
# Diameter


Diameter 1.1
September 20 2012

**September 20 2012**

# 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 3588. It supports arbitrary Diameter applications by way of a *dictionary* interface that allows messages and AVP's 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 peer 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 peer 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 peer (as identified by an Origin-Host AVP), transports can themselves be associated with capabilities AVP's so that a single service be used to implement more than one Diameter peer.

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 peer(s) it implements.

Each Diameter application is also configured with one or more dictionary modules that provide encode/decode functionality for outgoing/incoming Diameter messages. A module is generated from a *specification file* using the *diameterc* utility. Dictionaries for the RFC 3588 Diameter Common Messages, Base Accounting and Relay applications are provided by the diameter application.

## 1.2 Using the stack

To be written.

## 1.3 Examples

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

## 1.4 Standards Compliance

Known points of questionable or non-compliance.

### 1.4.1 RFC 3588

- The End-to-End Security framework (section 2.9) isn't implemented since it is largely unspecified. The document that was to describe it (reference [AAACMS]) was abandoned in an uncompleted state several years ago and the current draft RFC deprecates the framework, including the P Flag in the AVP header.
- There is no TLS support over SCTP. RFC 3588 requires that a Diameter server support TLS but in practise this seems to mean TLS over SCTP since there are limitations with running over SCTP: see RFC 6083 (DTLS over SCTP), which is a response to RFC 3436 (TLS over SCTP). The current RFC 3588 draft acknowledges this by equating TLS with TLS/TCP and DTLS/SCTP but we do not yet support DTLS.
- There is no explicit support for peer discovery (section 5.2). It can possibly be implemented on top of diameter as is but this is probably something that diameter should do. The current draft deprecates portions of the original RFC's mechanisms however.
- The peer state machine's election process (section 5.6.4) isn't implemented as specified since it assumes knowledge of a peer's Origin-Host before sending it a CER. (The identity becoming known upon reception of CEA.) The possibility of configuring the peer's Origin-Host could be added, along with handling of the case that it sends something else, but for many applications this will just be unnecessary configuration of a value that it has no control over.

### 1.4.2 RFC 3539

RFC 3539 is more difficult to comply to since it discusses problems as much as it requires functionality but all the MUST's are covered, the watchdog state machine being the primary one. Of the optional functionality, load balancing is left to the diameter user (since it's the one deciding who to send to) and there is no Congestion Manager.

# 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 creates a service that sends and receives messages using the Diameter protocol as defined in RFC 3588.

Basic usage consists of creating a representation of a locally implemented Diameter peer 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 application* and *Diameter application*. The former refers to the Erlang application named diameter whose main api is defined here, the latter to an application of the Diameter protocol in the sense of RFC 3588. More generally, capitalized Diameter refers to the RFC and diameter to this implementation.

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

## DATA TYPES

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

Types corresponding to RFC 3588 AVP Data Formats. Defined in *diameter_dict(4)*.

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

A name identifying a Diameter application in service configuration passed to *start_service/2* and passed to *call/4* when sending requests belonging to the application.

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

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

A module implementing the callback interface defined in *diameter_app(3)*, along with any extra arguments to be appended to those documented for the interface. Any extra arguments are appended to the documented list of arguments for each function. Note that additional arguments specific to an outgoing request be specified to *call/4*, in which case the call-specific arguments are appended to any specified with the callback module.

Specifying a `#diameter_callback{}` record allows individual functions to be configured in place of the usual *diameter_app(3)* callbacks, with default implementations provided by module `diameter_callback` unless otherwise specified. See that module for details.

```
application_opt()
```

Options defining a Diameter application as configured in an `application` option passed to *start_service/2*.

`{alias, application_alias()}`

An unique identifier for the application in the scope of the service. Optional, defaults to the value of the `dictionary` option.

`{dictionary, atom()}`

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

`{module, application_module()}`

A callback module with which messages of the Diameter application are handled. See *diameter_app(3)* for information on the required interface and semantics.

`{state, term()}`

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

`{call_mutates_state, true|false}`

Specifies whether or not the *pick_peer/4* application callback (following from a call to *call/4*) can modifiy state, Defaults to `false` if unspecified.

Note that *pick_peer* callbacks are serialized when these are allowed to modify state, 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 but sending its own requests should probably avoid it.

`{answer_errors, callback|report|discard}`

Determines the 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*, in the `errors` field of the `diameter_packet` record passed into 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 `report` if unspecified.

`call_opt()`

Options available to *call/4* when sending an outgoing Diameter request.

`{extra, list()}`

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

`{filter, peer_filter()}`

A filter to apply to the list of available peers before passing them 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`.

`{timeout, Unsigned32()}`

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

`detach`

Causes *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 used in outgoing CER/CEA messages during capabilities exchange. Can be configured both on a service and a transport, the latter taking precedence over the former.

`{'Origin-Host', DiameterIdentity()}`

Value of the Origin-Host AVP in outgoing messages.

`{'Origin-Realm', DiameterIdentity()}`

Value of the Origin-Realm AVP in outgoing messages.

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

Values of Host-IP-Address AVPs. Optional.

The list of addresses is available to the start function of a transport module, which either uses them as is or returns a new list (typically as configured as `transport_config()` on the transport module in question) in order for the correct list of addresses to be sent in capabilities exchange messages.

`{'Vendor-Id', Unsigned32()}`

Value of the Vendor-Id AVP sent in an outgoing capabilities exchange message.

`{'Product-Name', UTF8String()}`

Value of the Product-Name AVP sent in an outgoing capabilities exchange message.

`{'Origin-State-Id', Unsigned32()}`

Value of Origin-State-Id to be included in outgoing messages sent by diameter itself. In particular, the AVP will be included in CER/CEA and DWR/DWA messages. Optional.

Setting a value of `0` (zero) is equivalent to not setting a value as documented in RFC 3588. The function *origin_state_id/0* can be used as to retrieve a value that is set when the diameter application is started.

`{'Supported-Vendor-Id', [Unsigned32()]}`

Values of Supported-Vendor-Id AVPs sent in an outgoing capabilities exchange message. Optional, defaults to the empty list.

`{'Auth-Application-Id', [Unsigned32()]}`

Values of Auth-Application-Id AVPs sent in an outgoing capabilities exchange message. Optional, defaults to the empty list.

`{'Inband-Security-Id', [Unsigned32()]}`

Values of Inband-Security-Id AVPs sent in an outgoing capabilities exchange message. Optional, 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()]}`

Values of Acct-Application-Id AVPs sent in an outgoing capabilities exchange message. Optional, defaults to the empty list.

`{'Vendor-Specific-Application-Id', [Grouped()]}`

Values of Vendor-Specific-Application-Id AVPs sent in an outgoing capabilities exchange message. Optional, defaults to the empty list.

`{'Firmware-Revision', Unsigned32()}`

Value of the Firmware-Revision AVP sent in an outgoing capabilities exchange message. Optional.

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

`evaluable() = {M,F,A} | fun() | [evaluable() | 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 evaluable() `E` to an argument list `A` is meant in the sense of `eval([E|A])`.

Beware of using local funs (that is, fun expressions not of the form `fun Module: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.

`peer_filter() = term()`

A 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 at all.

`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 value, or all peers if the atom `any`.

`{eval, evaluable()}`

Matches only those peers for which the specified evaluable() returns `true` on 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 of the specified list.

`{any, [peer_filter()]}`

Matches only those peers matched by at least one filter of the specified list.

Note that the `host` and `realm` filters examine the outgoing request as passed to *call/4*, assuming that this is a record- or list-valued message() as documented in *diameter_app(3)*, and that the message contains 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. Note also that an empty host/realm (which should not be typical) is equivalent to an unspecified one for the purposes of filtering.

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

```
service_event() = #diameter_event{}
```

Event message sent to processes that have subscribed using *subscribe/1*.

The `info` field of the event record can be one of the following.

```
{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 open state. If a `diameter_packet` record is present in an `up` tuple then there has been an exchange of capabilities exchange messages and the record contains the received CER or CEA, otherwise the connection has reestablished without the loss or transport connectivity.

Note that a single up/down event for a given peer corresponds to as many *peer_up/peer_down* callbacks as there are Diameter applications shared by the peer, as determined during capablilities 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 `reconnect_timer` or `watchdog_timer` expiry.

```
{closed, Ref, Reason, Config}
```

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

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

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

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

An incoming CER has been answered with the indicated result code or discarded. The capabilities record contains pairs of values for the the local node and remote peer. The packet record contains the CER in question. In the case of rejection by a capabilities callback, the tuple indicates 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. The capabilities record contains only values for the the local node. The packet record contains the CER in question.

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

```
Result = integer() | 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. The capabilities record contains pairs of values for the the local node and remote peer. The packet record contains the CEA in question. In the case of rejection by a capabilities callback, the tuple indicates the rejecting callback.

`{'CEA', Caps, Pkt}`

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

An incoming CER contained errors and has been rejected. The capabilities record contains only values for the the local node. The packet record contains the CEA in question.

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

`service_name() = term()`

The 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()`

Options accepted by *start_service/2*. Can be any `capability()` tuple as well as the following.

`{application, [application_opt()]}`

Defines a Diameter application supported by the service.

A service must define one application for each Diameter application it intends to support. For an outgoing Diameter request, the application is specified by passing the desired application's `application_alias()` to *call/4*, while for an incoming request the application identifier in the message header determines the application (and callback module), the application identifier being specified in the *dictionary* file defining the application.

`transport_opt()`

Options accepted by *add_transport/2*.

`{transport_module, atom()}`

A module implementing a transport process as defined in *diameter_transport(3)*. Defaults to `diameter_tcp` if unspecified.

The interface required of a transport module is documented in *diameter_transport(3)*.

`{transport_config, term()}`

A 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 if unspecified.

`{applications, [application_alias()]}`

The list of Diameter applications to which usage of the transport should be restricted. Defaults to all applications configured on the service in question.

`{capabilities, [capability()]}`

AVP's used to construct outgoing CER/CEA messages. Any AVP specified takes precedence over a corresponding value specified for 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)* but not over SCTP as implemented by *diameter_sctp(3)*.

`{capabilities_cb, evaluable()}`

A 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 reference (as returned by *add_transport/2*) and `diameter_caps` record of the connection. Returning `ok` accepts the connection. Returning `integer()` causes an incoming CER to be answered with the specified Result-Code. Returning `discard` causes an incoming CER to be discarded. Returning `unknown` is 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.

`{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 if unspecified.

`{reconnect_timer, Tc}`

```
Tc = Unsigned32()
```

For a connecting transport, the RFC 3588 Tc timer, in milliseconds. Note that this timer determines the frequency with which the transport will attempt to establish a connection with its peer only *before* an initial connection is established: once there is an initial connection it's watchdog_timer that 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 a reestablishment, causing the RFC 3539 state machine to pass to state OPEN rather than REOPEN. Note that these semantics are not goverened by the RFC and that a listening transport's reconnect_timer should be greater than its peers's Tc plus jitter.

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

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*.

## Exports

**`add_transport(SvcName, {connect|listen, Options}) -> {ok, Ref} | {error, Reason}`**

Types:

```
SvcName = service_name()
Options = [transport_opt()]
Ref = ref()
Reason = term()
```

Add transport capability to a service.

The service will start a transport process(es) 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. 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 transports.

**`call(SvcName, App, Request, Options) -> ok | Answer | {error, Reason}`**

Types:

```
SvcName = service_name()
App = application_alias()
Request = diameter_app:message() | term()
Answer = term()
Options = [call_opt()]
```

Send a Diameter request message and possibly return the answer or error.

`App` identifies 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 has been specified to cause an earlier return, the call returns either when an answer message is received from the peer or an error occurs. In the case of an answer, the return value is as returned by a *handle_answer/4* callback. In the case of an error, 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 from 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 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 `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.

### `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`

Types:

```
SvcName = service_name()
Pred = Fun | MFA | ref() | list() | true | false
Fun = fun((reference(), connect|listen, list()) -> boolean())
    | fun((reference(), list()) -> boolean())
    | fun((list()) -> boolean())
MFA = {atom(), atom(), list()}
```

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 the corresponding call. The remaining forms are equivalent to an arity-3 fun as follows.

```
Pred = fun(reference(), list()):  fun(Ref, _, Opts) -> Pred(Ref, Opts) end
Pred = fun(list()):               fun(_, _, Opts) -> Pred(Opts) end
Pred = reference():               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 all associated transport connections to be broken. A base application DPR message with Disconnect-Cause `DO_NOT_WANT_TO_TALK_TO_YOU` will be sent to each connected peer before disassociating the transport configuration from the service and terminating the transport upon reception of DPA or timeout.

**service_info(SvcName, Item) -> Value**

Types:

**SvcName = service_name()**

**Value = term()**

Return information about a started service.

**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 3588. 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 will typically be 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 peer, specifying the capabilities of the peer to be used during capabilities exchange and the Diameter applications that it supports. Transports are added to a service using *add_transport/2*.

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

Stop the diameter application.

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

Types:

**SvcName = service_name()**

**Reason = term()**

Stop a diameter service.

**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.

**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 transform diameter *dictionary files* into Erlang source. The resulting source implements the interface diameter requires to encode and decode the dictionary's messages and AVP's.

## USAGE

diameterc [<options>] <file>

Transforms a single dictionary file. Valid options are as follows.

-o <dir>

Specifies the directory into which the generated source should be written. Defaults to the current working directory.

-i <dir>

Specifies a directory to add to the code path. Use to point at beam files corresponding to dictionaries inherited by the one being compiled using `@inherits` or `--inherits`. Inheritance is a beam dependency, not an erl/hrl dependency.

Multiple `-i` options can be specified.

-E

Supresses erl generation.

-H

Supresses hrl generation.

--name <name>

Set `@name` in the dictionary file. Overrides any setting in the file itself.

--prefix <prefix>

Set `@prefix` in the dictionary file. Overrides any setting in the file itself.

--inherits <dict>

Append an `@inherits` to the dictionary file. Specifying `'-'` as the dictionary has the effect of clearing any previous inherits, effectively ignoring previous inherits.

Multiple `--inherits` options can be specified.

## EXIT STATUS

Returns 0 on success, non-zero on failure.

## BUGS

The identification of errors in the source file is poor.

## SEE ALSO

*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 its application. The messages and AVPs of the Diameter 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.

> **Note:**
>
> The arities given 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 and remote Diameter peers 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() = record() | list()`

The representation of a Diameter message as passed to *diameter:call/4*. 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 (the record name minus any prefix specified in the relevant dictionary file) and whose tail is a list of `{FieldName, FieldValue}` pairs.

A third representation allows a message to be specified as a list whose head is a `diameter_header` record and whose tail is a list of `diameter_avp` records. 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 (since relays agents must handle arbitrary request): messages are sent exactly as specified.

`packet() = #diameter_packet{}`

A container for incoming and outgoing Diameters message that's passed through encode/decode and transport. Fields of a packet() record should not be set in return values except as documented.

`peer_ref() = term()`

A term identifying a transport connection with a Diameter peer. Should be treated opaquely.

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

A tuple representing a Diameter peer connection.

`service_name() = term()`

The service supporting the Diameter application. Specified to *diameter:start_service/2* when starting the service.

`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 = service_name()`**
**`Peer = peer()`**
**`State = NewState = state()`**

Invoked when a transport connection has been established and a successful capabilities exchange has indicated that the peer supports the Diameter application of the application on which the callback module in question has been configured.

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

Types:

**`SvcName = service_name()`**
**`Peer = peer()`**
**`State = NewState = state()`**

Invoked when a transport connection has been lost following a previous call to *peer_up/3*.

**`Mod:pick_peer(Candidates, Reserved, SvcName, State) -> {ok, Peer} | {Peer, NewState} | false`**

Types:

**`Candidates = [peer()]`**
**`Peer = peer() | false`**
**`SvcName = service_name()`**
**`State = NewState = state()`**

Invoked as a consequence of a call to *diameter:call/4* to select a destination peer for an outgoing request, the return value indicating the selected peer. A new application state can also be returned but only if the Diameter application in question was configured with the option `call_mutates_state` set to `true`, as documented for *diameter:start_service/2*.

The candidate peers list will only include those which are selected by any `filter` option specified in the call to *diameter:call/4*, and only those which have indicated support for the Diameter application in question. 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.

The return values `false` and `{false, State}` are equivalent when callback state is mutable, as are `{ok, Peer}` and `{Peer, State}`. Returning a peer as `false` causes `{error, no_connection}` to be returned from *diameter:call/4*. Returning a peer() from an initial pick_peer/4 callback will result in a *prepare_request/3* callback followed by either *handle_answer/4* or *handle_error/4* depending on whether or not an answer message is received from the peer. If transport with the peer is lost before this then a new *pick_peer/4* callback takes place to select an alternate peer.

Note that there is no guarantee that a *pick_peer/4* callback to select an alternate peer will be followed by any additional callbacks, only that the initial *pick_peer/4* will be, since a retransmission to an alternate peer is abandoned if an answer is received from a previously selected peer.

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

Types:

```
Packet = packet()
SvcName = service_name()
Peer = peer()
Action = {send, packet() | message()} | {discard, Reason} | discard
```

Invoked to return a request for encoding and transport. Allows the sender to access the selected peer's capabilities in order to set (for example) `Destination-Host` and/or `Destination-Realm` in the outgoing request, although the callback need not be limited to this usage. 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 module. Extra arguments passed to *diameter:call/4* can be used to communicate transport data to the callback. A returned packet() can also set the `header` field to a `diameter_header` record in order to specify values that should be preserved in the outgoing request, although this should typically not be necessary and allows the callback to set header values inappropriately. A returned `length`, `cmd_code` or `application_id` 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) -> Result`**

Types:

```
Packet = packet()
SvcName = service_name()
Peer = peer()
Result = {send, packet() | message()} | {discard, Reason} | discard
```

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 = service_name()
```

```
Peer = peer()
Result = term()
```

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

The decoded answer record is in the `msg` field of the argument packet(), the undecoded binary in the `packet` field. `Request` is the outgoing request message as was returned from *prepare_request/3* or *prepare_retransmit/3* before the request was passed to the transport.

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

By default, an incoming answer message that cannot be successfully decoded causes the request process in question to fail, causing the relevant call to *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. Application configuration may change this behaviour as described for *diameter:start_service/2*.

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

Types:

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

Invoked when an error occurs before an answer message is received from a peer in response to an outgoing request. The return value is returned from the call to *diameter:call/4* for which the callback takes place (unless the `detach` option was specified).

Reason `timeout` indicates that an answer message has not been received within the required time. Reason `failover` indicates that the transport connection to the peer to which the request has been sent has been lost but that not alternate node was available, possibly because a *pick_peer/4* callback returned false.

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

Types:

```
Packet = packet()
SvcName = term()
Peer = peer()
Action = Reply | {relay, Opts} | discard | {eval, Action, PostF}
Reply = {reply, message()} | {protocol_error, 3000..3999}
Opts = diameter:call_opts()
PostF = diameter:evaluable()
```

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` only 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 Result-Code's identifying errors that were encountered in decoding the request. In this case diameter will set both Result-Code and Failed-AVP AVP's in a returned answer message() before sending it to the peer: the returned message() need only set any other required AVP's. Note that the errors detected by diameter are all of the 5xxx series (Permanent Failures). 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 in the packet() containing any answer message sent back to the transport process unless another value is explicitly specified.

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

`{reply, message()}`

Send the specified answer message to the peer.

`{protocol_error, 3000..3999}`

Send an answer message to the peer containing the specified protocol error. Equivalent to

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

where `Avps` sets the Origin-Host, Origin-Realm, the specified Result-Code and (if the request sent one) Session-Id AVP's.

Note that RFC 3588 mandates that only answers with a 3xxx series Result-Code (protocol errors) may set the E bit. Returning a non-3xxx value in a `protocol_error` tuple will cause the request process in question to fail.

`{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 `diameter_packet` record 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.

`{eval, Action, PostF}`

Handle the request as if `Action` has been returned and then evaluate `PostF` in the request process.

Note that protocol errors detected by diameter will result in an answer message without `handle_request/3` being invoked.

# 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 *diameterc* and configuring the resulting dictionaries modules on a service.

The codec generation also results in a hrl file that defines records for the messages and grouped AVPs defined for the application, these records being what a user of the diameter application sends and receives. (Modulo other available 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 defines for the possible values of AVPs of type Enumerated.

The diameter application includes three dictionary modules corresponding to applications defined in section 2.4 of RFC 3588: `diameter_gen_base_rfc3588` for the Diameter Common Messages application with application identifier 0, `diameter_gen_accounting` 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 if unspecified. 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 RFC3588 AVPs from `diameter_gen_base_rfc3588`.

Example:

```
```

```
@inherits diameter_gen_base_rfc3588
```

`@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 *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
```

Note that the P flag has been deprecated by the Diameter Maintenance and Extensions Working Group of the IETF: diameter will set the P flag to 0 as mandated by the current draft standard.

`@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)` 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 and Data is the value to 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)` rather than `Mod:Name(encode|decode, Type, Data)`.

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 3588, "Command Code ABNF 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 3588, "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 hexidecimal.

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 3588 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 3588 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* callback upon reception of an incoming request.

*Basic AVP Data Formats*

```
OctetString() = [0..255]
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 3588 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()]
```

List elements are the UTF-8 encodings of the individual characters in the string. Invalid codepoints will result in encode/decode failure.

```
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 3588. 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_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).

## Exports

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

Types:

```
Type = connect | accept
Ref = reference()
Svc = #diameter_service{}
Opts = term()
Pid = pid()
LAddrs = [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.

The first argument indicates whether the transport process in question is being started for a connecting (`connect`) or listening (`accept`) transport. In the latter case, transport processes are started as required to accept connections from multiple peers. Ref is in each case the same value that was returned from the call to *diameter:add_transport/2* that has lead to starting of a transport process.

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.

Transport processes for a given service are started sequentially.

The start function should use the 'Host-IP-Address' list on the service, namely `Svc#diameter_service.host_ip_address`, and/or `Opts` to select an appropriate list of local IP addresses, and should return this list if different from the service addresses. The returned list is used to populate 'Host-IP-Address' AVPs in outgoing capabilities exchange messages, the service addresses being used otherwise.

## MESSAGES

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

A transport process can expect the following messages from diameter.

`{diameter, {send, Packet}}`

> An outbound Diameter message. Packet can be either `binary()` (the message to be sent) or a `#diameter_packet{}` whose `transport_data` field containes a value other than undefined.

`{diameter, {close, Pid}}`

> A request to close the transport connection. The transport process should terminate after closing the connection. 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 the following messages 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}}`

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.

`{diameter, {recv, Packet}}`

An inbound Diameter message. Packet can be either `binary()` (the message to be sent) or `#diameter_packet{}` whose `packet` field contains a `binary()`. Any value (other than undefined) set in the `transport_data` field will be passed back with a corresponding answer message in the case that the inbound message is a request unless the sender sets another value. How the `transport_data` is used/interpreted is up to the transport module.

`{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. 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, both as an upgrade following capabilities exchange as specified by RFC 3588 and at connection establishment as in the current draft standard.

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, [LAddr]} | {error, Reason}**

Types:

```
Type = connect | accept
Ref = reference()
Svc = #diameter_service{}
Opt = OwnOpt | SslOpt | OtherOpt
Pid = pid()
LAddr = ip_address()
Reason = term()
OwnOpt = {raddr, ip_address()} | {rport, integer()} | {port, integer()}
SslOpt = {ssl_options, true | list()}
OtherOpt = term()
```

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

The only diameter_tcp-specific argument is the options list. Options `raddr` and `rport` specify the remote address and port for a connecting transport and are not valid for a listening transport. Option `ssl_options` must be specified for a transport that must be able to 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 of ssl:ssl_accept/2 after capabilities exchange if TLS is negotiated. Remaining options are any accepted by ssl:connect/3 or gen_tcp:connect/3 for a connecting transport, or ssl:listen/3 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 if unspecified. Note that option `ip` specifies the local address.

An `ssl_options` list must be specified if and only if the transport in question has specified an Inband-Security-Id AVP with value TLS on the relevant call to *start_service/2* or *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.

If the service specifies more than one Host-IP-Address and option `ip` is unspecified then then the first of the service's addresses is used as the local address.

The returned local address list has length one.

## SEE ALSO

*diameter(3), diameter_transport(3)*

# diameter_sctp

Erlang module

This module implements diameter transport over SCTP using gen_sctp. 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 = reference()
Svc = #diameter_service{}
Opt = {raddr, ip_address()} | {rport, integer()} | term()
Pid = pid()
LAddr = ip_address()
Reason = term()
```

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

The only diameter_sctp-specific argument is the options list. 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. More than one `raddr` option can be specified, in which case the connecting transport in question attempts each in sequence until an association is established. 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 on the service are used. (In particular, one of these must be specified.) Option `port` defaults to 3868 for a listening transport and 0 for a connecting transport.

diameter_sctp uses the `transport_data` field of the `diameter_packet` record to communicate the stream on which an inbound message has been received, or on which an outbound message should be sent: the value will be of the form `{stream, Id}` on an inbound message passed to a *handle_request* or *handle_answer* callback. For an outbound message, either `undefined` (explicitly of by specifying the outbound message as a `binary()`) or a tuple should be set in the return value of *handle_request* (typically by retaining the value passed into this function) or *prepare_request*. The value `undefined` uses a "next outbound stream" id and then increments this modulo the total number outbound streams. That is, successive values of `undefined` cycle through all outbound streams.

## SEE ALSO

*diameter_transport(3)*