The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit

9: Networking and IPC

Last Update: August 16, 2011.

Connection Library [Library xconnect: include | src]

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

Includes a generic socket interface (SOCK), connection object (CONN), and specialized connector constructors (for sockets, files, HTTP, and services) to be used as engines for connections. It also provides access to the load-balancing daemon and NCBI named service dispatching facilities.

Although the core of the Connection Library is written in C and has an underlying C interface, the analogous C++ interfaces have been built to provide objects that work smoothly with the rest of the Toolkit.

Note: Because of security issues, not all links in the public version of this file are accessible by outside NCBI users.

- Overview
- Connections: notion of connection; different types of connections that the library provides; programming API.
  - Socket Connector
  - File Connector
  - HTTP Connector
  - Service Connector
- Debugging Tools and Troubleshooting
- C++ Connection Streams built on top of connection objects.
- Service mapping API: description of service name resolution API.
- Threaded Server Support

Chapter Outline

The following is an outline of the topics presented in this chapter:

- Debugging Tools and Troubleshooting Documentation
- C++ Interfaces to the Library
  - CONN-Based C++ Streams and Stream Buffers ncbi_conn_stream[.hpp | .cpp], ncbi_conn_streambuf[.hpp | .cpp]
  - Diagnostic Handler for E-Mailing Logs email_diag_handler[.hpp | .cpp]
  - Using the CONNECT Library with the C++ Toolkit ncbi_core_cxx[.hpp | .cpp]
  - Multithreaded Network Server Framework threaded_server[.hpp | .cpp]

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

- Basic Types and Functionality (for Registry, Logging and MT Locks) ncbi_core[.h | .c], ncbi_types[.h]
- Portable TCP/IP Socket Interface ncbi_socket[.h | .c]
- Connections and CONNECTORs
  - Open and Manage Connections to an Abstract I/O ncbi_connection[.h | .c]
  - Implement CONNECTOR for a ...
    - Abstract I/O ncbi_connector[.h | .c]
    - Network Socket ncbi_socket_connector[.h | .c]
    - FILE Stream ncbi_file_connector[.h | .c]
    - HTTP-based Network Connection ncbi_http_connector[.h | .c]
    - Named NCBI Service ncbi_service_connector[.h | .c]
    - In-memory CONNECTOR ncbi_memory_connector[.h | .c]
- Servers and Services
  - NCBI Server Meta-Address Info ncbi_server_info[.h | p.h | .c]
  - Resolve NCBI Service Name to the Server Meta-Address ncbi_service[.h | p.h | .c]
  - Resolve NCBI Service Name to the Server Meta-Address using NCBI Network Dispatcher (DISPD) ncbi_service[p_dispd.h | _dispd.c]
  - Resolve NCBI Service Name to the Server Meta-Address using NCBI Load-Balancing Service Mapper (LBSM) ncbi_service[p_lbsmd.h | _lbsmd.c | _lbsmd_stub.c]
  - NCBI LBSM client-server data exchange API ncbi_lbsm[.h | .c]
  - Implementation of LBSM Using SYSV IPC (shared memory and semaphores) ncbi_lbsm_ipc[.h | .c]
- Memory Management
  - Memory-Resident FIFO Storage Area ncbi_buffer[.h | .c]
  - Simple Heap Manager With Primitive Garbage Collection ncbi_heapmgr[.h | .c]
- Connection Library Utilities
  - Connection Utilities ncbi_connutil[.h | .c]
  - Send Mail (in accordance with RFC821 [protocol] and RFC822 [headers]) ncbi_sendmail[.h | .c]
  - Auxiliary (optional) Code for ncbi_core.[ch] ncbi_util[.h | .c]
  - Non-ANSI, Widely Used Functions ncbi_ansi_ext[.h | .c]

daemons [src/connect/daemons]

- LBSMD
- DISPD
- Firewall Daemon

Test Cases [src/connect/test]

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Overview

The NCBI C++ platform-independent connection library (src/connect and include/connect) consists of two parts:

- Lower-level library written in C (also used as a replacement of the existing connection library in the NCBI C Toolkit)
- Upper-level library written in C++ and using C++ streams

Functionality of the library includes:

- SOCK interface (sockets), which works interchangeably on most UNIX varieties, MS Windows, and Mac
- SERV interface, which provides mapping of symbolic service names into server addresses
- CONN interface, which allows the creation of a connection, the special object capable to do read, write, etc. I/O operations
- C++ streams built on top of the CONN interface

Note: The lowest level (SOCK) interface is not covered in this document. A well-commented API can be found in connect/ncbi_socket.h.

Connections

There are three simple types of connections: socket, file and http; and one hybrid type, service connection.

A connection is created with a call to CONN_Create(), declared in connect/ncbi_connection.h, and returned by a pointer to CONN passed as a second argument:

```
CONN conn; /* connection handle */
EIO_Status status = CONN_Create(connector, &conn);
```

The first argument of this function is a handle of a connector, a special object implementing functionality of the connection being built. Above, for each type of connection there is a special connector in the library. For each connector, one or more "constructors" are defined, each returning the connector's handle. Connectors' constructors are defined in individual header files, such as connect/ncbi_socket_connector.h, connect/ncbi_http_connector.h, connect/ncbi_service_connector.h, etc. CONN_Create() resets all timeouts to the default value kDefaultTimeout.

After successful creation with CONN_Create(), the following calls from CONN API connect/ncbi_connection.h become available. All calls (except CONN_GetTimeout() and CONN_GetType()) return an I/O completion status of type EIO_Status. Normal completion has code eIO_Success.

Note: There is no means to "open" a connection: it is done automatically when actually needed, and in most cases at the first I/O operation. But the forming of an actual link between source and destination can be postponed even longer. These details are hidden and made transparent to the connection's user. The connection is seen as a two-way communication channel, which is clear for use immediately after a call to CONN_Create().

Note: If for some reason CONN_Create() failed to create a connection (return code differs from eIO_Success), then the connector passed to this function is left intact, that is, its handle can be used again. Otherwise, if the connection is created successfully, the passed connector

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

handle becomes invalid and cannot be referenced anywhere else throughout the program (with one exception, however: it may be used as a replacing connector in a call to CONN_ReInit() for the same connection).

Note: There are no "destructors" on public connectors. A connector successfully put into connection is deleted automatically, along with that connection by CONN_Close(), or explicitly with a call to CONN_ReInit(), provided that the replacing connector is NULL or different from the original.

```
CONN_Read(CONN conn, void* buf, size_t size, size_t* n_read, EIO_ReadMethod how)
```

Read or peek data, depending on read method how, up to size bytes from connection to specified buffer buf, return (via pointer argument n_read) the number of bytes actually read. The last argument how can be one of the following:

- eIO_ReadPlain - to read data in a regular way, that is, extracting data from the connection;
- eIO_ReadPeek - to peek data from the connection, i.e., the next read operation will see the data again;
- eIO_ReadPersist - to read exactly (not less than) size bytes or until an error condition occurs.

A return value other than eIO_Success means trouble. Specifically, the return value eIO_Timeout indicates that the operation could not be completed within the allotted amount of time; but some data may, however, be available in the buffer (e.g., in case of persistent reading, as with eIO_ReadPersist), and this is actually the case for any return code.

```
CONN_ReadLine(CONN conn, char* line, size_t size, size_t* n_read)
```

Read up to size bytes from connection into the string buffer pointed to by line. Stop reading if either '\n' or an error is encountered. Replace '\n' with '\0'. Upon return *n_read contains the number of characters written to line, not including the terminating '\0'. If not enough space provided in line to accomodate the '\0'-terminated line, then all size bytes are used up and *n_read is equal to size upon return - this is the only case when line will not be '\0'-terminated.

Return code advises the caller whether another read can be attempted:

- eIO_Success -- read completed successfully, keep reading;
- other code -- an error occurred, and further attempt may fail.

This call utilizes eIO_Read timeout as set by CONN_SetTimeout().

```
CONN_Write(CONN conn, const void* buf, size_t size, size_t* n_written)
```

Write up to size bytes from the buffer buf to the connection. Return the number of actually written bytes in n_written. It may not return eIO_Success if no data at all can be written before the write timeout expired or an error occurred. Parameter how modifies the write behavior:

- eIO_WritePlain - return immediately after having written as little as 1 byte of data, or if an error has occurred;
- eIO_WritePersist - return only after having written all of the data from buf (eIO_Success), or if an error has occurred (fewer bytes written, non-eIO_Success).

Note: See CONN_SetTimeout() for how to set the write timeout.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

CONN_PushBack(CONN conn, const void* buf, size_t size)

Push back size bytes from the buffer buf into connection. Return eIO_Success on success, other code on error.

Note 1: The data pushed back may not necessarily be the same as previously obtained from the connection.

Note 2: Upon a following read operation, the pushed back data are taken out first.

CONN_GetPosition(CONN conn, EIO_Event event)

Get read (event == eIO_Read) or write (event == eIO_Write) position within the connection. Positions are advanced from 0 on, and only concerning I/O that has caused calling to the actual connector's "read" (i.e. pushbacks never considered, and peeks -- not always) and "write" methods. Special case: eIO_Open as event causes to clear both positions with 0, and to return 0.

CONN_Flush(CONN conn)

Explicitly flush connection from any pending data written by CONN_Write().

Note 1: CONN_Flush() effectively opens connection (if it wasn't open yet).

Note 2: Connection considered open if underlying connector's "Open" method has successfully executed; an actual data link may not yet exist.

Note 3: CONN_Read() always calls CONN_Flush() before proceeding; so does CONN_Close () but only if the connection is already open.

CONN_SetTimeout(CONN conn, EIO_Event action, const STimeout* timeout)

Set the timeout on the specified I/O action, eIO_Read, eIO_Write, eIO_ReadWrite, eIO_Open, and eIO_Close. The latter two actions are used in a phase of opening and closing the link, respectively. If the connection cannot be read (written, established, closed) within the specified period, eIO_Timeout would result from connection I/O calls. A timeout can be passed as the NULL-pointer. This special case denotes an infinite value for that timeout. Also, a special value kDefaultTimeout may be used for any timeout. This value specifies the timeout set by default for the current connection type.

CONN_GetTimeout(CONN conn, EIO_Event action)

Obtain (via the return value of type const STimeout*) timeouts set by the CONN_SetTimeout () routine, or active by default (i.e., special value kDefaultTimeout).

Caution: The returned pointer is valid only for the time that the connection handle is valid, i.e., up to a call to CONN_Close().

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

CONN_ReInit(CONN conn, CONNECTOR replacement)

This function clears the current contents of a connection and places ("immerse") a new connector into it. The previous connector (if any) is closed first (if open), then gets destroyed, and thus must not be referenced again in the program. As a special case, the new connector can be the same connector, which is currently active within the connection. It this case, the connector is not destroyed; instead, it will be effectively re-opened. If the connector passed as NULL, then the conn handle is kept existing but unusable (the old connector closed and destroyed) and can be CONN_ReInit()'ed later. None of the timeouts are touched by this call.

CONN_Wait(CONN conn, EIO_Event event, const STimeout* timeout)

Suspend the program until the connection is ready to perform reading (event =eIO_Read) or writing (event = eIO_Write), or until the timeout (if non-NULL) expires. If the timeout is passed as NULL, then the wait time is indefinite.

CONN_Status(CONN conn, EIO_Event direction)

Provide the information about recent low-level data exchange in the link. The operation direction has to be specified as either eIO_Read or eIO_Write. The necessity of this call arises from the fact that sometimes the return value of a CONN API function does not really tell that the problem has been detected: suppose, the user peeks data into a 100-byte buffer and gets 10 bytes. The return status eIO_Success signals that those 10 bytes were found in the connection okay. But how do you know whether the end-of-file condition occurred during the last operation? This is where CONN_Status() comes in handy. When inquired about the read operation, return value eIO_Closed denotes that EOF was actually hit while making the peek, and those 10 bytes are in fact the only data left untaken, no more are expected to come.

CONN_Close(CONN conn)

Close the connection by closing the link (if open), deleting underlying connector(s) (if any) and the connection itself. Regardless of the return status (which may indicate certain problems), the connection handle becomes invalid and cannot be reused.

CONN_Cancel(CONN conn)

Cancel the connection's I/O ability. This is **not** connection closure, but any data extraction or insertion (Read/Write) will be effectively rejected after this call (and eIO_Interrupt will result, same for CONN_Status()). CONN_Close() is still required to release internal connection structures.

CONN_GetType(CONN conn)

Return character string (null-terminated), verbally representing the current connection type, such as "HTTP", "SOCKET", "SERVICE/HTTP", etc. The unknown connection type gets returned as NULL.

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

CONN_Description(CONN conn)

Return a human-readable description of the connection as a character '\0'-terminated string. The string is not guaranteed to have any particular format and is intended solely for something like logging and debugging. Return NULL if the connection cannot provide any description information (or if it is in a bad state). Application program must call free() to deallocate space occupied by the returned string when the description is no longer needed.

CONN_SetCallback(CONN conn, ECONN_Callback type, const SCONN_Callback* new_cb, SCONN_Callback* old_cb)

Set user callback function to be invoked upon an event specified by callback type. The old callback (if any) gets returned via the passed pointer old_cb (if not NULL). Callback structure SCONN_Callback has the following fields: callback function func and void* data. Callback function func should have the following prototype:

typedef void (*FCONN_Callback)(CONN conn, ECONN_Callback type, void* data)

When called, both type of callback and data pointer are supplied. The callback types defined at the time of this writing are:

- eCONN_OnClose
- eCONN_OnRead
- eCONN_OnWrite
- eCONN_OnCancel

The callback function is always called prior to the event happening, e.g., a close callback is called when the connection is about to close.

Socket Connector

Constructors are defined in:

```
#include <connect/ncbi_socket_connector.h>
```

A socket connection, based on the socket connector, brings almost direct access to the SOCK API. It allows the user to create a peer-to-peer data channel between two programs, which can be located anywhere on the Internet.

To create a socket connection, user has to create a socket connector first, then pass it to CONN_Create(), as in the following example:

```
#include <connect/ncbi_socket_connector.h>
#include <connect/ncbi_connection.h>
#define MAX_TRY 3 /* Try to connect this many times before giving up */
unsigned short port = 1234;
CONNECTOR socket_connector = SOCK_CreateConnector("host.foo.com", port,
MAX_TRY);
if (!socket_connector)
 fprintf(stderr, "Cannot create SOCKET connector");
else {
 CONN conn;
```

```
 if (CONN_Create(socket_connector, &conn) != eIO_Success)
 fprintf(stderr, "CONN_Create failed");
 else {
 /* Connection created ok, use CONN_... function */
 /* to access the network */
 ...
 CONN_Close(conn);
 }
}
```

A variant form of this connector's constructor, SOCK_CreateConnectorEx(), takes three more arguments: a pointer to data (of type void*), data size (bytes) to specify the data to be sent as soon as the link has been established, and flags.

The CONN library defines two more constructors, which build SOCKET connectors on top of existing SOCK objects: SOCK_CreateConnectorOnTop() and SOCK_CreateConnectorOnTopEx(), the description of which is intentionally omitted here because SOCK is not discussed either. Please refer to the description in the Toolkit code.

File Connector

Constructors defined in:

```
#include <connect/ncbi_file_connector.h>

CONNECTOR file_connector = FILE_CreateConnector("InFile", "OutFile");
```

This connector could be used for both reading and writing files, when input goes from one file and output goes to another file. (This differs from normal file I/O, when a single handle is used to access only one file, but rather resembles data exchange via socket.)

Extended variant of this connector's constructor, FILE_CreateConnectorEx(), takes an additional argument, a pointer to a structure of type SFILE_ConnAttr describing file connector attributes, such as the initial read position to start from in the input file, an open mode for the output file (append eFCM_Append, truncate eFCM_Truncate, or seek eFCM_Seek to start writing at a specified file position), and the position in the output file, where to begin output. The attribute pointer passed as NULL is equivalent to a call to FILE_CreateConnector(), which reads from the very beginning of the input file and entirely overwrites the output file (if any), implicitly using eFCM_Truncate.

Connection-related parameters for higher-level connection protocols

The network information structure (from connect/ncbi_connutil.h) defines parameters of the connection point, where a server is running.

Note: Not all parameters of the structure shown below apply to every network connector.

```
/* Network connection related configurable info struct.
 * ATTENTION: Do NOT fill out this structure (SConnNetInfo) "from scratch"!
 * Instead, use ConnNetInfo_Create() described below to create
 * it, and then fix (hard-code) some fields, if really necessary.
 * NOTE1: Not every field may be fully utilized throughout the library.
 * NOTE2: HTTP passwords can be either clear text or Base64 encoded values
 * enclosed in square brackets [] (which are not Base-64 charset).
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
 * For encoding / decoding, one can use command line open ssl:
 * echo "password|base64value" | openssl enc {-e|-d} -base64
 * or an online tool (search the Web for "base64 online").
 */
typedef struct {
 char client_host[256]; /* effective client hostname ('\0'=def)*/
 EURLScheme scheme; /* only pre-defined types (limited) */
 char user[64]; /* username (if specified) */
 char pass[64]; /* password (if any) */
 char host[256]; /* host to connect to */
 unsigned short port; /* port to connect to, host byte order */
 char path[1024]; /* service: path(e.g. to a CGI script)*/
 char args[1024]; /* service: args(e.g. for a CGI script)*/
 EReqMethod req_method; /* method to use in the request (HTTP) */
 const STimeout* timeout; /* ptr to i/o tmo (infinite if NULL) */
 unsigned short max_try; /* max. # of attempts to connect (>= 1)*/
 char http_proxy_host[256]; /* hostname of HTTP proxy server */
 unsigned short http_proxy_port; /* port # of HTTP proxy server */
 char http_proxy_user[64]; /* http proxy username */
 char http_proxy_pass[64]; /* http proxy password */
 char proxy_host[256]; /* CERN-like (non-transp) f/w proxy srv*/
 EDebugPrintout debug_printout; /* printout some debug info */
 int/*bool*/ stateless; /* to connect in HTTP-like fashion only*/
 int/*bool*/ firewall; /* to use firewall/relay in connects */
 int/*bool*/ lb_disable; /* to disable local load-balancing */
 const char* http_user_header; /* user header to add to HTTP request */
 const char* http_referer; /* default referrer (when not spec'd) */

 /* the following field(s) are for the internal use only -- don't touch! */
 STimeout tmo; /* default storage for finite timeout */
 const char svc[1]; /* service which this info created for */
} SConnNetInfo;
```

Caution: Unlike other "static fields" of this structure, http_user_header (if non-NULL) is assumed to be dynamically allocated on the heap (via a call to malloc, calloc, or a related function, such as strdup).

ConnNetInfo convenience API

Although users can create and fill out this structure via field-by-field assignments, there is, however, a better, easier, much safer, and configurable way (the interface is defined in connect/ncbi_connutil.h) to deal with this structure:

- ConnNetInfo_Create(const char* service)

Create and return a pointer to new SConnNetInfo structure, filled with parameters specific either for a named service or by default (if the service is specified as NULL - most likely the case for ordinary HTTP connections). Parameters for the structure are taken from (in the order of precedence):

- Environment variables of the form <service>_CONN_<name>, where name is the name of the field;
- Service-specific registry section (see below the note about the registry) named [service] using the key CONN_<name>;

- environment variable of the form CONN_<name>
- registry section named [CONN] using name as a key
- default value applied, if none of the above resulted in a successful match

Search for the keys in both environment and registry is not case-sensitive, but the values of the keys are case sensitive (except for enumerated types and boolean values, which can be of any, even mixed, case). Boolean fields accept 1, "ON", "YES", and "TRUE" as true values; all other values are treated as false. In addition to a floating point number treated as a number of seconds, timeout can accept (case-insensitively) keyword "INFINITE".

Note: The first two steps in the above sequence are skipped if the service name is passed as NULL.

Caution: The library can not provide reasonable default values for path and args when the structure is used for HTTP connectors.

- ConnNetInfo_Destroy(SConnNetInfo* info)

Delete and free the info structure via a passed pointer (note that the HTTP user header http_user_header is freed as well).

- ConnNetInfo_SetUserHeader(SConnNetInfo* info, const char* new_user_header)

Set the new HTTP user header (freeing the previous one, if any) by cloning the passed string argument and storing it in the http_user_header field. New_user_header passed as NULL resets the field.

- ConnNetInfo_Clone(SConnNetInfo* info)

Create and return a pointer to a new SConnNetInfo structure, which is an exact copy of the passed structure. This function is recognizes the dynamic nature of the HTTP user header field.

Note about the registry. The registry used by the connect library is separate from the CNcbiRegistry class. To learn more about the difference and how to use both objects together in a single program, please see Using NCBI C and C++ Toolkits Together.

HTTP Connector

Constructors defined in:

```
#include <connect/ncbi_http_connector.h>
```

The simplest form of this connector's constructor takes three parameters:

```
CONNECTOR HTTP_CreateConnector(const SConnNetInfo* net_info,
 const char* user_header,
 THCC_Flags flags);
```

a pointer to the network information structure (can be NULL), a pointer to a custom HTTP tag-value(s) called a user-header, and a bitmask of various flags. The user-header has to be in the form "HTTP-Tag: Tag-value\r\n", or even multiple tag-values delimited and terminated by "\r\n". If specified, the user_header parameter overrides the corresponding field in the passed net_info.

The following fields of SConnNetInfo pertain to the HTTP connector: client_host, host, port, path, args, req_method (can be one of "GET", "POST", and "ANY"), timeout, max_try (analog of maximal try parameter for the socket connector), http_proxy_host, http_proxy_port, and

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

debug_printout (values are "NONE" to disable any trace printout of the connection data, "SOME" to enable printing of SConnNetInfo structure before each connection attempt, and "DATA" to print both headers and data of the HTTP packets in addition to dumps of SConnNetInfo structures). Values of other fields are ignored.

HTTP connector's flags

Argument flags in the HTTP connector's constructor is a bitwise OR of the following values:

- fHTTP_AutoReconnect Allow multiple request/reply HTTP transactions. (Otherwise, by default, only one request/reply is allowed.)
- fHTTP_SureFlush Always flush a request (may consist solely of HTTP header with no body at all) down to the HTTP server before preforming any read or close operations.
- fHTTP_KeepHeader By default, the HTTP connection sorts out the HTTP header and parses HTTP errors (if any). Thus, reading normally from the connection returns data from the HTTP body only. The flag disables this feature, and the HTTP header is not parsed but instead is passed "as is" to the application on a call to CONN_Read().
- fHTTP_UrlDecodeInput Decode input data passed in HTTP body from the HTTP server.
- fHTTP_UrlEncodeOutput Encode output data passed in the HTTP body to the HTTP server.
- fHTTP_UrlCodec Perform both decoding and encoding (fHTTP_UrlDecodeInput | fHTTP_UrlEncodeOutput).
- fHTTP_UrlEncodeArgs Encode URL if it contains special characters such as "+". By default, the arguments are passed "as is" (exactly as taken from SConnNetInfo).
- fHTTP_DropUnread Drop unread data, which might exist in connection, before making another request/reply HTTP shot. Normally, the connection first tries to read out the data from the HTTP server entirely, until EOF, and store them in the internal buffer, even if either application did not request the data for reading, or the data were read only partially, so that the next read operation will see the data.
- fHTTP_NoUpread Do not attempt to empty incoming data channel into a temporary intermediate buffer while writing to the outgoing data channel. By default, writing always makes checks that incoming data are available for reading, and those data are extracted and stored in the buffer. This approach avoids I/O deadlock, when writing creates a backward stream of data, which, if unread, blocks the connection entirely.
- fHTTP_Flushable By default all data written to the connection are kept until read begins (even though Flush() might have been called in between the writes); with this flag set, Flush() will result the data to be actually sent to server side, so the following write will form new request, and not get added to the previous one.
- fHTTP_InsecureRedirect For security reasons the following redirects comprise security risk and, thus, are prohibited: switching from https to http, and re-posting data (regardless of the transport, either http or https); this flag allows such redirects (if needed) to be honored.
- fHTTP_NoAutoRetry Do not attempt any auto-retries in case of failing connections (this flag effectively means having SConnNetInfo::max_try set to 1).
- fHTTP_DetachableTunnel SOCK_Close() won't close the OS handle.

The HTTP connection will be established using the following URL: http://host:port/path?args

Note that path has to have a leading slash "/" as the first character, that is, only "http://" and "?" are added by the connector; all other characters appear exactly as specified (but maybe encoded with fHTTP_UrlEncodeArgs). The question mark does not appear if the URL has no arguments.

A more elaborate form of the HTTP connector's constructor has the following prototype:

```
typedef int/*bool*/ (*FHTTP_ParseHeader)
(const char* http_header, /* HTTP header to parse, '\0'-terminated */
 void* user_data, /* supplemental user data */
 int server_error /* != 0 if HTTP error */
 );

typedef int/*bool*/ (*FHTTP_Adjust)
(SConnNetInfo* net_info, /* net_info to adjust (in place) */
 void* user_data, /* supplemental user data */
 unsigned int failure_count /* how many failures since open */
 );

typedef void (*FHTTP_Cleanup)
(void* user_data /* supplemental user data for cleanup */
 );

CONNECTOR HTTP_CreateConnectorEx
(const SConnNetInfo* net_info,
 THTTP_Flags flags,
 FHTTP_ParseHeader parse_header, /* may be NULL, then no addtl. parsing */
 void* user_data, /* user data for HTTP callbacks (CBs) */
 FHTTP_Adjust adjust, /* may be NULL, then no adjustments */
 FHTTP_Cleanup cleanup /* may be NULL, then no cleanup */
 );
```

This form is assumed to be used rarely by the users directly, but it provides rich access to the internal management of HTTP connections.

The first two arguments are identical to their counterparts in the arguments number one and three of HTTP_CreateConnector(). The HTTP user header field (if any) is taken directly from the http_user_header field of SConnNetInfo, a pointer to which is passed as net_info (which in turn can be passed as NULL, meaning to use the environment, registry, and defaults as described above).

The third parameter specifies a callback to be activated to parse the HTTP reply header (passed as a single string, with CR-LF (carriage return/line feed) characters incorporated to divide header lines). This callback also gets some custom data user_data as supplied in the fourth argument of the connector's constructor and a boolean value of true if the parsed response code from the server was not okay. The callback can return false (zero), which is considered the same way as having an error from the HTTP server. However, the pre-parsed error condition (passed in server_error) is retained, even if the return value of the callback is true, i.e. the callback is unable to "fix" the error code from the server. This callback is **not called** if fHTTP_KeepHeader is set in flags.

The fifth argument is a callback, which is in control when an attempt to connect to the HTTP server has failed. On entry, this callback has current SConnNetInfo, which is requested for an

adjusted in an attempt that the connection to the HTTP server will finally succeed. That is, the callback can change anything in the info structure, and the modified structure will be kept for all further connection attempts, until changed by this callback again. The number (starting from 1) of successive failed attempts is given in the argument of the last callback. The callback return value true (non-zero) means a successful adjustment. The return value false (zero) stops connection attempts and returns an error to the application.

When the connector is being destroyed, the custom object user_data can be destroyed in the callback, specified as the last argument of the extended constructor.

Note: Any callback may be specified as NULL, which means that no action is foreseen by the application, and default behavior occurs.

Service Connector

Constructors defined in:

```
#include <connect/ncbi_service_connector.h>
```

This is the most complex connector in the library. It can initiate data exchange between an application and a named NCBI service, and the data link can be either wrapped in HTTP or be just a byte-stream (similar to a socket). In fact, this connector sits on top of either HTTP or SOCKET connectors.

The library provides two forms of the connector's constructor:

```
SERVICE_CreateConnector(const char* service_name);
SERVICE_CreateConnectorEx(/* The registered name of an NCBI service */
 const char* service_name,
 /* Accepted server types, bitmask */
 TSERV_Type types,
 /* Connection parameters */
 const SConnNetInfo* net_info,
 /* Additional set of parameters, may be NULL */
 const SSERVICE_Extra* params);
```

The first form is equivalent to SERVICE_CreateConnectorEx(service_name, fSERV_Any, 0, 0). A named NCBI service is a CGI program or a stand-alone server (can be one of two supported types), which runs at the NCBI site and is accessible by the outside world. A special dispatcher (which runs on the NCBI Web servers) performs automatic switching to the appropriate server without having the client to know, a priori, the connection point, i.e. the client just uses the main entry gate of the NCBI Web (usually, www.ncbi.nlm.nih.gov) with a request to have a service "service_name", then, depending on the service availability, the request will be either honored (by switching and routing the client to the machine actually running the server: clicking on the previous link should bring you to a page containing "name=value" message, obtained from the special bouncing service as a result of the form submission), rejected, or declined. To the client, the entire process of dispatching is completely transparent (for example, try clicking several times on either of the latter two links and see that the error replies are actually sent from different hosts, and the successful processing of the first link is done by one of several hosts running the bouncing service).

Note: Services can be redirected.

The Dispatching Protocol per se is implemented on top of HTTP protocol and is parsed by a CGI program dispd.cgi (or another dispatching CGI), which is available on the NCBI Web. On every server running the named services, another program, called the load-balancing daemon (lbsmd), is executing. This daemon supports having the same service running on different machines and provides a choice of the one machine that is less loaded. When dispd.cgi receives a request for a named service, it first consults the load-balancing table, which is broadcasted by each load-balancing daemon and populated in a network-wide form on each server. When the corresponding server is found, the client request can be passed, or a dedicated connection to the server can be established. The dispatching is made in such a way that it can be also used directly from most Internet browsers.

The named service facility uses the following distinction of server types:

- HTTP servers, which are usually CGI programs:
  - HTTP_GET servers are those accepting requests only using the HTTP GET method.
  - HTTP_POST servers are those accepting requests only using the HTTP POST method.
  - HTTP servers are those accepting both of either GET or POST methods.
- NCBID servers are those run by a special CGI engine, called ncbid.cgi, a configurable program (now integrated within dispd.cgi itself) that can convert byte-stream output from another program (server) started by the request from a dispatcher, to an HTTP-compliant reply (that is, a packet having both HTTP header and body, and suitable, for example, for Web browsers).
- STANDALONE servers, similar to mailing daemons, are those listening to the network, on their own, for incoming connections.
- FIREWALL servers are the special pseudo-servers, not existing in reality, but that are created and used internally by the dispatcher software to indicate that only a firewall connection mode can be used to access the requested service.
- DNS servers are beyond the scope of this document because they are used to declare domain names, which are used internally at the NCBI site to help load-balancing based on DNS lookup (see here).

A formal description of these types is given in connect/ncbi_server_info.h:

```
/* Server types
 */
typedef enum {
 fSERV_Ncbid = 0x01,
 fSERV_Standalone = 0x02,
 fSERV_HttpGet = 0x04,
 fSERV_HttpPost = 0x08,
 fSERV_Http = fSERV_HttpGet | fSERV_HttpPost,
 fSERV_Firewall = 0x10,
 fSERV_Dns = 0x20
} ESERV_Type;

#define fSERV_Any 0
#define fSERV_StatelessOnly 0x80
typedef unsigned TSERV_Type; /* bit-wise OR of "ESERV_Type" flags */
```

The bitwise OR of the ESERV_Type flags can be used to restrict the search for the servers, matching the requested service name. These flags passed as argument types are used by the dispatcher when figuring out which server is acceptable for the client. A special value 0 (or, better fSERV_Any) states no such preference whatsoever. A special bit-value fSERV_StatelessOnly set, together with other bits or just alone, specifies that the servers should be of stateless (HTTP-like) type only, and it is the client who is responsible for keeping track of the logical sequence of the transactions.

The following code fragment establishes a service connection to the named service "io_bounce", using only stateless servers:

```
CONNECTOR c;
CONN conn;
if(!(c = SERVICE_CreateConnectorEx("io_bounce", fSERV_StatelessOnly, 0, 0)))
{
 fprintf(stderr, "No such service available");
} else if (CONN_Create(c, &conn) != eIO_Success) {
 fprintf(stderr, "Failed to create connection");
} else {
 static const char buffer[] = "Data to pass to the server";
 size_t n_written;
 CONN_Write(conn, buffer, sizeof(buffer) - 1, &n_written);
 ...
}
```

The real type of the data channel can be obtained via a call to CONN_GetType().

Note: In the above example, the client has no assumption of how the data actually passed to the server. The server could be of any type in principle, even a stand-alone server, which was used in the request/reply mode of one-shot transactions. If necessary, such wrapping would have been made by the dispatching facility as well.

The next-to-last parameter of the extended constructor is the network info, described in the section devoted to the HTTP connector. The service connector uses all fields of this structure, except for http_user_header, and the following assumptions apply:

- path specifies the dispatcher program (defaulted to dispd.cgi)
- args specifies parameters for the requested service (this is service specific, no defaults)
- stateless is used to set the fSERV_StatelessOnly flag in the server type bitmask, if it was not set there already (which is convenient for modifying the dispatch by using environment and/or registry, if the flag is not set, yet allows hardcoding of the flag at compile time by setting it in the constructor's types argument explicitly)
- lb_disable set to true (non-zero) means to always use the remote dispatcher (via network connection), even if locally running load-balancing daemon is available (by default, the local load-balancing deamon is consulted first to resolve the name of the service)
- firewall set to true (non-zero) disables the direct connection to the service; instead,
  - a connection to a proxy firewall daemon (fwdaemon), running at the NCBI site, is initiated to pass the data in stream mode;
  - or data get relayed via the dispatcher, if the stateless server is used
- http_user_header merged not to conflict with special dispatcher parameter.

As with the HTTP connector, if the network information structure is specified as NULL, the default values are obtained as described above, as with the call to ConnNetInfo_Create (service_name).

Normally, the last parameter of SERVICE_CreateConnectorEx() is left NULL, which sets all additional parameters to their default values. Among others, this includes the default procedure of choosing an appropriate server when the connector is looking for a mapping of the service name into a server address. To see how this parameter can be used to change the mapping procedure, please see the service mapping API section. The library provides an additional interface to the service mapper, which can be found in connect/ncbi_service.h.

Note: Requesting fSERV_Firewall in the types parameter effectively selects the firewall mode regardless of the network parameters, loaded via the SConnNetInfo structure.

Service Redirection

Services can be redirected without changing any code - for example, to test production code with a test service, or for debugging. Services are redirected using the <service>_CONN_SERVICE_NAME environment variable or the [<service>] CONN_SERVICE_NAME registry entry (see the connection library configuration section). The client application will use the original service name, but the connection will actually be made to the redirected-to service.

Debugging Tools and Troubleshooting

Each connector (except for the FILE connector) provides a means to view data flow in the connection. In case of the SOCKET connector, debugging information can be turned on by the last argument in SOCK_CreateConnectorEx() or by using the global routine SOCK_SetDataLoggingAPI() (declared in connect/ncbi_socket.h)

Note: In the latter case, every socket (including sockets implicitly used by other connectors such as HTTP or SERVICE) will generate debug printouts.

In case of HTTP or SERVICE connectors, which use SConnNetInfo, debugging can be activated directly from the environment by setting CONN_DEBUG_PRINTOUT to TRUE or SOME. Similarly, a registry key DEBUG_PRINTOUT with a value of either TRUE or SOME found in the section [CONN] will have the same effect: it turns on logging of the connection parameters each time the link is established. When set to ALL, this variable (or key) also turns on debugging output on all underlying sockets ever created during the life of the connection. The value FALSE (default) turns debugging printouts off. Moreover, for the SERVICE connector, the debugging output option can be set on a per-service basis using <service>_CONN_DEBUG_PRINTOUT environment variables or individual registry sections [<service>] and the key CONN_DEBUG_PRINTOUT in them.

Note: Debugging printouts can only be controlled in a described way via environment or registry if and only if SConnNetInfo is always created with the use of convenience routines.

Debugging output is always sent to the same destination, the CORE log file, which is a C object shared between both C and C++ Toolkits. As said previously, the logger is an abstract object, i.e. it is empty and cannot produce any output if not populated accordingly. The library defines a few calls gathered in connect/ncbi_util.h, which allow the logger to output via the FILE file pointer, such as stderr: CORE_SetLOGFILE() for example, as shown in test_ncbi_service_connector.c, or to be a regular file on disk. Moreover, both Toolkits define

interfaces to deal with registries, loggers, and locks that use native objects of each toolkit and use them as replacements for the objects of the corresponding abstract layers.

There is a common problem that has been reported several times and actually concerns network configuration rather than representing a malfunction in the library. If a test program, which connects to a named NCBI service, is not getting anything back from the NCBI site, one first has to check whether there is a transparent proxying/caching between the host and NCBI. Because the service dispatching is implemented on top of the ordinary HTTP protocol, the transparent proxying may latch unsuccessful service searches (which can happen and may not indicate a real problem) as error responses from the NCBI server. Afterwards, instead of actually connecting to NCBI, the proxy returns those cached errors (or sometimes just an empty document), which breaks the service dispatcher code. In most cases, there are configurable ways to exclude certain URLs from proxying and caching, and they are subject for discussion with a local network administrator.

Here is another tip: Make sure that all custom HTTP header tags (as passed, for example, in the SConnNetInfo::user_header field) have "\r\n" as tag separators (including the last tag). Many proxy servers (including transparent proxies, of which the user may not even be aware) are known to be sensitive to whether each and every HTTP tag is closed by "\r\n" (and not by a single "\n" character). Otherwise, the HTTP packet may be treated as a defective one and can be discarded.

Additional discussion on parameters of the service dispatcher as well as the trouble shooting tips can be found here.

C++ Connection Streams

Using connections and connectors (via the entirely procedural approach) in C++ programs overkills the power of the language. Therefore, we provide C++ users with the stream classes, all derived from a standard iostream class, and as a result, these can be used with all common stream I/O operators, manipulators, etc.

The declarations of the stream's constructors can be found in connect/ncbi_conn_stream.hpp. We tried to keep the same number and order of the constructor's parameters, as they appear in the corresponding connector's constructors in C.

The code below is a C++ style example from the previous section devoted to the service connector:

```
#include <connect/ncbi_conn_stream.hpp>
try {
 CConn_ServiceStream
 ios("io_bounce", fSERV_StatelessOnly, 0);
 ios << "Data to be passed to the server";
} STD_CATCH_ALL ("Connection problem");
```

Note: The stream constructor may show an exception if, for instance, the requested service is not found, or some other kind of problem arose. To see the actual reason, we used a standard toolkit macro STD_CATCH_ALL(), which prints the message and problem description into the log file (cerr, by default).

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

Service mapping API

The API defined in connect/ncbi_service.h is designed to map the required service name into the server address. Internally, the mapping is done either directly or indirectly by means of the load-balancing daemon, running at the NCBI site. For the client, the mapping is seen as reading from an iterator created by a call to SERV_Open(), similar to the following fragment (for more examples, please refer to the test program test_ncbi_disp.c):

```
#include <connect/ncbi_service.h>
SERV_ITER iter = SERV_Open("my_service", fSERV_Any, SERV_ANYHOST, 0);
int n = 0;
if (iter != 0) {
 SSERV_Info * info = SERV_GetNextInfo(iter);
 while (info != 0) {
 char* str = SERV_WriteInfo(info);
 printf("Server = `%s'\n", str);
 free(str);
 n++;
 }
 SERV_Close(iter);
}
if (!iter || !n)
 printf("Service not found\n");
```

Note: Services can be redirected.

Note: A non-NULL iterator returned from SERV_Open() **does not** yet guarantee that the service is available, whereas the NULL iterator definitely means that the service does not exist.

As shown in the above example, a loop over reading from the iterator results in the sequence of successive structures (none of which is to be freed by the program!) that along with the conversion functions SERV_ReadInfo(), SERV_WriteInfo() and others are defined in connect/ncbi_server_info.h. Structure SSERV_Info describes a server that implements the requested service. NULL gets returned when no more servers (if any) could be found. The iterator returns servers in the order that the load-balancing algorithm arranges them. Each server has a rating, and the larger the rating the better the chance for the server to be coming out first (but not necessarily in the order of the rates).

Note: Servers returned from the iterator are all of the requested type, with only one exception: they can include servers of type fSERV_Firewall, even if this type has not been explicitly requested. Therefore, the application must sort these servers out. But if fSERV_Firewall is set in the types, then the search is done for whichever else types are requested, and with the assumption that the client has chosen a firewall connection mode, regardless of the network parameters supplied in SConnNetInfo or read out from either the registry or environment.

Note: A search for servers of type fSERV_Dns is not inclusive with fSERV_Any specified as a server type. That is, servers of type DNS are only returned if specifically requested in the server mask at the time the iterator was opened.

There is a simplified version of SERV_Open(), called SERV_OpenSimple(), as well as an advanced version, called SERV_OpenEx(). The former takes only one argument, the service name. The latter takes two more arguments, which describe the set of servers **not** to be returned from the iterator (server descriptors that to be excluded).

There is also an advanced version of SERV_GetNextInfo(), called SERV_GetNextInfoEx(), that, via its second argument, provides the ability to get many host parameters, among which is so-called host environment; a "\0"-terminated string, consisting of a set of lines separated by "\n" characters and specified in the configuration file of the load-balancing daemon of the host, where the returned server has been found. The typical line within the set has a form "name=value" and resembles very much the shell environment, where its name comes after. The host environment could be very handy for passing additional information to applications if the host has some limitations or requires special handling, should the server be selected and used on this host. The example below shall give an idea. At the time of writing, getting the host information is only implemented when the service is obtained via direct access to the load-balancing daemon. Unlike returned server descriptors, the returned host information handle is not a constant object and must be explicitly freed by the application when no longer needed. All operations (getter methods) that are defined on the host information handle are declared in connect/ncbi_host_info.h. If the server descriptor was obtained using dispatching CGI (indirect dispatching, see below), then the host information handle is always returned as NULL (no host information available).

The back end of the service mapping API is split into two independent parts: *direct* access to LBSMD, if the one is both available on the current host and is not disabled by parameter lb_disable at the iterator opening. If LBSMD is either unavailable or disabled, the second (*indirect*) part of the back-end API is used, which involves a connection to the dispatching CGI, which in turn connects to LBSMD to carry out the request. An attempt to use the CGI is done only if the net_info argument is provided as non-NULL in the calls to SERV_Open() or SERV_OpenEx().

Note: In a call to SERV_OpenSimple(), net_info gets created internally before an upcall to SERV_Open() and thus CGI dispatching is likely to happen, unless either net_info could not be constructed from the environment, or the environment variable CONN_LB_DISABLE (or a service-specific one, or either of the corresponding registry keys) is set to TRUE.

Note: In the above conditions, the network service name resolution is also undertaken if the service name could not be resolved (because the service could not be found or because of some other error) with the use of locally running LBSMD.

The following code example uses both a service connector and the service mapping API to access certain services using an alternate way (other than the connector's default) of choosing appropriate servers. By default, the service connector opens an internal service iterator and then tries to connect to the next server, which SERV_GetNextInfo() returns when given the iterator. That is, the server with a higher rate is tried first. If user provides a pointer to structure SSERVICE_Extra as the last parameter of the connector's constructor, then the user-supplied routine (if any) can be called instead to obtain the next server. The routine is also given a supplemental custom argument data taken from SSERVICE_Extra. The (intentionally simplified) example below tries to create a connector to an imaginary service "my_service" with the restriction that the server has to additionally have a certain (user-specified) database present. The database name is taken from the LBSMD host environment keyed by "my_service_env", the first word of which is assumed to be the required database name.

```
#include <connect/ncbi_service_connector.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#define ENV_DB_KEY "my_service_env="
/* This routine gets called when connector is about to be destructed.
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
 */
static void s_CleanupData(void* data)
{
 free(data); /* we kept database name there */
}
/* This routine gets called on each internal close of the connector
 * (which may be followed by a subsequent internal open).
 */
static void s_Reset(void* data)
{
/* just see that reset happens by printing DB name */
 printf("Connection reset, DB: %s\n", (char*) data);
}
/* 'Data' is what we supplied among extra-parameters in connector's
 * constructor.
 * 'Iter' is an internal service iterator used by service connector;
 * it must remain open.
 */
static const SSERV_Info* s_GetNextInfo(void* data, SERV_ITER iter)
{
 const char* db_name = (const char*) data;
 size_t len = strlen(db_name);
 SSERV_Info* info;
 HOST_INFO hinfo;
 int reset = 0;
 for (;;) {
 while ((info = SERV_GetNextInfoEx(iter, &hinfo)) != 0) {
 const char* env = HINFO_Environment(hinfo);
 const char* c;
 for (c = env; c; c = strchr(c, '\n')) {
 if (strncmp(c == env ? c : ++c, ENV_DB_KEY,
 sizeof(ENV_DB_KEY)-1) == 0) {
 /* Our keyword has been detected in environment */
 /* for this host */
 c += sizeof(ENV_DB_KEY) - 1;
 while (*c && isspace(*c))
 c++;
 if (strncmp(c, db_name, len) == 0 && !isalnum(c + len)) {
 /* Database name match */
 free(hinfo); /* must be freed explicitly */
 return info;
 }
 }
 }
 if (hinfo)
 free(hinfo); /* must be freed explicitly */
 }
 if (reset)
 break; /* coming to reset 2 times in a row means no server fit */
 SERV_Reset(iter);
 reset = 1;
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
 }
 return 0; /* no match found */
}
int main(int argc, char* argv[])
{
 char* db_name = strdup(argv[1]);
 SSERVICE_Extra params;
 CONNECTOR c;
 CONN conn;
 memset(&params, 0, sizeof(params));
 params.data = db_name; /* custom data, anything */
 params.reset = s_Reset; /* reset routine, may be NULL */
 params.cleanup = s_CleanupData; /* cleanup routine, may be NULL*/
 params.get_next_info = s_GetNextInfo; /* custom iterator routine */
 if (!(c = SERVICE_CreateConnectorEx("my_service",
 fSERV_Any, NULL, &params))) {
 fprintf(stderr, "Cannot create connector");
 exit(1);
 }
 if (CONN_Create(c, &conn) != eIO_Success) {
 fprintf(stderr, "Cannot create connection");
 exit(1);
 }
 /* Now we can use CONN_Read(),CONN_Write() etc to deal with
 * connection, and we are assured that the connection is made
 * only to the server on such a host which has "db_name"
 * specified in configuration file of LBSMD.
 */
 ...
 CONN_Close(conn);
 /* this also calls cleanup of user data provided in params */
 return 0;
}
```

Note: No network (indirect) mapping occurs in the above example because net_info is passed as NULL to the connector's constructor.

Local specification of the LBSM table

The LBSM table can also be specified locally, in config file and/or environment variables.

Service lookup process now involves looking up through the following sources, in this order:

- Local environment/registry settings;
- LBSM table (only in-house; this step does not exist in the outside builds);
- Network dispatcher.

Only one source containing the information about the service is used; the next source is only tried if the previous one did not yield in any servers (for the service).

Step 1 is disabled by default, to enable it set CONN_LOCAL_ENABLE environment variable to "1" (or "ON, or "YES", or "TRUE") or add LOCAL_ENABLE=1 to [CONN] section in .ini file. Steps 2 and 3 are enabled by default. To disable them use CONN_LBSMD_DISABLE

and/or CONN_DISPD_DISABLE set to "1" in the environment or LBSMD_DISABLE=1 and/or DISPD_DISABLE=1 under the section "[CONN]" in the registry, respectively.

Note: Alternatively, and for the sake of backward compatibility with older application, the use of local LBSM table can be controlled by CONN_LB_DISABLE={0,1} in the environment or LB_DISABLE={0,1} in the "[CONN]" section of the registry, or individually on a per service basis:

For a service "ANAME", ANAME_CONN_LB_DISABLE={0,1} in the environment, or CONN_LB_DISABLE={0,1} in the section "[ANAME]" in the registry (to affect setting of this particular service, and no others).

The local environment / registry settings for service "ANAME" are screened in the following order:

- A value of environment variable "ANAME_CONN_LOCAL_SERVER_n";
- A value of registry key "CONN_LOCAL_SERVER_n" in the registry section "[ANAME]"

Note that service names are not case sensitive, yet the environment variables are looked up all capitalized.

An entry found in the environment takes precedence over an entry (for the same "n") found in the registry. "n" counts from 0 to 100, and need not to be sequential.

All service entries can be (optionally) grouped together in a list as a value of either:

- Environment variable "CONN_LOCAL_SERVICES", or
- Registry key "LOCAL_SERVICES" under the registry section "[CONN]".

The list of local services is only used in cases of wildcard searches, or in cases of reverse lookups, and is never consulted in regular cases of forward searches by a complete service name.

Examples:

1. In .ini file

```
[CONN]
LOCAL_ENABLE=yes
LOCAL_SERVICES="MSSQL10 MSSQL14 MSSQL15 MSSQL16 MSSQL17"

[MSSQL10]
CONN_LOCAL_SERVER_6="DNS mssql10:1433 L=yes"

[MSSQL15]
CONN_LOCAL_SERVER_9="DNS mssql15:1433 L=yes"
```

Note that entries for MSSQL14, 16, and 17 are not shown, and they are not required (inexistent definitions for declared services are simply ignored).

2. In environment set the following variables (equivalent to the .ini fragment above but having a higher precedence):

```
CONN_LOCAL_ENABLE=yes
CONN_LOCAL_SERVICES="MSSQL10 MSSQL14 MSSQL15 MSSQL16 MSSQL17"
```

The NCBI C++ Toolkit Book

The NCBI C++ Toolkit Book

```
MSSQL10_CONN_LOCAL_SERVER_6="DNS mssql10:1433 L=yes"
MSSQL15_CONN_LOCAL_SERVER_9="DNS mssql15:1433 L=yes"
```

You can also look at the detailed description of LBSMD and a sample configuration file.

Threaded Server Support

This library also provides CThreadedServer, an abstract base class for multithreaded network servers. Here is a demonstration of its use. Note that this class **does not** support multiplexing traffic over a single TCP connection; rather, each thread has an individual TCP connection created when a client connects to the server's listening port.