EPOCH Protocol client API C

Endpoint Oriented Channel application programming interface for C language

The header file epoch.h declares all the symbols defined in the next document.

struct epoch_s

epoch_s structure represents a connection state to the server. It has other fields that helps to build that state.

char ipv4[EPOCH_IPV4]

This field is used to set the IP version 4 of the server. If empty, then the IP version 6 is used to that end.

char ipv6[EPOCH_IPV6]

This field is used, if ipv4 field is empty, to set the servers IP version 6 address.

unsigned short port

Indicates the port in which the server is listening.

char key[EPOCH_KEYMAX]

Key field is a random generated session key by the user and will be used to encrypt/decrypt the transmitted data to and from the server.

int keylen

Specifies the session key length.

char *pubkey

This is the public key of the server. It is used to encrypt the random generated session and transmit it.

int rdtmout

In absence of a keepalive or heartbeat mechanism, this field provides a timeout for reading operations. Once the timeout expire, the read function will return an error.

int cntmout

This timeout represents the amount of time that the client is willing to wait for establish the tcp connection before drop the attempt.

struct netconn *nc

This is a handle to an obscure structure that contains fundamental fields for the connection. It must not be manipulated directly.

enum epoch_mode

epoch_mode enumeration has the modes in which the EPOCH manage the flowing data.

SINGLE_THREAD_SNDFIRST

This mode indicate to the EPOCH API that the data will be sent first to the server, and once is done, the client will receive the data sent by the server.

SINGLE_THREAD_RCVFIRST

This mode indicate to the EPOCH API that the data will be received first from the server, and once is done, the client will send its data to the server.

MULTI_THREAD

This mode indicate to the EPOCH API that the data will be sent from the client to the server at the same time the data will be received by the client from the server.

DETACHED

This mode indicate to the EPOCH API that the endpoint will run in a detached way -daemon mode-, so no read or write operations will be performed.

typedef void (*callback_snd**)(void ***sndbuf**, int64_t ***sndlen**)**

Callback function prototype for sending data.

void *sndbuf

This parameter points to an internal buffer for the data to be sent. The client should write this buffer before the callback returns. Once the callback returns, the EPOCH API will send it.

int64_t *sndlen

This parameter points to an internal 64bit signed integer that will be used to know how much data is in the *sndbuf parameter. Before the callback returns, the client must set it to indicate how much data is in the buffer. The length must be less than or equal to the buffer size specified in the epoch_endpoint function call. Once the client set *sndlen to 0 the EPOCH API will consider the upload operation done.

typedef void (*callback_rcv**)(void ***rcvbuf**, int64_t** rcvlen**)**

Callback function prototype for receiving data.

void *rcvbuf

This parameter points to an internal buffer for the data to be received. The client must copy the received data to a second buffer before the callback returns and should continue doing this until there is no more data to receive.

int64_t rcvlen

This parameter is set by the EPOCH API and indicate how much data is stored in the buffer. Once this parameter is set to 0 by the EPOCH API, there is no more data to receive.

int epoch_endpoint**(struct epoch_s ***e**, const char ***endpt**, const char ***auth**, enum epoch_mode** mode**, int64_t** bufsz**, callback_snd** snd_callback**, callback_rcv** rcv_callback, **const char ***wd, **int** *es**)**

The epoch_endpoint function connect to the server and run the endpoint indicated by the *endpt parameter. If everything goes well, it will keep calling snd_callback until there is no more data to send, and rcv_callback until there is no more data to receive.

struct epoch_s *e

A pointer to initialized structured with the parameters to establish the connection to the server.

const char *endpt

The endpoint ‘the program’ to execute at the server side with its parameters. An example could be like this: “/bin/ls -la” or just “/bin/ls”.

const char *auth

The authentication token is used by the server to determine which endpoints ‘programs’ and parameters can run.

int64_t bufsz

Sets the buffer size for communication. It has to be greater than or equal to 65536.

enum epoch_mode mode

Indicates the mode in which the server and the client will send and receive data.

callback_snd snd_callback

Specifies the callback for the sending operation. Client to server. If set to null, the EPOCH API will send immediately a 0 to the server indicating the end of sending transmission.

callback_rcv rcv_callback
: Specifies the callback for receiving operation. Server to client. If set to NULL, the EPOCH API will silently discard the arriving data until receive 0 from the server.

const char *wd
: Specifies the endpoint working directory. If NULL, don’t assume any particular working directory.

int *es
: A pointer to receive the endpoint exit status.

The return value is EPOCH_SUCCESS for the correct execution of the entire process. Otherwise it will return any of the following errors:

EPOCH_EBUFNULL
: An error has occurred creating communication buffers.

EPOCH_ECONNECT
: Error trying connecting to the server.

EPOCH_ESEND
: Error sending data and the connection is terminated.

EPOCH_ERECV
: Error receiving data and the connection is terminated.

EPOCH_EENDPTAUTH
: Error in endpoint authentication process. This could be either because the endpoint an its parameter are not allowed or because the authentication token is wrong

EPOCH_EMTINIT

If mode is set to MULTI_THREAD, this error indicate that an error has occurred in the multi-thread environment initialization.

EPOCH_EKEYMIN
: Is returned if the specified session key length is less than 16 bytes.

Note of caution: The client and the endpoint must keep in mine that one write or read at one end is not necessary equivalent to one read or write at the other end. It's crucial for both to continue reading, in the case of client, until the rcvlen parameter of the rcv_callback function is set to 0 by the EPOCH API; in the case of endpoint, until **read primitive** returns EOF or any signal indicating its standard input closure.

At the endpoint programming is important to close the standard output and the standard error descriptors once writing operations are done. This way we tell the EPOCH server we have finish doing writing.

int epoch_endpoint_bk**(struct epoch_s ***e**, const char ***endpt**, const char ***auth**, const char ***wd**)**

The epoch_endpoint_bk function connect to the server and run the endpoint indicated by the *endpt parameter. The parameters are the same of epoch_endpoint except for the removal of some of them: the callbacks, the buffer size, the exit status and the mode which is DETACHED. The possible return values are the same explained above. This version of the epoch_endpoint function implies that the endpoint will run detached from the epoch server. Therefor is not possible to write to the endpoint standard input or read from its standard output. If EPOCH_SUCCESS is returned, it means that EPOCH will do its best to execute the endpoint.

Expert mode

int epoch_endpoint_ex**(struct epoch_s ***e**, const char ***endpt**, const char ***auth**, **int64_t** bufsz**, const char ***wd**)**

The epoch_endpoint_ex function connect to the server and run the endpoint indicated by the *endpt parameter. The parameters are the same of epoch_endpoint except for the removal of some of them, especially the callbacks. The possible return values are the same explained above. This version of the epoch_endpoint function implies that the client should manually use the helper functions described below to receive from and send to the endpoint. That receive/send operations may proceed if and if only epoch_endpoint_ex returns EPOCH_SUCCESS.

int epoch_send_ex(**struct epoch_s ***e, **void ***buf, **int64_t** len)

This helper function is used to send data to the server. The len argument must always be less than or equal to the buffer specified in the epoch_endpoint_ex call. The client is responsible of send 0 as the value of the length to tell the server that sending operations are done.

struct epoch_s *e
: A pointer to initialized structured with the parameters to establish the connection to the server.

void *buf
: A pointer to the buffer with the data to be sent to the server.

int64_t len
: The length of the payload stored in the buf argument.

The return value is 0 for success, otherwise it return any of the following errors:

EPOCH_ESEND
: Error sending data and the connection is terminated.

EPOCH_EEXFLAG
: This error indicates that the function has sent already a 0 to the server so it cannot be called again.

int epoch_recv_ex(**struct epoch_s ***e, void *buf, int64_t *len)

This helper function is used to receive data from the server. The *len argument will be set to the size of the received payload. The client is responsible of continue calling epoch_recv_ex until it receives 0 as the value of *len. Once 0 has received, the server is telling to the client that sending data is done.

struct epoch_s *e
: A pointer to initialized structured with the parameters to establish the connection to the server.

void *buf
: A pointer to the buffer in which data will be received.

int64_t len
: The length of the payload stored in the buf argument.

The return value is 0 for success, otherwise it return any of the following errors:

EPOCH_ERECV
: Error receiving data and the connection is terminated.

EPOCH_EEXFLAG
: This error indicates that the function has received already a 0 from the server so it cannot be called again.

It is important to acknowledge that the previous helper functions can be called in any order, even in parallel, as long as a basic few things are respected:

1-) The client must respect the semantics of the communication that is defined in the endpoint -the program being executed at the server side inside EPOCH context-.
2-) Once the client decide that there is nothing else to send, it has to send a 0 as value of the len argument of the epoch_send_ex helper function.
3-) Once the client receive 0 as a value of the *len argument of the epoch_recv_ex helper function, the server has done sending data.
4-) After sent and receive 0 as len and *len argument the epoch_fin_ex must be called in order to receive the exit status of the executed endpoint and to perform the ending communication handshake.

int epoch_fin_ex(**struct epoch_s ***e, **int ***es)

This function must be called after sending and receiving data has done in order to receive the endpoint exit status and to finish gracefully the communication. If succeed it returns 0. Possible return values on error:

EPOCH_ERECV
Error while receiving the endpoint exit status.

int epoch_signal**(struct epoch_s ***e**, int** signo**)**

This function allow to send a signal to the endpoint -the program being executed in the epoch context-. If the server in which epoch runs support signals, it will be passed to the endpoint. This function cannot be called after the client sent 0, as the value of the header, to the server. The return value is 0 for success, otherwise it return any of the following errors:

EPOCH_ESEND
Error sending data and the connection is terminated.

EPOCH_ESIGCONXT
The signal has been sent out of context. This means that it was sent after the client sent 0, as the value of the header, to the server.

int resolvhn**(const char ***host**, char ***ip**, int** v6**, int** timeout**)**

This function is capable of resolve and ipv6 or ipv4 from a domain name. If the function success it return 0, otherwise -1.

const char *host
: The domain name to resolve, example “google.com”.

char *ip
: A buffer greater enough to store an ipv4 or an ipv6 in dotted format. “127.0.0.1” or “::1”.

int v6
: If set to non-zero indicates that the resolving must be done in the ipv6 address space. Otherwise, an ipv4 will be resolved.

int timeout
: A timeout in seconds indicating the time the client is willing to wait until resolution. If is 0, then there is no timeout.