Chapter 4
ConnMan Design Specification

Abstract

ConnMan manages building and destroying connections, allowing applications and to be written independent of the underlying session protocol.

Contents

Introduction 3
Requirements 3
Functional Overview 4

Design Description 5
Checking Existing Connections for a Match 5
Resolving a Name to an Address and Session ID 5
Building a Connection Entry 5
Calling SessMux to Establish Connection 7
Maintaining the Connection Database 7
LRUing of NCP NDS Connections 7

ConnMan API 10
Connection APIs 10
Connection Database APIs 11
VLM Compatibility APIs 12

NESL Events 13

Configuration 15

Performance 15

Deliverables 15

Introduction

ConnMan manages connections for the client. It works closely with the Name Services Multiplexor (NSMux), the Authentication Multiplexor (AuthMux), and the Session Multiplexor (SessMux) to build connections over different session protocols and name services.

The following diagram shows ConnMan in relation to the other Client32 Requester NLMs:

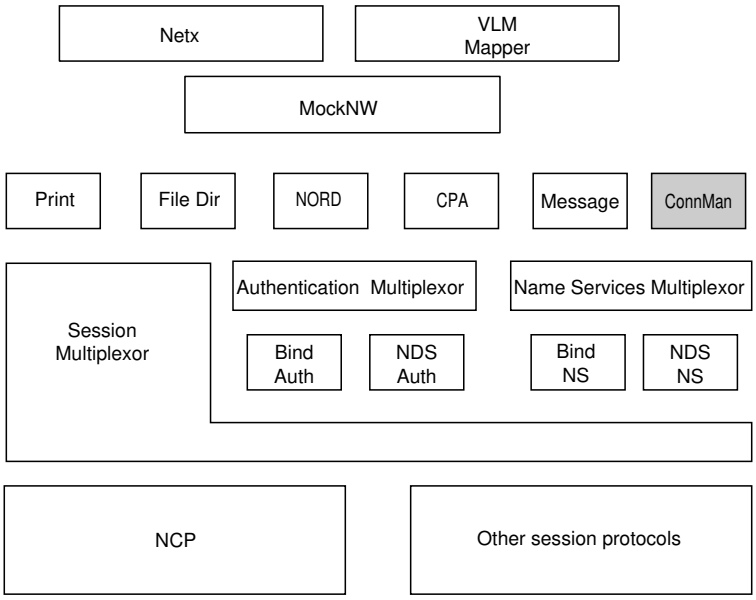

Figure 1. ConnMan and other Client32 NLMs.

Requirements

Like all Client32 Requester modules, ConnMan is written entirely in C and is portable to other Intel platforms.

Connman allows session protocol modules and authentication service modules to register and unregister dynamically. In addition, ConnMan dynamically increases its connection table to support more connections as needed.

Functional Overview

As a general overview of how ConnMan interacts with other Requester NLMs in establishing a connection, consider this example. (Each of these steps is discussed in greater detail in the *Design Description* section of this chapter.)

To establish a connection, ConnMan does the following:

1. Receives from an application a request to establish a connection with a server or tree of a given name.

2. Scans the existing connections to see if a connection to this entity already exists.

3. Calls the Name Services Multiplexor (NSMux) to resolve the server name to a transport address. The NSMux also returns a *sessionID*.

4. Builds a connection entry for the requested connection. This establishes a connection handle (*connHandle*) for the new connection.

5. Passes the *connHandle*, *sessionID*, and transport address to SessMux using **SESSConnectByAddress**. SessMux multiplexes the request to the correct session protocol provider, which actually establishes the connection and completes filling out the connection entry.

6. Returns the *connHandle* for the new connection to the application that requested it.

From here on, all session service requests (including file, directory, and printing functions) are completed via the Session Multiplexor and do not involve ConnMan, though ConnMan continues to maintain information about the connection itself.

Design Description

As seen in the Functional Overview, ConnMan coordinates the various steps needed to establish a connection. Each of those steps is described in detail here.

Checking Existing Connections for a Match

Before allocating a new connection, ConnMan first checks existing connections to see if the connection requested already exists. If such a connection does exist, ConnMan must also ascertain that the existing connection matches the scope requested by the new connection. That is, if the existing connection is private and the new connection is outside the scope of that private connection, a new connection will be established anyway.

The name used by ConnMan when checking for an existing connection is dependent on the name service being used. For Bindery connections, the server name is used (*ceServerName* in the Conn_Info structure); for NDS connections the domain name is used (*ceDomainName*). ConnMan determines which name service provider to use by checking the *NameSvcType* parameter that is passed in with the request for connection.

Resolving a Name to an Address and Session ID

The first thing ConnMan does when it receives a request for a connection (such as **CONNOpenByName**) is to call NSMux using **NSMResolveNametoAddress**. NSMux will return an address and session protocol ID. (Bindery, NDS, and PNW name service modules all return **NCP_SESSION_ID** for their session protocol IDs. Other name service modules could return a different session protocol ID, such as **SMB_SESSION_ID**.)

Building a Connection Entry

After retrieving an address and session ID from the NSMux, ConnMan builds a connection entry for this connection and assigns a *connHandle*. This *connHandle* will be used throughout the life of the connection to retrieve information about this particular connection; everything the Requester does with respect to this connection will use *connHandle* as a reference.

One step in filling out this connection table is to establish a pointer to the session-specific information that resides within the session protocol provider. This pointer in the connection entry structure is

called *ceSessionSpecPtr*, and is set by the session service provider when the connection is created (when **SESSConnectByAddress** is called).

The only way to modify or retrieve connection entry information is by using these ConnMan APIs:

CONNGetValue
CONNGetStructure
CONNSetValue
CONNSetStructure
CONNScanInfo

The following table lists all connection information that can be retrieved and set. (See CLIENT32.H).

Name	Define	Type	Who Reads	Who Writes
CONN_ENTRY_RETURN_NONE	0			
CONN_ENTRY_VERSION	1	UINT32	global	none
CONN_ENTRY_AUTH_SVC_ID	2	UINT32	global	AuthMux
CONN_ENTRY_BROADCAST_STATE	3	UINT32	global	global
CONN_ENTRY_REFERENCE	4	UINT32	global	none
CONN_ENTRY_DOMAIN_NAME	5	Struct	global	AuthMux
CONN_ENTRY_WORKGROUP_ID	6	Struct	global	AuthMux
CONN_ENTRY_SECURITY	7	UINT32	global	global
CONN_ENTRY_SERVER_CONN_NUM	8	UINT32	global	SessMux
CONN_ENTRY_AUTH_USER_ID	9	UINT32	global	AuthMux
CONN_ENTRY_SERVER_NAME	10	Struct	global	SessMux
CONN_ENTRY_TRAN_ADDR	11	Struct	global	SessMux
CONN_ENTRY_NDS_ABILITY	12	Flag	global	SessMux
CONN_ENTRY_MAX_IO	13	UINT32	global	SessMux
CONN_ENTRY_LICENSE	14	UINT32	global	global
CONN_ENTRY_PUBLIC_STATE	15	UINT32	global	NEVER
CONN_ENTRY_NAME_SVC_ID	16	UINT32	global	SessMux, NSMux
CONN_ENTRY_ROUND_TRIP	17	UINT32	global	SessMux
CONN_ENTRY_SERVER_VERSION	18	UINT32	global	SessMux
CONN_ENTRY_TRAN_ADDR_OBJ	19	UINT32	global	SessMux
CONN_ENTRY_SFT_LEVEL	20	UINT32	global	global
CONN_ENTRY_TTS_LEVEL	21	UINT32	global	global
CONN_ENTRY_SERVICE_NAME	22	Struct	global	AuthMux
CONN_ENTRY_PERM	23	Flag	global	ConnMan sets
CONN_ENTRY_AUTH	24	Flag	global	AuthMux sets
CONN_ENTRY_ANCHOR	25	Flag	global	ConnMan sets
CONN_ENTRY_SUSPENDED	26	Flag	global	ConnMan sets
CONN_ENTRY_RESOURCE_COUNT	27	UINT32	none	Global inc/dec
CONN_ENTRY_TRAN_SVC_ID	28	UINT32	global	SessMux
CONN_ENTRY_AUTH_HANDLE	29	UINT32	global	AuthMux
CONN_ENTRY_AUTH_SPEC_PTR	30	UINT32	Auth	AuthMux
CONN_ENTRY_SESS_SVC_ID	31	UINT32	global	SessMux
CONN_ENTRY_SESS_SPEC_PTR	32	UINT32	Sess	SessMux
CONN_ENTRY_ORDER_NUM	33	UINT32	global	SessMux
CONN_ENTRY_MAX_RW_IO	34	UINT32	global	SessMux
CONN_ENTRY_RETURN_ALL	65535	Struct	global	None
CONN_ENTRY_END_OF_TABLE	CONN_ENTRY_MAX_RW_IO		MAX VALUE	

If the user requests that all connection information be returned, the CONN_ENTRY_RETURN_ALL structure is used:

```
typedef struct _CONN_INFO_TYPE_ {
   UINT32   connInfoVersion;
   UINT32   connReference;
   UINT32   connMaxDomainNameLen;
   SPECT_DATA  connDomainName;
   UINT32   connNameSvcId;
   UINT32   connSecurity;
   UINT32   connServerConnNum;
   UINT32   connAuthUserId;
   UINT32   connAuthState;
   UINT32   connMaxServerNameLen;
   SPECT_DATA  connServerName;
   TRAN_ADDR_TYPE connTranAddr,
   UINT32   connMaxIo;
   UINT32   connLicense;
   UINT32   connMaxServiceNameLen;
   SPECT_DATA  connServiceName;
   UINT32   connRoundTrip;
   UINT32   connServerVersion;
} CONN_INFO_TYPE;
```

Calling SessMux to Establish Connection

ConnMan sends the request to the SessMux, which multiplexes it to the appropriate session protocol module. The session protocol module completes filling out the connection entry.

Maintaining the Connection Database

ConnMan maintains a database of information about each connection.

The number of connections maintained by ConnMan is dynamic. To begin, it's the number of Conn_Entry structures that can fit within a 4K block. If more connections are needed, more can be created by re-using existing connections or by allocating more memory. The algorithm to determine which is done (reuse or allocate new memory) will be determined later.

LRUing of NCP NDS Connections

ConnMan caches network connections. The cache is managed with a Least-Recently-Used (LRU) algorithm. LRUing connections allows clients with memory constraints to make the most efficient use of memory space for connections.
Also, LRUing provides improved performance during NDS operations.

The Conn_Entry structure includes maintaining two separate counts, an *in-use count* and a *resource count*, to track how connections are used and released. Also, it is still possible for certain connections to be marked *LONG_LIVED_CONNECTION*, meaning that they stay open even after their calling application closes.

With ConnMan, connections remain cached even if they are not currently being used, with the idea that they might be used again in the near future. If a connection request is received by the Requester and no unused connections are available, the least recently used cached connection will be torn down and used to fulfill the caller's request.

In-Use Counts. In previous Requester implementations, it was difficult to know when an application or library finished with a connection. The old API set (NWCALLS) gave actual connection

handles to every connection that an application even looked at, whether or not the application actually opened and used that connection. For example, if an application queried many connections to look for a certain attribute, it would be given connection handles to each connection, and there would be no way to know which connection it finally opened and used, and which handles were discarded.

The new API set solves this problem by giving an application only a *connection reference* (alias) instead of the actual connection handle. In order for an application to actually use a connection, the application has to get the actual connection handle by calling **CONNOpenByReference.**

By requiring an application to actively open a connection, the Requester can track how the application uses the connection. When the application is finished using the connection, it calls **CONNClose**. Opening and closing a connection increments and decrements a per-process *in-use* count in the connection entry for that connection.

Note: The VLM Mapper is allowed to allocate a connection and never close it. Though this is never done with the current APIs, it used to be done with the VLMs and so must be supported. To deal with this situation, ConnMan listens for the *task terminate* event, and decrements the *in-use* count for the appropriate connection when that event occurs. When the *in-use* count goes to zero, the connection is closed.

Long-Lived Connections. Long-lived connections are flagged differently depending on whether the application is a Netx/VLM application or a Directory Services application.

- **Netx/VLM.** This option provides backward-compatibility with the old API set. With the old APIs that did not use Directory Services, there was no way (or need) to signify a connection that needed to stay in place after an application closed. Since that is now an option, this flag is available to support those APIs.

  An example of such an API is **NWAttachToFileServer**. Though no hard resources may be allocated, it is necessary that this connection remain alive (not LRUed) until it is specifically closed with **NWDetachFromFileServer**.

  To make a connection long-lived, the caller sets a flag

parameter in the open and close APIs. Opening and closing a connection with this flag will set or clear the *cePermanentState* state indicator in the connection entry. This scheme has the advantage that the connection can be marked long-lived without using a separate call.

- **Directory Services.** Directory Services connections must be careful to never throw away the DS monitored connection, which is the connection to the DS database. This connection is protected by the *Anchor* flag, which is set by NDS.NLM and must not be manipulated by any other application.

Resource Count. Certain resources, such as mapped drives and redirected printer ports, remain intact even after an application terminates. Since applications can allocate these hard resources without the connection being permanent, ConnMan needs to be able to LRU a connection that has resources associated with it but an in-use count of zero. This is done by closing the connection (**CONNClose**) with the *LONG_LIVED_CONNECTION* flag. If, when the connection is closed, there are no other applications using it, then an event is broadcast for upper modules to free resources associated with the connection. The process of freeing the resources causes the *resource count* to go zero, and makes the connection LRUable.

When the *in-use* and *resource counts* go to zero, ConnMan will call the **SessDisconnect** routine of the appropriate session protocol module. The session protocol can either destroy the connection or place it on the LRU connection list. The NCP session protocol will place NetWare 4.x server connections on the LRU connection list. NetWare 3.x server connections will never be LRUed but instead are destroyed when the above counters all go to zero.

ConnMan API

There are three kinds of ConnMan APIs: connection APIs, database APIs, and VLM Compatibility APIs.

Connection APIs

These APIs establish connections with remote entities. If a connection is already established with the remote entity and the connection being requested is not private, the connection's "in-use" count is incremented and the connection handle to the already-established connection is returned.

The following connection-oriented APIs have been defined:

CONNClose
CONNOpenByAddress
CONNOpenByName
CONNOpenByRefererence

Connection Database APIs

These APIs are used to get, set, and reset connection entry fields.

CONNGetValue
CONNGetStructure
CONNSetValue
CONNSetStructure
CONNScanInfo
CONNIncInfo
CONNDecInfo

VLM Compatibility APIs

The following are miscellaneous APIs needed by system NLMs for VLM backward compatability or to implement session protocol independence.

CONNGetDefaultConnection
CONNGetNumConnections
CONNSetDefaultConnection

NESL Events

Events Produced

ConnMan produces NESL events to inform interested NLMs of new connections and destroyed connections. The specific events produced by ConnMan are:

EVENT_CONN_PRE_CREATED
Syntax: NESLProduceEvent(EVENT_CONN_PRE_CREATED, connHandle);

Description: ConnMan will produce this event before calling **SessConnectByAddress**. This will inform interested NLMs that a session connection is about to be made to a remote host.

EVENT_CONN_CREATED
Syntax: NESLProduceEvent(EVENT_CONN_CREATED, connHandle);

Description: ConnMan will produce this event after calling **SessConnectByAddress** if it's successful in creating a session connection to remote host.

EVENT_CONN_DESTROYED
Syntax: NESLProduceEvent(EVENT_CONN_DESTROYED, connHandle);

Description: ConnMan will produce this event after calling **SessDisconnect**.

EVENT_CONN_PRE_DESTROYED
Syntax: NESLProduceEvent(EVENT_CONN_PRE_DESTROYED, connHandle);

Description: ConnMan will produce this event before calling **SessDisconnect**.

EVENT_CONN_RENAME

Syntax: NESLProduceEvent(EVENT_CONN_RENAMED, *conn_Rename);

Description: When ConnMan places a connection on the LRU list, the connection handle for that connection is changed or renamed. This causes synchronization issues for those system NLMs that track those connections. This callout lets them synchronize their internal naming and tracking schemes with the new connHandle name and number.

Events Consumed

ConnMan consumes these events:

EVENT_CONN_RECONNECTED
EVENT_PROCESS_TERMINATE
EVENT_MOBILE_STATUS_CHANGE
EVENT_AUTH_UNREGISTER_SERVICE
EVENT_NAME_UNREGISTER_SERVICE
EVENT_SESS_UNREGISTER_SERVICE

Configuration

ConnMan currently has no configuration parameters that are set in NET.CFG.

Performance

ConnMan simply stores connection information and multiplexes requests to session protocol modules and to authentication service modules. Hence there is no performance measure available to measure its throughput. ConnMan should be coded efficiently so that it can store and retrieve items in the connection table.

Deliverables

The following are the deliverables for ConnMan:

Documents

CONNT.WP	Unit test plan describing tests and expected results that the ConnMan must pass before being accepted for cross- platform integration testing.

Executables

CONNMAN.NLM	NLM executable that implements this design specification.

Source Code

CONNMAN.C	C code that implements this design specification
CONN_INT.H	Header file that defines constants/structures defined in this design document.
CONNMAN.MAK	Makefile to build CONNMAN.NLM