Universal Data Transporter for NETWARE
Software Development Kit Specifications

Version 1.00a

Table of contents:

1. The Client/Server SDK for NETWARE 3.x & 4.x.

This software development kit was designed for those developers who need to write a client/server NETWARE based application.

The Universal Data Transporter SDK was designed to meet all the needs requiered by such client/server applications. The developer will be involved in writing only the request processing modules. These modules will be implemented upon both client and server machines.

1.1 Presenting the layers.

The architecture of a client/server application involves networking considerations. This architecture can be presented as a set of stacked up layers. Figure 1 shows these layers considering the Universal Data Transporter services.

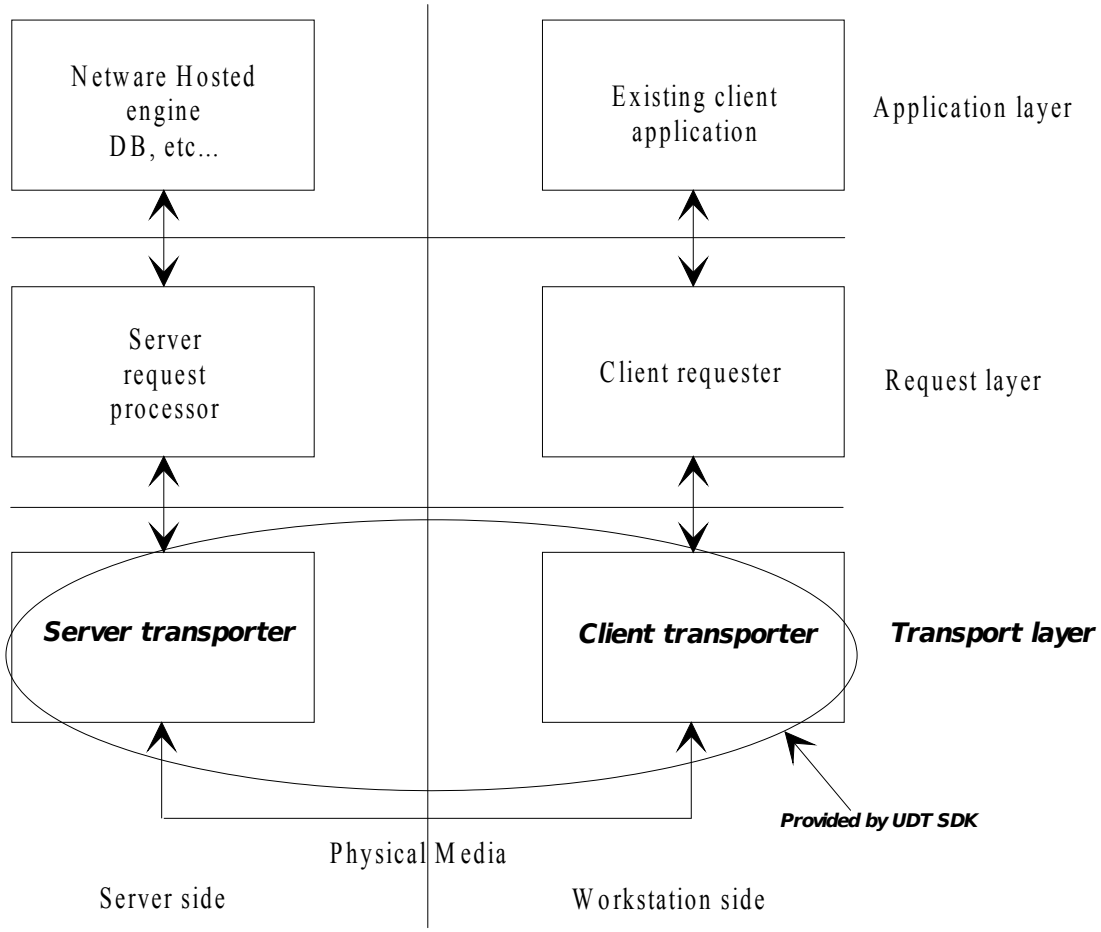

Figure 1.

Considering the application layer as the existing application, the two underneath layers are now needed to implement the client/server architecture.

The request layer will use an application-specific format. The layer will need formatting and unformatting the requests. **The Universal Data Transporter SDK gives the developer all the tools needed to implement this layer.**

The most critical layer is, of course, the transport one. As it involves in system specific resources, the development is a long effort. Also, tuning the performances is very difficult as it takes many tests. **This is the layer provided par the Universal Data Transporter.**

That's why this SDK provides an API definition, as well as callback functions for writing the request processing module on the server and direct calls on to the local Windows/MsDos station. Of course, the server request processor will be implemented in the form of a Netware Loadable Module (NLM).

The client local request processor can be implemented as a WINDOWS DLL or a DOS Lib file.

The two request processing modules (client and server) must implement a unique request format. These modules ignore, though, how these requests flow on the network. Each request must be formatted in a single buffer. This SDK gives you the proper tools to transmit these buffers from the client to the server and back, all transparently!

The SDK contains the following components:

⇒ A Netware NLM as the server request transporter.
⇒ Windows DLL and DOS static library as the client transporter.
⇒ Sources for the server request processor squeleton (implementing all the call-back functions with proper syntax ready to compile) with a compiled NLM version.
⇒ Documentation for all the public APIs.

1.2 The server transporter.

The main job for the server transporter is to stay running idle on the server machine, waiting for client stations to request connecting to it. At the same time the server supports managing requests receipt for all the clients simultaneously as well as sending back the results to the client. This concurrent task can be done by using the threads feature offered by the 32 bit NETWARE operating system.

The server transporter will notify the server request processor (implemented by you developer) when both a new client arrives and terminates the session so the module can do any client-related initialization and cleanup processing.

The server request transporter will collect all the received requests (from the client) into a linear address space and pass it up to the server request processor for processing. After passing the buffer, the server transporter stays waiting for the request to be fully completed.

Also, the transporter will notify the request processor at load/unload time. At load time, the request processor must give the number of simultaneous connections allowed.

The server transporter software does not use the NETWARE SAP services to make itself known over the network to all the NETWARE servers for ISDN optimizing considerations. Instead, a dynamic object into the NETWARE bindery will be created at program start time. These objects will be removed before finishing the program. The bindery dynamic object guarantees automatic deletion at remote server reboot time if any problem should occur locally on the server machine.

This module is implemented in the UDTSRV.NLM file.

Note that the server transporter NLM program can not run stand-alone. It will rely on the presence of a dynamic library as another NLM: the server request processor.

1.3 The server request processor squeleton.

The request processor will act as a library for the transporter**. This is the module that has to be written by the developer**.

The request processor will be responsible for supporting the following notifications reported by the UDTSRV module:

New client session (through a client-based handle provided by the transporter).
Normal client terminate session (through the handle previously given).
Critical client terminate session (through the handle previously given).
Request description text.
Request processing (through the handle previously given).
Respond if loading is possible at transporter load time (for licensing).
Down itself (or whatever) when the transporter downs.
Give the number of concurrent connections that the transport layer should accept.

These services will be implemented as public API functions and stand as the minimum squeleton for the server request processor implementation. The SDK comes shipped with a software version, ready-to-compile of this minimum implementation so the developer just has to fill the gaps...

This module must be implemented in the RQPROC.NLM file. The module is automatically loaded by UDTSRV.NLM module.

1.4 The client transporter.

The client transporter is responsible for opening/closing session with several servers simultanously, also for delivering the requests to the server transporter and receiving the results. That's why the client transporter will provide the following services:

Opening a session with a specified server.
Closing the session.
Sending a request/receive the result.
List all the available servers.

These services are implemented as public API functions and will be called by you developper from your client requester software.

This module is implemented in the UDTWIN16.DLL (for Windows 3.x) or UDTDOSMS.LIB (Microsoft) / UDTDOSBC.LIB (Borland) files (for DOS).

Note: The Windows NT version is under porting process. It will be available as UDTWIN32.DLL file. The OS/2 32bit version will be ported later.

1.5 The client requester.

The client requester is not part of the SDK. Instead the developer must code the request formatting/unformatting procedure inside it. The formatteded request is then passed to the client transporter that will send it to the server transporter.

This formatted request will be received as-is by the server request processor that will unformat it for processing. It will format the results back to the client so these results will be received back by the client requester. This formatting/unformatting rule remains the responsability of the developer.

1.6 The SDK development environment.

The developer needs the following components in order to write the server request processing modules:

A 486 based machine with 8mb of memory,
The Novell NLM SDK version 3.0 and above,
The Watcom C compiler version 10.0 and above.

The developer needs the following components in order to write the client requester modules:

A 486 based machine with 8mb of memory,
The Microsoft Visual C++ 1.0 and above,
The Microsoft Windows operating system version 3.1 and above.

2 Installation and operation.

2.1 Station requierements.

The station under DOS should be configured with the usual tools to connect to a file server. Note that the IPX layer must be version 3.10 and above.

The station under Windows should be configured with the Netware tools for Windows.

2.2 Client transporter parameterizing.

The Windows version can be parameterized in the SYSTEM.INI file as follows:

[Universal Data Transporter]
DisableWatchdog = <0-1> default is 0
MaximumECBs = <8-64> default is 16
TimeOutConnection = <1-20> default is 10

The SPX watchdog can be disabled by the local workstation to minimize network traffic, best for ISDN WAN networks.

The number of outstanding ECBs under Windows can be modified because one ECB consumes 576 (about 512 for the datas) bytes under the first megabyte wich is a critical resource for WINDOWS. Nevertheless, the higher the number the faster the communications with the server. If a single request buffer is bigger than the total size for the datas in the ECBs (ie 16 ECBs * 512 = 8kb) the client transporter has to enter an ECB polling algorithm that must wait ECBs to be released for continuing transmition of the datas.

The timeout at connection time can be extended or reduced, best for ISDN WAN networks.

2.3 Server requierements.

2.4 NLM error codes.

3 Definition of the SDK APIs.

The SDK contains three modules:

The server transporter.
The server request processor.
The client transporter.

The modules are delivered as executables and libraries.

Only two modules export public APIs to be used. These modules are:

The server request processor
The client transporter

The direction of the API calls is shown in the following figure:

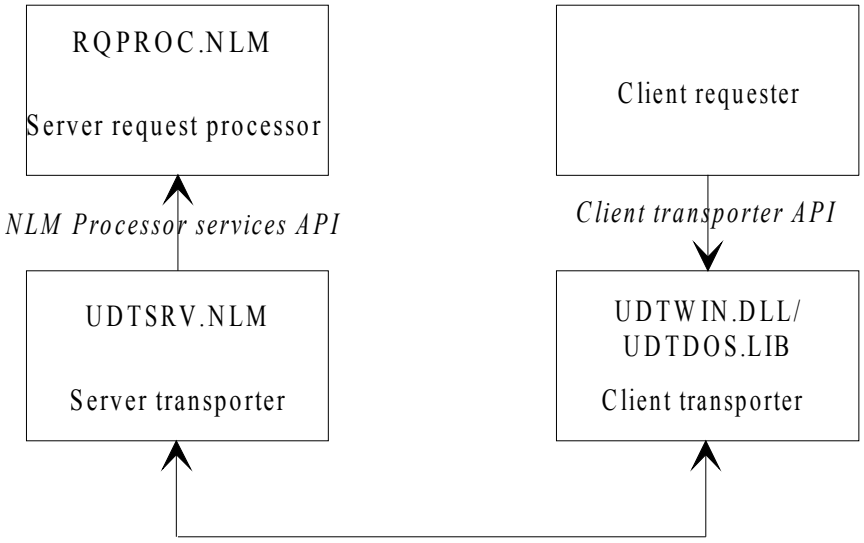

Figure 2.

The client requester (appearing in dot line) represents the client application that sends requests and receives the results.

Although the communication goes from the client to the server and comes back with the result informations, the API arrows show only the way from the client to the server. This is because the calling API always receives a buffer containing the result inline on return of API function.

3.1 The server request processor API.

This module (RQPROC.NLM) must be implemented by the developer. The module must implement the following APIs. The NLM produced acts as a Dynamic Library. When the UDTSRV.NLM (server request transporter) loads up, it automatically loads the RQPROC.NLM program. If all the APIs are not implemented, then UDTSRV.NLM cannot load up due to missing public references.

The APIs header file is RQPROC.H.

3.1.1 RQProcRegisterNewClient.

The **RQProcRegisterNewClient** function must accept a new client that just opened a session.

Syntax: int RQProcRegisterNewClient (HCLIENT hClient, const IPX_ADDR pstIPXAddress, WORD wNetwareConnection, const char * pszUserName)

hClient This is a handle provided by the transport layer. The application is expected to store this handle so to free any allocated resources for this client when it will unconnect (RQProcClientTermination).

pstIPXAddress The IPX address of the remote station.

wNetwareConnection The logical netware connection of the remote station.

return The function returns a value among the following:

RQP_OK	The connection is opened.
RQP_MEMORY	No more memory.
RQP_NOMOREUSERS	The limit of concurrent connections (commercial) was reached.

Comment The handle passed in parameter guarantees an uniqueness of identification. The transport layer will close the connection if the connection is not accepted. A message will be sent to the station. The transporter creates a new Netware thread group for the client, this function is called into the user's thread group.

3.1.2 RQProcClientTermination.

The **RQProcClientTermination** notifies a client disconnection. The application is expected to free all the resources allocated for this client.

Syntax: BOOL RQProcClientTermination (HCLIENT hClient)

hClient This is a handle provided by the transport layer.

return The function returns NULL if there happened any problem. Not NULL if the operation completed. This function is called into the user's thread group.

3.1.3 RQProcCriticalClientTermination.

The **RQProcCriticalClientTermination** notifies a client critical disconnection. The application is expected to free all the resources allocated for this client.

Syntax: BOOL RQProcCriticalClientTermination (HCLIENT hClient)

hClient This is a handle provided by the transport layer.

return The function returns NULL if there happened any problem. Not NULL if the operation completed. This function is called into the user's thread group.

3.1.4 RQProcGetDescriptionText.

The **RQProcGetDescriptionText** asks for a description of the request.

Syntax: BOOL RQProcGetDescriptionText (const char * pRequest, char * pDescription, int iLength)

pRequest The buffer containing the full request received.

pDescription The buffer to fill with the description text.

iLength The length not to go beyond when filling the description, including 0.

return The function returns NULL if there happened any problem. Not NULL if the operation completed.

Comment This description is used by the transport layer to display the current status of the connection. This function is called into the user's thread group.

3.1.5 RQProcProcessRequest.

The **RQProcProcessRequest** asks for processing the request.

Syntax: RQSTATUS RQProcProcessRequest (HCLIENT hClient, char ** ppRequest, LONG * pRequestLength)

pRequest The buffer containing the full request received. The allocation is a malloc call. The function is expected to call free function to release the buffer. The returned buffer is created using malloc function. The returned buffer address must be put back into the ppRequest variable. This returned buffer will be sent back to the station, next il will be removed by a free call by the server request transporter.

pRequestLength The buffer length is passed in input and output, like the request buffer.

return The function returns a value among the following:

RQP_OK	The request completed.
RQP_MEMORY	Not enough memory to complete the request.
RQP_BADFORMAT	Badly formatted request.
RQP_UNKNOWN	Unknown request.

Comment The application will process the request that was passed by the client requester on the remote workstation. The buffer is the one that was passed to the UdtSendBuffer function on the remote station. This function is called into the user's thread group.

3.1.6 RQProcLoadTransporter.

The **RQProcLoadTransporter** notifies the transporter loading. Upon called this API, the request processor is expected to accept or refuse loading (maybe for serialization violation).

Syntax: int RQProcLoadTransporter (char * pszCommandLine, char * pszStartupDirectory)

pszCommandLine The command line used to load the transporter.

pszStartupDirectory The startup directory of the transporter.

return The function returns a value among the following:

RQP_OK Loading is accepted.
RQP_ CANNOTLOAD Loading is refused.

Comment The application should not rely upon main, atexit, AtUnload functions calls to provide transparent portage for future versions of the Universal Data Transporter. Instead rely upon RQProcLoadTransporter and RQProcDownTransporter as all information is given to do initialization and cleanup processing.

3.1.7 RQProcDownTransporter.

The **RQProcDownTransporter** notifies the transporter termination. Upon called this API, the request processor is expected to end cleanly.

Syntax: void RQProcDownTransporter (void)

return No return value.

3.1.8 RQProcGetNumberOfConnections.

The **RQProcGetNumberOfUsers** returns the number of concurrent connections allowed by the transport layer.

Syntax: void RQProcDownTransporter (void)

return The number of connections.

Comment You have the guarantee that this function will be called after RQProcLoadTransporter so you can check any serialization information and validation relating to the maximum number of allowed connections.

3.2 The client requester API.

Theses APIs are provided in the libraries UDTWIN16.DLL and UDTDOSMS.LIB/UDTDOSBC.LIB. The APIs header file is UDTWIN16.H and UDTDOS.H.

Note that all the memory handling functions have been remapped so that the code is fully portable through the different platforms.

3.2.1 UdtOpenSession.

The **UdtOpenSession** opens a session with the server. The connection must be validated by the transport layer and by the request processor.

Syntax: int UdtOpenSession (LPHSESSION lphSession, LPUDTSERVER_INFO lpInfo)

lphSession Address of session handle that will be filled on return of function. This handle will be needed for further sevices.

lpInfo This information was created by a call to UdtGetServers function. If this pointer is set to NULL, the nearest server will be implicitly chosen.

return The function returns a value among the following:

UDT_OK	The connection is opened.
UDT_SERVERNOTPRESENT	The server transporter UDTSRV.NLM is not mounted.
UDT_IPXERROR	An IPX error occured.
UDT_CONNECTIONREJECTED	The server has reached the maximum client connections.

3.2.2 UdtCloseSession.

The **UdtCloseSession** closes a session with the server.

Syntax: int UdtCloseSession (HSESSION hSession)

hSession The handle of a valid session.

return The funtion returns a value among the following:

UDT_OK	The connection is closed.
UDT_BADCLIENTHANDLE	The handle is not valid.

3.2.3 UdtSendBuffer.

The **UdtSendBuffer** sends a buffer to the server. This buffer will contain the request that will be passed up to the server request processor for processing in the RQProcProcessRequest function.

Syntax: int UdtSendBuffer (HSESSION hSession, HPVOID FAR * lppBuffer, LPLONG lplBufferLength)

hSession The handle of a valid session.

lphBuffer The pointer of the allocation for both input and output (after processing getting back the results). The allocation must use UdtAlloc/UdtRealloc/UdtFree functions. The passed buffer will be freed after being sent to the server. The calling application is expected to free the given back buffer.

lplLength The pointer of the LONG type variable that contains the buffer length for input and outpout, like the request buffer.

return The funtion returns a value among the following:

UDT_OK	The connection is closed.
UDT_BADCLIENTHANDLE	The handle is not valid.
UDT_INVALIDMEMORY	The lppBuffer does not contain a valid allocation.
UDT_OUTOFMEMORY	The memory was not sufficient to complete the request.

3.2.4 UdtGetServers.

The **UdtGetServers** fills a buffer containing an array of servers description structures.

Syntax: int UdtGetServers (LPUDTSERVER_INFO FAR * lphServers)

lphServers The buffer that will contain the servers description is given at function return. This buffer is created by a UdtAlloc call. The calling application is expected to free the given back buffer by using the UdtFree function.

return The funtion returns the number of servers found.

The format of the structure containing the server informations is the following:

```
typedef struct tagUDTSERVER_INFO { char szServerName [ 48 ];
                                   BYTE uchIPXAddress [ 12 ];
                                 } UDTSERVER_INFO;
typedef UDTSERVER_INFO FAR * LPUDTSERVER_INFO;
```

3.2.5 UdtAlloc.

The **UdtAlloc** allocates a buffer for use by UDT. The allocation functions are portable through all the systems.

Syntax: void FAR * UdtAlloc (LONG lSize)

lSize The size to be allocated**. NOTE FOR DOS PLATFORM: The size must be limited to 64K maximum otherwise the allocation will fail.**

return The funtion returns the pointer allocated.

3.2.6 UdtFree.

The **UdtFree** frees a buffer for use by UDT. The allocation functions are portable through all the systems.

Syntax: BOOL UdtFree (HPVOID lpBuffer)

lpBuffer The buffer to be freed.

return The funtion returns not NULL if successful.

3.2.7 UdtReAlloc.

The **UdtReAlloc** re-allocates a buffer for use by UDT. The allocation functions are portable through all the systems.

Syntax: void FAR * UdtReAlloc (HPVOID lpBuffer, LONG lSize)

lpBuffer The buffer to be re-allocated.

lSize The new size to be allocated.

return The funtion returns the pointer allocated.

3.2.8 UdtCopyMemory.

The **UdtCopyMemory** makes a memory block copy.

Syntax: void UdtCopyMemory (HPVOID lpDstination, HPVOID lpSource, LONG lLength)

lpDestination The destination buffer.

lpSource The source buffer.

lLength The number of bytes to copy.

return No return.

4 Comprehensive index.