

Diameter

Copyright © 2011-2018 Ericsson AB. All Rights Reserved.

Diameter 2.1.4

April 17, 2018

Copyright © 2011-2018 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

April 17, 2018

1 Diameter Users Guide

The diameter application is a framework for building applications on top of the Diameter protocol.

1.1 Introduction

The diameter application is an implementation of the Diameter protocol as defined by RFC 6733. It supports arbitrary Diameter applications by way of a **dictionary** interface that allows messages and AVPs to be defined and input into diameter as configuration. It has support for all roles defined in the RFC: client, server and agent. This chapter provides a short overview of the application.

A Diameter node is implemented by configuring a **service** and one or more **transports** using the interface module *diameter*. The service configuration defines the Diameter applications to be supported by the node and, typically, the capabilities that it should send to remote peers at capabilities exchange upon the establishment of transport connections. A transport is configured on a service and provides protocol-specific send/receive functionality by way of a transport interface defined by diameter and implemented by a transport module. The diameter application provides two transport modules: *diameter_tcp* and *diameter_sctp* for transport over TCP (using *gen_tcp*) and SCTP (using *gen_sctp*) respectively. Other transports can be provided by any module that implements diameter's *transport interface*.

While a service typically implements a single Diameter node (as identified by an Origin-Host AVP), transports can themselves be associated with capabilities AVPs so that a single service can be used to implement more than one Diameter node.

Each Diameter application defined on a service is configured with a callback module that implements the *application interface* through which diameter communicates the connectivity of remote peers, requests peer selection for outgoing requests, and communicates the reception of incoming Diameter request and answer messages. An application using diameter implements these application callback modules to provide the functionality of the Diameter node(s) it implements.

Each Diameter application is also configured with a dictionary module that provide encode/decode functionality for outgoing/incoming Diameter messages belonging to the application. A dictionary module is generated from a *dictionary file* using the *diameterc* utility. Dictionaries for the RFC 6733 Diameter Common Messages, Base Accounting and Relay applications are provided with the diameter application.

1.2 Usage

To be written.

1.3 Examples

Example code can be found in the diameter application's `examples` subdirectory.

1.4 Standards Compliance

The table below summarizes the diameter application's compliance with RFC 6733. Since the diameter application isn't a Diameter node on its own, compliance is strictly the responsibility of the user in many cases, diameter providing the means for the user to be compliant rather than being compliant on its own.

The Compliance column notes **C** (Compliant) if the required functionality is implemented, **PC** (Partially Compliant) if there are limitations, **NC** (Not Compliant) if functionality is not implemented, or a dash if text is informational or only places requirements that must be met by the user's implementation.

Capitalized **Diameter** refers to the protocol, lowercase **diameter** to the Erlang application.

1.4.1 RFC 6733 - Diameter Base Protocol

Section	Title	Compliance	Notes
1	Introduction	—	
1.1	Diameter Protocol	—	
1.1.1	Description of the Document Set	—	
1.1.2	Conventions Used in This Document	—	
1.1.3	Changes from RFC 3588	—	It is possible to configure a 3588 dictionary in order to get 3588 semantics, where the differ from 6733.
1.2	Terminology	—	
1.3	Approach to Extensibility	—	The dictionary interface documented in *diameter_dict(4)* provides extensibility, allowing the user to defined new AVPs, commands, and applications. Ready dictionaries are provided for the RFC 6733 common message, base accounting, and relay applications, as well as for RFC 7683, Diameter Overload Indicator Conveyance.
1.3.1	Defining New AVP Values	—	
1.3.2	Creating New AVPs	—	New AVPs can be defined using the dictionary interface. Both both RFC data formats and extensions are supported.

1.3.3	Creating New Commands	—	New commands can be defined using the dictionary interface.
1.3.4	Creating New Diameter Applications	—	New applications can be defined using the dictionary interface.
2	Protocol Overview	—	Session state is the responsibility of the user. The role of a Diameter node is determined by the user's implementation.
2.1	Transport	PC	Ports are configured by the user: diameter places no restrictions. The transport interface documented in *diameter_transport(3)* allows the user to implement their own methods. Ready support is provided for TCP, TCP/TLS, and SCTP, but not DTLS/SCTP. Multiple connections to the same peer is possible. ICMP messages are not interpreted.
2.1.1	SCTP Guidelines	C	Unordered sending is configurable in *diameter_sctp(3)*. There is no special handling of DPR/DPA: since a user that cares about pending answers should wait for them before initiating DPR. A PPID can be configured with a a gen_sctp sctp_default_send_param option.

2.2	Securing Diameter Messages	PC	DTLS is not supported by *diameter_sctp(3)*. See also 2.1.
2.3	Diameter Application Compliance	—	
2.4	Application Identifiers	C	The user configures diameter with the identifiers to send at capabilities exchange, along with corresponding dictionaries defining the messages of the applications.
2.5	Connections vs. Sessions	C	Connections are realized by configuring transport. Sessions are the responsibility of the user.
2.6	Peer Table	PC	Routing is implemented by the user in callbacks documented in *diameter_app(3)*. A peer table of the documented form is not exposed to the user.
2.7	Routing Table	PC	See 2.6. A routing table of the documented form is not exposed to the user.
2.8	Role of Diameter Agents	C	Most role-specific behaviour is implemented by the user. How a node advertises itself at capabilities exchange is determined by user configuration.
2.8.1	Relay Agents	C	
2.8.2	Proxy Agents	C	
2.8.3	Redirect Agents	C	
2.8.4	Translation Agents	C	
2.9	Diameter Path Authorization	—	Authorization is the responsibility of the user.

3	Diameter Header	C	Hop-by-Hop and End-to-End Identifiers are set by diameter when sending outgoing requests.
3.1	Command Codes	C	
3.2	Command Code Format Specification	C	Commands are defined as CCF specifications in dictionary files.
3.3	Diameter Command Naming Conventions	—	
4	Diameter AVPs	C	Any required padding is added by diameter when encoding outgoing messages.
4.1	AVP Header	C	
4.1.1	Optional Header Elements	C	
4.2	Basic AVP Data Formats	C	
4.3	Derived AVP Data Formats	C	Arbitrary derived data formats are supported by the dictionary interface.
4.3.1	Common Derived AVP Data Formats	C	Beware that RFC 6733 changed the DiameterURI transport/port defaults specified in RFC3588. Relying on the defaults can result in interoperability problems.
4.4	Grouped AVP Values	C	The M-bit on a component AVP of a Grouped AVP that does not set M is ignored: such AVPs are not regarded as erroneous at decode. Grouped AVPs are defined as CCF specifications in dictionary files.
4.4.1	Example AVP with a Grouped Data Type	—	

4.5	Diameter Base Protocol AVPs	C	The base AVPs are defined in the common dictionary provided by diameter. There are common dictionaries for both RFC 3588 and RFC 6733 since the latter made changes to both syntax and semantics.
5	Diameter Peers	—	
5.1	Peer Connections	PC	A peer's DiameterIdentity is not required when initiating a connection: the identify is received at capabilities exchange, at which time the connection can be rejected if the identity is objectionable. The number of connections established depends on the user's configuration. Multiple connections per peer is possible.
5.2	Diameter Peer Discovery	NC	No form of peer discovery is implemented. The user can implement this independently of diameter if required.
5.3	Capabilities Exchange	C	All supported applications are sent in CEA. The user can reject an incoming CER or CEA in a configured callback. Both transport security at connection establishment and negotiated via an Inband-Security AVP are supported.
5.3.1	Capabilities-Exchange-Request	C	CER is sent and received by diameter.
5.3.2	Capabilities-Exchange-Answer	C	CEA is sent and received by diameter.
5.3.3	Vendor-Id AVP	C	

5.3.4	Firmware-Revision AVP	C	
5.3.5	Host-IP-Address AVP	C	
5.3.6	Supported-Vendor-Id AVP	C	
5.3.7	Product-Name AVP	C	
5.4	Disconnecting Peer Connections	C	DPA will not be answered with error: a peer that wants to a avoid a race can wait for pending answers before sending DPR.
5.4.1	Disconnect-Peer-Request	C	DPR is sent by diameter in response to configuration changes requiring a connection to be broken. The user can also send DPR.
5.4.2	Disconnect-Peer-Answer	C	DPR is answered by diameter.
5.4.3	Disconnect-Cause AVP	C	
5.5	Transport Failure Detection	—	
5.5.1	Device-Watchdog-Request	C	DWR is sent and received by diameter. Callbacks notify the user of transitions into and out of the OKAY state.
5.5.2	Device-Watchdog-Answer	C	DWA is sent and received by diameter.
5.5.3	Transport Failure Algorithm	C	
5.5.4	Failover and Failback Procedures	C	
5.6	Peer State Machine	PC	The election process is modified as described in 5.6.4.
5.6.1	Incoming Connections	C	
5.6.2	Events	—	

5.6.3	Actions	—	
5.6.4	The Election Process	PC	As documented, the election assumes knowledge of a peer's DiameterIdentity when initiating a connection, which diameter doesn't require. Connections will be accepted if configuration allows multiple connections per peer to be established or there is no existing connection. Note that the election process is only applicable when multiple connections per peer is disallowed.
6	Diameter Message Processing	—	
6.1	Diameter Request Routing Overview	—	Routing is performed by the user. A callback from diameter provides a list of available suitable peer connections.
6.1.1	Originating a Request	C	Requests are constructed by the user; diameter sets header fields as defined in the relevant dictionary.
6.1.2	Sending a Request	C	
6.1.3	Receiving Requests	C	Loops are detected by diameter when the return value of a request callback asks that a request be forwarded. Loop detection in other cases is the responsibility of the user.
6.1.4	Processing Local Requests	C	The user decides whether or not to process a request locally in the request callback from diameter.
6.1.5	Request Forwarding	PC	See 2.6.
6.1.6	Request Routing	PC	See 2.7.

6.1.7	Predictive Loop Avoidance	C	See 6.1.3.
6.1.8	Redirecting Requests	PC	See 2.6.
6.1.9	Relaying and Proxying Requests	C	A Route-Record AVP is appended by diameter when the return value of a request callback asks that a request be forwarded. Appending the AVP in other cases is the responsibility of the user.
6.2	Diameter Answer Processing	C	Answer message are constructed by the user, except in the case of some protocol errors, in which case the procedures are followed.
6.2.1	Processing Received Answers	C	Answers with an unknown Hop-by-Hop Identifier are discarded.
6.2.2	Relaying and Proxying Answers	—	Modifying answers is the responsibility of the user in callbacks from diameter.
6.3	Origin-Host AVP	C	The order of AVPs in an encoded message is determined by the CCF of the message in question. AVPs defined in the RFC are defined in dictionaries provided by diameter. Their proper use in application messages is the responsibility of the user.
6.4	Origin-Realm AVP	C	
6.5	Destination-Host AVP	C	
6.6	Destination-Realm AVP	C	
6.7	Routing AVPs	—	
6.7.1	Route-Record AVP	C	
6.7.2	Proxy-Info AVP	C	

6.7.3	Proxy-Host AVP	C	
6.7.4	Proxy-State AVP	C	
6.8	Auth-Application-Id AVP	C	
6.9	Acct-Application-Id AVP	C	
6.10	Inband-Security-Id AVP	C	See 2.1.
6.11	Vendor-Specific-Application-Id AVP	C	Note that the CCF of this AVP is not the same as in RFC 3588.
6.12	Redirect-Host AVP	C	
6.13	Redirect-Host-Usage AVP	C	
6.14	Redirect-Max-Cache-Time AVP	C	
7	Error Handling	C	Answers are formulated by the user in most cases. Answers setting the E-bit can be sent by diameter itself in response to a request that cannot be handled by the user.
7.1	Result-Code AVP	C	
7.1.1	Informational	C	
7.1.2	Success	C	
7.1.3	Protocol Errors	C	Result codes 3001, 3002, 3005, and 3007 can be sent in answers formulated by diameter, if configured to do so.
7.1.4	Transient Failures	C	Result code 4003 is sent in CEA if there is an existing connection to the peer in question and configuration does not allow more than one.
7.1.5	Permanent Failures	C	Message reception detects 5001, 5004, 5005, 5008, 5009, 5010, 5011, 5014, 5015, and 5017 errors. It ignores 5013 errors at the

			admonition of sections 3 and 4.1. Note that RFC 3588 did not allow 5xxx result codes in answers setting the E-bit, while RFC 6733 does. This is a potential interoperability problem since the Diameter protocol version has not changed.
7.2	Error Bit	C	
7.3	Error-Message AVP	C	The user can include this AVP as required.
7.4	Error-Reporting-Host AVP	C	The user can include this AVP as required.
7.5	Failed-AVP AVP	C	The user constructs application-specific messages, but diameter provides failed AVPs in message callbacks. Failed component AVPs are grouped within the relevant Grouped AVPs.
7.6	Experimental-Result AVP	C	
7.7	Experimental-Result-Code AVP	C	
8	Diameter User Sessions	—	Authorization and accounting AVPs are defined in provided dictionaries. Their proper use is the responsibility of the user.
8.1	Authorization Session State Machine	—	Authorization is the responsibility of the user: diameter does not implement this state machine.
8.2	Accounting Session State Machine	—	Accounting is the responsibility of the user: diameter does not

			implement this state machine.
8.3	Server-Initiated Re-Auth	—	
8.3.1	Re-Auth-Request	C	
8.3.2	Re-Auth-Answer	C	
8.4	Session Termination	—	Session-related messages and AVPs are defined in provided dictionaries. Their proper use is the user's responsibility.
8.4.1	Session-Termination-Request	C	
8.4.2	Session-Termination-Answer	C	
8.5	Aborting a Session	—	Session-related messages and AVPs are defined in provided dictionaries. Their proper use is the user's responsibility.
8.5.1	Abort-Session-Request	C	
8.5.2	Abort-Session-Answer	C	
8.6	Inferring Session Termination from Origin-State-Id	—	Session-related messages and AVPs are defined in provided dictionaries. Their proper use is the user's responsibility.
8.7	Auth-Request-Type AVP	C	
8.8	Session-Id AVP	C	
8.9	Authorization-Lifetime AVP	C	
8.10	Auth-Grace-Period AVP	C	
8.11	Auth-Session-State AVP	C	
8.12	Re-Auth-Request-Type AVP	C	
8.13	Session-Timeout AVP	C	

8.14	User-Name AVP	C	
8.15	Termination-Cause AVP	C	
8.16	Origin-State-Id AVP	C	
8.17	Session-Binding AVP	C	
8.18	Session-Server-Failover AVP	C	
8.19	Multi-Round-Time-Out AVP	C	
8.20	Class AVP	C	
8.21	Event-Timestamp AVP	C	
9	Accounting	—	Accounting-related messages and AVPs are defined in provided dictionaries. Their proper use is the user's responsibility.
9.1	Server Directed Model	—	
9.2	Protocol Messages	—	
9.3	Accounting Application Extension and Requirements	—	
9.4	Fault Resilience	—	
9.5	Accounting Records	—	
9.6	Correlation of Accounting Records	—	
9.7	Accounting Command Codes	—	
9.7.1	Accounting-Request	C	
9.7.2	Accounting-Answer	C	
9.8	Accounting AVPs	—	
9.8.1	Accounting-Record-Type AVP	C	
9.8.2	Acct-Interim-Interval AVP	C	

9.8.3	Accounting-Record-Number AVP	C	
9.8.4	Acct-Session-Id AVP	C	
9.8.5	Acct-Multi-Session-Id AVP	C	
9.8.6	Accounting-Sub-Session-Id AVP	C	
9.8.7	Accounting-Realtime-Required AVP	C	
10	AVP Occurrence Tables	—	
10.1	Base Protocol Command AVP Table	—	
10.2	Accounting AVP Table	—	
11	IANA Considerations	—	
11.1	AVP Header	—	
11.1.1	AVP Codes	—	
11.1.2	AVP Flags	—	
11.2	Diameter Header	—	
11.2.1	Command Codes	—	
11.2.2	Command Flags		
11.3	AVP Values	—	
11.3.1	Experimental-Result-Code AVP	—	
11.3.2	Result-Code AVP Values	—	
11.3.3	Accounting-Record-Type AVP Values	—	
11.3.4	Termination-Cause AVP Values	—	
11.3.5	Redirect-Host-Usage AVP Values	—	
11.3.6	Session-Server-Failover AVP Values	—	

11.3.7	Session-Binding AVP Values	—	
11.3.8	Disconnect-Cause AVP Values	—	
11.3.9	Auth-Request-Type AVP Values	—	
11.3.10	Auth-Session-State AVP Values	—	
11.3.11	Re-Auth-Request-Type AVP Values	—	
11.3.12	Accounting-Realtime-Required AVP Values	—	
11.3.13	Inband-Security-Id AVP (code 299)	—	
11.4	_diameters Service Name and Port Number Registration	—	
11.5	SCTP Payload Protocol Identifiers	—	
11.6	S-NAPTR Parameters	—	
12	Diameter Protocol-Related Configurable Parameters	—	
13	Security Considerations	PC	See 2.1. IPsec is transparent to diameter.
13.1	TLS/TCP and DTLS/ SCTP Usage	PC	See 2.1.
13.2	Peer-to-Peer Considerations	—	
13.3	AVP Considerations	—	
14	References	—	
14.1	Normative References	—	
14.2	Informative References	—	

Table 4.1: RFC 6733 Compliance

2 Reference Manual

The Diameter application is a framework for building applications on top of the Diameter protocol.

diameter

Erlang module

This module provides the interface with which a user can implement a Diameter node that sends and receives messages using the Diameter protocol as defined in RFC 6733.

Basic usage consists of creating a representation of a locally implemented Diameter node and its capabilities with *start_service/2*, adding transport capability using *add_transport/2* and sending Diameter requests and receiving Diameter answers with *call/4*. Incoming Diameter requests are communicated as callbacks to a *diameter_app(3)* callback modules as specified in the service configuration.

Beware the difference between **diameter** (not capitalized) and **Diameter** (capitalized). The former refers to the Erlang application named diameter whose main api is defined here, the latter to Diameter protocol in the sense of RFC 6733.

The diameter application must be started before calling most functions in this module.

DATA TYPES

```
Address()
DiameterIdentity()
Grouped()
OctetString()
Time()
Unsigned32()
UTF8String()
```

Types corresponding to RFC 6733 AVP Data Formats. Defined in *diameter_dict(4)*.

```
application_alias() = term()
```

Name identifying a Diameter application in service configuration. Passed to *call/4* when sending requests defined by the application.

```
application_module() = Mod | [Mod | ExtraArgs] | #diameter_callback{}
```

```
Mod = atom()
ExtraArgs = list()
```

Module implementing the callback interface defined in *diameter_app(3)*, along with any extra arguments to be appended to those documented. Note that extra arguments specific to an outgoing request can be specified to *call/4*, in which case those are are appended to any module-specific extra arguments.

Specifying a `#diameter_callback{}` record allows individual functions to be configured in place of the usual *diameter_app(3)* callbacks. See `diameter_callback.erl` for details.

```
application_opt()
```

Options defining a Diameter application. Has one the following types.

`{alias, application_alias()}`

Unique identifier for the application in the scope of the service. Defaults to the value of the `dictionary` option.

`{dictionary, atom()}`

Name of an encode/decode module for the Diameter messages defined by the application. These modules are generated from files whose format is documented in *diameter_dict(4)*.

`{module, application_module()}`

Callback module in which messages of the Diameter application are handled. See *diameter_app(3)* for the required interface and semantics.

`{state, term()}`

Initial callback state. The prevailing state is passed to some *diameter_app(3)* callbacks, which can then return a new state. Defaults to the value of the `alias` option.

`{call_mutates_state, true|false}`

Whether or not the *pick_peer/4* application callback can modify the application state. Defaults to `false`.

Warning:

pick_peer/4 callbacks are serialized when this option is `true`, which is a potential performance bottleneck. A simple Diameter client may suffer no ill effects from using mutable state but a server or agent that responds to incoming request should probably avoid it.

`{answer_errors, callback|report|discard}`

Manner in which incoming answer messages containing decode errors are handled.

If `callback` then errors result in a *handle_answer/4* callback in the same fashion as for *handle_request/3*, with errors communicated in the `errors` field of the `#diameter_packet{}` passed to the callback. If `report` then an answer containing errors is discarded without a callback and a warning report is written to the log. If `discard` then an answer containing errors is silently discarded without a callback. In both the `report` and `discard` cases the return value for the *call/4* invocation in question is as if a callback had taken place and returned `{error, failure}`.

Defaults to `discard`.

`{request_errors, answer_3xxx|answer|callback}`

Manner in which incoming requests are handled when an error other than 3007 (DIAMETER_APPLICATION_UNSUPPORTED, which cannot be associated with an application callback module), is detected.

If `answer_3xxx` then requests are answered without a *handle_request/3* callback taking place. If `answer` then even 5xxx errors are answered without a callback unless the connection in question has configured the RFC 3588 common dictionary as noted below. If `callback` then a *handle_request/3* callback always takes place and its return value determines the answer sent to the peer, if any.

Defaults to `answer_3xxx`.

Note:

Answers sent by diameter set the E-bit in the Diameter Header. Since RFC 3588 allows only 3xxx result codes in an `answer-message`, `answer` has the same semantics as `answer_3xxx` when the transport in question has been configured with `diameter_gen_base_rfc3588` as its common dictionary. Since RFC 6733 allows both 3xxx and 5xxx result codes in an `answer-message`, a transport with `diameter_gen_base_rfc6733` as its common dictionary does distinguish between `answer_3xxx` and `answer`.

`call_opt()`

Options available to *call/4* when sending an outgoing Diameter request. Has one of the following types.

`{extra, list()}`

Extra arguments to append to callbacks to the callback module in question. These are appended to any extra arguments configured on the callback itself. Multiple options append to the argument list.

`{filter, peer_filter()}`

Filter to apply to the list of available peers before passing it to the *pick_peer/4* callback for the application in question. Multiple options are equivalent a single `all` filter on the corresponding list of filters. Defaults to `none`.

`{peer, diameter_app:peer_ref()}`

Peer to which the request in question can be sent, preempting the selection of peers having advertised support for the Diameter application in question. Multiple options can be specified, and their order is respected in the candidate lists passed to a subsequent *pick_peer/4* callback.

`{timeout, Unsigned32()}`

Number of milliseconds after which the request should timeout. Defaults to 5000.

`detach`

Cause *call/4* to return `ok` as soon as the request in question has been encoded, instead of waiting for and returning the result from a subsequent *handle_answer/4* or *handle_error/4* callback.

An invalid option will cause *call/4* to fail.

`capability()`

AVP values sent in outgoing CER or CEA messages during capabilities exchange. Can be configured both on a service and a transport, values on the latter taking precedence. Has one of the following types.

```
{'Origin-Host', DiameterIdentity()}
{'Origin-Realm', DiameterIdentity()}
{'Host-IP-Address', [Address()]}
```

An address list is available to the start function of a *transport module*, which can return a new list for use in the subsequent CER or CEA. Host-IP-Address need not be specified if the transport module in question communicates an address list as described in *diameter_transport(3)*

```
{'Vendor-Id', Unsigned32()}
{'Product-Name', UTF8String()}
{'Origin-State-Id', Unsigned32()}
```

Origin-State-Id is optional but, if configured, will be included in outgoing CER/CEA and DWR/DWA messages. Setting a value of `0` (zero) is equivalent to not setting a value, as documented in RFC 6733. The function *origin_state_id/0* can be used as to retrieve a value that is computed when the diameter application is started.

```
{'Supported-Vendor-Id', [Unsigned32()]}
{'Auth-Application-Id', [Unsigned32()]}
{'Inband-Security-Id', [Unsigned32()]}
```

Inband-Security-Id defaults to the empty list, which is equivalent to a list containing only 0 (NO_INBAND_SECURITY). If 1 (TLS) is specified then TLS is selected if the CER/CEA received from the peer offers it.

```
{'Acct-Application-Id', [Unsigned32()]}
{'Vendor-Specific-Application-Id', [Grouped()]}
{'Firmware-Revision', Unsigned32()}
```

Note that each tuple communicates one or more AVP values. It is an error to specify duplicate tuples.

`eval() = {M,F,A} | fun() | [eval() | A]`

An expression that can be evaluated as a function in the following sense.

```
eval([{M,F,A} | T]) ->
    apply(M, F, T ++ A);
eval([[F|A] | T]) ->
    eval([F | T ++ A]);
eval([F|A]) ->
    apply(F, A);
eval(F) ->
    eval([F]).
```

Applying an *`eval()`* `E` to an argument list `A` is meant in the sense of `eval([E|A])`.

Warning:

Beware of using fun expressions of the form `fun Name/Arity` in situations in which the fun is not short-lived and code is to be upgraded at runtime since any processes retaining such a fun will have a reference to old code. In particular, such a value is typically inappropriate in configuration passed to *start_service/2* or *add_transport/2*.

`peer_filter() = term()`

Filter passed to *call/4* in order to select candidate peers for a *pick_peer/4* callback. Has one of the following types.

`none`

Matches any peer. This is a convenience that provides a filter equivalent to no filter.

`host`

Matches only those peers whose Origin-Host has the same value as Destination-Host in the outgoing request in question, or any peer if the request does not contain a Destination-Host AVP.

`realm`

Matches only those peers whose Origin-Realm has the same value as Destination-Realm in the outgoing request in question, or any peer if the request does not contain a Destination-Realm AVP.

`{host, any|`*`DiameterIdentity()`*`}`

Matches only those peers whose Origin-Host has the specified value, or all peers if the atom `any`.

`{realm, any|`*`DiameterIdentity()`*`}`

Matches only those peers whose Origin-Realm has the specified value, or all peers if the atom `any`.

`{eval, `*`eval()`*`}`

Matches only those peers for which the specified *`eval()`* returns `true` when applied to the connection's `diameter_caps` record. Any other return value or exception is equivalent to `false`.

`{neg, `*`peer_filter()`*`}`

Matches only those peers not matched by the specified filter.

`{all, [`*`peer_filter()`*`]}`

Matches only those peers matched by each filter in the specified list.

`{any, [`*`peer_filter()`*`]}`

Matches only those peers matched by at least one filter in the specified list. The resulting list will be in match order, peers matching the first filter of the list sorting before those matched by the second, and so on.

`{first, [peer_filter()]}`

Like `any`, but stops at the first filter for which there are matches, which can be much more efficient when there are many peers. For example, the following filter causes only peers best matching both the host and realm filters to be presented.

```
{first, [{all, [host, realm]}, realm]}
```

An invalid filter is equivalent to `{any,[]}`, a filter that matches no peer.

Note:

The `host` and `realm` filters cause the Destination-Host and Destination-Realm AVPs to be extracted from the outgoing request, assuming it to be a record- or list-valued *`diameter_codec:message()`*, and assuming at most one of each AVP. If this is not the case then the `{host|realm, DiameterIdentity()}` filters must be used to achieve the desired result. An empty *`DiameterIdentity()`* (which should not be typical) matches all hosts/realms for the purposes of filtering.

Warning:

A `host` filter is not typically desirable when setting Destination-Host since it will remove peer agents from the candidates list.

`service_event() = #diameter_event{service = service_name(), info = service_event_info()}`

An event message sent to processes that have subscribed to these using *subscribe/1*.

`service_event_info() = term()`

The `info` field of a *service_event()* record. Can have one of the following types.

`start`
`stop`

The service is being started or stopped. No event precedes a `start` event. No event follows a `stop` event, and this event implies the termination of all transport processes.

`{up, Ref, Peer, Config, Pkt}`
`{up, Ref, Peer, Config}`
`{down, Ref, Peer, Config}`

```
Ref    = transport_ref()
Peer   = diameter_app:peer()
Config = {connect|listen, [transport_opt()]}
Pkt    = #diameter_packet{}
```

The RFC 3539 watchdog state machine has transitioned into (`up`) or out of (`down`) the OKAY state. If a `#diameter_packet{}` is present in an `up` event then there has been a capabilities exchange on a newly established transport connection and the record contains the received CER or CEA.

Note that a single `up` or `down` event for a given peer corresponds to multiple *peer_up/3* or *peer_down/3* callbacks, one for each of the Diameter applications negotiated during capabilities exchange. That is, the event communicates connectivity with the peer as a whole while the callbacks communicate connectivity with respect to individual Diameter applications.

`{reconnect, Ref, Opts}`

```
Ref  = transport_ref()
Opts = [transport_opt()]
```

A connecting transport is attempting to establish/reestablish a transport connection with a peer following *connect_timer* or *watchdog_timer* expiry.

`{closed, Ref, Reason, Config}`

```
Ref = transport_ref()
Config = {connect|listen, [transport_opt()]}
```

Capabilities exchange has failed. `Reason` can have one of the following types.

`{'CER', Result, Caps, Pkt}`

```
Result = ResultCode | {capabilities_cb, CB, ResultCode|discard}
Caps = #diameter_caps{}
Pkt  = #diameter_packet{}
ResultCode = integer()
CB = eval()
```

An incoming CER has been answered with the indicated result code, or discarded. `Caps` contains pairs of values, for the local node and remote peer respectively. `Pkt` contains the CER in question. In the case of rejection by a capabilities callback, the tuple contains the rejecting callback.

`{'CER', Caps, {ResultCode, Pkt}}`

```
ResultCode = integer()
Caps = #diameter_caps{}
Pkt  = #diameter_packet{}
```

An incoming CER contained errors and has been answered with the indicated result code. `Caps` contains values for the local node only. `Pkt` contains the CER in question.

`{'CER', timeout}`

An expected CER was not received within *capx_timeout* of connection establishment.

`{'CEA', Result, Caps, Pkt}`

```
Result = ResultCode | atom() | {capabilities_cb, CB, ResultCode|discard}
Caps = #diameter_caps{}
Pkt  = #diameter_packet{}
ResultCode = integer()
```

An incoming CEA has been rejected for the indicated reason. An integer-valued `Result` indicates the result code sent by the peer. `Caps` contains pairs of values for the local node and remote peer. `Pkt` contains the CEA in question. In the case of rejection by a capabilities callback, the tuple contains the rejecting callback.

`{'CEA', Caps, Pkt}`

```
Caps = #diameter_caps{}
Pkt  = #diameter_packet{}
```

An incoming CEA contained errors and has been rejected. `Caps` contains only values for the local node. `Pkt` contains the CEA in question.

`{'CEA', timeout}`

An expected CEA was not received within *capx_timeout* of connection establishment.

`{watchdog, Ref, PeerRef, {From, To}, Config}`

```
Ref = transport_ref()
PeerRef = diameter_app:peer_ref()
From, To = initial | okay | suspect | down | reopen
Config = {connect|listen, [transport_opt()]}
```

An RFC 3539 watchdog state machine has changed state.

`any()`

For forward compatibility, a subscriber should be prepared to receive info fields of forms other than the above.

`service_name() = term()`

Name of a service as passed to *start_service/2* and with which the service is identified. There can be at most one service with a given name on a given node. Note that *erlang:make_ref/0* can be used to generate a service name that is somewhat unique.

`service_opt()`

Option passed to *start_service/2*. Can be any *`capability()`* as well as the following.

`{application, [application_opt()]}`

A Diameter application supported by the service.

A service must configure one tuple for each Diameter application it intends to support. For an outgoing request, the relevant *`application_alias()`* is passed to *call/4*, while for an incoming request the application identifier in the message header determines the application, the identifier being specified in the application's *dictionary* file.

Warning:

The capabilities advertised by a node must match its configured applications. In particular, `application` configuration must be matched by corresponding *capability()* configuration, of *-Application-Id AVPs in particular.

`{decode_format, record | list | map | none}`

The format of decoded messages and grouped AVPs in the `msg` field of diameter_packet records and `value` field of diameter_avp records respectively. If `record` then a record whose definition is generated from the dictionary file in question. If `list` or `map` then a `[Name | Avps]` pair where `Avps` is a list of AVP name/values pairs or a map keyed on AVP names respectively. If `none` then the atom-value message name, or `undefined` for a Grouped AVP. See also *diameter_codec:message()*.

Defaults to `record`.

Note:

AVPs are decoded into a list of diameter_avp records in `avps` field of diameter_packet records independently of `decode_format`.

`{restrict_connections, false | node | nodes | [node()] | eval()}`

The degree to which the service allows multiple transport connections to the same peer, as identified by its Origin-Host at capabilities exchange.

If `[node()]` then a connection is rejected if another already exists on any of the specified nodes. Types `false`, `node`, `nodes` and *eval()* are equivalent to `[]`, `[node()]`, `[node()|nodes()]` and the evaluated value respectively, evaluation of each expression taking place whenever a new connection is to be established. Note that `false` allows an unlimited number of connections to be established with the same peer.

Multiple connections are independent and governed by their own peer and watchdog state machines.

Defaults to `nodes`.

`{sequence, {H,N} | eval()}`

A constant value `H` for the topmost `32-N` bits of of 32-bit End-to-End and Hop-by-Hop Identifiers generated by the service, either explicitly or as a return value of a function to be evaluated at *start_service/2*. In particular, an identifier `Id` is mapped to a new identifier as follows.

```
(H bsl N) bor (Id band ((1 bsl N) - 1))
```

Note that RFC 6733 requires that End-to-End Identifiers remain unique for a period of at least 4 minutes and that this and the call rate places a lower bound on appropriate values of `N`: at a rate of `R` requests per second, an `N`-bit counter traverses all of its values in `(1 bsl N) div (R*60)` minutes, so the bound is `4*R*60 =< 1 bsl N`.

`N` must lie in the range `0..32` and `H` must be a non-negative integer less than `1 bsl (32-N)`.

Defaults to `{0,32}`.

Warning:

Multiple Erlang nodes implementing the same Diameter node should be configured with different sequence masks to ensure that each node uses a unique range of End-to-End and Hop-by-Hop Identifiers for outgoing requests.

`{share_peers, boolean() | [node()] | eval()}`

Nodes to which peer connections established on the local Erlang node are communicated. Shared peers become available in the remote candidates list passed to *pick_peer/4* callbacks on remote nodes whose services are configured to use them: see `use_shared_peers` below.

If `false` then peers are not shared. If `[node()]` then peers are shared with the specified list of nodes. If `eval()` then peers are shared with the nodes returned by the specified function, evaluated whenever a peer connection becomes available or a remote service requests information about local connections. The value `true` is equivalent to `fun erlang:nodes/0`. The value `node()` in a list is ignored, so a collection of services can all be configured to share with the same list of nodes.

Defaults to `false`.

> **Note:**
>
> Peers are only shared with services of the same name for the purpose of sending outgoing requests. Since the value of the *application_opt()* `alias`, passed to *call/4*, is the handle for identifying a peer as a suitable candidate, services that share peers must use the same aliases to identify their supported applications. They should typically also configure identical *capabilities()*, since by sharing peer connections they are distributing the implementation of a single Diameter node across multiple Erlang nodes.

`{strict_arities, boolean() | encode | decode}`

Whether or not to require that the number of AVPs in a message or grouped AVP agree with those specified in the dictionary in question when passing messages to *diameter_app(3)* callbacks. If `true` then mismatches in an outgoing messages cause message encoding to fail, while mismatches in an incoming message are reported as 5005/5009 errors in the errors field of the diameter_packet record passed to *handle_request/3* or *handle_answer/4* callbacks. If `false` then neither error is enforced/detected. If `encode` or `decode` then errors are only enforced/detected on outgoing or incoming messages respectively.

Defaults to `true`.

> **Note:**
>
> Disabling arity checks affects the form of messages at encode/decode. In particular, decoded AVPs are represented as lists of values, regardless of the AVP's arity (ie. expected number in the message/AVP grammar in question), and values are expected to be supplied as lists at encode. This differs from the historic decode behaviour of representing AVPs of arity 1 as bare values, not wrapped in a list.

`{string_decode, boolean()}`

Whether or not to decode AVPs of type *OctetString()* and its derived types *DiameterIdentity()*, *DiameterURI()*, *IPFilterRule()*, *QoSFilterRule()*, and *UTF8String()*. If `true` then AVPs of these types are decoded to string(). If `false` then values are retained as binary().

Defaults to `true`.

> **Warning:**
>
> This option should be set to `false` since a sufficiently malicious peer can otherwise cause large amounts of memory to be consumed when decoded Diameter messages are passed between processes. The default value is for backwards compatibility.

`{traffic_counters, boolean()}`

Whether or not to count application-specific messages; those for which *diameter_app(3)* callbacks take place. If false then only messages handled by diameter itself are counted: CER/CEA, DWR/DWA, DPR/DPA.

Defaults to `true`.

> **Note:**
>
> Disabling counters is a performance improvement, but means that the omitted counters are not returned by *service_info/2*.

`{use_shared_peers, boolean() | [node()] | eval()}`

Nodes from which communicated peers are made available in the remote candidates list of *pick_peer/4* callbacks.

If `false` then remote peers are not used. If `[node()]` then only peers from the specified list of nodes are used. If `eval()` then only peers returned by the specified function are used, evaluated whenever a remote service communicates information about an available peer connection. The value `true` is equivalent to `fun erlang:nodes/0`. The value `node()` in a list is ignored.

Defaults to `false`.

Note:

A service that does not use shared peers will always pass the empty list as the second argument of *pick_peer/4* callbacks.

Warning:

Sending a request over a peer connection on a remote node is less efficient than sending it over a local connection. It may be preferable to make use of the *service_opt()* `restrict_connections` and maintain a dedicated connection on each node from which requests are sent.

`transport_opt()`

Any transport option except `applications` or `capabilities`. Used as defaults for transport configuration, values passed to *add_transport/2* overriding values configured on the service.

`transport_opt()`

Option passed to *add_transport/2*. Has one of the following types.

`{applications, [application_alias()]}`

Diameter applications to which the transport should be restricted. Defaults to all applications configured on the service in question. Applications not configured on the service in question are ignored.

Warning:

The capabilities advertised by a node must match its configured applications. In particular, setting `applications` on a transport typically implies having to set matching *-Application-Id AVPs in a *capabilities()* tuple.

`{avp_dictionaries, [module()]}`

A list of alternate dictionary modules with which to encode/decode AVPs that are not defined by the dictionary of the application in question. At decode, such AVPs are represented as diameter_avp records in the `'AVP'` field of a decoded message or Grouped AVP, the first alternate that succeeds in decoding the AVP setting the record's value field. At encode, values in an `'AVP'` list can be passed as AVP name/value 2-tuples, and it is an encode error for no alternate to define the AVP of such a tuple.

Defaults to the empty list.

> **Note:**
>
> The motivation for alternate dictionaries is RFC 7683, Diameter Overload Indication Conveyance (DOIC), which defines AVPs to be piggybacked onto existing application messages rather than defining an application of its own. The DOIC dictionary is provided by the diameter application, as module `diameter_gen_doic_rfc7683`, but alternate dictionaries can be used to encode/decode any set of AVPs not known to an application dictionary.

`{capabilities, [`*`capability()`*`]}`

AVPs used to construct outgoing CER/CEA messages. Values take precedence over any specified on the service in question.

Specifying a capability as a transport option may be particularly appropriate for Inband-Security-Id, in case TLS is desired over TCP as implemented by *diameter_tcp(3)*.

`{capabilities_cb,` *`eval()`*`}`

Callback invoked upon reception of CER/CEA during capabilities exchange in order to ask whether or not the connection should be accepted. Applied to the *`transport_ref()`* and `#diameter_caps{}` record of the connection.

The return value can have one of the following types.

`ok`

Accept the connection.

`integer()`

Causes an incoming CER to be answered with the specified Result-Code.

`discard`

Causes an incoming CER to be discarded without CEA being sent.

`unknown`

Equivalent to returning `3010`, DIAMETER_UNKNOWN_PEER.

Returning anything but `ok` or a 2xxx series result code causes the transport connection to be broken. Multiple *capabilities_cb* options can be specified, in which case the corresponding callbacks are applied until either all return `ok` or one does not.

`{capx_timeout,` *`Unsigned32()`*`}`

Number of milliseconds after which a transport process having an established transport connection will be terminated if the expected capabilities exchange message (CER or CEA) is not received from the peer. For a connecting transport, the timing of connection attempts is governed by *connect_timer* or *watchdog_timer* expiry. For a listening transport, the peer determines the timing.

Defaults to 10000.

`{connect_timer, Tc}`

```
Tc = Unsigned32()
```

For a connecting transport, the RFC 6733 Tc timer, in milliseconds. This timer determines the frequency with which a transport attempts to establish an initial connection with its peer following transport configuration. Once an initial connection has been established, *watchdog_timer* determines the frequency of reconnection attempts, as required by RFC 3539.

For a listening transport, the timer specifies the time after which a previously connected peer will be forgotten: a connection after this time is regarded as an initial connection rather than reestablishment, causing the RFC 3539 state machine to pass to state OKAY rather than REOPEN. Note that these semantics are not governed by the RFC and that a listening transport's *connect_timer* should be greater than its peer's Tw plus jitter.

Defaults to 30000 for a connecting transport and 60000 for a listening transport.

`{disconnect_cb, eval()}`

Callback invoked prior to terminating the transport process of a transport connection having watchdog state `OKAY`. Applied to `application|service|transport` and the *`transport_ref()`* and *`diameter_app:peer()`* in question: `application` indicates that the diameter application is being stopped, `service` that the service in question is being stopped by *stop_service/1*, and `transport` that the transport in question is being removed by *remove_transport/2*.

The return value can have one of the following types.

`{dpr, [option()]}`

Send Disconnect-Peer-Request to the peer, the transport process being terminated following reception of Disconnect-Peer-Answer or timeout. An `option()` can be one of the following.

`{cause, 0|rebooting|1|busy|2|goaway}`

Disconnect-Cause to send, `REBOOTING`, `BUSY` and `DO_NOT_WANT_TO_TALK_TO_YOU` respectively. Defaults to `rebooting` for `Reason=service|application` and `goaway` for `Reason=transport`.

`{timeout, Unsigned32()}`

Number of milliseconds after which the transport process is terminated if DPA has not been received. Defaults to the value of *dpa_timeout*.

`dpr`

Equivalent to `{dpr, []}`.

`close`

Terminate the transport process without Disconnect-Peer-Request being sent to the peer.

`ignore`

Equivalent to not having configured the callback.

Multiple *disconnect_cb* options can be specified, in which case the corresponding callbacks are applied until one of them returns a value other than `ignore`. All callbacks returning `ignore` is equivalent to not having configured them.

Defaults to a single callback returning `dpr`.

`{dpa_timeout, Unsigned32()}`

Number of milliseconds after which a transport connection is terminated following an outgoing DPR if DPA is not received.

Defaults to 1000.

`{dpr_timeout, Unsigned32()}`

Number of milliseconds after which a transport connection is terminated following an incoming DPR if the peer does not close the connection.

Defaults to 5000.

`{incoming_maxlen, 0..16777215}`

Bound on the expected size of incoming Diameter messages. Messages larger than the specified number of bytes are discarded.

Defaults to `16777215`, the maximum value of the 24-bit Message Length field in a Diameter Header.

`{length_errors, exit|handle|discard}`

How to deal with errors in the Message Length field of the Diameter Header in an incoming message. An error in this context is that the length is not at least 20 bytes (the length of a Header), is not a multiple of 4 (a valid length) or is not the length of the message in question, as received over the transport interface documented in *diameter_transport(3)*.

If `exit` then the transport process in question exits. If `handle` then the message is processed as usual, a resulting *handle_request/3* or *handle_answer/4* callback (if one takes place) indicating the `5015` error (DIAMETER_INVALID_MESSAGE_LENGTH). If `discard` then the message in question is silently discarded.

Defaults to `exit`.

> **Note:**
>
> The default value reflects the fact that a transport module for a stream-oriented transport like TCP may not be able to recover from a message length error since such a transport must use the Message Length header to divide the incoming byte stream into individual Diameter messages. An invalid length leaves it with no reliable way to rediscover message boundaries, which may result in the failure of subsequent messages. See *diameter_tcp(3)* for the behaviour of that module.

`{pool_size, pos_integer()}`

Number of transport processes to start. For a listening transport, determines the size of the pool of accepting transport processes, a larger number being desirable for processing multiple concurrent peer connection attempts. For a connecting transport, determines the number of connections to the peer in question that will be attempted to be establshed: the *service_opt()*: `restrict_connections` should also be configured on the service in question to allow multiple connections to the same peer.

`{spawn_opt, [term()]}`

Options passed to *erlang:spawn_opt/2* when spawning a process for an incoming Diameter request. Options `monitor` and `link` are ignored.

Defaults to the empty list.

`{strict_capx, boolean()]}`

Whether or not to enforce the RFC 6733 requirement that any message before capabilities exchange should close the peer connection. If false then unexpected messages are discarded.

Defaults to true. Changing this results in non-standard behaviour, but can be useful in case peers are known to be behave badly.

`{strict_mbit, boolean()}`

Whether or not to regard an AVP setting the M-bit as erroneous when the command grammar in question does not explicitly allow the AVP. If `true` then such AVPs are regarded as 5001 errors, DIAMETER_AVP_UNSUPPORTED. If `false` then the M-bit is ignored and policing it becomes the receiver's responsibility.

Defaults to `true`.

> **Warning:**
>
> RFC 6733 is unclear about the semantics of the M-bit. One the one hand, the CCF specification in section 3.2 documents AVP in a command grammar as meaning **any** arbitrary AVP; on the other hand, 1.3.4 states that AVPs setting the M-bit cannot be added to an existing command: the modified command must instead be placed in a new Diameter application.
>
> The reason for the latter is presumably interoperability: allowing arbitrary AVPs setting the M-bit in a command makes its interpretation implementation-dependent, since there's no guarantee that all implementations will understand the same set of arbitrary AVPs in the context of a given command. However, interpreting `AVP` in a command grammar as any AVP, regardless of M-bit, renders 1.3.4 meaningless, since the receiver can simply ignore any AVP it thinks isn't relevant, regardless of the sender's intent.
>
> Beware of confusing mandatory in the sense of the M-bit with mandatory in the sense of the command grammar. The former is a semantic requirement: that the receiver understand the semantics of the AVP in the context in question. The latter is a syntactic requirement: whether or not the AVP must occur in the message in question.

`{transport_config, term()}`
`{transport_config, term(), Unsigned32() | infinity}`

Term passed as the third argument to the *start/3* function of the relevant *transport module* in order to start a transport process. Defaults to the empty list.

The 3-tuple form additionally specifies an interval, in milliseconds, after which a started transport process should be terminated if it has not yet established a connection. For example, the following options on a connecting transport request a connection with one peer over SCTP or another (typically the same) over TCP.

```
{transport_module, diameter_sctp}
{transport_config, SctpOpts, 5000}
{transport_module, diameter_tcp}
{transport_config, TcpOpts}
```

To listen on both SCTP and TCP, define one transport for each.

`{transport_module, atom()}`

Module implementing a transport process as defined in *diameter_transport(3)*. Defaults to `diameter_tcp`.

Multiple `transport_module` and *transport_config* options are allowed. The order of these is significant in this case (and only in this case), a `transport_module` being paired with the first *transport_config* following it in the options list, or the default value for trailing modules. Transport starts will be attempted with each of the modules in order until one establishes a connection within the corresponding timeout (see below) or all fail.

`{watchdog_config, [{okay|suspect, non_neg_integer()}]}`

Configuration that alters the behaviour of the watchdog state machine. On key `okay`, the non-negative number of answered DWR messages before transitioning from REOPEN to OKAY. On key `suspect`, the number of watchdog timeouts before transitioning from OKAY to SUSPECT when DWR is unanswered, or 0 to not make the transition.

Defaults to `[{okay, 3}, {suspect, 1}]`. Not specifying a key is equivalent to specifying the default value for that key.

> **Warning:**
>
> The default value is as required by RFC 3539: changing it results in non-standard behaviour that should only be used to simulate misbehaving nodes during test.

`{watchdog_timer, TwInit}`

```
TwInit = Unsigned32()
       | {M,F,A}
```

The RFC 3539 watchdog timer. An integer value is interpreted as the RFC's TwInit in milliseconds, a jitter of ± 2 seconds being added at each rearming of the timer to compute the RFC's Tw. An MFA is expected to return the RFC's Tw directly, with jitter applied, allowing the jitter calculation to be performed by the callback.

An integer value must be at least 6000 as required by RFC 3539. Defaults to 30000.

Unrecognized options are silently ignored but are returned unmodified by *service_info/2* and can be referred to in predicate functions passed to *remove_transport/2*.

`transport_ref() = reference()`

Reference returned by *add_transport/2* that identifies the configuration.

Exports

`add_transport(SvcName, {connect|listen, [Opt]}) -> {ok, Ref} | {error, Reason}`

Types:

`SvcName = service_name()`
`Opt = transport_opt()`
`Ref = transport_ref()`
`Reason = term()`

Add transport capability to a service.

The service will start transport processes as required in order to establish a connection with the peer, either by connecting to the peer (`connect`) or by accepting incoming connection requests (`listen`). A connecting transport establishes transport connections with at most one peer, an listening transport potentially with many.

The diameter application takes responsibility for exchanging CER/CEA with the peer. Upon successful completion of capabilities exchange the service calls each relevant application module's *peer_up/3* callback after which the caller can exchange Diameter messages with the peer over the transport. In addition to CER/CEA, the service takes responsibility for the handling of DWR/DWA and required by RFC 3539, as well as for DPR/DPA.

The returned reference uniquely identifies the transport within the scope of the service. Note that the function returns before a transport connection has been established.

> **Note:**
>
> It is not an error to add a transport to a service that has not yet been configured: a service can be started after configuring its transports.

call(SvcName, App, Request, [Opt]) -> Answer | ok | {error, Reason}

Types:

SvcName = *service_name()*
App = *application_alias()*
Request = *diameter_codec:message()*
Answer = term()
Opt = *call_opt()*

Send a Diameter request message.

`App` specifies the Diameter application in which the request is defined and callbacks to the corresponding callback module will follow as described below and in *diameter_app(3)*. Unless the `detach` option is specified, the call returns either when an answer message is received from the peer or an error occurs. In the answer case, the return value is as returned by a *handle_answer/4* callback. In the error case, whether or not the error is returned directly by diameter or from a *handle_error/4* callback depends on whether or not the outgoing request is successfully encoded for transmission to the peer, the cases being documented below.

If there are no suitable peers, or if *pick_peer/4* rejects them by returning `false`, then `{error,no_connection}` is returned. Otherwise *pick_peer/4* is followed by a *prepare_request/3* callback, the message is encoded and then sent.

There are several error cases which may prevent an answer from being received and passed to a *handle_answer/4* callback:

- If the initial encode of the outgoing request fails, then the request process fails and `{error,encode}` is returned.
- If the request is successfully encoded and sent but the answer times out then a *handle_error/4* callback takes place with `Reason = timeout`.
- If the request is successfully encoded and sent but the service in question is stopped before an answer is received then a *handle_error/4* callback takes place with `Reason = cancel`.
- If the transport connection with the peer goes down after the request has been sent but before an answer has been received then an attempt is made to resend the request to an alternate peer. If no such peer is available, or if the subsequent *pick_peer/4* callback rejects the candidates, then a *handle_error/4* callback takes place with `Reason = failover`. If a peer is selected then a *prepare_retransmit/3* callback takes place, after which the semantics are the same as following an initial *prepare_request/3* callback.
- If an encode error takes place during retransmission then the request process fails and `{error,failure}` is returned.
- If an application callback made in processing the request fails (pick_peer, prepare_request, prepare_retransmit, handle_answer or handle_error) then either `{error,encode}` or `{error,failure}` is returned depending on whether or not there has been an attempt to send the request over the transport.

Note that `{error,encode}` is the only return value which guarantees that the request has **not** been sent over the transport connection.

origin_state_id() -> Unsigned32()

Return a reasonable value for use as Origin-State-Id in outgoing messages.

The value returned is the number of seconds since 19680120T031408Z, the first value that can be encoded as a Diameter *`Time()`*, at the time the diameter application was started.

remove_transport(SvcName, Pred) -> ok | {error, Reason}

Types:

SvcName = *service_name()*

```
Pred = Fun | MFA | transport_ref() | list() | true | false
Fun = fun((transport_ref(), connect|listen, list()) -> boolean())
    | fun((transport_ref(), list()) -> boolean())
    | fun((list()) -> boolean())
MFA = {atom(), atom(), list()}
Reason = term()
```

Remove previously added transports.

`Pred` determines which transports to remove. An arity-3-valued `Pred` removes all transports for which `Pred(Ref, Type, Opts)` returns `true`, where `Type` and `Opts` are as passed to *add_transport/2* and `Ref` is as returned by it. The remaining forms are equivalent to an arity-3 fun as follows.

```
Pred = fun(transport_ref(), list()):  fun(Ref, _, Opts) -> Pred(Ref, Opts) end
Pred = fun(list()):                   fun(_, _, Opts) -> Pred(Opts) end
Pred = transport_ref():               fun(Ref, _, _)  -> Pred == Ref end
Pred = list():                        fun(_, _, Opts) -> [] == Pred -- Opts end
Pred = true:                          fun(_, _, _) -> true end
Pred = false:                         fun(_, _, _) -> false end
Pred = {M,F,A}:  fun(Ref, Type, Opts) -> apply(M, F, [Ref, Type, Opts | A]) end
```

Removing a transport causes the corresponding transport processes to be terminated. Whether or not a DPR message is sent to a peer is controlled by value of *disconnect_cb* configured on the transport.

service_info(SvcName, Info) -> term()

Types:

```
SvcName = service_name()
Info = Item | [Info]
Item = atom()
```

Return information about a started service. Requesting info for an unknown service causes `undefined` to be returned. Requesting a list of items causes a tagged list to be returned.

`Item` can be one of the following.

```
'Origin-Host'
'Origin-Realm'
'Vendor-Id'
'Product-Name'
'Origin-State-Id'
'Host-IP-Address'
'Supported-Vendor'
'Auth-Application-Id'
'Inband-Security-Id'
'Acct-Application-Id'
'Vendor-Specific-Application-Id'
'Firmware-Revision'
```

Return a capability value as configured with *start_service/2*.

`applications`

Return the list of applications as configured with *start_service/2*.

`capabilities`

Return a tagged list of all capabilities values as configured with *start_service/2*.

`transport`

Return a list containing one entry for each of the service's transport as configured with *add_transport/2*. Each entry is a tagged list containing both configuration and information about established peer connections. An example return value with for a client service with Origin-Host "client.example.com" configured with a single transport connected to "server.example.com" might look as follows.

```
[[{ref,#Ref<0.0.0.93>},
  {type,connect},
  {options,[{transport_module,diameter_tcp},
            {transport_config,[{ip,{127,0,0,1}},
                               {raddr,{127,0,0,1}},
                               {rport,3868},
                               {reuseaddr,true}]}]},
  {watchdog,{<0.66.0>,-576460736368485571,okay}},
  {peer,{<0.67.0>,-576460736357885808}},
  {apps,[{0,common}]},
  {caps,[{origin_host,{"client.example.com","server.example.com"}},
         {origin_realm,{"example.com","example.com"}},
         {host_ip_address,{[{127,0,0,1}],[{127,0,0,1}]}},
         {vendor_id,{0,193}},
         {product_name,{"Client","Server"}},
         {origin_state_id,{[],[]}},
         {supported_vendor_id,{[],[]}},
         {auth_application_id,{[0],[0]}},
         {inband_security_id,{[],[0]}},
         {acct_application_id,{[],[]}},
         {vendor_specific_application_id,{[],[]}},
         {firmware_revision,{[],[]}},
         {avp,{[],[]}}]},
  {port,[{owner,<0.69.0>},
         {module,diameter_tcp},
         {socket,{{127,0,0,1},48758}},
         {peer,{{127,0,0,1},3868}},
         {statistics,[{recv_oct,656},
                      {recv_cnt,6},
                      {recv_max,148},
                      {recv_avg,109},
                      {recv_dvi,19},
                      {send_oct,836},
                      {send_cnt,6},
                      {send_max,184},
                      {send_avg,139},
                      {send_pend,0}]}]},
  {statistics,[{{{0,258,0},recv},3},
               {{{0,258,1},send},3},
               {{{0,258,0},recv,{'Result-Code',2001}},3},
               {{{0,257,0},recv},1},
               {{{0,257,1},send},1},
               {{{0,257,0},recv,{'Result-Code',2001}},1},
               {{{0,280,1},recv},2},
               {{{0,280,0},send},2},
               {{{0,280,0},send,{'Result-Code',2001}},2}]}]]
```

Here `ref` is a *`transport_ref()`* and `options` the corresponding *`transport_opt()`* list passed to *add_transport/2*. The `watchdog` entry shows the state of a connection's RFC 3539 watchdog state machine. The `peer` entry identifies the *`diameter_app:peer_ref()`* for which there will have been *peer_up/3* callbacks for the Diameter applications identified by the `apps` entry, `common` being the *`application_alias()`*. The `caps` entry identifies the capabilities sent by the local node and received from the peer during capabilities exchange. The `port` entry displays socket-level information about the transport connection. The `statistics`

entry presents Diameter-level counters, an entry like `{{{0,280,1},recv},2}` saying that the client has received 2 DWR messages: `{0,280,1} = {Application_Id, Command_Code, R_Flag}`.

Note that `watchdog`, `peer`, `apps`, `caps` and `port` entries depend on connectivity with the peer and may not be present. Note also that the `statistics` entry presents values accumulated during the lifetime of the transport configuration.

A listening transport presents its information slightly differently since there may be multiple accepted connections for the same *`transport_ref()`*. The `transport` info returned by a server with a single client connection might look as follows.

```
[[{ref,#Ref<0.0.0.61>},
  {type,listen},
  {options,[{transport_module,diameter_tcp},
            {transport_config,[{reuseaddr,true},
                               {ip,{127,0,0,1}},
                               {port,3868}]}]},
  {accept,[[{watchdog,{<0.56.0>,-576460739249514012,okay}},
            {peer,{<0.58.0>,-576460638229179167}},
            {apps,[{0,common}]},
            {caps,[{origin_host,{"server.example.com","client.example.com"}},
                   {origin_realm,{"example.com","example.com"}},
                   {host_ip_address,{[{127,0,0,1}],[{127,0,0,1}]}},
                   {vendor_id,{193,0}},
                   {product_name,{"Server","Client"}},
                   {origin_state_id,{[],[]}},
                   {supported_vendor_id,{[],[]}},
                   {auth_application_id,{[0],[0]}},
                   {inband_security_id,{[],[]}},
                   {acct_application_id,{[],[]}},
                   {vendor_specific_application_id,{[],[]}},
                   {firmware_revision,{[],[]}},
                   {avp,{[],[]}}]},
            {port,[{owner,<0.62.0>},
                   {module,diameter_tcp},
                   {socket,{{127,0,0,1},3868}},
                   {peer,{{127,0,0,1},48758}},
                   {statistics,[{recv_oct,1576},
                                {recv_cnt,16},
                                {recv_max,184},
                                {recv_avg,98},
                                {recv_dvi,26},
                                {send_oct,1396},
                                {send_cnt,16},
                                {send_max,148},
                                {send_avg,87},
                                {send_pend,0}]}]}],
           [{watchdog,{<0.72.0>,-576460638229717546,initial}}]]},
  {statistics,[{{{0,280,0},recv},7},
               {{{0,280,1},send},7},
               {{{0,280,0},recv,{'Result-Code',2001}},7},
               {{{0,258,1},recv},3},
               {{{0,258,0},send},3},
               {{{0,258,0},send,{'Result-Code',2001}},3},
               {{{0,280,1},recv},5},
               {{{0,280,0},send},5},
               {{{0,280,0},send,{'Result-Code',2001}},5},
               {{{0,257,1},recv},1},
               {{{0,257,0},send},1},
               {{{0,257,0},send,{'Result-Code',2001}},1}]}]]
```

The information presented here is as in the `connect` case except that the client connections are grouped under an `accept` tuple.

Whether or not the *transport_opt()* `pool_size` has been configured affects the format of the listing in the case of a connecting transport, since a value greater than 1 implies multiple transport processes for the same *`transport_ref()`*, as in the listening case. The format in this case is similar to the listening case, with a `pool` tuple in place of an `accept` tuple.

`connections`

Return a list containing one entry for every established transport connection whose watchdog state machine is not in the `down` state. This is a flat view of `transport` info which lists only active connections and for which Diameter-level statistics are accumulated only for the lifetime of the transport connection. A return value for the server above might look as follows.

```
[[{ref,#Ref<0.0.0.61>},
  {type,accept},
  {options,[{transport_module,diameter_tcp},
            {transport_config,[{reuseaddr,true},
                               {ip,{127,0,0,1}},
                               {port,3868}]}]},
  {watchdog,{<0.56.0>,-576460739249514012,okay}},
  {peer,{<0.58.0>,-57646063822917916 7}},
  {apps,[{0,common}]},
  {caps,[{origin_host,{"server.example.com","client.example.com"}},
         {origin_realm,{"example.com","example.com"}},
         {host_ip_address,{[{127,0,0,1}],[{127,0,0,1}]}},
         {vendor_id,{193,0}},
         {product_name,{"Server","Client"}},
         {origin_state_id,{[],[]}},
         {supported_vendor_id,{[],[]}},
         {auth_application_id,{[0],[0]}},
         {inband_security_id,{[],[]}},
         {acct_application_id,{[],[]}},
         {vendor_specific_application_id,{[],[]}},
         {firmware_revision,{[],[]}},
         {avp,{[],[]}}]},
  {port,[{owner,<0.62.0>},
         {module,diameter_tcp},
         {socket,{{127,0,0,1},3868}},
         {peer,{{127,0,0,1},48758}},
         {statistics,[{recv_oct,10124},
                      {recv_cnt,132},
                      {recv_max,184},
                      {recv_avg,76},
                      {recv_dvi,9},
                      {send_oct,10016},
                      {send_cnt,132},
                      {send_max,148},
                      {send_avg,75},
                      {send_pend,0}]}]},
  {statistics,[{{{0,280,0},recv},62},
               {{{0,280,1},send},62},
               {{{0,280,0},recv,{'Result-Code',2001}},62},
               {{{0,258,1},recv},3},
               {{{0,258,0},send},3},
               {{{0,258,0},send,{'Result-Code',2001}},3},
               {{{0,280,1},recv},66},
               {{{0,280,0},send},66},
               {{{0,280,0},send,{'Result-Code',2001}},66},
               {{{0,257,1},recv},1},
               {{{0,257,0},send},1},
               {{{0,257,0},send,{'Result-Code',2001}},1}]}]]
```

Note that there may be multiple entries with the same `ref`, in contrast to `transport` info.

`statistics`

Return a `{{Counter, Ref}, non_neg_integer()}` list of counter values. `Ref` can be either a *`transport_ref()`* or a *`diameter_app:peer_ref()`*. Entries for the latter are folded into corresponding entries for the former as peer connections go down. Entries for both are removed at *remove_transport/2*. The Diameter-level statistics returned by `transport` and `connections` info are based upon these entries.

`diameter_app:peer_ref()`

Return transport configuration associated with a single peer, as passed to *add_transport/2*. The returned list is empty if the peer is unknown. Otherwise it contains the `ref`, `type` and `options` tuples as in `transport` and `connections` info above. For example:

```
[{ref,#Ref<0.0.0.61>},
 {type,accept},
 {options,[{transport_module,diameter_tcp},
           {transport_config,[{reuseaddr,true},
                              {ip,{127,0,0,1}},
                              {port,3868}]}]}]
```

`services() -> [SvcName]`

Types:

`SvcName = service_name()`

Return the list of started services.

`session_id(Ident) -> OctetString()`

Types:

`Ident = DiameterIdentity()`

Return a value for a Session-Id AVP.

The value has the form required by section 8.8 of RFC 6733. Ident should be the Origin-Host of the peer from which the message containing the returned value will be sent.

`start() -> ok | {error, Reason}`

Start the diameter application.

The diameter application must be started before starting a service. In a production system this is typically accomplished by a boot file, not by calling `start/0` explicitly.

`start_service(SvcName, Options) -> ok | {error, Reason}`

Types:

`SvcName = service_name()`

`Options = [service_opt()]`

`Reason = term()`

Start a diameter service.

A service defines a locally-implemented Diameter node, specifying the capabilities to be advertised during capabilities exchange. Transports are added to a service using *add_transport/2*.

> **Note:**
>
> A transport can both override its service's capabilities and restrict its supported Diameter applications so "service = Diameter node as identified by Origin-Host" is not necessarily the case.

`stop() -> ok | {error, Reason}`

Stop the diameter application.

`stop_service(SvcName) -> ok | {error, Reason}`

Types:

`SvcName = service_name()`

`Reason = term()`

Stop a diameter service.

Stopping a service causes all associated transport connections to be broken. A DPR message with be sent as in the case of *remove_transport/2*.

> **Note:**
>
> Stopping a service does not remove any associated transports: *remove_transport/2* must be called to remove transport configuration.

`subscribe(SvcName) -> true`

Types:

`SvcName = service_name()`

Subscribe to `service_event()` messages from a service.

It is not an error to subscribe to events from a service that does not yet exist. Doing so before adding transports is required to guarantee the reception of all transport-related events.

`unsubscribe(SvcName) -> true`

Types:

`SvcName = service_name()`

Unsubscribe to event messages from a service.

SEE ALSO

diameter_app(3), diameter_transport(3), diameter_dict(4)

diameterc

Command

The diameterc utility is used to compile a diameter *dictionary file* into Erlang source. The resulting source implements the interface diameter required to encode and decode the dictionary's messages and AVPs.

The module *diameter_make(3)* provides an alternate compilation interface.

USAGE

diameterc [<options>] <file>

Compile a single dictionary file to Erlang source. Valid options are as follows.

-i <dir>

Prepend the specified directory to the code path. Use to point at beam files compiled from inherited dictionaries, `@inherits` in a dictionary file creating a beam dependency, not an erl/hrl dependency.

Multiple `-i` options can be specified.

-o <dir>

Write generated source to the specified directory. Defaults to the current working directory.

-E
-H

Suppress erl and hrl generation, respectively.

--name <name>
--prefix <prefix>

Transform the input dictionary before compilation, setting `@name` or `@prefix` to the specified string.

--inherits <arg>

Transform the input dictionary before compilation, appending `@inherits` of the specified string.

Two forms of `--inherits` have special meaning:

```
--inherits -
--inherits Prev/Mod
```

The first has the effect of clearing any previous inherits, the second of replacing a previous inherits of `Prev` to one of `Mod`. This allows the semantics of the input dictionary to be changed without modifying the file itself.

Multiple `--inherits` options can be specified.

EXIT STATUS

Returns 0 on success, non-zero on failure.

SEE ALSO

diameter_make(3), diameter_dict(4)

diameter_app

Erlang module

A diameter service as started by *diameter:start_service/2* configures one of more Diameter applications, each of whose configuration specifies a callback that handles messages specific to the application. The messages and AVPs of the application are defined in a dictionary file whose format is documented in *diameter_dict(4)* while the callback module is documented here. The callback module implements the Diameter application-specific functionality of a service.

A callback module must export all of the functions documented below. The functions themselves are of three distinct flavours:

- *peer_up/3* and *peer_down/3* signal the attainment or loss of connectivity with a Diameter peer.
- *pick_peer/4*, *prepare_request/3*, *prepare_retransmit/3*, *handle_answer/4* and *handle_error/4* are (or may be) called as a consequence of a call to *diameter:call/4* to send an outgoing Diameter request message.
- *handle_request/3* is called in response to an incoming Diameter request message.

The arities for the the callback functions here assume no extra arguments. All functions will also be passed any extra arguments configured with the callback module itself when calling *diameter:start_service/2* and, for the call-specific callbacks, any extra arguments passed to *diameter:call/4*.

DATA TYPES

`capabilities() = #diameter_caps{}`

A record containing the identities of the local Diameter node and the remote Diameter peer having an established transport connection, as well as the capabilities as determined by capabilities exchange. Each field of the record is a 2-tuple consisting of values for the (local) host and (remote) peer. Optional or possibly multiple values are encoded as lists of values, mandatory values as the bare value.

`message() = diameter_codec:message()`

The representation of a Diameter message as passed to *diameter:call/4* or returned from a *handle_request/3* callback.

`packet() = diameter_codec:packet()`

A container for incoming and outgoing Diameter messages that's passed through encode/decode and transport. Fields should not be set in return values except as documented.

`peer_ref() = term()`

A term identifying a transport connection with a Diameter peer.

`peer() = {peer_ref(), capabilities()}`

A tuple representing a Diameter peer connection.

`state() = term()`

The state maintained by the application callback functions *peer_up/3*, *peer_down/3* and (optionally) *pick_peer/4*. The initial state is configured in the call to *diameter:start_service/2* that configures the application on a service. Callback functions returning a state are evaluated in a common service-specific process while those not returning state are evaluated in a request-specific process.

Exports

`Mod:peer_up(SvcName, Peer, State) -> NewState`

Types:

`SvcName = diameter:service_name()`
`Peer = peer()`
`State = NewState = state()`

Invoked to signal the availability of a peer connection on the local Erlang node. In particular, capabilities exchange with the peer has indicated support for the application in question, the RFC 3539 watchdog state machine for the connection has reached state `OKAY` and Diameter messages can be both sent and received.

> **Note:**
>
> A watchdog state machine can reach state `OKAY` from state `SUSPECT` without a new capabilities exchange taking place. A new transport connection (and capabilities exchange) results in a new peer_ref().

> **Note:**
>
> There is no requirement that a callback return before incoming requests are received: *handle_request/3* callbacks must be handled independently of *peer_up/3* and *peer_down/3*.

`Mod:peer_down(SvcName, Peer, State) -> NewState`

Types:

`SvcName = diameter:service_name()`
`Peer = peer()`
`State = NewState = state()`

Invoked to signal that a peer connection on the local Erlang node is no longer available following a previous call to *peer_up/3*. In particular, that the RFC 3539 watchdog state machine for the connection has left state `OKAY` and the peer will no longer be a candidate in *pick_peer/4* callbacks.

`Mod:pick_peer(LocalCandidates, RemoteCandidates, SvcName, State) -> Selection | false`

Types:

`LocalCandidates = RemoteCandidates = [peer()]`
`SvcName = diameter:service_name()`
`State = NewState = state()`
`Selection = {ok, Peer} | {Peer, NewState}`
`Peer = peer() | false`

Invoked as a consequence of a call to *diameter:call/4* to select a destination peer for an outgoing request. The return value indicates the selected peer.

The candidate lists contain only those peers that have advertised support for the Diameter application in question during capabilities exchange, that have not be excluded by a `filter` option in the call to *diameter:call/4* and whose watchdog state machine is in the `OKAY` state. The order of the elements is unspecified except that any peers whose Origin-Host and Origin-Realm matches that of the outgoing request (in the sense of a `{filter, {all, [host, realm]}}` option to *diameter:call/4*) will be placed at the head of the list. `LocalCandidates` contains peers

whose transport process resides on the local Erlang node while `RemoteCandidates` contains peers that have been communicated from other nodes by services of the same name.

A callback that returns a peer() will be followed by a *prepare_request/3* callback and, if the latter indicates that the request should be sent, by either *handle_answer/4* or *handle_error/4* depending on whether or not an answer message is received from the peer. If the transport becomes unavailable after *prepare_request/3* then a new *pick_peer/4* callback may take place to failover to an alternate peer, after which *prepare_retransmit/3* takes the place of *prepare_request/3* in resending the request. There is no guarantee that a *pick_peer/4* callback to select an alternate peer will be followed by any additional callbacks since a retransmission to an alternate peer is abandoned if an answer is received from a previously selected peer.

The return values `false` and `{false, State}` (that is, `NewState = State`) are equivalent, as are `{ok, Peer}` and `{Peer, State}`.

Note:

The *diameter:service_opt()* `use_shared_peers` determines whether or not a service uses peers shared from other nodes. If not then `RemoteCandidates` is the empty list.

Warning:

The return value `{Peer, NewState}` is only allowed if the Diameter application in question was configured with the *diameter:application_opt()* `{call_mutates_state, true}`. Otherwise, the `State` argument is always the initial value as configured on the application, not any subsequent value returned by a *peer_up/3* or *peer_down/3* callback.

`Mod:prepare_request(Packet, SvcName, Peer) -> Action`

Types:

```
Packet = packet()
SvcName = diameter:service_name()
Peer = peer()
Action = Send | Discard | {eval_packet, Action, PostF}
Send = {send, packet() | message()}
Discard = {discard, Reason} | discard
PostF = diameter:eval()}
```

Invoked to return a request for encoding and transport. Allows the sender to use the selected peer's capabilities to modify the outgoing request. Many implementations may simply want to return `{send, Packet}`

A returned *packet()* should set the request to be encoded in its `msg` field and can set the `transport_data` field in order to pass information to the transport process. Extra arguments passed to *diameter:call/4* can be used to communicate transport (or any other) data to the callback.

A returned *packet()* can set the `header` field to a `#diameter_header{}` to specify values that should be preserved in the outgoing request, values otherwise being those in the header record contained in `Packet`. A returned `length`, `cmd_code` or `application_id` is ignored.

A returned `PostF` will be evaluated on any encoded `#diameter_packet{}` prior to transmission, the `bin` field containing the encoded binary. The return value is ignored.

Returning `{discard, Reason}` causes the request to be aborted and the *diameter:call/4* for which the callback has taken place to return `{error, Reason}`. Returning `discard` is equivalent to returning `{discard, discarded}`.

`Mod:prepare_retransmit(Packet, SvcName, Peer) -> Action`

Types:

```
Packet = packet()
SvcName = diameter:service_name()
Peer = peer()
Action = Send | Discard | {eval_packet, Action, PostF}
Send = {send, packet() | message()}
Discard = {discard, Reason} | discard
PostF = diameter:eval()}
```

Invoked to return a request for encoding and retransmission. Has the same role as *prepare_request/3* in the case that a peer connection is lost an an alternate peer selected but the argument *packet()* is as returned by the initial *prepare_request/3*.

Returning `{discard, Reason}` causes the request to be aborted and a *handle_error/4* callback to take place with `Reason` as initial argument. Returning `discard` is equivalent to returning `{discard, discarded}`.

`Mod:handle_answer(Packet, Request, SvcName, Peer) -> Result`

Types:

```
Packet = packet()
Request = message()
SvcName = diameter:service_name()
Peer = peer()
Result = term()
```

Invoked when an answer message is received from a peer. The return value is returned from *diameter:call/4* unless the `detach` option was specified.

The decoded answer record and undecoded binary are in the `msg` and `bin` fields of the argument *packet()* respectively. `Request` is the outgoing request message as was returned from *prepare_request/3* or *prepare_retransmit/3*.

For any given call to *diameter:call/4* there is at most one *handle_answer/4* callback: any duplicate answer (due to retransmission or otherwise) is discarded. Similarly, only one of *handle_answer/4* or *handle_error/4* is called.

By default, an incoming answer message that cannot be successfully decoded causes the request process to fail, causing *diameter:call/4* to return `{error, failure}` unless the `detach` option was specified. In particular, there is no *handle_error/4* callback in this case. The *diameter:application_opt()* `answer_errors` can be set to change this behaviour.

`Mod:handle_error(Reason, Request, SvcName, Peer) -> Result`

Types:

```
Reason = timeout | failover | term()
Request = message()
SvcName = diameter:service_name()
Peer = peer()
Result = term()
```

Invoked when an error occurs before an answer message is received in response to an outgoing request. The return value is returned from *diameter:call/4* unless the `detach` option was specified.

Reason `timeout` indicates that an answer message has not been received within the time specified with the corresponding *diameter:call_opt()*. Reason `failover` indicates that the transport connection to the peer to which the request has been sent has become unavailable and that not alternate peer was not selected.

`Mod:handle_request(Packet, SvcName, Peer) -> Action`

Types:

`Packet = packet()`

`SvcName = term()`

`Peer = peer()`

`Action = Reply | {relay, [Opt]} | discard | {eval|eval_packet, Action, PostF}`

`Reply = {reply, packet() | message()} | {answer_message, 3000..3999|5000..5999} | {protocol_error, 3000..3999}`

`Opt = diameter:call_opt()`

`PostF = diameter:eval()`

Invoked when a request message is received from a peer. The application in which the callback takes place (that is, the callback module as configured with *diameter:start_service/2*) is determined by the Application Identifier in the header of the incoming request message, the selected module being the one whose corresponding dictionary declares itself as defining either the application in question or the Relay application.

The argument *packet()* has the following signature.

```
#diameter_packet{header = #diameter_header{},
                 avps   = [#diameter_avp{}],
                 msg    = record() | undefined,
                 errors = [Unsigned32() | {Unsigned32(), #diameter_avp{}}],
                 bin    = binary(),
                 transport_data = term()}
```

The `msg` field will be `undefined` in case the request has been received in the relay application. Otherwise it contains the record representing the request as outlined in *diameter_dict(4)*.

The `errors` field specifies any results codes identifying errors found while decoding the request. This is used to set Result-Code and/or Failed-AVP in a returned answer unless the callback returns a `#diameter_packet{}` whose `errors` field is set to either a non-empty list of its own, in which case this list is used instead, or the atom `false` to disable any setting of Result-Code and Failed-AVP. Note that the errors detected by diameter are of the 3xxx and 5xxx series, Protocol Errors and Permanent Failures respectively. The `errors` list is empty if the request has been received in the relay application.

The `transport_data` field contains an arbitrary term passed into diameter from the transport module in question, or the atom `undefined` if the transport specified no data. The term is preserved if a *message()* is returned but must be set explicitly in a returned *packet()*.

The semantics of each of the possible return values are as follows.

`{reply, packet() | message()}`

Send the specified answer message to the peer. In the case of a *packet()*, the message to be sent must be set in the `msg` field and the `header` field can be set to a `#diameter_header{}` to specify values that should be preserved in the outgoing answer, appropriate values otherwise being set by diameter.

`{answer_message, 3000..3999|5000..5999}`

Send an answer message to the peer containing the specified Result-Code. Equivalent to

```
{reply, ['answer-message' | Avps]
```

where `Avps` sets the Origin-Host, Origin-Realm, the specified Result-Code and (if the request contained one) Session-Id AVPs, and possibly Failed-AVP as described below.

Returning a value other than 3xxx or 5xxx will cause the request process in question to fail, as will returning a 5xxx value if the peer connection in question has been configured with the RFC 3588 common dictionary `diameter_gen_base_rfc3588`. (Since RFC 3588 only allows 3xxx values in an answer-message.)

When returning 5xxx, Failed-AVP will be populated with the AVP of the first matching Result-Code/AVP pair in the `errors` field of the argument *packet()*, if found. If this is not appropriate then an answer-message should be constructed explicitly and returned in a `reply` tuple instead.

`{relay, Opts}`

Relay a request to another peer in the role of a Diameter relay agent. If a routing loop is detected then the request is answered with 3005 (DIAMETER_LOOP_DETECTED). Otherwise a Route-Record AVP (containing the sending peer's Origin-Host) is added to the request and *pick_peer/4* and subsequent callbacks take place just as if *diameter:call/4* had been called explicitly. The End-to-End Identifier of the incoming request is preserved in the header of the relayed request.

The returned `Opts` should not specify `detach`. A subsequent *handle_answer/4* callback for the relayed request must return its first argument, the *packet()* containing the answer message. Note that the `extra` option can be specified to supply arguments that can distinguish the relay case from others if so desired. Any other return value (for example, from a *handle_error/4* callback) causes the request to be answered with 3002 (DIAMETER_UNABLE_TO_DELIVER).

`discard`

Discard the request. No answer message is sent to the peer.

`{eval, Action, PostF}`

Handle the request as if `Action` has been returned and then evaluate `PostF` in the request process. The return value is ignored.

`{eval_packet, Action, PostF}`

Like `eval` but evaluate `PostF` on any encoded `#diameter_packet{}` prior to transmission, the `bin` field containing the encoded binary. The return value is ignored.

`{protocol_error, 3000..3999}`

Equivalent to `{answer_message, 3000..3999}`.

Note:

Requests containing errors may be answered by diameter, without a callback taking place, depending on the value of the *diameter:application_opt()* `request_errors`.

diameter_codec

Erlang module

Incoming Diameter messages are decoded from binary() before being communicated to *diameter_app(3)* callbacks. Similarly, outgoing Diameter messages are encoded into binary() before being passed to the appropriate *diameter_transport(3)* module for transmission. The functions documented here implement the default encode/decode.

Warning:

The diameter user does not need to call functions here explicitly when sending and receiving messages using *diameter:call/4* and the callback interface documented in *diameter_app(3)*: diameter itself provides encode/decode as a consequence of configuration passed to *diameter:start_service/2*, and the results may differ from those returned by the functions documented here, depending on configuration.

The *header()* and *packet()* records below are defined in diameter.hrl, which can be included as follows.

```
-include_lib("diameter/include/diameter.hrl").
```

Application-specific records are defined in the hrl files resulting from dictionary file compilation.

DATA TYPES

```
uint8()  = 0..255
uint24() = 0..16777215
uint32() = 0..4294967295
```

8-bit, 24-bit and 32-bit integers occurring in Diameter and AVP headers.

`avp() = #diameter_avp{}`

The application-neutral representation of an AVP. Primarily intended for use by relay applications that need to handle arbitrary Diameter applications. A service implementing a specific Diameter application (for which it configures a dictionary) can manipulate values of type *message()* instead.

Fields have the following types.

```
code = uint32()
is_mandatory = boolean()
need_encryption = boolean()
vendor_id = uint32() | undefined
```

Values in the AVP header, corresponding to AVP Code, the M flag, P flags and Vendor-ID respectively. A Vendor-ID other than `undefined` implies a set V flag.

`data = iolist()`

The data bytes of the AVP.

`name = atom()`

The name of the AVP as defined in the dictionary file in question, or `undefined` if the AVP is unknown to the dictionary file in question.

`value = term()`

The decoded value of an AVP. Will be `undefined` on decode if the data bytes could not be decoded, the AVP is unknown, or if the *decode format* is `none`. The type of a decoded value is as document in *diameter_dict(4)*.

`type = atom()`

The type of the AVP as specified in the dictionary file in question (or one it inherits). Possible types are `undefined` and the Diameter types: `OctetString`, `Integer32`, `Integer64`, `Unsigned32`, `Unsigned64`, `Float32`, `Float64`, `Grouped`, `Enumerated`, `Address`, `Time`, `UTF8String`, `DiameterIdentity`, `DiameterURI`, `IPFilterRule` and `QoSFilterRule`.

`dictionary() = module()`

The name of a generated dictionary module as generated by *diameterc(1)* or *diameter_make:codec/2*. The interface provided by a dictionary module is an implementation detail that may change.

`header() = #diameter_header{}`

The record representation of the Diameter header. Values in a *packet()* returned by *decode/2* are as extracted from the incoming message. Values set in an *packet()* passed to *encode/2* are preserved in the encoded binary(), with the exception of `length`, `cmd_code` and `application_id`, all of which are determined by the *dictionary()* in question.

Note:

It is not necessary to set header fields explicitly in outgoing messages as diameter itself will set appropriate values. Setting inappropriate values can be useful for test purposes.

Fields have the following types.

```
version = uint8()
length = uint24()
cmd_code = uint24()
application_id = uint32()
hop_by_hop_id = uint32()
end_to_end_id = uint32()
```

Values of the Version, Message Length, Command-Code, Application-ID, Hop-by-Hop Identifier and End-to-End Identifier fields of the Diameter header.

```
is_request = boolean()
is_proxiable = boolean()
is_error = boolean()
is_retransmitted = boolean()
```

Values corresponding to the R(equest), P(roxiable), E(rror) and T(Potentially re-transmitted message) flags of the Diameter header.

`message() = record() | maybe_improper_list()`

The representation of a Diameter message as passed to *diameter:call/4* or returned from a *handle_request/3* callback. The record representation is as outlined in *diameter_dict(4)*: a message as defined in a dictionary file is encoded as a record with one field for each component AVP. Equivalently, a message can also be encoded as a list whose head is the atom-valued message name (as specified in the relevant dictionary file) and whose tail is either a list of AVP name/values pairs or a map with values keyed on AVP names. The format at decode is determined by *diameter:service_opt() decode_format*. Any of the formats is accepted at encode.

Another list-valued representation allows a message to be specified as a list whose head is a *header()* and whose tail is an *avp()* list. This representation is used by diameter itself when relaying requests as directed by the return value of a *handle_request/3* callback. It differs from the other other two in that it bypasses the checks for messages that do not agree with their definitions in the dictionary in question: messages are sent exactly as specified.

`packet() = #diameter_packet{}`

A container for incoming and outgoing Diameter messages. Fields have the following types.

`header = header() | undefined`

The Diameter header of the message. Can be (and typically should be) `undefined` for an outgoing message in a non-relay application, in which case diameter provides appropriate values.

`avps = [avp()] | undefined`

The AVPs of the message. Ignored for an outgoing message if the `msg` field is set to a value other than `undefined`.

`msg = message() | undefined`

The incoming/outgoing message. For an incoming message, a term corresponding to the configured *decode format* if the message can be decoded in a non-relay application, `undefined` otherwise. For an outgoing message, setting a `[header() | avp()]` list is equivalent to setting the `header` and `avps` fields to the corresponding values.

> **Warning:**
>
> A value in the `msg` field does **not** imply an absence of decode errors. The `errors` field should also be examined.

`bin = binary()`

The incoming message prior to encode or the outgoing message after encode.

`errors = [5000..5999 | {5000..5999, avp()}]`

Errors detected at decode of an incoming message, as identified by a corresponding 5xxx series Result-Code (Permanent Failures). For an incoming request, these should be used to formulate an appropriate answer as documented for the *handle_request/3* callback in *diameter_app(3)*. For an incoming answer, the *diameter:application_opt()* `answer_errors` determines the behaviour.

`transport_data = term()`

An arbitrary term of meaning only to the transport process in question, as documented in *diameter_transport(3)*.

Exports

`decode(Mod, Bin) -> Pkt`

Types:

`Mod = dictionary()`
`Bin = binary()`
`Pkt = packet()`

Decode a Diameter message.

encode(Mod, Msg) -> Pkt

Types:

Mod = *dictionary()*

Msg = *message()* | *packet()*

Pkt = *packet()*

Encode a Diameter message.

SEE ALSO

diameterc(1), diameter_app(3), diameter_dict(4), diameter_make(3)

diameter_dict

Name

A diameter service, as configured with *diameter:start_service/2*, specifies one or more supported Diameter applications. Each Diameter application specifies a dictionary module that knows how to encode and decode its messages and AVPs. The dictionary module is in turn generated from a file that defines these messages and AVPs. The format of such a file is defined in *FILE FORMAT* below. Users add support for their specific applications by creating dictionary files, compiling them to Erlang modules using either *diameterc(1)* or *diameter_make(3)* and configuring the resulting dictionaries modules on a service.

Dictionary module generation also results in a hrl file that defines records for the messages and Grouped AVPs defined by the dictionary, these records being what a user of the diameter application sends and receives, modulo other possible formats as discussed in *diameter_app(3)*. These records and the underlying Erlang data types corresponding to Diameter data formats are discussed in *MESSAGE RECORDS* and *DATA TYPES* respectively. The generated hrl also contains macro definitions for the possible values of AVPs of type Enumerated.

The diameter application includes five dictionary modules corresponding to applications defined in section 2.4 of RFC 6733: `diameter_gen_base_rfc3588` and `diameter_gen_base_rfc6733` for the Diameter Common Messages application with application identifier 0, `diameter_gen_accounting` (for RFC 3588) and `diameter_gen_acct_rfc6733` for the Diameter Base Accounting application with application identifier 3 and `diameter_gen_relay` the Relay application with application identifier 0xFFFFFFFF.

The Common Message and Relay applications are the only applications that diameter itself has any specific knowledge of. The Common Message application is used for messages that diameter itself handles: CER/CEA, DWR/DWA and DPR/DPA. The Relay application is given special treatment with regard to encode/decode since the messages and AVPs it handles are not specifically defined.

FILE FORMAT

A dictionary file consists of distinct sections. Each section starts with a tag followed by zero or more arguments and ends at the the start of the next section or end of file. Tags consist of an ampersand character followed by a keyword and are separated from their arguments by whitespace. Whitespace separates individual tokens but is otherwise insignificant.

The tags, their arguments and the contents of each corresponding section are as follows. Each section can occur multiple times unless otherwise specified. The order in which sections are specified is unimportant.

`@id Number`

Defines the integer Number as the Diameter Application Id of the application in question. Can occur at most once and is required if the dictionary defines `@messages`. The section has empty content.

The Application Id is set in the Diameter Header of outgoing messages of the application, and the value in the header of an incoming message is used to identify the relevant dictionary module.

Example:

```
@id 16777231
```

`@name Mod`

Defines the name of the generated dictionary module. Can occur at most once and defaults to the name of the dictionary file minus any extension. The section has empty content.

Note that a dictionary module should have a unique name so as not collide with existing modules in the system.

Example:

```
@name etsi_e2
```

`@prefix Name`

Defines Name as the prefix to be added to record and constant names (followed by a `'_'` character) in the generated dictionary module and hrl. Can occur at most once. The section has empty content.

A prefix is optional but can be be used to disambiguate between record and constant names resulting from similarly named messages and AVPs in different Diameter applications.

Example:

```
@prefix etsi_e2
```

`@vendor Number Name`

Defines the integer Number as the the default Vendor-Id of AVPs for which the V flag is set. Name documents the owner of the application but is otherwise unused. Can occur at most once and is required if an AVP sets the V flag and is not otherwise assigned a Vendor-Id. The section has empty content.

Example:

```
@vendor 13019 ETSI
```

`@avp_vendor_id Number`

Defines the integer Number as the Vendor-Id of the AVPs listed in the section content, overriding the `@vendor` default. The section content consists of AVP names.

Example:

```
@avp_vendor_id 2937

WWW-Auth
Domain-Index
Region-Set
```

`@inherits Mod`

Defines the name of a dictionary module containing AVP definitions that should be imported into the current dictionary. The section content consists of the names of those AVPs whose definitions should be imported from the dictionary, an empty list causing all to be imported. Any listed AVPs must not be defined in the current dictionary and it is an error to inherit the same AVP from more than one dictionary.

Note that an inherited AVP that sets the V flag takes its Vendor-Id from either `@avp_vendor_id` in the inheriting dictionary or `@vendor` in the inherited dictionary. In particular, `@avp_vendor_id` in the inherited dictionary is ignored. Inheriting from a dictionary that specifies the required `@vendor` is equivalent to using `@avp_vendor_id` with a copy of the dictionary's definitions but the former makes for easier reuse.

All dictionaries should typically inherit RFC 6733 AVPs from `diameter_gen_base_rfc6733`.

Example:

```
@inherits diameter_gen_base_rfc6733
```

`@avp_types`

Defines the name, code, type and flags of individual AVPs. The section consists of definitions of the form

`Name Code Type Flags`

where Code is the integer AVP code, Type identifies an AVP Data Format as defined in section *DATA TYPES* below, and Flags is a string of V, M and P characters indicating the flags to be set on an outgoing AVP or a single `'-'` (minus) character if none are to be set.

Example:

```
@avp_types

Location-Information   350  Grouped     MV
Requested-Information  353  Enumerated   V
```

Warning:

The P flag has been deprecated by RFC 6733.

`@custom_types Mod`

Specifies AVPs for which module Mod provides encode/decode functions. The section contents consists of AVP names. For each such name, `Mod:Name(encode|decode, Type, Data, Opts)` is expected to provide encode/decode for values of the AVP, where Name is the name of the AVP, Type is it's type as declared in the `@avp_types` section of the dictionary, Data is the value to encode/decode, and Opts is a term that is passed through encode/decode.

Example:

```
@custom_types rfc4005_avps

Framed-IP-Address
```

`@codecs Mod`

Like `@custom_types` but requires the specified module to export `Mod:Type(encode|decode, Name, Data, Opts)` rather than `Mod:Name(encode|decode, Type, Data, Opts)`.

Example:

```
@codecs rfc4005_avps

Framed-IP-Address
```

`@messages`

Defines the messages of the application. The section content consists of definitions of the form specified in section 3.2 of RFC 6733, "Command Code Format Specification".

```
@messages

RTR ::= < Diameter Header: 287, REQ, PXY >
        < Session-Id >
        { Auth-Application-Id }
        { Auth-Session-State }
        { Origin-Host }
        { Origin-Realm }
        { Destination-Host }
        { SIP-Deregistration-Reason }
        [ Destination-Realm ]
        [ User-Name ]
      * [ SIP-AOR ]
      * [ Proxy-Info ]
      * [ Route-Record ]
      * [ AVP ]

RTA ::= < Diameter Header: 287, PXY >
        < Session-Id >
        { Auth-Application-Id }
        { Result-Code }
        { Auth-Session-State }
        { Origin-Host }
        { Origin-Realm }
        [ Authorization-Lifetime ]
        [ Auth-Grace-Period ]
        [ Redirect-Host ]
        [ Redirect-Host-Usage ]
        [ Redirect-Max-Cache-Time ]
      * [ Proxy-Info ]
      * [ Route-Record ]
      * [ AVP ]
```

`@grouped`

Defines the contents of the AVPs of the application having type Grouped. The section content consists of definitions of the form specified in section 4.4 of RFC 6733, "Grouped AVP Values".

Example:

```
@grouped

SIP-Deregistration-Reason ::= < AVP Header: 383 >
                              { SIP-Reason-Code }
                              [ SIP-Reason-Info ]
                            * [ AVP ]
```

Specifying a Vendor-Id in the definition of a grouped AVP is equivalent to specifying it with `@avp_vendor_id`.

`@enum Name`

Defines values of AVP Name having type Enumerated. Section content consists of names and corresponding integer values. Integer values can be prefixed with 0x to be interpreted as hexadecimal.

Note that the AVP in question can be defined in an inherited dictionary in order to introduce additional values to an enumeration otherwise defined in another dictionary.

Example:

```
@enum SIP-Reason-Code

PERMANENT_TERMINATION    0
NEW_SIP_SERVER_ASSIGNED  1
SIP_SERVER_CHANGE        2
REMOVE_SIP_SERVER        3
```

`@end`

Causes parsing of the dictionary to terminate: any remaining content is ignored.

Comments can be included in a dictionary file using semicolon: characters from a semicolon to end of line are ignored.

MESSAGE RECORDS

The hrl generated from a dictionary specification defines records for the messages and grouped AVPs defined in `@messages` and `@grouped` sections. For each message or grouped AVP definition, a record is defined whose name is the message or AVP name, prefixed with any dictionary prefix defined with `@prefix`, and whose fields are the names of the AVPs contained in the message or grouped AVP in the order specified in the definition in question. For example, the grouped AVP

```
SIP-Deregistration-Reason ::= < AVP Header: 383 >
                              { SIP-Reason-Code }
                              [ SIP-Reason-Info ]
                             * [ AVP ]
```

will result in the following record definition given an empty prefix.

```
-record('SIP-Deregistration-Reason' {'SIP-Reason-Code',
                                     'SIP-Reason-Info',
                                     'AVP'}).
```

The values encoded in the fields of generated records depends on the type and number of times the AVP can occur. In particular, an AVP which is specified as occurring exactly once is encoded as a value of the AVP's type while an AVP with any other specification is encoded as a list of values of the AVP's type. The AVP's type is as specified in the AVP definition, the RFC 6733 types being described below.

DATA TYPES

The data formats defined in sections 4.2 ("Basic AVP Data Formats") and 4.3 ("Derived AVP Data Formats") of RFC 6733 are encoded as values of the types defined here. Values are passed to *diameter:call/4* in a request record when sending a request, returned in a resulting answer record and passed to a *handle_request/3* callback upon reception of an incoming request.

In cases in which there is a choice between string() and binary() types for OctetString() and derived types, the representation is determined by the value of *diameter:service_opt() string_decode*.

Basic AVP Data Formats

```
OctetString() = string() | binary()
Integer32()   = -2147483647..2147483647
Integer64()   = -9223372036854775807..9223372036854775807
Unsigned32()  = 0..4294967295
Unsigned64()  = 0..18446744073709551615
Float32()     = '-infinity' | float() | infinity
Float64()     = '-infinity' | float() | infinity
Grouped()     = record()
```

On encode, an OctetString() can be specified as an iolist(), excessively large floats (in absolute value) are equivalent to `infinity` or `'-infinity'` and excessively large integers result in encode failure. The records for grouped AVPs are as discussed in the previous section.

Derived AVP Data Formats

```
Address() = OctetString()
          | tuple()
```

On encode, an OctetString() IPv4 address is parsed in the usual x.x.x.x format while an IPv6 address is parsed in any of the formats specified by section 2.2 of RFC 2373, "Text Representation of Addresses". An IPv4 tuple() has length 4 and contains values of type 0..255. An IPv6 tuple() has length 8 and contains values of type 0..65535. The tuple representation is used on decode.

```
Time() = {date(), time()}

where

  date() = {Year, Month, Day}
  time() = {Hour, Minute, Second}

  Year   = integer()
  Month  = 1..12
  Day    = 1..31
  Hour   = 0..23
  Minute = 0..59
  Second = 0..59
```

Additionally, values that can be encoded are limited by way of their encoding as four octets as required by RFC 6733 with the required extension from RFC 2030. In particular, only values between `{{1968,1,20},{3,14,8}}` and `{{2104,2,26},{9,42,23}}` (both inclusive) can be encoded.

```
UTF8String() = [integer()] | binary()
```

List elements are the UTF-8 encodings of the individual characters in the string. Invalid codepoints will result in encode/decode failure. On encode, a UTF8String() can be specified as a binary, or as a nested list of binaries and codepoints.

```
DiameterIdentity() = OctetString()
```

A value must have length at least 1.

```
DiameterURI() = OctetString()
              | #diameter_URI{type = Type,
                              fqdn = FQDN,
                              port = Port,
                              transport = Transport,
                              protocol  = Protocol}

where

  Type = aaa | aaas
  FQDN = OctetString()
  Port = integer()
  Transport = sctp | tcp
  Protocol  = diameter | radius | 'tacacs+'
```

On encode, fields port, transport and protocol default to 3868, sctp and diameter respectively. The grammar of an OctetString-valued DiameterURI() is as specified in section 4.3 of RFC 6733. The record representation is used on decode.

```
Enumerated() = Integer32()
```

On encode, values can be specified using the macros defined in a dictionary's hrl file.

```
IPFilterRule()  = OctetString()
QoSFilterRule() = OctetString()
```

Values of these types are not currently parsed by diameter.

SEE ALSO

diameterc(1), diameter(3), diameter_app(3), diameter_codec(3), diameter_make(3)

diameter_make

Erlang module

The function *codec/2* is used to compile a diameter *dictionary file* into Erlang source. The resulting source implements the interface diameter requires to encode and decode the dictionary's messages and AVPs.

The utility *diameterc(1)* provides an alternate compilation interface.

Exports

codec(File :: iolist() | binary(), [Opt]) -> ok | {ok, [Out]} | {error, Reason}

Compile a single dictionary file. The input `File` can be either a path or a literal dictionary, the occurrence of newline (ascii NL) or carriage return (ascii CR) identifying the latter. `Opt` determines the format of the results and whether they are written to file or returned, and can have the following types.

`parse | forms | erl | hrl`

Specifies an output format. Whether the output is returned or written to file depends on whether or not option `return` is specified. When written to file, the resulting file(s) will have extensions `.D`, `.F`, `.erl`, and `.hrl` respectively, basenames defaulting to `dictionary` if the input dictionary is literal and does not specify *`@name`*. When returned, results are in the order of the corresponding format options. Format options default to `erl` and `hrl` (in this order) if unspecified.

The `parse` format is an internal representation that can be passed to *flatten/1* and *format/1*, while the `forms` format can be passed to *compile:forms/2*. The `erl` and `hrl` formats are returned as iolists.

`{include, string()}`

Prepend the specified directory to the code path. Use to point at beam files compiled from inherited dictionaries, *`@inherits`* in a dictionary file creating a beam dependency, not an erl/hrl dependency.

Multiple `include` options can be specified.

`{outdir, string()}`

Write generated source to the specified directory. Defaults to the current working directory. Has no effect if option `return` is specified.

`return`

Return results in a `{ok, [Out]}` tuple instead of writing to file and returning `ok`.

`{name|prefix, string()}`

Transform the input dictionary before compilation, setting *`@name`* or *`@prefix`* to the specified string.

`{inherits, string()}`

Transform the input dictionary before compilation, appending *`@inherits`* of the specified string.

Two forms have special meaning:

```
{inherits, "-"}
{inherits, "Prev/Mod"}
```

The first has the effect of clearing any previous inherits, the second of replacing a previous inherits of `Prev` to one of `Mod`. This allows the semantics of the input dictionary to be changed without modifying the file itself.

Multiple `inherits` options can be specified.

Note that a dictionary's *`@name`*, together with the `outdir` option, determine the output paths when the `return` option is not specified. The *`@name`* of a literal input dictionary defaults to `dictionary`.

A returned error reason can be converted into a readable string using *format_error/1*.

`format(Parsed) -> iolist()`

Turns a parsed dictionary, as returned by *codec/2*, back into the dictionary format.

`flatten(Parsed) -> term()`

Reconstitute a parsed dictionary, as returned by *codec/2*, without using *`@inherits`*. That is, construct an equivalent dictionary in which all AVP's are definined in the dictionary itself. The return value is also a parsed dictionary.

`format_error(Reason) -> string()`

Turn an error reason returned by *codec/2* into a readable string.

BUGS

Unrecognized options are silently ignored.

SEE ALSO

diameterc(1), *diameter_dict(4)*

diameter_transport

Erlang module

A module specified as a `transport_module` to *diameter:add_transport/2* must implement the interface documented here. The interface consists of a function with which diameter starts a transport process and a message interface with which the transport process communicates with the process that starts it (aka its parent).

DATA TYPES

`message() = binary() | diameter_codec:packet()`

A Diameter message as passed over the transport interface.

For an inbound message from a transport process, a *diameter_codec:packet()* must contain the received message in its `bin` field. In the case of an inbound request, any value set in the `transport_data` field will passed back to the transport module in the corresponding answer message, unless the sender supplies another value.

For an outbound message to a transport process, a *diameter_codec:packet()* has a value other than `undefined` in its `transport_data` field and has the binary() to send in its `bin` field.

Exports

`Mod:start({Type, Ref}, Svc, Config) -> {ok, Pid} | {ok, Pid, LAddrs} | {error, Reason}`

Types:

`Type = connect | accept`
`Ref = diameter:transport_ref()`
`Svc = #diameter_service{}`
`Config = term()`
`Pid = pid()`
`LAddrs = [inet:ip_address()]`
`Reason = term()`

Start a transport process. Called by diameter as a consequence of a call to *diameter:add_transport/2* in order to establish or accept a transport connection respectively. A transport process maintains a connection with a single remote peer.

`Type` indicates whether the transport process in question is being started for a connecting (`Type=connect`) or listening (`Type=accept`) transport. In the latter case, transport processes are started as required to accept connections from multiple peers.

Ref is the value that was returned from the call to *diameter:add_transport/2* that has lead to starting of a transport process.

`Svc` contains capabilities passed to *diameter:start_service/2* and *diameter:add_transport/2*, values passed to the latter overriding those passed to the former.

`Config` is as passed in `transport_config` tuple in the *diameter:transport_opt()* list passed to *diameter:add_transport/2*.

The start function should use the `Host-IP-Address` list in `Svc` and/or `Config` to select and return an appropriate list of local IP addresses. In the connecting case, the local address list can instead be communicated in a `connected` message (see *MESSAGES* below) following connection establishment. In either case, the local address list is used

to populate `Host-IP-Address` AVPs in outgoing capabilities exchange messages if `Host-IP-Address` is unspecified.

A transport process must implement the message interface documented below. It should retain the pid of its parent, monitor the parent and terminate if it dies. It should not link to the parent. It should exit if its transport connection with its peer is lost.

MESSAGES

All messages sent over the transport interface are of the form `{diameter, term()}`.

A transport process can expect messages of the following types from its parent.

`{diameter, {send, message()}}`

An outbound Diameter message.

`{diameter, {close, Pid}}`

A request to terminate the transport process after having received DPA in response to DPR. The transport process should exit. `Pid` is the pid() of the parent process.

`{diameter, {tls, Ref, Type, Bool}}`

Indication of whether or not capabilities exchange has selected inband security using TLS. `Ref` is a reference() that must be included in the `{diameter, {tls, Ref}}` reply message to the transport's parent process (see below). `Type` is either `connect` or `accept` depending on whether the process has been started for a connecting or listening transport respectively. `Bool` is a boolean() indicating whether or not the transport connection should be upgraded to TLS.

If TLS is requested (`Bool=true`) then a connecting process should initiate a TLS handshake with the peer and an accepting process should prepare to accept a handshake. A successful handshake should be followed by a `{diameter, {tls, Ref}}` message to the parent process. A failed handshake should cause the process to exit.

This message is only sent to a transport process over whose `Inband-Security-Id` configuration has indicated support for TLS.

A transport process should send messages of the following types to its parent.

`{diameter, {self(), connected}}`

Inform the parent that the transport process with `Type=accept` has established a connection with the peer. Not sent if the transport process has `Type=connect`.

`{diameter, {self(), connected, Remote}}`
`{diameter, {self(), connected, Remote, [LocalAddr]}}`

Inform the parent that the transport process with `Type=connect` has established a connection with a peer. Not sent if the transport process has `Type=accept`. `Remote` is an arbitrary term that uniquely identifies the remote endpoint to which the transport has connected. A `LocalAddr` list has the same semantics as one returned from *start/3*.

`{diameter, {recv, message()}}`

An inbound Diameter message.

`{diameter, {tls, Ref}}`

Acknowledgment of a successful TLS handshake. `Ref` is the reference() received in the `{diameter, {tls, Ref, Type, Bool}}` message in response to which the reply is sent. A transport must exit if a handshake is not successful.

SEE ALSO

diameter_tcp(3), *diameter_sctp(3)*

diameter_tcp

Erlang module

This module implements diameter transport over TCP using *gen_tcp(3)*. It can be specified as the value of a `transport_module` option to *diameter:add_transport/2* and implements the behaviour documented in *diameter_transport(3)*. TLS security is supported, either as an upgrade following capabilities exchange or at connection establishment.

Note that the ssl application is required for TLS and must be started before configuring TLS capability on diameter transports.

Exports

`start({Type, Ref}, Svc, [Opt]) -> {ok, Pid} | {ok, Pid, [LAddr]} | {error, Reason}`

Types:

`Type = connect | accept`

`Ref = diameter:transport_ref()`

`Svc = #diameter_service{}`

`Opt = OwnOpt | SslOpt | TcpOpt`

`Pid = pid()`

`LAddr = inet:ip_address()`

`Reason = term()`

`OwnOpt = {raddr, inet:ip_address()} | {rport, integer()} | {accept, Match} | {port, integer()} | {fragment_timer, infinity | 0..16#FFFFFFFF} | {message_cb, diameter:eval()} | {sender, boolean()}`

`SslOpt = {ssl_options, true | list()}`

`TcpOpt = term()`

`Match = inet:ip_address() | string() | [Match]`

The start function required by *diameter_transport(3)*.

Options `raddr` and `rport` specify the remote address and port for a connecting transport and are not valid for a listening transport.

Option `accept` specifies remote addresses for a listening transport and is not valid for a connecting transport. If specified, a remote address that does not match one of the specified addresses causes the connection to be aborted. Multiple `accept` options can be specified. A string-valued `Match` that does not parse as an address is interpreted as a regular expression.

Option `ssl_options` must be specified for a transport that should support TLS: a value of `true` results in a TLS handshake immediately upon connection establishment while `list()` specifies options to be passed to *ssl:connect/2* or *ssl:ssl_accept/2* after capabilities exchange if TLS is negotiated.

Option `fragment_timer` specifies the timeout, in milliseconds, of a timer used to flush messages from the incoming byte stream even if the number of bytes indicated in the Message Length field of its Diameter Header have not yet been accumulated: such a message is received over the transport interface after two successive timeouts without the reception of additional bytes. Defaults to 1000.

Option `sender` specifies whether or not to use a dedicated process for sending outgoing messages, which avoids the possibility of send blocking reception. Defaults to `false`. If set to `true` then a `message_cb` that avoids the possibility of messages being queued in the sender process without bound should be configured.

Option `message_cb` specifies a callback that is invoked on incoming and outgoing messages, that can be used to implement flow control. It is applied to two arguments: an atom indicating the reason for the callback (`send`, `recv`, or `ack` after a completed send), and the message in question (binary() on `recv`, binary() or diameter_packet record on `send` or `ack`, or `false` on `ack` when an incoming request has been discarded). It should return a list of actions and a new callback as tail; eg. `[fun cb/3, State]`. Valid actions are the atoms `send` or `recv`, to cause a following message-valued action to be sent/received, a message to send/receive (binary() or diameter_packet record), or a boolean() to enable/disable reading on the socket. More than one `send`/`recv`/message sequence can be returned from the same callback, and an initial `send`/`recv` can be omitted if the same as the value passed as the callback's first argument. Reading is initially enabled, and returning `false` does not imply there cannot be subsequent `recv` callbacks since messages may already have been read. An empty tail is equivalent to the prevailing callback. Defaults to a callback equivalent to `fun(ack, _) -> []; (_, Msg) -> [Msg] end`.

Remaining options are any accepted by *ssl:connect/3* or *gen_tcp:connect/3* for a connecting transport, or *ssl:listen/2* or *gen_tcp:listen/2* for a listening transport, depending on whether or not `{ssl_options, true}` has been specified. Options `binary`, `packet` and `active` cannot be specified. Also, option `port` can be specified for a listening transport to specify the local listening port, the default being the standardized 3868. Note that the option `ip` specifies the local address.

An `ssl_options` list must be specified if and only if the transport in question has set `Inband-Security-Id` to 1 (`TLS`), as specified to either *diameter:start_service/2* or *diameter:add_transport/2*, so that the transport process will receive notification of whether or not to commence with a TLS handshake following capabilities exchange. Failing to specify an options list on a TLS-capable transport for which TLS is negotiated will cause TLS handshake to fail. Failing to specify TLS capability when `ssl_options` has been specified will cause the transport process to wait for a notification that will not be forthcoming, which will eventually cause the RFC 3539 watchdog to take down the connection.

The first element of a non-empty `Host-IP-Address` list in `Svc` provides the local IP address if an `ip` option is not specified. The local address is either returned from*start/3* or passed in a `connected` message over the transport interface.

SEE ALSO

diameter(3), *diameter_transport(3)*, *gen_tcp(3)*, *inet(3)*, *ssl(3)*

diameter_sctp

Erlang module

This module implements diameter transport over SCTP using *gen_sctp(3)*. It can be specified as the value of a transport_module option to *diameter:add_transport/2* and implements the behaviour documented in *diameter_transport(3)*.

Exports

start({Type, Ref}, Svc, [Opt]) -> {ok, Pid, [LAddr]} | {error, Reason}

Types:

Type = connect | accept
Ref = *diameter:transport_ref()*
Svc = #diameter_service{}
Opt = OwnOpt | SctpOpt
Pid = pid()
LAddr = *inet:ip_address()*
Reason = term()
OwnOpt = {raddr, *inet:ip_address()*} | {rport, integer()} | {accept, Match} | {unordered, boolean() | pos_integer()} | {packet, boolean() | raw} | {message_cb, *diameter:eval()*} | {sender, boolean()}
SctpOpt = term()
Match = *inet:ip_address()* | string() | [Match]

The start function required by *diameter_transport(3)*.

Options `raddr` and `rport` specify the remote address and port for a connecting transport and not valid for a listening transport: the former is required while latter defaults to 3868 if unspecified. Multiple `raddr` options can be specified, in which case the connecting transport in question attempts each in sequence until an association is established.

Option `accept` specifies remote addresses for a listening transport and is not valid for a connecting transport. If specified, a remote address that does not match one of the specified addresses causes the association to be aborted. Multiple `accept` options can be specified. A string-valued `Match` that does not parse as an address is interpreted as a regular expression.

Option `unordered` specifies whether or not to use unordered delivery, integer `N` being equivalent to `N =< OS`, where `OS` is the number of outbound streams negotiated on the association in question. Regardless of configuration, sending is ordered on stream 0 until reception of a second incoming message, to ensure that a peer receives capabilities exchange messages before any other. Defaults to `false`.

Option `packet` determines how/if an incoming message is packaged into a diameter_packet record. If `false` then messages are received as binary(). If `true` then as a record with the binary() message in the `bin` field and a `{stream, Id}` tuple in the `transport_data` field, where `Id` is the identifier of the inbound stream the message was received on. If `raw` then as a record with the received ancillary sctp_sndrcvinfo record in the `transport_data` field. Defaults to `true`.

Options `message_cb` and `sender` have semantics identical to those documented in *diameter_tcp(3)*, but with the message argument to a `recv` callback being as directed by the `packet` option.

An `{outstream, Id}` tuple in the `transport_data` field of a outgoing diameter_packet record sets the outbound stream on which the message is sent, modulo the negotiated number of outbound streams. Any other value causes successive such sends to cycle though all outbound streams.

Remaining options are any accepted by *gen_sctp:open/1*, with the exception of options `mode`, `binary`, `list`, `active` and `sctp_events`. Note that options `ip` and `port` specify the local address and port respectively.

Multiple `ip` options can be specified for a multihomed peer. If none are specified then the values of `Host-IP-Address` in the `diameter_service` record are used. Option `port` defaults to 3868 for a listening transport and 0 for a connecting transport.

Warning:

An small receive buffer may result in a peer having to resend incoming messages: set the *inet(3)* option `recbuf` to increase the buffer size.

An small send buffer may result in outgoing messages being discarded: set the *inet(3)* option `sndbuf` to increase the buffer size.

SEE ALSO

diameter(3), *diameter_transport(3)*, *gen_sctp(3)*, *inet(3)*