Messaging system.

Replacement of transport consists of 3 main steps. Developer need to do following:

1. Implement you read and write functions and send them as callback functions to mbedtls\_ssl\_set\_bio.
2. Implement functions equivalent to socket functions: connect, accept, close – these functions are called from application code. Other socket functions may be required for some protocols – those functions are in net\_socket.c module. For this messaging implementation not used functions are turned off – code branch are excluded by defining macro USE\_SHARED\_MEMORY
3. Introduce synchronization for sending messages between client and server.

This implementation uses shared memory for message exchange between client and server. Shared memory uses windows implementation: “memory mapped file” windows kernel object that uses swap file to avoid creating file in file system.(See file mbedtls\programs\ssl\mmf\_communication.c/mmf\_communication.h)

To switch between socket and shared memory macro is defined: USE\_SHARED\_MEMORY

Turning it on: shared memory will be used. Turning off: sockets are used.

Step 1.

Read/write callback are in mbedtls\programs\ssl\mmf\_communication.c

Those callbacks should be passed to following function (called from application code: dtls\_mmf\_server.c/ dtls\_mmf\_client.c) – parameters in grey are not used in current implementation. Description is copied from original functions and not all of parameters are used in this simple implementation. But may be important in other messaging systems.

mbedtls\_ssl\_set\_bio( &ssl, &client\_fd,

mbedtls\_net\_send\_mmf,

mbedtls\_net\_recv\_mmf,

mbedtls\_net\_recv\_timeout\_mmf);

/\*\*

\* \brief **Read** at most 'len' characters. If no error occurs,

\* the actual amount read is returned.

\*

\* \param ctx Socket

\* \param buf The buffer to write to

\* \param len Maximum length of the buffer

\*

\* \return the number of bytes received,

\* or a non-zero error code; with a non-blocking socket,

\* MBEDTLS\_ERR\_SSL\_WANT\_READ indicates read() would block.

\*/

int mbedtls\_net\_recv\_mmf(void\* ctx, unsigned char\* buf, size\_t len);

/\*\*

\* \brief **Write** at most 'len' characters. If no error occurs,

\* the actual amount read is returned.

\*

\* \param ctx Socket

\* \param buf The buffer to read from

\* \param len The length of the buffer

\*

\* \return the number of bytes sent,

\* or a non-zero error code; with a non-blocking socket,

\* MBEDTLS\_ERR\_SSL\_WANT\_WRITE indicates write() would block.

\*/

int mbedtls\_net\_send\_mmf(void\* ctx, const unsigned char\* buf, size\_t len);

/\*\*

\* \brief **Read** at most 'len' characters, blocking for at most

\* 'timeout' seconds. If no error occurs, the actual amount

\* read is returned.

\*

\* \param ctx Socket

\* \param buf The buffer to write to

\* \param len Maximum length of the buffer

\* \param timeout Maximum number of milliseconds to wait for data

\* 0 means no timeout (wait forever)

\*

\* \return the number of bytes received,

\* or a non-zero error code:

\* MBEDTLS\_ERR\_SSL\_TIMEOUT if the operation timed out,

\* MBEDTLS\_ERR\_SSL\_WANT\_READ if interrupted by a signal.

\*

\* \note This function will block (until data becomes available or

\* timeout is reached) even if the socket is set to

\* non-blocking. Handling timeouts with non-blocking reads

\* requires a different strategy.

\*/

int mbedtls\_net\_recv\_timeout\_mmf(void\* ctx, unsigned char\* buf, size\_t len,

uint32\_t timeout);

Step 2.

Functions below are specific to shared memory implementation. They creates/uses shared memory – between two processes – one process writes buffer of mmf – another reads from buffer. First int value (4 bytes) contains length of record.

|  |  |
| --- | --- |
| HANDLE create\_mmf(); | Create memory mapped file (mmf) - shared memory |
| PVOID map\_mmf (HANDLE hFileMap); | Maps file to process address space – in the begging of program |
| void unmap\_mmf (PVOID pView); | Unmap file in the end of program |
| void write\_mmf (PVOID pView, void\* buf, int nbytes); | Writes to mmf – called from callback above |
| int read\_mmf (PVOID pView, void\* buf); | Read from mmf – called from callback above |
| void close\_mmf (HANDLE hFileMap); | Closes mmf in the end. |

Step 3. Synchronization.

Following events are used:

ghConnectedEvent - signals to server that connection from client accepted.

ghSignalAboutEvent – signals that message is written – set in write operation

ghWaitForEvent – blocks read operation until message arrives.

|  |  |
| --- | --- |
| void create\_event\_mmf(enum PointOfView pointOfView); | Creates all 3 events. PointOfView meams point of view: server or client because the same event is treated differently – event set in write op of client is used for waiting for message in server. And vice versa. |
|  |  |
| void accept\_connection\_mmf(); | Server blocks on this operation waiting for connection from client |
| void connect\_mmf(); | Establishes connection with server by setting ghConnectedEvent to signaled state |
| void close\_connection\_mmf(); | Close all connections by resetting ghConnectedEvent to non-signaled state |

To verify/ test messaging implementation SysInternals [tcpview](https://docs.microsoft.com/en-us/sysinternals/downloads/tcpview) is used (or netstat) and following can be done:

1. Test socket version by undefining USE\_SHARED\_MEMORY . and rebuild. Run dtls\_mmf\_server.bat and dtls\_mmf\_commissioner\_J1.bat and observe that TCPView shows connection on port 4433. Output will show that protocol works.
2. Define USE\_SHARED\_MEMORY and rebuild. Repeat the same as in 1). Output will show additional records for mmf read and write functions and will show that protocol works – message sent from client and received back. TCPView will do not show connection because data is sent via shared memory.