Client side implementation of a server

This code shows how is a client side implementation of a server.

Overview

In the page How to create a server from scratch we have seen how to implement a server from scratch. When programming with servers we do not directly access the API implemented on that example. Instead, we use a 'R' class to do this. The 'R' class (see R class) are used to handle resources maintained elsewhere. Talking about servers, R classes are used to manipulate and access the server. Although the use of this kind of class is very common, few people know how they are built (I was one who didn't know until now :)). This example shows how a simple R class is implemented. This example deals with synchronous requests. Asynchronous requests implementation can be found here Implementing asynchronous requests.

This snippet can be self-signed.

Preconditions

Know the Symbian Client-Server Framework.

How to implement a R class

When a R class is associated with a server it is derived from RSessionBase. This means that the R class we're going to implement is a session. Session to what? To the server of course. Is through this session that the communication among client and servers occurs. Is this session really necessary? Yes because servers and clients runs in separate process which means they have insulated memory address spaces. Ok, now we know sessions are necessary and we also know we need to inherit from RSessionBase. The following example shows how this can be done. This example is the client side implementation of the How to create a server from scratch.

```
// CLASS DECLARATION
/**
* RTimeServerSession
*  This class provides the client-side interface to the server session
*/
class RTimeServerSession : public RSessionBase
    {
    public: // Constructors and destructors

        /**
        * RTimeServerSession.
        * Constructs the object.
        */
        RTimeServerSession();

    public: // New functions

        /**
        * Connect.
        * Connects to the server and create a session.
        * @return Error code.
        */
        TInt Connect();

        /**
        * Version.
        * Gets the version number.
        * @return The version.
        */
```

```
        TVersion Version() const;

        /**
        * RequestTime.
        * Issues a request for the time to the server.
        * @param aTime The location for the server to write data into.
        */
        void RequestTime( TTime& aTime ) const;

    };
```

Nice. We have a R class definition. As you probably have already observed, most part of the R class that are session to servers have a *Connect* method (I haven't seen any that hasn't). In our case this is not different. This method is used to connect to the server and create the session. This means: the connect method is used to start the server (CREATE THE SERVER PROCESS uuoou) and create the session. This is what happens with the so called 'transient servers'. This servers runs under demand, i.e., they don't start at boot. So, when the first client calls *Connect()*, the server process is started and the session is created. After that, subsequent calls to *Connect()* only create the session.

We have the *Version()* method that is used to verify the compatibility among the client side implementation and the server.

After specifying the *Connect* and *Version* methods, is time to specify the API for clients to access the server. In this example there is only one synchronous API method (*RequestTime*). The *RequestTime* method is used to obtain the time from the server. The implementation of these methods are showed as follows.

```
// ---------------------------------------------------------------------------
// RTimeServerSession::RTimeServerSession()
// C++ default constructor can NOT contain any code, that might leave.
// ---------------------------------------------------------------------------
//
RTimeServerSession::RTimeServerSession()
: RSessionBase()
    {
    // No implementation required
    }

// ---------------------------------------------------------------------------
// RTimeServerSession::RequestTime()
// Issues a request for the time to the server.
// ---------------------------------------------------------------------------
//
void RTimeServerSession::RequestTime( TTime& aTime ) const
    {
    // Create descriptor to enable copying data between client and server.
    // Note: This can be local since this is a synchronous call.
    // Note : Using TPtr8 since this is binary information.
    TPtr8 descriptor( reinterpret_cast<TUint8*>( &aTime ), sizeof( aTime ),
                      sizeof( aTime ) );

    // Package message arguments before sending to the server
    TIpcArgs args( &descriptor );

    // This call waits for the server to complete the request before
    // proceeding. When it returns, the new time will be in aTime.
    SendReceive( ETimeServRequestTime, args );
    }

// ---------------------------------------------------------------------------
```

```
// RTimeServerSession::Connect()
// Connects to the server and create a session.
// -------------------------------------------------------------------------
//
TInt RTimeServerSession::Connect()
    {
    TInt error = ::StartServer();

    if ( KErrNone == error )
        {
        error = CreateSession( KTimeServerName,
                               Version(),
                               KDefaultMessageSlots );
        }
    return error;
    }
```

The first thing we see in the code is the constructor of the derived class calling the parent class constructor. No implementation is required for the constructor. The next thing to take a look is the API method. This method send a TTime object reference to the server to be filled with the current time. There are two important things to talk about this method and how things happens:

First - How transfer the arguments from client to server: To do this, the arguments must be packed in descriptors. The arguments can be descriptors or they can be transformed in descriptors used constructors (as in the example) or special classes to do this as TPckgBuf. After the argument is encapsulated in a descriptor, it is packed in a TIpcArgs object. This object can pack many different arguments.

Second - Send the request: To do this the function *SendReceive* of the RSessionBase is used. This function is overloaded to accept different arguments, but the main area the arguments and the index for the function in the server that should be called. This index is defined in a enumeration shared in a header file by both client side implementation and server. An example of shared header file follows. It shows the enumeration shared by client side implementation and server implementation so the client makes request to the correct functions in the server.

```
// DATA TYPES
// Opcodes used in message passing between client and server
enum TTimeServRqst
    {
    ETimeServRequestTime
    };
```

The last thing we see in the code is the *Connect* method. The first thing done by this method is to start the server (will be discussed later on). After it creates the session calling the base method *CreateSession* giving the name of the server (the file name of the .exe), the version and the number of slots that specifies the total number of requests a client may have with the server at any time.

How to start the server

Before starting the server we test if the server is already running. This is done using a standard query using a TFindServer giving the server name to look for. If this server is already running, the *StartServer* returns. If the server is not running, then the process is created. This is done using an RProcess instance. This instance can create new process through the *Create* method provided that receives the file name of the server to be created, a descriptor with the parameters to be passed to it and the UID. The following code shows these steps to create the server.

```
//prototypes
static TInt StartServer();
static TInt CreateServerProcess();
```

```
// ----------------------------------------------------------------------------
// StartServer()
// Starts the server if it is not already running
// ----------------------------------------------------------------------------
//
static TInt StartServer()
    {
    TFindServer findTimeServer( KTimeServerName );
    TFullName name;

    TInt result( findTimeServer.Next( name ) );
    if ( result == KErrNone )
        {
        // Server already running
        return KErrNone;
        }

    RSemaphore semaphore;
    result = semaphore.CreateGlobal( KTimeServerSemaphoreName, 0 );
    if ( result != KErrNone )
        {
        return  result;
        }

    result = CreateServerProcess();
    if ( result != KErrNone )
        {
        return  result;
        }

    semaphore.Wait();
    semaphore.Close();

    return KErrNone;
    }

// ----------------------------------------------------------------------------
// CreateServerProcess()
// Creates a server process
// ----------------------------------------------------------------------------
//
static TInt CreateServerProcess()
    {
    const TUidType serverUid( KNullUid, KNullUid, KServerUid3 );

    RProcess server;

    TInt result( server.Create( KTimeServerFilename, KNullDesC, serverUid ) );

    if ( result != KErrNone )
        {
        return  result;
        }

    server.Resume();
    server.Close();
```

```
    return  KErrNone;
    }
```

Postconditions

This code shows the main steps to create a handle to a server. The reader sould be prepared to start implementing its own version of its handle based on this example.

Internal Links:

- How to create a server from scratch
- Client-Server Framework
- Inter Process Communication in Symbian