LibOutreach

Extends the Power of

High Performance Computing to MS Excel

CyberXpert, Inc.

Printed in the United States of America

Table of Contents

1. Introduction

Microsoft® Excel has been one of the primary analytical tools for analysts in various fields. Excel's evolution has been in tandem with the desktop computers' processing power and operating system capabilities. A decade ago, desktops did not have multi-processing and networking capabilities. Consequently, applications involving heavy-duty calculations, significant database operations and inter-process communications have been developed to run in Unix Server environments. Now, because of high-speed processors and Windows NT® /Windows® 2000 platforms, the desktops have significant processing power. However, migrating server applications to Excel is not a trivial task. It takes considerable amount of development time and entails significant costs. ***LibOutreach*** offers an ideal solution by providing the ability to access and execute remote server (UNIX as well as Windows NT) applications in Excel at a considerably lower cost to the organization.

LibOutreach brings the processing power of remote servers to Excel applications. It provides a seamless interface in Excel for remote function invocation. *LibOutreach* manages all the data communications involved and keeps all complex issues transparent to the desktop users. *LibOutreach* is easy to install and use. With minimal efforts from C++/VBA developers, all the power and potential of remote servers are unlocked.

Highlights

LibOutreach enhances spreadsheet applications by providing additional capabilities. Some of these capabilities are:

- Remote function execution within Excel
- Sharing of VBA modules by spreadsheet applications
- Access to multiple servers and their libraries
- Sequential, Parallel and Stream modes for efficient execution

Using these features, *LibOutreach* frees the developers/users from the following:

- Version control issues due to duplicate function definitions
- Developing routines for client-server communication
- Software distribution, license control and maintenance

2. Installation

LibOutreach is a collection of Microsoft Excel Add-Ins, DLLs (Dynamic Link Libraries) and Server programs. LibOutreach has two components, a desktop component ***LibClient*** and a server component ***LibServer***. LibClient needs to be installed on all users' desktop and LibServer needs to be installed on all service providing remote servers.

LibClient Installation

LibClient is compatible with Microsoft® Excel 97, Excel 2000, Excel XP and Excel 2003. MS Excel must be installed prior to LibClient installation. Please click the link *LibOutreach–Microsoft Excel*, and run the self extraction program. Installation brings up a dialog and asks for a directory in which *LibClient* is to be installed. The default directory is "*C:\Program Files\CyberXpert\LibOutreach_Client*".

The Excel component of *LibClient* is implemented as an Excel Add-In, ***LibClient.xla.*** This add-in gets added during the final stage of installation. If you encounter any error during the final step, please manually add it following these steps:

From Excel command menu select,

Tools ->
Add-Ins …
(from dialog box) Browse…
open < installation path>\Addins\LibClient.xla.

If you are prompted with "create a local copy?" by the operating system, please say NO[1]. A new tool bar shown below gets added to your MS Excel toolbars.

Please close all MS Excel applications and start afresh again[2]. You will see this tool bar again. Your have successfully installed LibClient.

LibServer Installation

LibServer runs in Solaris and Linux operating systems. Please download LibOutreach_SunOS.tar for Solaris or LibOutreach_Linux.tar for Linux, and untar to a target directory. Once the directory has been copied, edit the file **LibServer.sh** and change the resource values to reflect your system configuration. The *LibServer.sh* configuration is detailed in the "LibServer" section.

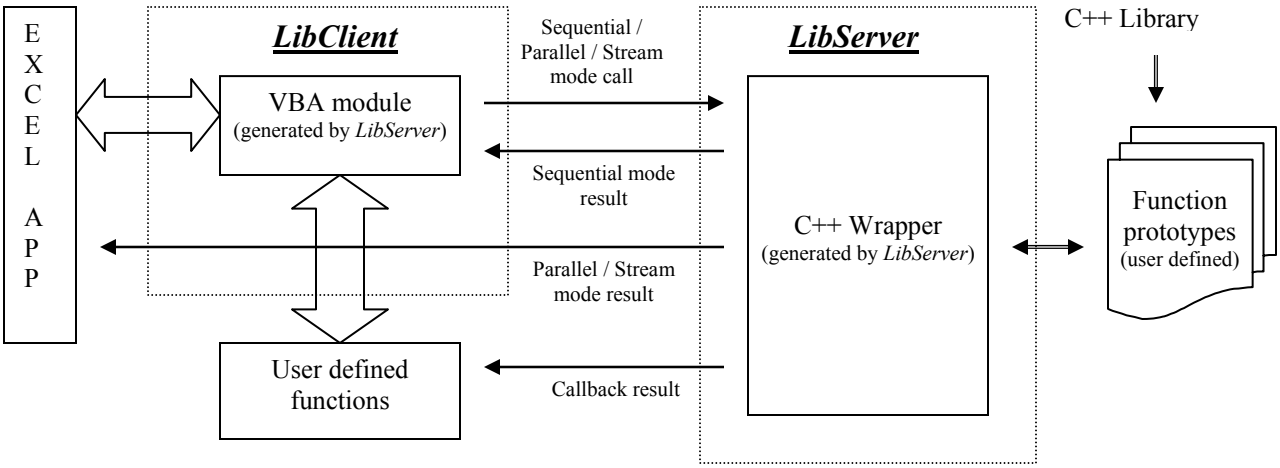
[1] Operating system copies only the libraries and not all the required files to a local directory.

[2] Operating system saves the installed configuration.

3. Architecure

LibOutreach enables Excel applications to launch remote server library functions and receive their results. LibOutreach has two components: *LibServer,* a program runs at server and *LibClient,* a set of libraries and Excel-addins, runs as a part of MSExcel. One instance of *LibServer* runs on a remote server and communicates with *LibClients*. LibServer is provided with information about libraries, library functions and their execution modes, in a special file called ***prototype*** file. A *LibServer* can handle multiple *prototype* files. Excel applications can communicate with many *LibServer*s, running at multiple locations. Thus, *LibOutreach* provides scalability and reusability of library resources. The remote functions can be called from workbooks or VBA routines.

LibOutreach



LibOutreach provides three types of function execution methodologies. The first type is **'Sequential mode'**, a function is called, and further execution waits till the called function returns its result. The second type is **'Parallel mode'**, a thread is created for the called function and its identity is sent back immediately to the calling cell; further execution of other cell values continue. Once the called thread execution is over, the result is sent to the appropriate cell which called it. The third type is **'Stream mode'**, a thread is created for the called function and the thread posts the initial results and further updates to the calling function. If execution of a function/thread terminates abruptly without returning a result, #LIB_ERROR message will be displayed in the cell which initiated the call.

The following three messages can appear in a cell with remote function call.
#THREAD (*thread Identifier*) - A 'Parallel mode' thread has been created and the cell is waiting for its result.
#STREAM (*thread Identifier*) - A 'Stream mode' thread has been created and the cell is waiting for its first result.
#LIB_ERROR - Function/Thread terminated abruptly due to library routine errors or communication problems.

4. LibServer

LibServer can be started as part of system boot sequence or at the command prompt. The executable and startup script files are in <*LibOutreach_installed_directory*>/ directory. The startup script file is "LibServer.sh". This script file can be copied to other locations; can be edited to reconfigure *LibServer*'s parameters. LibServer can be stared by running LibServer.sh script.

% LibServer.sh [test | start | stop | restart]

test: This option is used for validating *LibServer*'s configuration and users' prototype file descriptions. If there is any modification to *LibServer* configuration or prototype file, it is advisable to use this option before starting *LibServer*. In test mode, syntax errors and configuration errors are reported.
start: Starts the *LibServer* process. The server is ready to accept requests from LibClients.
stop: Stops the *LibServer*.
restart: Stops and restarts *LibServer*.

Once *LibServer* is installed in a machine, multiple instances of it can be run. A separate startup script file is required for each instance. Environment variables defined at the beginning of the script file need to be set without conflicting with other script files. The variables are:

LIBSERVER_BIN_DIR

Specifies the directory where the *LibServer* is installed. It defaults to '.', the current directory.

LIBSERVER_LOG_DIR

Specifies the directory where *LibServer* can create temporary files. It defaults to "./log/".

LIBSERVER_PORT

Port at which *LibServer* waits for *LibClient*'s request. It defaults to 9999.

PROTOTYPE_FILES

One or more files having prototype information. The file names are separated by spaces.

EXECUTION_MODEL

LibOutreach supports two execution models namely *PROCESS_POOL* and *PROCESS_PER_USER*.

PROCESS_POOL: In this model, any process will pickup requests from *LibClients*. This model is useful when the server serves many clients and there are no 'global' variables in the library.

PROCESS_PER_USER: In this model, a new process is created at the first request from a *LibClient* and this process is dedicated to serve all future requests from that *LibClient*. This model can be used when a library has global variables or user would like to share values between sessions.

MAX_THREADS_PER_PROCESS

LibServer creates threads in order to speedup processing. A *LibServer* stops creating threads once the limit set by this parameter is reached. Multi-threading can be disabled by setting this parameter value to 0. If a server library is not multi-threaded, this option must be set to 0.

MAX_PROCESSES

This value can be set to limit the number of processes created by a *LibServer* to achieve optimum performance. Assigning a value of 1 makes the *LibServer* to run as a single process. Single process mode with disabled multi-threaded configuration will simplify debugging user libraries and *LibServer* integration.

Prototype File

Prototype files have information about shared libraries and the useful functions defined in that library. A prototype file has multiple sections for describing ***User defined types, Libraries***, ***Functions, Threads, Streams*** and ***Modules***. These sections are optional and can be written in any order. The sections *Libraries*, *Functions*, *Threads, Streams* and *Modules* provide information to Excel applications about the availability of library routines and their execution modes. The section *Modules* is not a part of the library, but specifies the file names of VBA routines used in applications.

```
Types {
        User defined type specifications
}

Libraries {
        Full path information about the C/C++ libraries used by LibServer
 }

Functions {
        Functions' prototype available from the C/C++ library that run in 'Sequential mode'
}

Threads {
        Functions' prototype available from C/C++ library that run in 'Parallel mode'
}

Streams {
        Functions' prototype available from C/C++ library that run in 'Stream mode'
}

Modules {
```

```
        File names of Excel-VBA routines
}
```

The primary objective of a *LibServer* is to provide interface to library routines. A sample description of the prototype file of a *LibServer* is given below:

```
Types {
        complex (int  real, int  imaginary) ;
        Matrix (double *vector,  int  columns, int  rows) ;
}

Libraries {
        /opt/LibOutreach/libraries/libcomplex.so ;
        /opt/LibOutreach/libraries/libprime.so ;
}

Functions {
        int find_next_prime (int begin) ;
        char *reverse (char *string) ;
        double average (double *array);
        int *prime_numbers (int begin, int count);

        //  comment
        Matrix inverse_matrix  (const Matrix  input_matrix ) ;
}

Threads {
        int  num_of_prime (int begin, int end);
}

Streams {
        int random_number (ExcelChannel<int>,  int seed);
}
```

The above prototype description specifies the availability of two shared libraries, libcomplex and libprime. The string "//"is used for commenting the rest of the line. Each block can have multiple statements delimited with ";".

Parameter Types

LibOutreach supports two types of parameter passing convention, pass by reference and pass by value.

Pass by Reference, where the values are passed to the library function and the changes in the arguments are sent back to the caller. The change made within the function will affect the associated VBA variable or Work Sheet cell value. Pass by reference is useful when a huge spread sheet values are directly updated in-place without explicit copying back to the work sheets. Reference values are specified with '&'.

```
int  get_eod_summary (char *portfolio_name, double &market_value, double &dtd_PandL, doube &mtd_PandL, double &ytd_PandL);
```

In the above formula, the values market_value, dtd_PandL, mtd_PandL, and ytd_PandL are set in the function get_eod_summary, and the updated values will be available to the caller.

Pass by value, the argument values are passed to the remote library function and the changes in its arguments are discarded. Pass by Value is used when the changes are not required in the associated VBA variables or Work Sheet cells. This method is more efficient, as the values are not transferred back to the application.

Pass by value parameters are specified with "const" before its type as shown in the following example

```
functions {
        double  average (const  double * ) ;
}
```

Pass by Value is the default for non-*array* (those don't have "*" in their definition) arguments, unless they are explicitly specified with "&". Array type values (those uses "*") are Pass by Reference by default, unless defined with a "const".

```
double   number_of_days;            // pass by value
double  &number_of_days;            // pass by reference
vector<doube>  historical_prices;   //   pass by value
vector<double> &historical_prices;  // pass by reference

double *historical_prices;          //  pass by reference
const double *historical_prices;    // pass by value
```

We encourage the use of STL vector class against array type, as C++ array object does not provide enough information about its size.

VBA Macros

VBA macros associated with the remote library functions can be more closely packaged with the Library Functions. It is a way of extending the object-oriented methodology, where data, its C++ and VBA methods can be combined into a package, increasing the scope of the application.

LibServer also acts as a source for distributing Excel-VBA modules. When a *LibClient* downloads information from a *LibServer*, these modules are automatically downloaded into the current workbook. This automated download helps application developers to ensure the availability of VBA routines on all *LibClient* applications.

Modules are included in the prototype file by specifying their file name within the tag "Modules {" and "}".

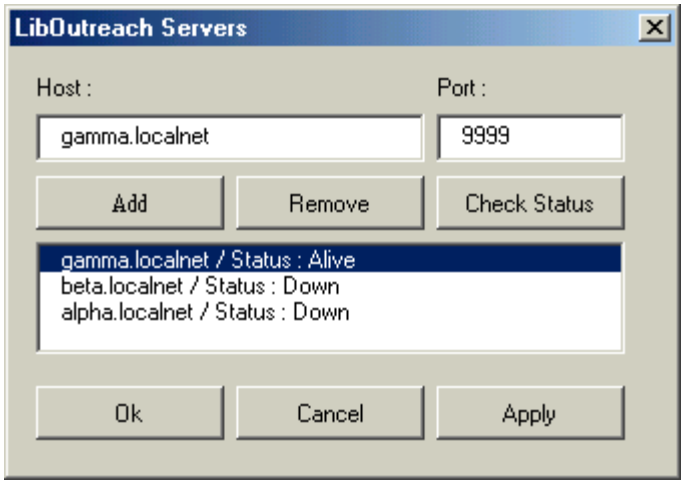
```
Modules {
        /opt/LibOutreach/excel/prime/primes.bas;
        /opt/LibOutreach/excel/complex/complex.bas;
}
```

5. LibClient

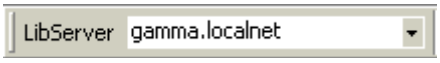
LibClient provides a toolbar for downloading the necessary information from *LibServers*. This section describes the steps in using *LibClient*. LibClient tool bar:

LibServer Specification

LibClient applications can communicate with more than one LibServer. A predefined list of servers can be added by clicking LibServer button, upon which you will see a dialog as shown below. The values shown in the dialog are for illustration purpose only and are not real server names[3].



New servers can be added, and servers no longer in use can be deleted. The server configuration dialog provides an editable text field for specifying the server name and its port number. In the above snapshot, we are using an instance of LibServer running at *gamma.localnet* and listening at port 9999. The system wide default port information is specified in the file, "<LibClient_installed_directory>/conf/LibClient.conf". The server status is displayed along with the server names. The selected server name will appear in the toolbar as shown.



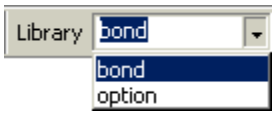
If an application needs modules from multiple servers, the modules need to be downloaded from one server at a time.

[3] These are the server names in use at CyberXpert.

Library Selection

After a *LibServer* has been selected, clicking the 'Library' button gets all libraries available from the specified server. These library names will be displayed next to the 'Library' button in a drop down list.

The first library name in the list will be the default selection. Additional library names in the list can be viewed by clicking the list expansion arrow button, and making a selection.



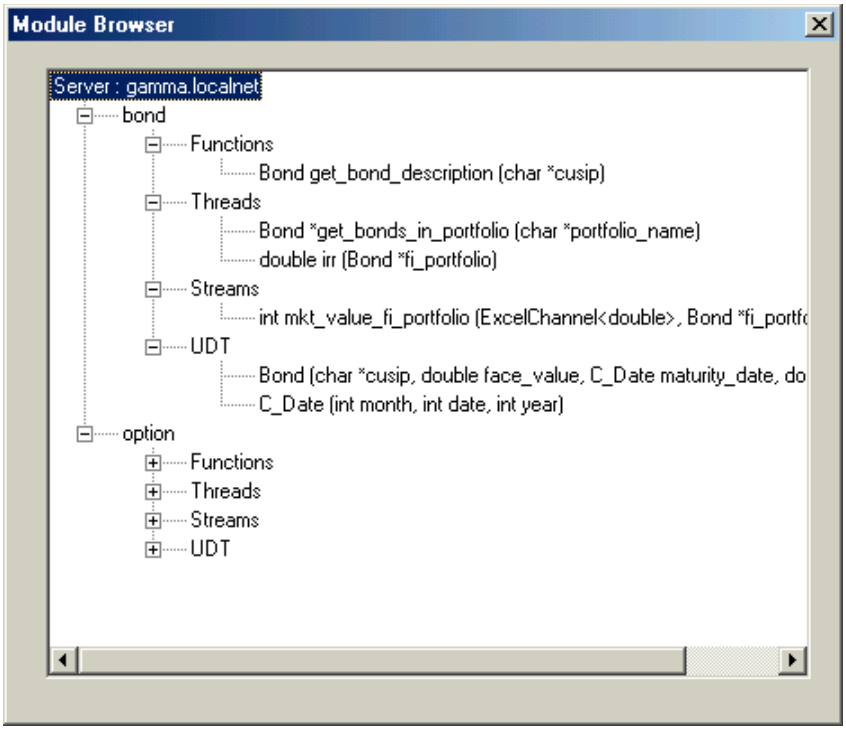
Download Button

The download button is used for getting client-side wrapper routines from the selected server. These wrapper routines are added to the Modules section of the active workbook. For example, if the library name is "bond", the Excel VBA module name would be "bond_WRAPPER". First three lines of any wrapper module contain information about the *LibServer* name, the server's communication port number and the last updated date/time of its prototype file. This date/time stamp is useful in determining the version of the VBA module. The versions can also be easily verified using "Consistency check" button.

Once the wrapper routines are downloaded, we can start using all the *functions/threads/streams* from these downloaded modules in cell formulas and VBA macros. These modules can be viewed using Excel-VBA editor.

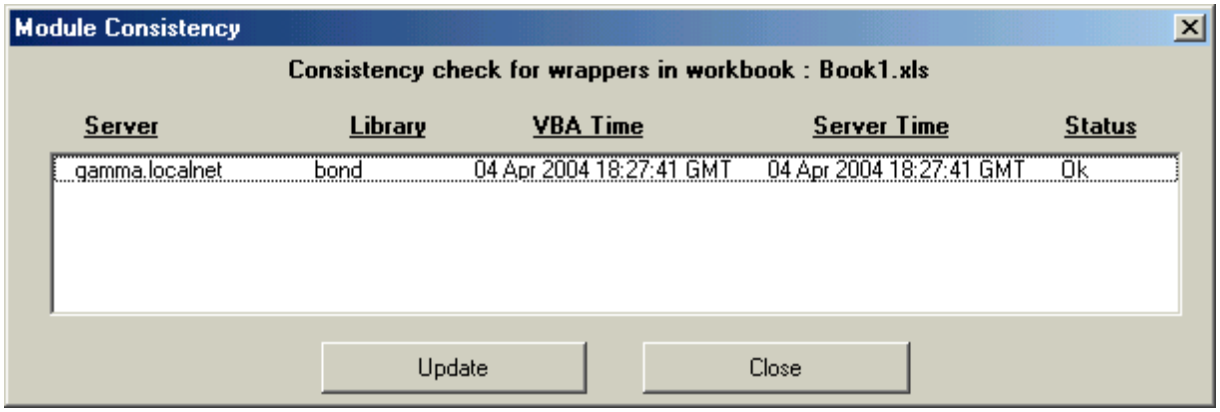
Browse Button

LibClient provides a convenient interface, where we can browse the libraries and library functions available at a server. On clicking browse button, a dialog listing all the libraries and the details of each library, in an expandable tree format. Developers will find it very useful while coding Excel macros.



Consistency Check Button 

When an application is in development, the application's version and server's version may differ, and causes an application error or data integrity error. Consistency check button helps us to compare the module uploaded time and it's updated time at the server. Depending on the status, the inconsistent module can be updated.



6. Basic Data Types

LibServer provides remote library function accessibility to Excel Applications. The remote functions can be run in *sequential, parallel* or *stream* mode. Their arguments can be of basic types (*int, long, float, double, char, char**) or an array of these basic types or user defined types (UDTs). They can return a single value or an array of values. When the result from a remote function is an array, additional wrapper functions are needed to determine the array size and retrieve array elements.

We encourage the use of "vector" and string classes wherever possible, instead of array types. If you use vectors, the "object-wrapping" technique discussed in this section can be totally avoided. As a rule of thumb, if the dimension of an array is known in advance, then use vector class. If the array dimension is not known or want to create values on the fly, then Range class is worth the effort. For an example, if you want to return a file's content to the caller, you can use "Range" class and send the files line by line, without caching them in a vector and returning the vector.

Library Wrappers for Handling Array Results:

LibServer uses an object-oriented interface class named ***Range***. This *Range* class needs to wrap each function returning array values. *Range* class is defined in the header file "LibServer_stub.hxx" in the *LibServer* installation directory.

The definition of Range class is as follows:

```
template <class T>
class Range {
   public :
         virtual GetSize () = NULL;
         virtual T GetValue (int offset)  = NULL;

         // default destructor
         ~Range () {
         }
};
```

There are two methods in the *Range* class, namely, GetSize() and GetValue(offset). These methods need to be appropriately coded for each returned data type. ***GetSize*** () returns the size of the array and ***GetValue*** *(offset)* returns the array element at the given *offset*. *LibServer* uses *GetSize*() to get the array size; then, uses *GetValue*(offset) to get the array elements one by one in the increasing order of offset.

Sample code of a function returning an array of basic type Type is shown below:

```
// if  the function is a  'C' library function, please uncomment following lines
// extern "C" {
//    <return_type>  library_function_name (<library function's argument types>);
// };

class  function_name_class : public  Range <Type> {
```

```
        int     f_size;
        Type *  f_values;
  Public :
        // constructor
        function_name_class (<arguments as in the prototype file>) {
                f_result = function_name (<arguments as in the library function>);
                //  insert here the code to determine  f_size & f_values
                //  from f_result or from reference parameters in the above library function call
        }
        int GetSize () {
                return f_size;
        }
        Type GetValue (int i) {
                return f_values [i];
        }
        ~function_name_class () {
                delete f_values;     // if f_value is an object. Otherwise use 'free (f_values);'
        }
};

Range <Type> *function_name (<arguments, as in the prototype file>) {
        function_name_class *object_instance;
        object_instance  = new function_name_class (<arguments as in the prototype file >);
        Return object_instance;
}
```

Some times, it may be difficult to code the logic for GetSize(). In such cases, rewrite *GetSize()* to return "UNKNOWN" (integer value -1); *GetValue(offset)* to return value for valid offsets, otherwise throw an exception with the error code "NOTVALID" (integer value -1). Sample code illustrating this scenario is shown below:

```
class  function_name_class : public Range <Type> {
        int     f_size;
        Type *  f_values;

  public :
        // constructor
       function_name_class (<arguments as in the prototype file>) {
                f_result = library_function_name (<arguments as in the library function>);
        }

        int GetSize () {
                return UNKNOWN;  // or return -1;
        }
        Type GetValue (int i) {
              // insert here the code to fetch  f_values[i]
                if f_values[I] is available,
                        return f_values [i];
                else
                        throw (UNDEFINED);  // or  throw (-1);
        }
        ~function_name_class () {
                delete f_values;
        }
};
```

The capability of having remote functions returning arrays creates news avenues in application development and expands the reach of Excel based spreadsheet applications.

7. User Defined Types (UDTs)

User defined types (UDTs) are ordered collection of basic types (int, char, char *, string, float, long and double). UDTs improve the programming structure in passing data to libraries and manipulating them in VBA macros. UDTs are associated with "Type" in VBA and "Class" in C++.

An UDT must have a same name across C++ program, VBA module and prototype file. UDTs can be nested but cannot be recursive. A UDT "**Bond**" can be defined in C++, in VBA module and in prototype files as follows:

C++ definitions

```
Class C_Date[4] {
  int  month;
  int  date;
  int  year;
  // member functions
};

Class Bond {
  char    *cusip;
  double   faceValue;
  C_Date   maturity;
  double   coupon;
  // member functions
};
```

Prototype definitions

```
Types {
  C_Date(int   month,
         int  day,
         int   year);

  Bond(char     *cusip,
       double  faceValue,
       C_Date maturity,
       double  coupon );
}
```

VBA definitions

```
Type C_Date
  Month as  Integer
  Date    as  Integer
  Year    as  Integer
End Type

Type Bond
 Cusip        as   String
 FaceValue  as   Double
  Maturity    as   C_Date
  Coupon     as   Double
End type
```

An UDT must be defined before it is used in a nested definition. In the above example, ***C_Date*** is defined before ***Bond*** definition, as C_Date is used in the Bond definition.

LibServer generate VBA wrapper code for the types corresponding to the C++ types. This wrapper code will be added along with the module, if these types are not defined already in the application. For every VBA type definition, a new UDT Array type suffixed with "_Array", and two type converter functions will be generated. One type converter converts Variant to UDT, and the other converts Variant to UDT Array. For the above example, two new types are

```
Type C_Date_Array
        Value () as C_Date
End Type
Type Bond_Array
        Value() as Bond
```

[4] The type name C_Date is used to avoid type conflicts with built-in Date type in C++ & VBA

End Type

, and the conversion methods for the UDT and UDT arrays are

Function C_Date (arg as variant) C_Date

Function C_Date_Array (arg as variant) C_Date()

and

Function Bond (arg as variant) Bond

Function Bond_Array (arg as variant) Bond()

The use of these types and the data retrieval methods are explained in the next chapter on Threads.

UDT types should not have Arrays, but can have vectors.

```
class  A {
        double  *close_price;
};    // It is invalid.
```

But

```
Class A {
        vector<double> close_price;
};   // It is valid.
```

This caveat is due the lack of array size information available to LibServer.

8. Threads

Threads are multiple control instances that run in parallel. Creating a thread returns the thread identifier and starts a new control instance that executes the given routine. The caller routine of the new thread gets the thread identity and proceeds further without waiting for its result. When the thread completes its execution, it provides the result back to the caller.

If a spreadsheet is used for intensive computational analysis, the analytical functions may take significant time for computing cell values and thus put the spreadsheet on hold. This freeze can be overcome by having the functions executed in thread mode. Under the thread mode execution, the spreadsheet can be used for other tasks while the threads are running in parallel.

When a routine runs in thread mode, the *LibServer* creates a thread and returns its identity. The calling cell value is set to "#THREAD(thread identity)" initially. When the computed value arrives, *LibClient* updates the cell value with the new result. The cell formula remains the same all through these stages.

Creating threads from VBA modules

We can call remote functions from your VBA modules. If the function runs in 'Sequential mode', the caller routine waits till it gets its result. If the function runs in 'Parallel mode', the caller routine gets the thread description, and not the result. We need to associate a VBA routine (called ***Callback*** routine) with this thread description, using ***SetCallback*** method. This method associates the Callback routine with the thread descriptor, and this callback routine will be called when the result is available from the thread running at the remote server.

Example: Associating thread descriptors with a callback routine:

```
Dim thread_description as String
Dim user_data              as String

thread_description =  Remote_function  (arguments)

user_data  = "our_data, which may be required in the callback routine"
Application.Run ("SetCallback" , thread_description,  "workbook!callback_routine",  user_data)
```

The remote_function_call can also be in-lined as

```
Application.Run("SetCallback", _
        remote_function_call(arguments),"workbook!callback_routine", user_data)
```

and the callback routine has the following prototype

```
Sub Callback_module (UserData as String, CallbackData() as Variant)
   …
End sub
```

When the result is available, the callback routine will be called with UserData and CallbackData. CallbackData has the result of the thread execution.

Remote execution can be cancelled from VBA routine, by issuing a ***CancelCallback*** request, with the thread description as its argument. This call sends a cancel request to the server.

Example:

```
Application.Run ("CancelCallback", thread_description)
```

UDTs in Threads

Handling UDTs is similar to handling primitive types. But handling of UDT arrays differ as VBA cannot bind variants to UDT arrays. It can be overcome by using UDT data retrieval function. For example, to get an array of bond, Get_Bond_Array data retrieval method can be used. The following example shows the usage of Bond array in the callback function.

'Call a library function to get all callable bonds in the year 2002

```
Sub find_callable_bonds ()
  Dim result as Integer
  Result =  Application.Run( "SetCallback", callable_bonds (2002),  _
                              "callable_bond_callback_function",   "callable in 2002" )
End  Sub


Sub  callable_bond_callback_function (UserData as String,  CallbackData () as Variant)
  Dim bonds ()  as Bond

  bonds  = Get_Bond_Array (CallbackData (0))
  ' logic to plug the data in worksheet from 'bonds'.

End Sub
```

Pass By Reference in Threads

It is common for library functions return more than one value by "pass by reference". It is very intuitive in the case of a function calls. In the case of "thread" mode execution, all outbound variables (return values and pass by reference parameters) are packed into the CallbackData, in such a way,

CallbackData (0) corresponds to the return value

CallbackData (1) corresponds to the name of the 1st pass by argument

CallbackData (2) corresponds to the value of the 1st pass by argument

…

CallbackData(N) corresponds to the name of the N^{th} pass by argument (and N is an odd number)
CallbackData(N+1) corresponds to the value of the N^{th} pass by argument

Program developer can retrieve value by going through the argument name. It is inefficient, but ensures the correct value is being retrieved.

Example:

```
int pass_by_ref_example (int arg_1, double &arg_2, vector<double> arg_2, vector<double> &arg_4) {
…
        arg_2 = 10;
        arg_4[0] = 20; arg_4[1] = …
        return 30;
}
```

The callback data will have

CallbackData (0) = 30
CallbackData(1) = “arg_2”
CallbackData(2) = 10
CallbackData(3) = “arg_4”
CallbackData(4) = [20, …]

UDTs and Pass by References are handled the same way in the Stream callback routines.

9. Streams

Streams are multiple control instances that keep updating clients (excel applications - workbook cells or VBA macros) with the computed results. Streams, also called 'stream-threads', are a special type of 'threads' discussed in the previous chapter. Streams can run without termination and keep posting updates. Streams are useful for 'server-side' monitoring applications, where changes at a server need to be immediately communicated to clients.

A stream-thread returns the stream identifier and starts a new control instance to execute the given routine. The caller routine gets the stream identifier and proceeds further without waiting for its result. When the stream-thread is ready to deliver a result, it sends its result to the caller.

If a spreadsheet is used for intensive, continuous computational analysis, the analytical functions may take significant time to compute cell values and thus put the spreadsheet on hold for every recalculation. Under stream mode execution, you may continue using the spreadsheet for other tasks while the functions are executing in parallel, still abreast with updates.

When you call *LibServer* to run a function in stream mode, the *LibServer* creates a stream-thread and returns its identity. The calling cell value is set to "#STREAM (thread identity)". When the computed value arrives at the first time, *LibClient* updates the cell value with the new result. The cell formula remains the same, but displays the recently updated value.

Creating stream-threads from VBA modules

Stream-threads are created within VBA modules in the same way as 'threads'. Depending on the 'prototype' file specifications, the *LibServer* determines whether a routine is to be run in stream mode. When a routine runs in 'Stream mode', the caller routine gets the stream identifier initially, and not the result. You need to use ***SetCallback*** method to associate a VBA routine (called ***Callback*** routine) with this stream identifier. This *callback* routine is automatically called whenever result/update is available from the thread running at the remote server. You can cancel the stream-thread execution anytime within the VBA modules by issuing a ***CancelCallback*** request with the stream identifier as its argument.

Implementing Stream-threads in LibServer

Functions, threads and stream-threads are started by a *LibServer* upon client request. Functions and threads terminate after sending the result back to the clients. However, stream-threads continue to run and keep communicating the updates. A *LibServer* accomplishes this persistent link by sending the communication handle, called 'Excel-Channel', as an argument to the "stream mode" library routine at its invocation time. Once the results are available, the library routine sends the results through this 'Excel-Channel' back to the client.
A function's prototype in the 'prototype file' for 'random_number' to be run in stream mode, it can be specified as

```
Streams {
        int random_number (ExcelChannel<double> *excel_channel,  int seed) ;
}
```

The function 'random_number' takes a 'seed' as the input and updates clients with a value of 'double'. The function can be implemented in 'C++' as

```
#include "libserver_stub.hxx"

int random_number (ExcelChannel<double> *excel_channel, int seed) {
    double  new_random_number;
    while (1) {
        new_random_number = rand (seed);   // rand  is a random number generator
        excel_channel->Send (new_random_number);
    }
    return 0;
}
```

Results can be of type Array. Array needs to be a subclass of 'Range' class as described in the section '***Basic Data Types***'.

Stream mode functions send their results to *LibClients* through Excel-Channel and not through standard return value convention.

10. Examples – C++ Interface

Example programs and prototypes are distributed along with the package and can be found under the directory "*samples"* at both *LibServer* and *LibClient*.

This section illustrates an implementation of **reverse_range.** The '**reverse_range**' function reverses the input array and returns the reversed array. The '**reverse_range_in_thread_mode'**, a thread-mode execution simply waits (*sleeps*) for some time and calls the ***reverse_range*** function. The **reverse_range_in_stream_mode**, a stream-mode execution keeps calling ***reverse_range*** function and keeps sending the result. All three functions take an **array** argument and return an **array** result. Since their return type is an array, the result has to be wrapped with **Range** class. In this example, we consider the result array elements are to be of type *double* and call the wrapper class as '*DoubleRange*'.

```
class DoubleRange : public Range <double> {
    int         size;
    double   *values;

  public :
    DoubleRange (int _size, double _values[] ) {
          size    = _size;
          values = _values;
    }

    int GetSize  () {
          return size;
    }

    double GetValue (int i) {
          return values [i];
    }

    ~DoubleRange () {
          delete [] values;
    }
};

DoubleRange *reverse_range (double *input_range, int range_size) {
          // reverses the input range.
          double *output_range = new double [range_size];
          for (int i = 0; i < range_size; i++) {
                    output_range [i] = input_range [range_size  - i - 1];
          }
          DoubleRange *result_range = new DoubleRange (range_size, output_range);
          return result_range;
}

DoubleRange *reverse_range_in_thread_mode (int time_gap, double *input_range, int range_size) {
          // delay for the specified time.
          sleep (time_gap);
```

```
        // and calls the reverse_range function; simulating as if 'reverse' range is a time consuming task.
        return reverse_range (input_range, range_size);
}

int  reverse_range_in_stream_mode (ExcelChannel<DoubleRange *> *excel_channel, int time_gap,
                                                double *input_range, int range_size) {
        DoubleRange *result;

        while (1) {
                // delay for the specified time
                sleep (time_gap);
                // call the reverse range function;
                result = reverse_range (input_range, range_size);
                excel_channel->Send (result);
        }
}
```

11. Examples – VBA Interface

This section illustrates the usage of Excel-VBA interface to call remote functions. Routines ***reverse_range*** and ***reverse_range_in_thread_mode*** are remote server library functions that run in "Sequential mode" and "Parallel mode" respectively. These two functions take an array of numbers and return the array in reverse order. Interface for "Stream mode" execution is identical to that of "Parallel mode" execution.

VBA routine for reversing a range in Sequential mode

In this example, we directly pass the array ***CellRange*** extracted from the Excel Range object to ***reverse_range*** function (for execution in the remote server). Upon completing function execution, the server returns us the result. The result returned by the function is then used to populate our target Excel Range.

```
Sub vba_reverse_range ()
   Dim CellRange      As Range
   Dim Retvalue       As Variant
   Dim Count          As Integer

   On Error GoTo ErrLab
   ' Get input values
   Set CellRange = Range("B4:G4")

   ' Call remote function
   Retvalue = reverse_range (CellRange, CellRange.Count)

   ' Clear output cell values
   Set CellRange = Range("B7:G7")
   CellRange.ClearContents

   ' Copy results
   Set CellRange = Range("B7")
   For Count = 0 To UBound(Retvalue)
      CellRange.Value = Retvalue(Count)
      Set CellRange = CellRange.Offset(0, 1)
   Next Count

  Exit Sub

ErrLab:
   MsgBox "Error " + Err.Description
End Sub
```

VBA routine for reversing a range in Parallel/Stream mode

In this example, we are directing the remote server to run the ***reverse_range_in_thread_mode*** function in Parallel mode and upon its completion invokes the subroutine "LibServer_demo_sample.xls!***reverse_range_cb***" **(**specified in the call back routine ***reverse_range_callback_routine*****)**. The result from the server function is passed to ***reverse_range_cb*** as the second argument **(*CBResult*)** of the subroutine.

```
Sub create_reverse_range_thread()
   Dim CellRange              As Range
   Dim ReturnValue            As Integer

   Set CellRange = Range ("B4:G4")
   ' Set Callback, asking server to reverse the given range.
   ReturnValue = Application.Run("SetCallback", _
                 reverse_range_in_thread_mode (CellRange, CellRange.Count), _
                 ThisWorkbook.Name + "!reverse_range_cb", _
                 "Result for the range B4:G4")

   ' Display the status
   Set CellRange = Range("B7")
   If (ReturnValue = 1) Then
      CellRange.Value = "Waiting for reverse_range_in_thread_mode's result"
   Else
      CellRange.Value = "Unable to set Callback!!!"
   End If
End Sub


Sub reverse_range_cb (MyData As String, CallbackData() As Variant)
On Error GoTo ErrLab
   Dim Count As Integer
   Dim CellRange As Range
   Dim  Result () as Double

   Set CellRange = Range("B7")
   Result = CallbackData (0)

   ' populate the result
   For Count = 0 To UBound(Result)
      CellRange.offset(0, Count).Value = Result(Count)
   Next Count

   Exit Sub
ErrLab:
   MsgBox "Error :" + Err.Description
End Sub
```

12. Summary

LibOutreach enables Microsoft® Excel applications to access UNIX library functions. Excel – UNIX integration eliminates migrating server applications to desktop environment. Excel's dynamic value updates and report generation coupled with remote-servers' capabilities give you a powerful, user-friendly application development and deployment platform.

Parallel execution of time-consuming processes speedup the results without slowing down your desktop. Dynamic updates from servers to Excel spreadsheets, along with *ExcelOutreach*'s event notification, put you ahead of others.

You can significantly reduce client maintenance and simplify infrastructure environment by running all the proprietary and third party applications under their own resource requirements and making them available from Excel.

Please contact our Customer Support department for any clarifications, enhancements, customizations and suggestions.

CyberXpert, Inc.
Phone # (732) 713 2629
Email: support@CyberXpert.com

1

13. Index