

# Transient server template for Symbian OS v9.x

Hamish Willee, Andrew Thoelke, Adrian Taylor

Version 1.3

October 2006

## 1 Introduction

Symbian OS makes much use of the client-server model. A transient server is one that exists only when it is needed; it is started up the first time a client attempts to connect, and then shuts itself down some point after the last client session has been closed.

The code used to start and shut down the server is almost boiler-plate. This paper and associated code deliver a template that can be used as the basis for Symbian OS v9 transient servers that launch the server in a separate process to the client; note that the start-up code for a server that runs in the same process as the client is not covered in this paper.

Many Symbian OS servers are transient – the advantages of reduced memory consumption often outweigh the increased initial server boot time, especially when the service provided by the server is required infrequently.

Transient servers become especially important in Symbian OS v9.x, where platform security means that there is sometimes a need to put access to services behind a process boundary. For example, implementation of a service which requires more capabilities than you need from the clients of the service.

## 2 What does the server do?

The server comprises of two parts:

- **t-client.dll**, a client side DLL (with client API) and
- **t-server.exe**, the server executable

The client API (`RMySession`, defined in **t-client.h**) provides methods to: connect to the server, send a message synchronously, receive a message asynchronously, and to cancel an outstanding asynchronous request to receive a message.

```
class RMySession : public RSessionBase
{
public:
    IMPORT_C TInt Connect();
    IMPORT_C TInt Send(const TDesC& aMessage);
    IMPORT_C void Receive(TRequestStatus& aStatus, TDes& aMessage);
    IMPORT_C void CancelReceive();
};
```

From the perspective of transient servers, the most interesting method is `Connect()`. This first attempts to connect to the server and create a session. If this fails, for example if the server does not yet exist, then the function attempts to start the server, and then again to create a session.

```

EXPORT_C TInt RMySession::Connect()
//
// Connect to the server, attempting to start it if necessary
//
{
    TInt retry=2;
    for (;;)
    {
        TInt r=CreateSession(KMyServerName, TVersion(0, 0, 0), 1);
        if (r!=KErrNotFound && r!=KErrServerTerminated)
            return r;
        if (--retry==0)
            return r;
        r=StartServer();
        if (r!=KErrNone && r!=KErrAlreadyExists)
            return r;
    }
}

```

CMyServer::NewSessionL() is called server-side to create the server-side session object. In this example that's all it does – in a real server you would normally check that the version of server that the client is expecting is correct, and also possibly that the client has the correct capabilities to use this server.

The server maintains a count of sessions. When a session is destroyed the CMyServer::DropSession() function is called to decrement the count – when the count reaches zero a shutdown timer is started. The shutdown timer is reset if another session is created, otherwise it shuts down the server when it times out.

### 3 What's in the template code?

The template consists of an example client-server interface and a console test executable that verifies its operation. Both test code and example code can be built from the command line in the \Transient\ directory using the normal build invocation:

```

bldmake bldfiles
abld build [gcce] | [armv5]

```

#### 3.1 Server files

The server is stored in the directory \Transient\Server\. The following describes each of the server files.

Directory	File	Description
src\	client.cpp	Source code for client side DLL
	server.cpp	Source code for server
inc\	server.h	Server definition
	clientserver.h	Shared client/server definitions
	t-client.h	Client –side API. This is exported to /epoc32/include/
group\	t-server.mmp	Project file for server exe
	t-client.mmp	Project file for client side DLL
sis\	server_gcce.pkg	S60 Server PKG file – imports GCCE binaries
	server_armv5.pkg	S60 Server PKG file – imports ARMv5 binaries

### 3.2 Test console files

The test code is stored in the directory `\Transient\Test_ConsoleExe\`. The test code is a console executable **T-Test.exe** which loads **t-test.dll**; this in turn calls each of the server's client-side APIs.

The following describes each of the files in the test code.

Directory	File	Description
src\	test.cpp	Source code for test exe
	testclient.cpp	Source code for test dll
inc\	plugin.h	Test DLL definitions
	testclient.h	Test DLL definitions
group\	t-test.mmp	Project file for test exe
	t-testc.mmp	Project file for test DLL
sis\	TestConsoleExe_gcce.pkg	S60 test PKG file – imports GCCE binaries
	TestConsoleExe_armv5.pkg	S60 test PKG file – imports ARMv5 binaries

### 3.3 Installing on a device

The PKG files provided in the `\sis\` subdirectories allow you to build installation files for the server and test exe that can be run on both S60 3rd Edition and UIQ3 devices.

The PKG files specify relative paths, so you will need to use **makesis** as described in [FAQ-1113: How do I avoid using absolute paths in my package file?](#) - e.g. "makesis -d%EPOCROOT% server\_gcce.pkg"

You may also need to sign the files, depending on which device you are using.

## 4 What does the test code do?

The test code runs three main tests:

#### Validate interface

This tests normal use of the interface, including connecting to the server when it hasn't yet started, sending and receiving messages, creating a second session (server already started). Finally it tests that closing both of the sessions allows the transient server to terminate.

#### Simultaneous start

This test confirms that the start-up code succeeds when two sessions are started "simultaneously".

#### Start while stopping

This test validates that the start-up code behaves correctly when the server is has shut down or is shutting down e.g. it attempts to restart with dead thread/server and after cleanup.

## 5 How do I run the test code?

The console exe is not a GUI application, and so will not show up in the UI shell for either S60 or UIQ. You can launch it from another (visible) application using code as below:

```

RProcess testexe;
_LIT(KTestCodeFileName, "\\sys\\bin\\t-test.exe");
_LIT(KNullDesc, "");
 TInt err= testexe.Create(KTestCodeFileName, KNullDesc);
 if (err==KErrNone)
 {
  TRequestStatus stat;
  testexe.Rendezvous(stat);
  if (stat!=KRequestPending)
    testexe.Kill(0); // abort startup
  else
    testexe.Resume(); // logon OK - start the server
  User::WaitForRequest(stat); // wait for start or death
  testexe.Close();
 }

```

The test code has run successfully on a Nokia E61 phone. The SIS files were signed using a DevCert with only basic capabilities. The code has not been run on a UIQ3 device, but there is no reason that it should not install and run correctly.

## 6 How do I extend this template?

### 6.1 Platform security

The client DLL and server EXE currently have no capabilities. When you create your server from this template you will need to give the EXE any capabilities it needs to use any protected APIs. You will need to give the client DLL the capabilities of all its possible clients.

If you also want to control access to your server, you should instead use the server base class [CPolicyServer](#) instead of [CServer2](#). This makes implementing secure servers, or securing existing servers, relatively simple.

It is possible to secure APIs using [CServer2](#) directly – for example, the fragment below shows how to use `_LIT_SECURITY_POLICY_S0` to check the client SID:

```

// Check that SID matches required application.
// Use security policy check so behaviour obeys global security policy.
static _LIT_SECURITY_POLICY_S0(mySidPolicy, KRequiredSecureId);
r = mySidPolicy().CheckPolicy(aMessage);
if (!r)
{
  User::Leave(KErrPermissionDenied);
}

```

The server currently has an unprotected name, which means that it could be impersonated by another server that advertises itself as “t-server”. If the client might pass sensitive information to the server, then the client needs to be able to ensure that only the correct server is loaded. You can do this by giving the server a protect name – one which starts with “!”. If you do this, you will need to request the `ProtServ` capability. See below for more information on the server name.

### 6.2 UIDs

The example uses UID/SIDs in the example range. For a commercial application you would need to use UIDs allocated from [www.symbiansigned.com](http://www.symbiansigned.com).

## 6.3 Client-server issues

Below are several points about modifications you might make to the template as part of your server design.

### 6.3.1 Client-side API

The client-side API, Inter-Process Communication (IPC) and server side implementation is dependent on what your server needs to do. Many servers will copy data between client and server in a similar way to those shown by the APIs in this example.

You should rename your files, class and method names to be meaningful for your context.

### 6.3.2 Number of message slots:

`RMySession::Connect()` calls `RSessionBase::CreateSession()`, specifying a message slot count of `KServerDefaultMessageSlots`. The number of message slots is an important parameter in server design because it defines the number of outstanding requests the client may have with the server at any one time. In this case there is only 1 asynchronous function, so only 1 request can be outstanding, and hence only one message slot is required.

The maximum number of slots is 255. If `aAsyncMessageSlots== -1` then this indicates that the session should use messages from the global free pool of messages.

### 6.3.3 Retry count

`RMySession::Connect()` first tries to create a session with the server, and if this fails it tries to re-start the server (which may never have existed, or which may have been terminated). This process repeats until a session is created, session creation or server start fails with an unexpected or unrecoverable error, or the retry count (`KServerRetryCount`) is reached.

`KServerRetryCount` (**client.cpp**) could theoretically be infinity; the current value of '2' seems to eliminate 99% of all startup/shutdown race problems – however if specific problems are discovered, it may be necessary to change this value.

### 6.3.4 Server name

You will need to edit the server name (`KMyServerName`) defined in **clientserver.h**. You may wish to make the server name protected, as described above in 6.1.

Other values in **clientserver.h** can all be modified, as appropriate for your design (e.g. the IPC function enumeration).

## 7 How do I report errors or queries about this template code

The best place to raise queries is on Symbian's [feedback.api](#) newsgroup.

[Back to Developer Library](#)

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.