The parameters are:

sessionId A pointer to the session id for the session.

Return values are:

AAA_ERR_SUCCESS If the session was successfully closed.

AAA_ERR_NOT_FOUND If the handle is invalid.

3.4.3.4. AAAAbortSession()

The following function, sent by the server, terminates a session:

```
AAAReturnCode AAAAbortSession(AAASessionId *sessionId);
```

The parameters are:

sessionId A pointer to the session id for the session.

Return values are:

AAA_ERR_SUCCESS If the session was successfully closed.

AAA_ERR_NOT_FOUND If the handle is invalid.

3.4.3.5. AAALookupServer()

The function looks up the AAA server based on IP address and port number. Server connections are created from the configuration file:

AAAServer *AAALookupServer(IP_ADDR ipAddr);

The parameters are:

ipAddr The IP address/Port/Family of the server.

The return value is either a valid server pointer or the NULL if the server can't be found.

3.4.3.6. AAASetSessionMessageTimeout()

The following function sets the timeout, in seconds, for all AAAMessages in a particular session:

AAAReturnCode AAASetSessionMessageTimeout(AAASessionId *id, time_t timeout);

The parameters are:

id The session id for the session whose timeout should be changed.

timeout The session timeout. The default timeout is 120 seconds.

The return values are:

AAA_ERR_SUCCESS If setting the timeout succeeded.

AAA_ERR_FAILURE If the setting the timeout failed.

3.4.3.7. AAAGetDomainInterconnectType()

The following function returns the domain interconnect type for a particular domain name and type of service:

AAAResultCode AAAGetDomainInterconnectType(AAAMessage *message, char

*domainName, char *type);

The parameters are:

domainName The name of the domain.

type The type of service. This must be one of the strings "LOCAL", "PROXY", or "BROKER".

3.4.4. Dictionary Lookup

The functions in this section are used to look up AVPs and commands in the dictionary. The client is responsible for supplying the structure memory into which the dictionary information is copied.

3.4.4.1. AAADictionaryEntryFromAVPCode()

This function looks up a dictionary entry using a command code and a vendor id:

AAAReturnCode AAADictionaryEntryFromAVPCode(AAA_AVPCode avpCode, AAAVendorId vendorId, AAADictionaryEntry *entry);

The parameters are:

avpCode The code number of the AVP.

vendorId The vendor id of the AVP.

entry an AAADictionaryEntry structure for returning the entry.

The return value is one of:

AAA_ERR_SUCCESS If the query succeeded.

AAA_ERR_FAILURE If no matching dictionary entry was found.

3.4.4.2. AAADictionaryEntryFromName()

This function looks up a dictionary entry using command code name and vendor id:

AAAReturnCode AAADictionaryEntryFromName(char *avpName, AAAVendorId vendorId, AAADictionaryEntry *entry);

The parameters are:

avpName The name of the AVP.

vendorId The vendor id of the AVP.

entry an AAADictionaryEntry structure for returning the entry.

The return value is one of:

AAA_ERR_SUCCESS If the query succeeded.

AAA_ERR_FAILURE If no matching dictionary entry was found.

3.4.4.3. AAAValueFromName()

This function looks up an AVP value using the AVP name and vendor name:

```
AAAValue AAAValueFromName(char *avpName, char *vendorName, char
*valueName);
```

The parameters are:

avpName The name of the AVP.

vendorName The name of the vendor.

valueName The name of the value.

The return value is the id of the AVP, or AAA_ERR_NOT_FOUND if no match was found.

3.4.4.4. AAAValueFromAVPCode()

This function looks up an AVP value using the AVP id and vendor id, and the value name:

```
AAAValue AAAValueFromAVPCode(AAA_AVPCode avpCode, AAAVendorId
vendorId, char *valueName);
```

The parameters are:

avpCode The code of the AVP.

vendorId The id of the vendor.

valueName The name of the value.

The return value is id of the AVP, or AAA_ERR_NOT_FOUND if no match was found.

3.4.4.5. AAALookupValueNameUsingValue()

This function returns the AVP value name:

```
const char *AAALookupValueNameUsingValue(AAA_AVPCode avpCode,
AAAVendorId vendorId, AAAValue value);
```

The parameters are:

avpCode The code of the AVP.

vendorId The id of the vendor.

value The value.

The value name is returned, or NULL if no match occurred.

3.4.4.6. AAAGetCommandCode()

This function returns the command code and vendor id based on a string:

```
boolean_t AAAGetCommandCode(char *commandName, AAACommandCode
*commandCode, AAAVendorId *vendorId);
```

The parameters are:

commandName A string containing the command name.

commandCode Pointer that on return holds the command code if the command was found.

vendorId Pointer that on return holds the vendor id if the command was found.

The return value is _B_TRUE if the command was found.

3.4.5. Message Management

The functions in this section allow the client to create messages, add AVPs, and traverse AVP lists.

3.4.5.1. AAANewMessage()

This function allocates an AAAMessage and returns a pointer to it. If a command code is provided, this function adds the command code AVP. If the session identifier handle is provided, the Session-Id AVP is also added. Lastly, if this message is allocated in response to a request, the request's message can be provided, and the new message is initialized to match the request, for fields such as the identifier, the server identifier, etc. If this is a new message, the request parameter is NULL:

```
AAAMessage *AAANewMessage(AAACommandCode commandCode, AAAVendorId
vendorId, AAASessionId *sessionId, AAAExtensionId extensionId,
AAAMessage *request);
```

The parameters are:

commandCode The command code.

vendorId The vendor identifier.

sessionId Session identifier.

extensionId The extension identifier.

request A pointer to a request message, if this message is being allocated in response to a request.

The function returns a pointer to the message or NULL if a failure occurred.

3.4.5.2. AAAFreeMessage()

This function frees a message allocated through AAANewMessage():

```
AAAReturnCode AAAFreeMessage(AAAMessage **message);
```

The parameters are:

message A pointer to a pointer to the allocated message.

The return value is the AAA status code AAA_ERR_SUCCESS.

3.4.5.3. AAARespondToMessage()

This function is called to set the AAA Message to the appropriate values, and to inform the library that this message is a locally generated response

AAAReturnCode AAARespondToMessage(AAAMessage* message, AAACommandCode commandCode, AAAVendorId vendorId, AAAResultCode resultCode);

The parameters are:

message The AAAMessage to respond to.

commandCode The command code of the response.

vendorId The vendor identifier (of the command code).

resultCode The result code of the response.

The function returns AAA_ERR_SUCCESS upon completion, or AAA_ERR_PARAMETER if a NULL pointer was provided.

3.4.5.4. AAAAddProxyState()

This function will add a Proxy-State AVP to a message, that contains the FQDN of the source of the message.

AAAReturnCode AAAAddProxyState(AAAMessage *message);

The parameters are:

message The AAAMessage to add state to.

The function returns AAA_ERR_SUCCESS upon completion, or AAA_ERR_FAILURE if an error occured.

3.4.5.5. AAACreateAVP()

This function creates an AVP and returns a pointer to it. The AVP is initialized from the arguments:

AAAReturnCode AAACreateAVP(AAA_AVP **avp, AAA_AVPCode code, AAA_AVPFlag flags, AAAVendorId vendorId, char *data, size_t length);

The parameters are:

avp On return, contains a pointer to the allocated AVP, or NULL if no AVP was allocated.

code The AVP's code.

flags Any AVP flags that must be passed.

vendorId The vendor id of the AVP. If no vendor id, then AAA_NO_VENDOR_ID.

data A buffer containing the AVP data.

length The length of the data buffer.

Return values are:

AAA_ERR_SUCCESS Upon success.

AAA_ERR_PARAMETER If an invalid parameter was passed.

AAA_ERR_PROTO If a protocol error occurred.

AAA_ERR_NOMEM Indicating a memory failure.

3.4.5.6. AAACreateAndAddAVPToList()

This function creates an AVP and adds it to an AVP list. It returns a pointer to the list in the avpList argument. The AVP is initialized from the arguments:

```
AAAReturnCode AAACreateAndAddAVPToList(AAA_AVP_LIST **avpList,
AAA_AVPCode code, AAA_AVPFlag flags, AAAVendorId vendorId, char
*data, size_t length);
```

The parameters are:

avpList The list to which the AVP should be added.

code The AVP's code.

flags Any AVP flags that must be passed.

vendorId The vendor id of the AVP. If no vendor id, then AAA_NO_VENDOR_ID.

data A buffer containing the AVP data.

length The length of the data buffer.

Return values are:

AAA_ERR_SUCCESS Upon success.

AAA_ERR_PARAMETER If an invalid parameter was passed.

AAA_ERR_NOMEM Indicating a memory failure.

3.4.5.7. AAAAddAVPToList()

AAAReturnCode AAAAddAVPToList(AAA_AVP_LIST **avpList, AAA_AVP *avp, AAA_AVP *position);

Insert the AVP avp into this avpList after position. If position is NULL, the AVP is added to the beginning of the list.

If *avpList is NULL, a list will be allocated, and *avpList will point to it.

The parameters are:

avpList Pointer to a pointer for list. If *avpList is NULL, list memory is allocated.

avp AAA_AVP to add to list.

position AAA_AVP pointer to add data after, or NULL if the new AVP should go at the beginning of the list.

The return value is one of:

AAA_ERR_SUCCESS Upon success.

AAA_ERR_PARAMETER If an invalid parameter was passed.

AAA_ERR_NOMEM Indicates a memory failure.

3.4.5.8. AAAFindMatchingAVP()

This function finds an AVP with matching code and vendor id. If none match, the function returns NULL:

AAA_AVP *AAAFindMatchingAVP(AAA_AVP_LIST *avpList, AAA_AVP *startAvp, AAA_AVPCode avpCode, AAAVendorId vendorId, AAASearchType searchType);

The parameters are:

avp A pointer to the head of the AVP list.

avpCode The code of the sought after AVP.

vendorId The vendor id of the sought after AVP.

The return value is a pointer to the found AVP, or NULL if none is found.

3.4.5.9. AAAJoinAVPLists()

The following function joins together two AVP lists:

```
AAAReturnCode AAAJoinAVPLists(AAA_AVP_LIST *dest, AAA_AVP_LIST
*source, AAA_AVP *position);
```

The parameters are:

dest The destination list (All of the avps in sourc will be moved here).

source The source list to be added to dest.

position The position to add the AVPs to, or NULL for the beginning of the list.

The return value is one of:

AAA_ERR_SUCCESS Upon success.

AAA_ERR_PARAMETER If an invalid parameter was passed.

3.4.5.10. AAARemoveAVPFromList()

This function removes an AVP from a list:

```
AAAReturnCode AAARemoveAVPFromList(AAA_AVP_LIST *avpList, AAA_AVP
*avp);
```

The parameters are:

avpList The head of the list from which to remove the AVP.

avp Contains a pointer to the AVP to remove.

The return value is one of:

AAA_ERR_SUCCESS Upon success.

AAA_ERR_PARAMETER If an invalid parameter was passed.

3.4.5.11. AAAFreeAVP()

The function frees an AVP:

AAAReturnCode AAAFreeAVP(AAA_AVP **avp);

The parameters are:

avp Contains a pointer to a pointer to the AVP to free.

The return value is one of:

AAA_ERR_SUCCESS Upon success.

AAA_ERR_PARAMETER If an invalid parameter was passed.

3.4.5.12. AAAGetFirstAVP()

This function returns a pointer to the first AVP in the list:

AAA_AVP *AAAGetFirstAVP(AAA_AVP_LIST *avpList);

The parameters are:

avpList A pointer to the list.

The return value is a pointer to the found AVP, or NULL if none is found.

3.4.5.13. AAAGetLastAVP()

This function returns a pointer to the last AVP in the list:

AAA_AVP *AAAGetLastAVP(AAA_AVP_LIST *avpList);

The parameters are:

avpList A pointer to the list.

The function returns a pointer to the found AVP, or NULL if none is found.

3.4.5.14. AAAGetNextAVP()

This function returns a pointer to the next AVP in the list.

AAA_AVP *AAAGetNextAVP(AAA_AVP *avp);

The parameters are:

avp A pointer to the AVP prior to the one sought.

The return value is the next AVP in the list, or NULL if the parameter is the last element in the list.

3.4.5.15. AAAGetPrevAVP()

This function returns a pointer to the previous AVP in the list:

AAA_AVP *AAAGetPrevAVP(AAA_AVP *avp);

The parameters are:

avp A pointer to the AVP after the one sought.

The return value is the previous AVP in the list or NULL if the parameter is the first element in the list.

3.4.5.16. AAAConvertAVPToString()

This function converts the data in the AVP to a format suitable for log or display functions.

char *AAAConvertAVPToString(AAA_AVP *avp, char *dest, size_t destLen);

The parameters are:

avp The AVP to display.

dest A used supplied destination buffer.

destLen The length of the user supplied buffer.

The return value is the passed in destination buffer.

3.4.5.17. AAASetMessageResultCode()

This function decapsulates an encapsulated AVP, and populates the list with the correct pointers.

AAAResultCode AAASetMessageResultCode(AAAMessage *message,
AAAResultCode resultCode);

The parameters are:

message A pointer to the allocated message.

resultCode The AAA Result Code

The eturn value is one of:

AAA_ERR_SUCCESS Upon success.

AAA_ERR_PARAMETER If an invalid parameter was passed.

3.4.6. Message Control

The following functions allow the client to direct a message to a particular server, determine the server for a message, etc.

3.4.6.1. AAASetServer()

This function sets the server to which the message is sent:

AAAReturnCode AAASetServer(AAAMessage *message, IP_ADDR host);

The parameters are:

message The message to be sent.

ipVersion The version number of the IP address.

host The IP address / port / family of the server.

The return value is

AAA_ERR_SUCCESS If the server was found.

AAA_ERR_NOT_FOUND If the server was not found.

3.4.6.2. AAASendMessage()

The following function sends a message to the server assigned to the message by AAASetServer(). The message identifier is assigned to the message. If no server id has been assigned to the message, a server id is selected. If no servers are currently active, the message is queued for later sending.

AAAReturnCode AAASendMessage(AAAMessage *message);

The parameter is the message to send.

The return codes are:

AAA_ERR_SUCCESS Upon completion.

AAA_ERR_FAILURE If an error occurred.

3.4.7. Accounting

The following functions allow the client to direct a message to a particular server, determine the server for a message, etc.

3.4.7.1. AAASendAcctRequest()

The following function sends an accounting message to an accounting server.

AAAReturnCode AAASendAcctRequest(AAASessionId *aaaSessionId, AAAExtensionId extensionId, AAA_AVP_LIST *acctAvpList, AAAAcctMessageType msgType)

The parameters are:

aaaSessionId The session id that this accounting data corresponds to.

extensionId The extension type associated with this accounting message.

acctAvpList A list of AVPs to send in the accounting message.

msgType The type of accounting message.

The return codes are:

AAA_ERR_SUCCESS Upon completion.

AAA_ERR_PARAMETER If a parameter is invalid.

3.4.8. Security Functions

The following functions control the security/encryption features of the connection.

All encryption/signing of packets is handled internally by the library, based on the settings in the dictionary.

3.4.8.1. AAAGetPeerSecurityStatus()

The following function will return the security characteristics of the current connection.

AAASecurityStatus AAAGetPeerSecurityStatus(IP_ADDR remoteHost);

The parameters are:

remoteHost The host that is being inquired.

The possible return values are defined in Section 3.1.21.

3.5. Implementation Notes

The following are some implementation notes that library designers may want to keep in mind.

3.6. Grouped AVPs

In order to create grouped AVPs, an application creates an AAA_AVP_LIST that is not attached to an AAAMessage structure (also known as an orphaned AAA_AVP_LIST). All of the necessary AVPs within the Group are added to the orphaned AAA_AVP_LIST using the existing list manipulation functions. Lastly, the grouped AVP is added to the AAAMessage structure.

The following is an example that adds a Proxy-State Grouped AVP to an existing AAAMessage structure.

```
addProxyState(AAAMessage *message, ipaddr_t *ourAddress,
    void *state, size_t stateLen)
{
        AAA_AVP_LIST *avpList = NULL;

        /*
         * Add the Proxy-Address AVP to the AAAList
         */
        if (AAACreateAndAddAVPToList(&avpList,
            DIAM_AVP_PROXY_ADDRESS, AAA_AVPI_FLAG_NONE,
            NO_VENDOR_ID, (char *) ourAddress,
            sizeof (ipaddr_t))) {
                loggerSyslog(LOG_AVP_PROBLEMS,
                    "Unable to add Proxy-Address AVP");
                return (AAA_ERR_FAILURE);
        }
        /*
         * Now we add the Proxy-Info AVP to the AAAList
         */
        if (AAACreateAndAddAVPToList(&avpList,
            DIAM_AVP_PROXY_INFO, AAA_AVPI_FLAG_NONE, NO_VENDOR_ID,
            state, stateLen)) {
                loggerSyslog(LOG_AVP_PROBLEMS,
                    "Unable to add Proxy-Info AVP");
                return (AAA_ERR_FAILURE);
        }
        /*
         * Now the AAAList is added to the AAAMessage as
         * a Proxy-State AVP.
         */
        if (AAACreateAndAddAVPToList(&message->avpList,
            DIAM_AVP_PROXY_STATE, AAA_AVPI_FLAG_NONE,
            NO_VENDOR_ID, (char *)avpList,
            AAA_AVP_GROUPED_LENGTH)) {
                loggerSyslog(LOG_AVP_PROBLEMS,
                    "Unable to add Proxy-State AVP");
                return (AAA_ERR_FAILURE);
        }

        return (AAA_ERR_SUCCESS);
}
```

As shown above, the procedures is to create a new AAA_AVP_LIST structure, adding all of the necessary AVPs that are within the Grouped AVP, then calling AAACreateAndAddAVPToList() to add the AAA_AVP_LIST as a Grouped AVP to the AAAMessage.

Note that the AAA_AVP_LIST pointed to by orphaned avpList MUST NOT be

accessed by the application after the Grouped avp has been created. The list will be freed along with the AVP by the AAA Library.

In order to parse a Grouped AVP, the AAA_AVP data field contains a pointer to an AAA_AVP_LIST, as shown below.

```
boolean_t
isProxyStateOurs(AAA_AVP *proxyState, ipaddr_t *ourAddress)
{
        AAA_AVP_LIST *avpList;
        AAA_AVP *proxyAddress;
        AAA_AVP *proxyInfo;
        ipaddr_t *proxyAddress;

        /*
         * Get the pointer to the Grouped AAA_AVP_LIST
         */
        avpList = (AAA_AVP_LIST *)proxyState->data;

        /*
         * First, for the Proxy-Address, and see if it is ours.
         */
        if ((proxyAddress = AAAFindMatchingAVP(avpList, NULL,
            DIAM_AVP_PROXY_ADDRESS, NO_VENDOR_ID,
            AAA_FORWARD_SEARCH)) != NULL) {
                /*
                 * Check if this one is ours.
                 */
                address2 = (ipaddr_t *)proxyAddress->data;
                if (*address2 == *address) {
                        /*
                         * This one is ours... return TRUE
                         */
                        return (B_TRUE);
                }
        }

        return (B_FALSE);
}
```

3.7. Extended AAA_AVP structure

The AAA_AVP structure that is defined in this specification is a subset of the structure used by the internal library. The internal structure, known as the extended AAA_AVP, may contain many private fields, such as pointers to AAA_AVPs. Applications do not directly access the next (and previous) AAA_AVP pointers directly, but instead access them via the AAAGetNextAVP() and AAAGetPreviousAVP()

functions.

The following is an example of an extended AAA_AVP structure:

```
typedef struct {
        // API Public variables here
} AAA_AVP;


typedef struct xavp {
        AAA_AVP      avp;
        struct xavp *next;
        struct xavp *prev;
        int          privateFlags;
} Extended_AAA_Avp;
```

Of course, when AAACreateAVP is called, sufficient memory is allocated for the extended AAA_AVP structure, however the function returns a pointer to the AAA_AVP.

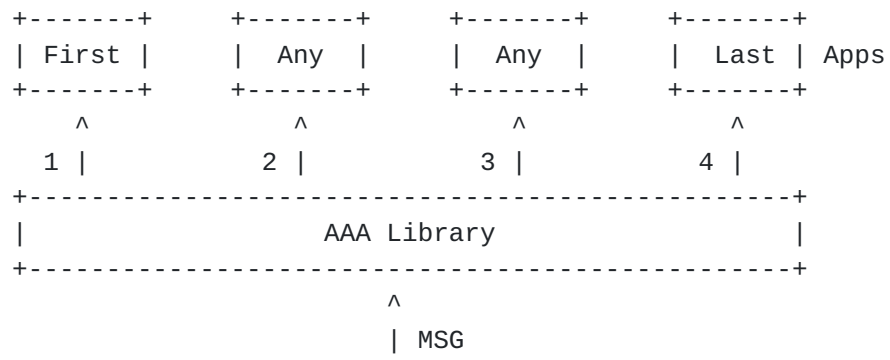
3.8. Avoiding AVP Copying

The AAA_AVP struct does not provide an exact mapping to the Diameter protocol AVP packet format; however, library implementors can avoid having to copy the AVP data by putting a pointer to a packet format structure into a hidden part of the AAA_AVP struct. A pointer to the AVP data is then deposited into the AAA_AVP data field. This allows proper deallocation of the packet format structure when the AAA_AVP structure is deallocated.

3.9. Callback Processing Order

The C API allows API clients to register message processors, or callbacks, that are invoked before and after the bulk of the processing functions for a message. Only one pre- or post-processor is allowed for all incoming messages, regardless of command or extension type. If the API client adds another, any existing pre- and post-processors are removed.

Message processing can be best explained by the following diagram:



In the above diagram, "First", "Any", and "Last" are added by the API client. The message processor labeled "First" is given access to the message before any other, the message processor "Last" after all others are finished. There is no guarantee on ordering for the other message processors. If the client adds a new "First" or "Last" message processor, the existing ones removed. There is one "First" and "Last" processor for all commands regardless of type; whereas, the "Any" processors are command-specific.

If one of the "Any" processors completes successfully, the message is not passed on any further. A successful completion means the success return code is returned from the C API callback, but the callback is responsible for freeing the message before returning.

4. Security Considerations

This document describes an API and therefore depends on the security mechanisms defined in the Diameter protocol [1].

5. References

[1] Calhoun, Loughney, Guttman, Zorn, and Arkko, "Diameter Base Protocol", draft-ietf-aaa-diameter-14.txt , October 2002, <http://www.ietf.org/internet-drafts/draft-ietf-aaa-diameter-14.txt>.

[2] Calhoun, Farrell, and Bulley, "Diameter CMS Security Application", draft-ietf-aaa-cms-sec-04.txt , March 2002, <http://www.ietf.org/internet-drafts/draft-ietf-aaa-diameter-cms-sec-04.txt>.

[3] Guttman, Perkins, Veizades, and Day, "Service Location Protocol, Version 2", rfc 2608, June 1999, <http://www.ietf.org/rfc/rfc2608.txt>.

[4] "Diameter XML Dictionary", , June 1999, <http://www.diameter.org/XML/>.

[5] "Standard Java APIs", , June 1999, <http://www.javasoft.com/xml/jaxp-docs-1.1/readme.html>.

[6] Gilligan, Thomson, Bound, and Stevens, "Basic Socket Interface Extensions for IPv6", rfc 2553, March 1999, <http://www.ietf.org/rfc/rfc2553.txt>.

Authors' Addresses

Pat R. Calhoun
Cisco Systems, Inc.
170 West Tasman
San Jose, CA 95134

Phone: +1 408-853-5269
Email: pcalhoun@airespace.com

David Frascone
Cisco Systems, Inc.
605 N. Frances Street
Terrell, TX 75160

Phone: +1 972-524-6346
Fax: +1 978-334-0249
Email: dave@frascone.com

James Kempf
DoCoMo Labs, USA
180 Metro Drive, Suite 300
San Jose, CA 95110

Phone: +1 408-451-4711
Email: kempf@docomolabs-usa.com

Intellectual Property Statement

Disclaimer of Validity

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.