How to Load Test RBS Applications with HP LoadRunner

Applicable Releases:

Sybase Unwired Platform 2.1.2 or above
SAP Mobile Platform 2.3 or above

Version 1.0
May 2013
**Document History**

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>First official release of this guide</td>
</tr>
</tbody>
</table>
## Typographic Conventions

<table>
<thead>
<tr>
<th>Type Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example Text</td>
<td>Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Cross-references to other documentation.</td>
</tr>
<tr>
<td>Example text</td>
<td>Emphasized words or phrases in body text, graphic titles, and table titles.</td>
</tr>
<tr>
<td>Example text</td>
<td>File and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.</td>
</tr>
<tr>
<td>Example text</td>
<td>User entry texts. These are words or characters that you enter in the system exactly as they appear in the documentation.</td>
</tr>
<tr>
<td>&lt;Example text&gt;</td>
<td>Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.</td>
</tr>
<tr>
<td>EXAMPLE TEXT</td>
<td>Keys on the keyboard, for example, F2 or ENTER.</td>
</tr>
</tbody>
</table>

## Icons

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>🚨</td>
<td>Caution</td>
</tr>
<tr>
<td>📌</td>
<td>Note or Important</td>
</tr>
<tr>
<td>📚</td>
<td>Example</td>
</tr>
<tr>
<td>🔄</td>
<td>Recommendation or Tip</td>
</tr>
</tbody>
</table>
Table of Contents

1. Prerequisites .................................................................................................................. 1
   1.1 Hardware and Software Requirements ...................................................................... 1
   1.2 Knowledge Prerequisites .......................................................................................... 1

2. Introduction ...................................................................................................................... 2
   2.1 Introduction to the Load Test Method ....................................................................... 2
   2.2 Types of Load Tests .................................................................................................. 2
   2.3 Test Preparation ....................................................................................................... 2

3. Create the Skeleton of the Test Script ............................................................................. 4
   3.1 Load Test Scenario .................................................................................................. 4
   3.2 Generate Client Code ............................................................................................... 5
   3.3 Create a C# .Net Project with LoadRunner VuGen ................................................... 5
   3.4 Edit the Vuser Project with Microsoft Visual Studio 2012 ....................................... 6
      3.4.1 Add Runtime References .................................................................................... 6
      3.4.2 Change Target Framework to 3.5 (optional) ....................................................... 7
      3.4.3 Add Compile-time References ........................................................................... 8
   3.5 Copy the Generated Code to the Project .................................................................... 9
   3.6 Create a Runnable Console Application .................................................................. 10
   3.7 Preparation before Creating Test Scripts ................................................................. 12
      3.7.1 Add Namespaces Used in the Test Script ........................................................... 12
      3.7.2 Toggle between Visual Studio and LoadRunner ............................................... 13
      3.7.3 Copy Helper Classes into the Project ................................................................. 14

4. Implement the Test Logic ............................................................................................... 15
   4.1 vuser_init - Initialization Logic ............................................................................... 15
   4.2 Track Status with Callback Handlers ........................................................................ 18
   4.3 Action – Test Logic .................................................................................................. 23
      4.3.1 Code Snippet of Action.cs ................................................................................. 23
      4.3.2 Measure Synchronization Response Time ......................................................... 26
      4.3.3 Break Down Synchronization Time into Operation Replay and Download Sync ... 27
   4.4 vuser_end – Clean-up Logic ..................................................................................... 27
   4.5 Run Single User Test with Visual Studio ................................................................. 28

5. Continue to Build the Script with LoadRunner ............................................................. 30
   5.1 Convert Project from Visual Studio .......................................................................... 30
   5.2 Switch the Run-time Environment ........................................................................... 30
   5.3 Configure Run-time Settings for the Script .............................................................. 30
   5.4 Run the Script in VuGen ......................................................................................... 32
   5.5 Parameterize the Test Script .................................................................................... 32

6. Perform Load Tests with LoadRunner Controller ......................................................... 38
   6.1 Prepare the Test Environment.................................................................................... 38
   6.2 Create a Test Scenario .............................................................................................. 38
      6.2.1 Create a New Test Scenario .............................................................................. 38

SAP The Best-Run Businesses Run SAP™
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.2</td>
<td>Configure Run-time Settings</td>
<td>39</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Configure Test Schedule</td>
<td>42</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Distribute the Load to Load Generators</td>
<td>43</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Configure OS Monitoring for the Test</td>
<td>44</td>
</tr>
<tr>
<td>6.2.6</td>
<td>Configure Results Settings</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>Run the Test Scenario</td>
<td>44</td>
</tr>
<tr>
<td>7.</td>
<td><strong>Advanced Topics</strong></td>
<td>46</td>
</tr>
<tr>
<td>7.1</td>
<td>Run Test Scripts on a Remote Load Generator</td>
<td>46</td>
</tr>
<tr>
<td>7.2</td>
<td>Run Load Tests on Multiple Load Generators</td>
<td>47</td>
</tr>
<tr>
<td>7.3</td>
<td>Pace Vusers with Rendezvous Points</td>
<td>50</td>
</tr>
<tr>
<td>7.4</td>
<td>Use a RAM Disk</td>
<td>52</td>
</tr>
<tr>
<td>7.5</td>
<td>Load Test via IP-based Load Balancer</td>
<td>52</td>
</tr>
<tr>
<td>7.6</td>
<td>Simulate Mobile Networks in the Test</td>
<td>53</td>
</tr>
<tr>
<td>7.7</td>
<td>Clean Up Test Data</td>
<td>54</td>
</tr>
<tr>
<td>8.</td>
<td><strong>Monitor SMP during Load Tests</strong></td>
<td>55</td>
</tr>
<tr>
<td>8.1</td>
<td>Collect OS Performance Metrics</td>
<td>55</td>
</tr>
<tr>
<td>8.2</td>
<td>Monitor SMP with MobiLink Monitor</td>
<td>56</td>
</tr>
<tr>
<td>8.3</td>
<td>Monitor SMP with CA Wily Introscope</td>
<td>58</td>
</tr>
<tr>
<td>9.</td>
<td><strong>Analyze Test Results</strong></td>
<td>61</td>
</tr>
<tr>
<td>10.</td>
<td><strong>Appendix</strong></td>
<td>65</td>
</tr>
<tr>
<td>10.1</td>
<td>Create the Sample MBO Model</td>
<td>65</td>
</tr>
<tr>
<td>10.2</td>
<td>Source Code of ApplicationCallbackHandler.cs</td>
<td>69</td>
</tr>
<tr>
<td>10.3</td>
<td>Source Code of SMPCallbackHandler.cs</td>
<td>71</td>
</tr>
</tbody>
</table>
1. Prerequisites

This guide is written based on the following software versions. The steps for other software versions are mostly the same or similar, and the differences are pointed out wherever necessary.

- SAP Mobile Platform (SMP) 2.3 SP01
- HP LoadRunner 11.50
- Visual Studio 2012 for Desktop

1.1 Hardware and Software Requirements

- Sybase Unwired Platform (SUP) 2.1.2 or above, or SMP 2.3
  Make sure that the installed license supports the defined number of virtual users.
- HP LoadRunner
  - LoadRunner Controller running on Windows Server OS
  - LoadRunner Load Generator, 4 or more CPUs, 8GB or more RAM, gigabit network adapter, Windows Server OS (64bit OS preferred)
  - Additional Load Generator machines, depending on the scenario or test plan
  - HP LoadRunner 9.52 or above. HP LoadRunner 11.50 preferred.
  - .NET Record/Replay License or C# .NET Template Bundle License for the planned number of Vusers
  - Gigabit network connectivity between Load Generators and SMP Server, Relay Server or its load balancer, if applicable
  - Visual Studio 2010 (C#) or Visual Studio 2012 for Desktop

1.2 Knowledge Prerequisites

You should be familiar with the following knowledge in order to follow the steps described in this guide:

- Good knowledge of SMP MBO modeling and Object API
- C# and Microsoft .Net programming
- Basic knowledge of HP LoadRunner
2. Introduction

2.1 Introduction to the Load Test Method

You are using SAP Mobile Platform for connectivity and synchronization of data for your native RBS (Replication-based Synchronization) applications. Prior to releasing these applications to production, you want to be able to test the applications and the platform to ensure that the application and the system work and perform as needed when running the applications under anticipated load.

The volume testing technique described in this document is often called the “mini-app” approach. A rudimentary client running on Windows is created based on the generated C#/.NET code from the Mobile Business Object (MBO) model. This client can be started multiple times using a test harness. For this document, HP LoadRunner is used as the test harness.

Note that with this approach, the device-side performance metrics such as UI rendering, Object Queries, etc., are not tested. Only the server-side components including SMP servers, backend systems and load balancers, are tested.

2.2 Types of Load Tests

Load tests, depending on the test goal, can be categorized into the following three types:

- Scalability test -- helps you understand a software system’s performance behavior under various loads. Typically, a scalable system’s response time should increase only slightly (or even stay unchanged) when the number of concurrent users increases within a reasonably large range. Typically, there is no fixed response time KPI set for such tests.
- Sizing verification test -- verifies that the system was sized properly and the response time under the estimated load is within an acceptable range, typically defined by business requirements. Sizing verification test is usually conducted prior to system go-live.
- Stress test -- finds out where the system breaks. For example, under what size of load does the system response time increase dramatically when the number of concurrent users increases only slightly. A stress test usually consists of a series of tests at various loads, in order to determine where the inflection point is on the response time vs. number of users curve.

Carefully consider what type of load test you are conducting, and prepare the test plan accordingly.

2.3 Test Preparation

SAP recommends that you prepare for testing by:

- Ensuring the applications are functionally tested and tuned for single user access prior to load testing.
- Verifying that the test system and the applications are identical to the production system. In other words, the server hardware, software versions, network bandwidth, etc., should be the same as those of the production system. This is usually the QA system or pre-production system. If there are additional components in the system landscape, such as Relay Server or load balancer, they should have a similar setup as in the production system as well.
- Preparing test users and test data. From the load testing perspective, the test data may contain fillers but it is important to make sure that each test user get / send realistic amount of
data during synchronizations. You may consider adding certain flags to the test data, so that the test data can be easily identified and cleaned up after the load tests.

- Fine-tuning the SMP server and its supporting components such as Relay Servers with baseline settings. For SMP 2.3, see the details about the baseline performance-related settings below:
  - Performance Tuning and Testing

For other versions of SMP, refer to the corresponding SMP product documentation for details.
3. Create the Skeleton of the Test Script

This chapter walks you through the steps to build the skeleton of a load test script project, based on a simple test scenario. The project is created by LoadRunner Virtual User Generator, and then edited within Visual Studio to add the generated code from the Mobile SDK and the referenced libraries. A console application is created to instantiate the test flow so that it can run in the Visual Studio environment.

1. Load Test Scenario
2. Generate Client Code
3. Create a C# .Net Project with LoadRunner VuGen
4. Edit the Vuser Project with Microsoft Visual Studio 2012
5. Copy the Generated Code to the Project
6. Create a Runnable Console Application
7. Preparation before Creating Test Scripts

3.1 Load Test Scenario

The procedure in this document uses an application named SMP101 as an example to explain how to conduct a load test against RBS applications on SMP 2.3 from end to end. The steps to create the MBO model of this application can be found in the Appendix section.

There are three MBOs in this model: Customer, Product and Sales_order. Customer and Product are in the SG_MasterData synchronization group, and Sales_order is in the SG_Order synchronization group.

Within the anticipated 10-minute peak load, there are 25 concurrent users who synchronize with SG_MasterData and SG_Orders once and then create five sales orders (i.e. synchronizing with SG_Orders).

The test should run via a Relay Server so that the load is equally distributed among all the SMP servers.
3.2 Generate Client Code

Generate Win32 C# code from the SAP Mobile SDK based on the application’s MBO model. Check these options at the bottom of the dialog:

- Generate metadata classes
- Including object manager classes

![Figure 1: Generate C# Code](image)

3.3 Create a C# .Net Project with LoadRunner VuGen

Launch LoadRunner Virtual User Generator (VuGen), and create a new script. Select .NET as the protocol and name the script `SMP101Script`.  

3.4 Edit the Vuser Project with Microsoft Visual Studio 2012

In VuGen, click 🔄 on the toolbar to open the project in Microsoft Visual Studio 2012 for Desktop.

3.4.1 Add Runtime References

Copy the following DLLs from the generated C# code directory to the newly-created Vuser project’s root directory. Putting them in the root directory makes sure the executable is able to find the runtime dependencies when running in the LoadRunner environment.

- sup-client.dll
- CMessagingClient.2.3.0.dll
- ECTrace.dll
- libeay32.dll
- ssleay32.dll
- TravelerLib.dll
- zlib1.dll
- iAnywhere.Data.UltraLite.dll
- ulnet12.dll
- mlcrsa12.dll
- mlczlib12.dll

**NOTE:** Set Copy to Output Directory to Copy Always. This configuration satisfies runtime reference requirements when running and testing the code as a console application later in this document.
3.4.2 Change Target Framework to 3.5 (optional)

If you are using a version of LoadRunner prior to 11.50, perform these steps:
1. Right-click the Script project, and click Properties.
2. Change the Target Framework to .NET Framework 3.5.
Note: make sure that the "var" keyword is not used anywhere in the script. In other words, types of all variables must be declared explicitly.

3.4.3 Add Compile-time References

For the Script project, add compile-time references to the following:

- iAnywhere.Data.UltraLite.dll
- sup-client.dll
- System
- System.Data
3.5 Copy the Generated Code to the Project

Create a folder in the project, for example, “mbo”, and copy all the generated C# code from Mobile SDK into this folder.
3.6 Create a Runnable Console Application

Visual Studio provides an excellent IDE for creating and debugging C# applications; therefore it is recommended to create most of the code in Visual Studio and run it with single user successfully, before moving the project back into LoadRunner VuGen.

To do this, follow the steps below:

1. Create a console application under the same solution and name it ConsoleTest.
2. If the LoadRunner version to be used is below 11.50, change the Target Framework of the ConsoleTest project to .NET Framework 3.5, as the Script project, and remove Microsoft.CSharp from the list of references.
3. Reference the project Script in the ConsoleTest project.
Figure 7: Reference the Script project from the ConsoleTest project

4. Set the console project as the startup project of the solution.

Figure 8: Set the ConsoleTest project as the startup project
5. All the test scripts are placed in the vuser_*.cs files. In order to run the test script from a console application, we simply need to instantiate VuserClass in Program.cs, and run the methods vuser_init(), Action() and vuser_end() sequentially. Here is a code snippet of Program.cs:

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleTest
{
    class Program
    {
        static void Main(string[] args)
        {
            Script.VuserClass myvuser = new Script.VuserClass();
            myvuser.vuser_init();
            myvuser.Action();
            myvuser.vuser_end();
            Console.ReadKey();
        }
    }
}
```

3.7 Preparation before Creating Test Scripts

`VuserClass` is a partial class that spans across four files:

- vuser_init.cs
- Action.cs
- vuser_end.cs
- globalsFixed.cs

3.7.1 Add Namespaces Used in the Test Script

Add namespaces used in the script in each of the above files, particularly, the namespace (`SMP101` in this example) specified when generating the C# code from the Mobile SDK. In this example, the following namespaces are added:

```csharp
vuser_init.cs
using System;
using System.IO;
using System.Threading;
using System.Collections.Concurrent;
using Sybase.Mobile;
using Sybase.MessagingClientLib;
using Sybase.Persistence;
using Sybase.Collections;
```
3.7.2 Toggle between Visual Studio and LoadRunner

You can easily toggle between the LoadRunner runtime environment and the Visual Studio runtime environment.

1. In globalsFixed.cs file, comment out either of the two lines in bold. When running in the Visual Studio environment, comment out the second line, i.e. the line that instantiates the `LoadRunner.LrApi` object as well as declaring the variable `lr`. When running in the LoadRunner environment, comment out the first line, i.e. the line that declares the variable `lr`.

```csharp
namespace Script
{
    public partial class VuserClass
    {
        private LoadRunner.LrApi lr;
        //private LoadRunner.LrApi lr = new LoadRunner.LrApi(); // Initialize LR-API Interface
        public void DATASET_XML(int arg)
        {
        }
    }
}
```

2. Add a flag named `isRunninginLR` in vuser_init.cs to indicate whether the script is running in LoadRunner or in Visual Studio.

Use this flag to control whether to invoke any LoadRunner-specific API calls, such as `lr.start_transaction()`, so that only one code base needs to be maintained for running in both Visual Studio and LoadRunner. Here is a code snippet of vuser_init.cs:

```csharp
namespace Script
{
    class VuserInit
    {
        public void VUSER_INIT()
        {
            isRunninginLR = false;
            // only execute LR specific code if in LR
            if (isRunninginLR) {
                lr.start_transaction();
            }
        }
    }
}
```
How to Load Test RBS Applications with HP LoadRunner

May 2013

3.7.3 Copy Helper Classes into the Project

1. Copy ApplicationCallbackHandler.cs and SMPCallbackHandler.cs into the project. The source code of those file can be found in the Appendix section of this document.

   - Class ApplicationCallbackHandler inherits DefaultApplicationCallback, and tracks the detailed status of registration and connection by implementing OnConnectionStatusChanged() and OnRegistrationStatusChanged(). It then raises an event and passes along the status with the event.

   - Class SMPCallbackHandler inherits DefaultCallbackHandler, and tracks the detailed status of synchronizations by implementing OnSynchronize(), OnReplaySuccess() and OnReplayFailure(). It then raises an event and passes along the status with the event.

2. Replace the namespace value of SMP101 with your app’s namespace in these files accordingly, if you are testing another application.
4. Implement the Test Logic

Eventually when the script runs in the LoadRunner environment, all initialization code in the `vuser_init()` are executed only once at the beginning of a test run for each Vuser, and all application logic in `Action()` are executed a number of iterations for each Vuser, and the cleanup code in `vuser_end()` are executed only once before the end of a test run for each Vuser.

This chapter goes through the steps to implement the test logic in VuserClass and run a single-user test with Visual Studio. A few critical techniques related to status tracking and synchronization response time measurement are also discussed here in detail.

1. **vuser_init - Initialization Logic**
2. **Track Status with Callback Handlers**
3. **Action – Test Logic**
4. **vuser_end – Clean-up Logic**
5. **Run Single User Test with Visual Studio**

4.1 vuser_init - Initialization Logic

The initialization logic should usually contain application registration and initial synchronization. Here is a code snippet of the initialization logic:

```csharp
public partial class VuserClass
{
    public bool isRunninginLR = false;
    public bool IsRegisterRunning;
    public bool IsSyncRunning;
    public bool IsSyncSuccessful;
    public bool IsConnecting;
    public bool IsOperationReplaySuccessful;

    private String HOST = "relayserver"; // Relay Server host
    //private int PORT = 5001; // SMP Port
    private int PORT = 80; // Relay Server Port
    private String DOMAIN = "test";

    private String APP_ID = "SMP101";
    private String SECURITY = "nosec"; // Security setting according to the SMP setup
    private String USERNAME = "testuser";
    private String PASSWORD = "abcd1234";
    private String NET_PROTOCOL = "http";
    private String URL_PREFIX = "/rs/client/rs_client.dll";

    private String MBSFarmID = "mbs.test.sap";
    private String RBSFarmID = "rbs.test.sap";

    // The Messaging Client Application
```
private Sybase.Mobile.Application app;

private String USER_IDENTIFIER;
private String USER_DATA_DIRECTORY;
private String USER_DATA_DIRECTORY_BASE = "C:\LoadTestResults\SMP101";
private int sales_rep;
private String region;
private int customer_id;
private int test_run_no = 1;

public int vuser_init()
{
    if (isRunningInLR)
    {
        sales_rep = 299; //To be parameterized by LoadRunner
        region = "Eastern"; //To be parameterized by LoadRunner
        USER_IDENTIFIER = "Client" + sales_rep;
        USER_DATA_DIRECTORY = USER_DATA_DIRECTORY_BASE + "\" + USER_IDENTIFIER;
    }
    else
    {
        sales_rep = 299;
        region = "Eastern";
        USER_IDENTIFIER = "Client" + sales_rep;
        USER_DATA_DIRECTORY = USER_DATA_DIRECTORY_BASE + "\" + USER_IDENTIFIER;
    }

    // initialize client library
    app = Sybase.Mobile.Application.GetInstance(USER_IDENTIFIER,
        USER_DATA_DIRECTORY);
    app.ApplicationIdentifier = APP_ID;

    // setup application messaging callbacks
    ApplicationCallbackHandler appCallback = new ApplicationCallbackHandler();
    appCallback.OnApplicationMessage += HandleAppOnSyncMessage;
    app.ApplicationCallback = appCallback;

    // setup SUP messaging callbacks
    SMPCallbackHandler smpCallback = new SMPCallbackHandler();
    smpCallback.OnSyncMessage += HandleSMPOnSyncMessage;
    SMP101DB.RegisterCallbackHandler(smpCallback);

    ConnectionProfile cp = SMP101DB.GetConnectionProfile();
    cp.SetProperty("databaseFile", USER_DATA_DIRECTORY + "\data.udb");
    //cp.Save();

    ConnectionProperties connProps = app.ConnectionProperties;
    LoginCredentials loginCredentials = new LoginCredentials(USERNAME,
        PASSWORD);
    connProps.LoginCredentials = loginCredentials;
connProps.ServerName = HOST;
connProps.PortNumber = PORT;
connProps.SecurityConfiguration = SECURITY;
connProps.NetworkProtocol = NET_PROTOCOL;
connProps.FarmId = MBSFarmID;
connProps.UrlSuffix = URL_PREFIX;

SMP101DB.SetApplication(app);
mainThreadId = Thread.CurrentThread.ManagedThreadId;

if (app.RegistrationStatus != RegistrationStatus.REGISTERED)
{
    LogMessage("Registering device...");
    IsRegisterRunning = true;
    IsConnecting = true;
    app.RegisterApplication();
    while (IsRegisterRunning)
    {
        LogMessage("Waiting for registration to complete...");
        _signal.WaitOne(1000);
        while (_MessageQueue.TryDequeue(out messageItem))
        {
            LogMessage(messageItem);
        }
    }
}
else
{
    IsConnecting = true;
    app.StartConnection();
    while (IsConnecting)
    {
        LogMessage("Waiting for application to connect...");
        _signal.WaitOne(1000);
        while (_MessageQueue.TryDequeue(out messageItem))
        {
            LogMessage(messageItem);
        }
    }
}

ConnectionProfile syncProfile = SMP101DB.GetSynchronizationProfile();
syncProfile.ServerName = HOST;
syncProfile.DomainName = DOMAIN;
syncProfile.Credentials = loginCredentials;
syncProfile.PortNumber = PORT;
syncProfile.NetworkProtocol = NET_PROTOCOL;
syncProfile.StreamParams.Url_Suffix = URL_PREFIX + "" + RBSFarmID;
syncProfile.AsyncReplay = false;
SMP101DB.EnableChangeLog();
In the LoadRunner environment, multiple Vusers, i.e. multiple app instances, are launched from the same Load Generator. Therefore it is critical to distinguish the app instances by using the overloaded method:

```csharp
```

In this sample script, the variable `USER_IDENTIFIER` is set to “Client” + `sales_rep` (a unique value), and `USER_DATA_DIRECTORY` is set to `USER_DATA_DIRECTORY_BASE + USER_IDENTIFIER`. When conducting tests in which no unique user identifier is available, use an arbitrary unique ID, for example, a manually-created line number in a parameter file. For more details about parameterization, see section 5.5.

During the load test, multiple Vusers are launched from the same directory, i.e. root folder of the Script project. It is therefore vital to store each Vuser’s (i.e. each app instance’s) UltraLite database in a unique location. Use the below method to store each Vuser’s database file in a different path:

```csharp
ConnectionProfile.SetProperty("databaseFile", USER_DATA_DIRECTORY + "\data.udb")
```

Do not store the device-side UltraLite database files in the subfolders of the script, as by default they would be automatically deleted by LoadRunner after running the script on a remote Load Generator. This step should be done before calling `SMP101DB.SetApplication(app)`.

After setting the `databaseFile` property in the `ConnectionProfile`, do **NOT** perform `ConnectionProfile.Save()`. This method saves the configuration to a file in the LoadRunner script’s root directory. When running multiple Vusers at the same time, multiple Vusers could attempt to write to this file at the same time, thus causing failures.

### 4.2 Track Status with Callback Handlers

It is very important for the script to be able to track the status of application registration, connection establishment and synchronization, in order to run certain LoadRunner API calls based on the retrieved status, including outputting log messages, measuring operation replay time within a synchronization call, etc.
The VuserClass implements two event handlers (contained in vuser_init.cs), HandleAppOnSyncMessage() and HandleSMPOnSyncMessage(), to handle events raised from ApplicationCallbackHandler and SMPCallbackHandler, respectively. HandleAppOnSyncMessage() maintains the flag variables IsRegisterRunning and IsConnecting. HandleSMPOnSyncMessage() maintains the flag variables IsSyncRunning, IsSyncSuccessful and IsOperationReplaySuccessful, and records the data that is being synchronized or replayed. The flag variables are used to control script flow, and to set LoadRunner transactions to PASS/FAIL accordingly.

Note that event handlers above are often called from a background thread, while many LoadRunner APIs, e.g. lr.message() which outputs log messages, can only be invoked from the main thread. Therefore this sample script implements a cross-thread messaging mechanism for the background thread to notify the main thread of new status messages. The main thread can then take appropriate actions based on the status.

The cross-thread messaging mechanism is based on the thread-safe FIFO collection System.Collections.Concurrent.ConcurrentQueue<T> (_MessageQueue) and the event-notification facility System.Threading.AutoResetEvent (_signal).

Be aware that ConcurrentQueue<T> is only available on .NET Framework 4 or above, so if you are testing with LoadRunner versions that do not support .NET Framework 4 (LoadRunner 9.52 for example), consider using System.Collections.Generic.Queue<T> (a FIFO collection that is not thread-safe) together with a locking mechanism.

Here is the code snippet that implements the event handlers and the cross-thread messaging mechanism. Note that the lines of the cross-thread messaging mechanism are marked in bold.

```csharp
public partial class VuserClass
{
  
  private readonly AutoResetEvent _signal = new AutoResetEvent(true);
  private readonly ConcurrentQueue<string> _MessageQueue = new ConcurrentQueue<string>();
  private string messageItem;
  private int mainThreadId = 0;

  public int vuser_init()
  {
    mainThreadId = Thread.CurrentThread.ManagedThreadId;
    if (app.RegistrationStatus != RegistrationStatus.REGISTERED)
    {
      LogMessage("Registering device...");
      IsRegisterRunning = true;
      IsConnecting = true;
      app.RegisterApplication();
      while (IsRegisterRunning)
      {
        LogMessage("Waiting for registration to complete...");
        _signal.WaitOne(1000);
      }
    }
}
```
while (_MessageQueue.TryDequeue(out messageItem))
{
    LogMessage(messageItem);
}
}
else
{
    IsConnecting = true;
    app.StartConnection();
    while (IsConnecting)
    {
        LogMessage("Waiting for application to connect...");
        _signal.WaitOne(1000);
        while (_MessageQueue.TryDequeue(out messageItem))
        {
            LogMessage(messageItem);
        }
    }
    ...
}

internal void LogMessage(string message)
{
    if (mainThreadId == Thread.CurrentThread.ManagedThreadId)
    {
        if (isRunninginLR)
            lr.message(DateTime.Now.ToString("hh:mm:ss.ffffff ") + message);
        else
            Console.WriteLine(DateTime.Now.ToString("hh:mm:ss.ffffff " + message);
    } else
    {
        _MessageQueue.Enqueue(message);
        _signal.Set();
    }
}

void HandleAppOnSyncMessage(string s, CallbackType cbt, CallbackSource cbs,
object o)
{
    object[] objects = (object[])o;
    LogMessage(cbs.ToString() + ": " + cbt.ToString());
    switch (cbt)
    {
        case CallbackType.CONNECTED:
            IsConnecting = false;
            LogMessage("Connection established.");
            break;

        default:
            LogMessage("Unknown callback type.");
            break;
    }
}
```csharp
private void HandleSMPOnSyncMessage(string s, CallbackType cbt, CallbackSource cbs, object o)
{
    try
    {
        LogMessage(s);
        if (cbt == CallbackType.IMPORT_DATA)
        {
            if (o != null)
            {
                if (o.GetType() == typeof(LogRecordImpl))
                {
                    LogRecordImpl log = (LogRecordImpl)o;
                    LogMessage(
                        string.Format(
                            "Level: {0}, Code: {1}, Operation: {2}, Message: {3}",
                }
            }
        }
        else if (cbt == CallbackType.SYNC_STATUS)
        {
            if (IsSyncRunning && (o.GetType() == typeof(Sybase.Persistence.SynchronizationContext)))
            {
                Sybase.Persistence.SynchronizationContext context = (Sybase.Persistence.SynchronizationContext)o;
                if (context.Status == SynchronizationStatus.FINISHING)
                {
```
IsSyncSuccessful = true;
IsSyncRunning = false;
}
else if (context.Status == SynchronizationStatus.ERROR)
{
    IsSyncSuccessful = false;
    IsSyncRunning = false;
}
}
else if (cbt == CallbackType.REPLAY_SUCCESS)
{
    IsOperationReplaySuccessful = true;
    ProcessReplayResult(true, o, cbt);
}
else if (cbt == CallbackType.REPLAY_FAILED)
{
    IsOperationReplaySuccessful = false;
    ProcessReplayResult(false, o, cbt);
}
else if (cbt == CallbackType.REPLAY_ABORTED)
{
    IsOperationReplaySuccessful = false;
    ProcessReplayResult(false, o, cbt);
}
}
catch (Exception e)
{
    LogMessage(String.Format("Exception Message=" + e.Message));
    LogMessage(String.Format("Exception StackTrace=" + e.StackTrace));
}
}
private void ProcessReplayResult(bool success, object o, CallbackType cbt)
{
    if (o.GetType() == typeof(Sales_order))
    {
        LogMessage("MBO entity replayed:");
        Sales_order mboSO = (Sales_order)o;
        LogMessage(string.Format("Id = {0}, Cust_Id = {1}, Order_date = {2}, Fin_code_id = {3}, Region= {4}, Sales_rep = {5}",
            mboSO.Id, mboSO.Cust_id, mboSO.Order_date, mboSO.Fin_code_id, mboSO.Region, mboSO.Sales_rep));
    }
}
4.3 Action – Test Logic

In this section, a series of tasks that represent a typical user’s daily work are created. Usually the logic consists of certain upsert operations and synchronizations of particular synchronization groups. Each synchronization call is defined as a LoadRunner transaction, and each Operation Replay is defined as a sub-transaction of the synchronization transaction, if applicable. LoadRunner automatically measures the duration of the transactions and sub-transactions when executing the test.

4.3.1 Code Snippet of Action.cs

The following code snippet based on the SMP101 app covers downloading master data and creating five sales orders.

```csharp
public int Action()
{
    if (isRunninginLR)
    {
        lr.rendezvous("InitializationComplete");
        lr.think_time(450);
    }
    // download syncs
    SyncMasterData();
    SyncOrders();

    // Create 5 sales orders in a max time span of 150 seconds
    for (int i = 0; i < 5; i++)
    {
        if (isRunninginLR)
            lr.think_time(30);
        SyncOrders_Upsert(i);
    }
    return 0;
}

private void SyncMasterData()
{
    string Sync_Group_Name = "SG_MasterData";
    string transactionName = "SMP101_" + Sync_Group_Name;
    LogMessage("Start syncing " + Sync_Group_Name + "...");
    IsSyncRunning = true;
    IsSyncSuccessful = false;
    if (isRunninginLR)
        lr.start_transaction(transactionName);
    SMP101DB.Synchronize(Sync_Group_Name);
    if (IsSyncSuccessful)
    {
        LogMessage("Sync successful: " + transactionName);
    }
    if (isRunninginLR)
        lr.think_time(30);
}
```
private void SyncOrders()
{
    string Sync_Group_Name = "SG_Orders";
    string transactionName = "SMP101_" + Sync_Group_Name;
    Sales_orderSynchronizationParameters so_sp = Sales_order.SynchronizationParameters;
    so_sp.ParamRegion = region;
    so_sp.Save();
    LogMessage("Start syncing " + transactionName + "...");
    IsSyncRunning = true;
    IsSyncSuccessful = false;
    if (isRunninginLR)
        lr.start_transaction(transactionName);
    SMP101DB.Synchronize(Sync_Group_Name);
    if (IsSyncSuccessful)
    {
        LogMessage("Sync successful: " + transactionName);
        if (isRunninginLR)
            lr.end_transaction(transactionName, lr.PASS);
    }
    else
    {
        if (isRunninginLR)
            lr.end_transaction(transactionName, lr.FAIL);
        LogMessage("Sync failed: " + transactionName);
    }
}

private void SyncOrders_Upsert(int iterationNo)
{
    string Sync_Group_Name = "SG_Orders";
    string transactionName = "SMP101_" + Sync_Group_Name + "_Upload";
    if (isRunninginLR)
        customer_id = lr.eval_int("{paramCustId}");
    else
        customer_id = 200;
    Sales_orderSynchronizationParameters so_sp = Sales_order.SynchronizationParameters;
    so_sp.ParamRegion = region;
    so_sp.Save();
Sales_order so = new Sales_order();
if (isRunninginLR)
    so.Id = test_run_no * 10000 + lr.vuser_id * 100 + iterationNo;
else
    so.Id = test_run_no * 10000 + iterationNo;
so.Cust_id = customer_id;
so.Order_date = DateTime.Now;
so.Fin_code_id = "r1";
so.Region = region;
so.Sales_rep = sales_rep;
so.Create();
so.SubmitPending();

LogMessage("Start syncing " + transactionName + "...");
IsSyncRunning = true;
IsSyncSuccessful = false;
IsOperationReplaySuccessful = false;
GenericList<ISynchronizationGroup> syncGroups = new GenericList<ISynchronizationGroup>();
syncGroups.Add(SynchronizationGroup.GetInstance(Sync_Group_Name));
if (isRunningInLR)
    lr.start_transaction(transactionName);
SMP101DB.BeginSynchronize(syncGroups, "");
while (IsSyncRunning)
{
    LogMessage("Waiting for sync to complete...");
    _signal.WaitOne(1000);
    while (_MessageQueue.TryDequeue(out messageItem))
    {
        if (isRunningInLR)
        {
            if (messageItem.Contains("STATUS=STARTING"))
            {
                lr.start_sub_transaction("SalesOrder_BackendReplay",
transactionName);
            }
            if (messageItem.Contains("OnReplaySuccess"))
            {
                lr.end_sub_transaction("SalesOrder_BackendReplay",
lr.PASS);
                lr.start_sub_transaction("SalesOrder_DownloadAfterReplay",
transactionName);
            }
            if (messageItem.Contains("OnReplayFailure"))
            {
                lr.end_sub_transaction("SalesOrder_BackendReplay",
lr.FAIL);
                lr.start_sub_transaction("SalesOrder_DownloadAfterReplay",
transactionName);
            }
        }
    }
}
The id attribute of the Sales_order MBO corresponds to the primary key of the sales_order table in the backend database. The variables test_run_no, vuser_id and iterationNo are used to make sure that the test does not generate duplicate keys in the backend database. Therefore it is important to increase the test_run_no by 1 before each test run.

The test logic used in a load test needs to simulate the load that a real user would generate against the SMP server. However it does not have to perfectly resemble the actual app. As long as it generates the same amount of load on the server and does not break any server-side restrictions, you can use filler data. For example, if there is an MBO that contains location information (longitude and latitude) and the location information does not affect how much data is being synchronized, data fillers of 0 can be used for the location information.

### 4.3.2 Measure Synchronization Response Time

There are multiple forms of synchronization API calls. SMP101DB.synchronize(SyncGroup) is a blocking call (synchronous call), while SMP101DB.BeginSynchronize() is a non-blocking asynchronous call. For download-only synchronizations, the blocking form should be used to simplify measurement of response time. Just add the start/stop transaction statements before and after the synchronize call like below, and the synchronization duration is then automatically measured by LoadRunner:
How to Load Test RBS Applications with HP LoadRunner

May 2013

```csharp
lr.start_transaction(transactionName);
SMP101B.Synchronize(Sync_Group_Name);
lr.end_transaction(transactionName);
```

Add appropriate Think Time between transactions to simulate the idle time between two user actions.

If you plan to run multiple scripts at the same time, add a namespace value, e.g. SMP101, to the names of the LoadRunner transactions in order to avoid potential name conflicts when reporting transaction response times. Do not use “.” as the delimiter, as it is reserved by LoadRunner for sub-transactions.

4.3.3 Break Down Synchronization Time into Operation Replay and Download Sync

For synchronizations that contain operation replays, by default the operation replay is executed asynchronously, irrespective of whether a blocking synchronize call or a non-blocking synchronize call is performed. In order to precisely measure how much time in total the synchronization takes including operation replay, set AsyncReplay property of SynchronizationProfile to false (done in vuser_init.cs), which makes operation replay synchronous.

Keep in mind that in the device app, synchronization is usually performed asynchronously, and the operation replay is often executed asynchronously as well. So the time measured here should only be used to understand SMP and backend server performance, and not used to understand the end user experience.

A synchronization call with operation replay consists of two steps: operation replay and a download sync. It is very useful to break it down and measure these two steps separately, in order to tell whether a performance problem lies in the backend system or the SMP server. In this case, the asynchronous form of the synchronization API should be used, i.e. BeginSynchronize(). Make use of the synchronization status messages sent across via the cross-thread messaging mechanism (see details in section 4.2), and start/stop sub-transactions for operation replay and the download sync accordingly. In the SyncOrders_Upsert() method, two sub-transactions are created: SalesOrder_BackendReplay and SalesOrder_DownloadAfterReplay.

4.4 vuser_end – Clean-up Logic

In the clean-up logic, always stop the connection to SMP. Optionally, if you would like the next test run to start from scratch (re-register, redo initial sync, etc), unregister the application and delete all the directories created by the tests. The following is a code snippet of the vuser_end() method:

```csharp
public partial class VuserClass {
    public int vuser_end() {
        if (app.ConnectionStatus == ConnectionStatus.CONNECTED)
            app.StopConnection();
    }
}
```
4.5 Run Single User Test with Visual Studio

Once the test script is finished, run it in Visual Studio and debug the application if anything is not running as expected. Verify the data by dumping data before/after synchronization using Object APIs and by inspecting the UltraLite database with Sybase Central. Once you are satisfied that the script is working completely as expected, it is time to move the project over to LoadRunner.

When generating the C# code from the MBO model (see details in section 3.2), you have checked the option Generate metadata classes and Including object manager classes. These additional classes can help you understand the table structure in the device UltraLite database when inspecting the database with Sybase Central.
Figure 9: Run single user test with Visual Studio
5. Continue to Build the Script with LoadRunner

This chapter continues to build the script in LoadRunner VuGen to make the script ready for multi-user tests. It first switches the script from Visual Studio to LoadRunner VuGen, and then parameterizes the test script for multi-user execution.

1. Convert Project from Visual Studio
2. Switch the Run-time Environment
3. Configure Run-time Settings for the Script
4. Run the Script in VuGen
5. Parameterize the Test Script

5.1 Convert Project from Visual Studio

If you are running the scripts using LoadRunner versions below 11.50, open the solution file Script.sln with a text editor and change the version number "12.00" in the first line "Microsoft Visual Studio Solution File, Format Version 12.00" to "9.00".

5.2 Switch the Run-time Environment

Open the project in LoadRunner VuGen. Note that the referenced libraries and the generated C# code files are not listed in the VuGen's Solution Explorer.

To switch the runtime environment from Visual Studio to LoadRunner, perform the following steps:

1. In globalsFixed.cs, comment out the declaration of lr, and uncomment the line that declares AND initializes lr.

```csharp
namespace Script
{
    public partial class VuserClass
    {
        //private LoadRunner.LrApi lr;
        private LoadRunner.LrApi lr = new LoadRunner.LrApi(); // Initialize LR-API Interface
        public void DATASET_XML(int arg)
        {
        }
    }
}
```

2. In vuser_init.cs, set isRunningInLR to true.

5.3 Configure Run-time Settings for the Script

In Solution Explorer, double-click Run-time Settings, and the following window pops up:
1. In the General -> Miscellaneous screen, set Run Vuser as a process, and uncheck Define each action as a transaction.

2. In the General -> Log screen, check Enable logging, and select Always send messages. Choose Extended log, and check Parameter substitution and Advanced trace.
5.4 Run the Script in VuGen

If everything has been done correctly, the Vuser can be compiled and played from within LoadRunner VuGen. Click on the play button on the toolbar to run the script. All the logs and messages can be found in the Output window at the bottom. Verify the log and the UltraLite database to make sure the script is working as expected.

![Running the script in VuGen](image)

Figure 12: Running the script in VuGen

5.5 Parameterize the Test Script

The script that we have created so far works well in single user tests with some hard-coded values for the synchronization parameter and fields for newly-created MBO entities. However if we want to run the test with hundreds or thousands of users to simulate real users, in most cases each individual user should have different variables. LoadRunner provides parameterization capability to customize your script to run in a multi-user environment.

Take the field *Sales_rep* in the MBO *Sales_order* in the above script as an example. It is important to set a distinct value to this parameter to distinguish each virtual user. To parameterize the variables including *sales_rep*, follow the steps below:

1. Highlight the value set to *sales_rep*, right-click it and select *Replace with a Parameter* in the context menu.
2. In the pop-up window, set the parameter’s name, e.g. to `paramSales_rep`, and set the Parameter type to `File`.

3. Click `Properties`, and enter the values for all Vusers. Set `Select next row to Unique`, and `Update value on` to `Once`. This ensures that every Vuser get a unique value from the parameter list, provided that the parameter file contains enough values for all the Vusers.

4. Make sure that after parameterization, this line should look like below:

   ```java
   sales_rep = lr.eval_int("{paramSales_rep}");
   ```

   You can store more than one parameter in the same parameter file by specifying which column corresponds to which parameter. We can go ahead to parameterize the variable `region` by following a similar procedure, and the end results should look like the following:

   ```java
   sales_rep = lr.eval_int("{paramSales_rep}");
   region = lr.eval_string("{paramRegion}");
   ```
**Figure 15: Parameter properties for paramSales_rep**

<table>
<thead>
<tr>
<th>Sales Rep</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eastern</td>
</tr>
<tr>
<td>2</td>
<td>Eastern</td>
</tr>
<tr>
<td>3</td>
<td>South</td>
</tr>
<tr>
<td>4</td>
<td>Eastern</td>
</tr>
<tr>
<td>5</td>
<td>Eastern</td>
</tr>
<tr>
<td>6</td>
<td>Eastern</td>
</tr>
<tr>
<td>7</td>
<td>Eastern</td>
</tr>
</tbody>
</table>

- **File path:** sales_rep.dat
- **Parameter type:** File
- **Column delimiter:** Comma
- **First data line:** Select column by number
- **Update value on:** Once
- **Allocate values in the browser:**
  - Allocate values for each
  - Automatically allocate block size

---
Figure 16: Parameter properties for paramRegion

Note that for the second parameter, i.e. paramRegion, we set the column number to 2, and Select next row to Same line as paramSales_rep. Together with the paramSales_rep parameter setting, this makes sure that each Vuser uses one line of data to feed the variables sales_rep and region, respectively. If your script uses a different logic when retrieving parameters, configure the parameter settings accordingly.

Similarly, parameterize customer_id in the SyncOrders_Upset() method in Action.cs. The parameter should look like this:
Note that *Select next row* is set to *Sequential* and *Update value on* is set to *Each occurrence*. This means that every time the parameter is accessed, a new value is retrieved from the parameter file. When out of values, LoadRunner goes back to the beginning of the parameter file to retrieve the first line of data.

After parameterization, this section of code should look like the following:

```plaintext
if (isRunningInLR)
    customer_id = lr.eval_int("{paramCustId}");
else
    customer_id = 200;
```

Simulate the parameters by clicking on *Simulate Parameter*..., and make sure the parameters can be retrieved correctly. Re-run the test script after creating the necessary parameters, and verify the parameters are retrieved as expected and the script works properly.
Figure 18: VuGen log with parameter substitution
6. Perform Load Tests with LoadRunner Controller

This chapter steps you through the process to setup a load test scenario in LoadRunner Controller using the script built from the previous chapter.

1. Prepare the Test Environment
2. Create a Test Scenario
3. Run the Test Scenario

6.1 Prepare the Test Environment

On the LoadRunner Generator machines:
- Delete any existing old data directories
- Delete any old device keys in the following registry locations as these entries may interfere with the Vusers.
  - HKEY_CURRENT_USER/Sybase/MessagingClient
  - HKEY_CURRENT_USER/Sybase/MessagingClientLib

6.2 Create a Test Scenario

6.2.1 Create a New Test Scenario

Launch LoadRunner Controller. Create a new Scenario and add the Vuser script to the scenario.

![Image of LoadRunner Controller interface for creating a new scenario](image_url)

Figure 19: Create a load test scenario
If you would like to precisely control the number of Vusers among scripts, uncheck *Use the Percentage Mode to distribute the Vusers among the scripts*. This enables Vuser Group Mode, which may become very useful when running multiple scripts on multiple Load Generators.

### 6.2.2 Configure Run-time Settings

Highlight the script in the scenario, and open the Run-time Settings window by clicking the button on the toolbar. The scenario has automatically inherited the run-time settings of the script from VuGen; however you can override those settings by setting them again in the scenario.

1. Set *Miscellaneous* -> *Run Vuser as a process*.
2. Uncheck *Miscellaneous* - > *Define each action as a transaction*.

#### Figure 20: Runtime settings -> Miscellaneous

3. In *General* - > *Log*, check *Enable Logging*, and select *Always send messages*. Choose *Extended log*, and select *Parameter substitution* and *Advanced trace*.
4. In *General* -> *Think Time*, select *Replay think time* -> *Use random percentage of recorded think time*, and set the percentage from 1% to 100%.
Figure 22: Run-time Settings -> Think Time

5. If you would like the test to run a certain number of iterations for each Vuser, set the number in **General -> Pacing -> Number of Iterations**.
Figure 23: Run-time Settings -> Pacing

6.2.3 Configure Test Schedule

Configure the schedule for the test in the Design tab of the scenario.

Figure 24: Load test schedule

1. Specify the number of Vusers and how fast to ramp up Vusers. Click Start Vusers to specify the required values.
2. Specify the duration of the test by clicking on *Duration*. If you would like to run a certain number of iterations (configured in run-time settings) instead of certain duration, set the *Duration* to *Run until completion*.

6.2.4 Distribute the Load to Load Generators

If no Load Generator is listed, click *Add* and add the Load Generators, for example, `localhost`.

![Figure 25: Number of Vusers and ramp-up speed](image)

![Figure 26: Test duration](image)

![Figure 27: Distribute scripts to Load Generators](image)
6.2.5 Configure OS Monitoring for the Test

Switch to the Run tab, and add OS performance metrics that you would like to collect during the load test along with the transaction response time, e.g. SMP servers’ CPU utilization, memory consumption, disk queue length, etc. See more details in section 8.1.

6.2.6 Configure Results Settings

In the main menu, go to Results -> Results Settings…, and configure where to store the test results. Give a different name for each test run. The load test result can be analyzed later by using LoadRunner Analysis.

Save the test scenario and give it a name, for example, SMP101Scenario.

6.3 Run the Test Scenario

Run the test scenario by clicking on the toolbar. You can monitor the test in real time and see the synchronization response time, among many other metrics, in the chart Trans Response Time. You can also dig into each individual Vuser by clicking on the button.
Figure 30: Running the test scenario
7. Advanced Topics

This chapter takes a deep dive into several frequently-used techniques for successful load testing in complex scenarios.

1. Run Test Scripts on a Remote Load Generator
2. Run Load Tests on Multiple Load Generators
3. Pace Vusers with Rendezvous Points
4. Use a RAM Disk
5. Load Test via IP-based Load Balancer
6. Simulate Mobile Networks in the Test
7. Clean Up Test Data

7.1 Run Test Scripts on a Remote Load Generator

When a script is to run on a remote Load Generator, the LoadRunner Controller first copies the script’s files to the remote Load Generator for local compilation and execution. However the LoadRunner Controller does not automatically recognize all the supporting files in the project, such as the SMP DLLs and the generated C# code files. Therefore we need to manually add those files. To do so, first highlight the script in the LoadRunner Controller scenario, and click on the Details button on the toolbar. A pop-up window like below appears:

![Figure 31: Add supporting files to a script](image)

Use the *Add File*... menu to add SMP DLL files and the generated C# code files.

Note: if the load test script project was originally created on a VuGen other than the one installed on the LoadRunner Controller machine, adding files in this step may fail due to hard links to the root folder of the script. In this case, open VuGen on the LoadRunner Controller machine and save a copy of the script. Using the copied script instead solves this issue.
7.2 Run Load Tests on Multiple Load Generators

Depending on the hardware capacity of the Load Generators, you may need to distribute the Vusers to multiple Load Generators. This poses a challenge when running a series of load tests that depend on existing app registration and device database files.

In a realistic test scenario, it is often necessary to run a second test based on the existing application registration and device database files (.udb files) populated from a previous test run. The second test may be running a different test case, or the same test case as before but with different number of Vusers.

When running a test with multiple load generators, each Vuser’s registration information is stored in the load generator’s local Windows registry, and its corresponding UltraLite database is stored on the load generator’s local disk drive, e.g. F:\LoadTestResults\Client\<UniqueID>\data.udb. As explained earlier in this document, here the UniqueID may come from a parameter file. When the test scenario is executed a second time or a third time, LoadRunner does not guarantee that Vusers running on this particular load generator will get this parameter again, so the Vuser that gets a certain unique ID from the parameter file may be running on a different load generator that does not have the corresponding app registration and device database. As a result, the Vuser may have to perform another registration and synchronize with SMP with an empty device database, which is of course not the intended test case.

To solve this issue, one straightforward way is to make copies of the script and configure each script to run on one Load Generator only. Follow the detailed steps below:

1. Use File -> Save As… in VuGen to copy the script. Configure the copied script to consume only a certain set of parameters by setting the First data line as below. For example, suppose you have 20 lines of parameters and you would like to run the scenario on two Load Generators. Specify the second (i.e. the copied) script to use parameters from line 11.
2. Create a LoadRunner test scenario and add both scripts. Make sure that each script runs on one Load Generator only. Assign the right number of Vusers for each script so that each Load Generator uses its own set of parameters. For the above example, assign 10 Vusers to the first script, and another 10 Vusers to the second (i.e. copied) script.
One way to avoid the entire complexity described above is to use only one Load Generator machine that is powerful enough (mainly CPU and RAM). Create a large RAM disk to store all the device DB files and folders and eliminate the I/O bottleneck on the Load Generator. See details about using RAM disk in section 7.4.
7.3 Pace Vusers with Rendezvous Points

In a typical load test with LoadRunner, it is often the case that the logic in the Vuser’s Action() method is iterated multiple times; however this may not apply to certain RBS apps. Suppose we are running a test for a read-only application which only downloads data from SMP regularly. There may not be delta data to download after a recent synchronization. If we repeat the synchronizations using LoadRunner iterations, the script will not pose a high-enough load on SMP because no new data will be transferred after the first iteration. In this case, it is often reasonable to run only one iteration for each Vuser and pace them precisely so that they all start the download synchronizations during the same time frame (i.e. peak load period).

Precise pacing can be achieved by setting up a rendezvous point at the beginning of the Action() method, i.e. after the application connection has been established and the first sync with “system” synchronization group has finished. Follow the detailed steps below:

1. Add a rendezvous point and give a randomized think time after it. This makes sure that all Vuser activities are performed within the “peak load period”. Below is the script snippet that adds the rendezvous point and a think time.

   ```java
   public int Action()
   {
       if (isRunninginLR)
       {
           lr.rendezvous("InitializationComplete");
           lr.think_time(500); //this value is close to a defined peak load period of 10 min
       }
   }
   ```

2. In the Run-time setting of the script in the LoadRunner scenario, configure a randomized think time between 1 and 100 percent. In conjunction with the think time of 500 seconds, the effect of this setting is that all Vuser activities are started within about 8 minutes, and the load is evenly distributed within the 8-minute time frame.

3. LoadRunner needs to run the script from the LoadRunner Controller once in order to recognize the rendezvous points in the scripts. Run the scenario with one user per script would suffice the need. In LoadRunner Controller, go to the menu `Scenario -> Rendezvous...`, and make sure all the rendezvous points have been recognized and the scripts and users are listed correctly.
4. Click on the button Policy…, and set the Rendezvous policy to Release when 100% of all Vusers arrive at the rendezvous. Do not use the default setting of Release 100% of all running Vusers arrive at the rendezvous as it may cause unexpected release of Vusers.

5. Configure the Timeout between Vusers to a high value such as 1000 seconds to avoid release of Vusers due to delays in the vuser_init section.

If for any reason a Vuser gets stuck during in the vuser_init stage, you can always manually release all other Vusers from the rendezvous point.
How to Load Test RBS Applications with HP LoadRunner

May 2013

52

The rendezvous point is also valid for multiple-script scenarios. If you would like to have all the scripts hitting the SMP server during the same peak load period, make sure their rendezvous names are identical across all scripts.

In order for LoadRunner to identify the rendezvous points in the scripts, the script has to be parsed successfully by LoadRunner. Certain statements, although correct in syntax, may cause the script parsing to fail. For example, putting a string expression instead of a string constant or a string variable as the input parameter of `lr.start_transaction()` causes failure in script parsing; as a result, rendezvous points in the script are not recognized by LoadRunner. Make sure the Script parse passed status is displayed in the status bar of VuGen.

In the test script, store all Vusers’ working folders and device databases on the RAM disk to improve I/O performance.

7.4 Use a RAM Disk

Running hundreds of RBS app instances simultaneously is I/O intensive on the Load Generator machine. In order to avoid the Load Generator boxes becoming the performance bottleneck, RAM disks, such as ImDisk, can be used to boost the Load Generator’s I/O performance, if the Load Generator has sufficient RAM installed.

In the test script, store all Vusers’ working folders and device databases on the RAM disk to improve I/O performance.

7.5 Load Test via IP-based Load Balancer

If you are using an IP-based load balancer in front of the SMP server nodes, it is important that the requests from the Load Generators come from different IP addresses, in order for the load balancer to
evenly distribute the load onto all the SMP server nodes during the load test. If this is the case, consider using LoadRunner’s IP Spoofing feature, by which the load generators reserve a pool of IP addresses, and the Vusers appear to come from different IP addresses.

The IP address pool can be configured by LoadRunner’s IP Wizard as below:

![Figure 38: IP Wizard](image)

For more information about LoadRunner’s IP Spoofing feature, refer to LoadRunner documentation.

7.6 Simulate Mobile Networks in the Test

So far we have been discussing tests over a high-quality local area network only. This kind of test serves the purpose well if you want to test the server-side performance only. However, if you wish to add the mobile network factors into the test in order to measure the end user experience, consider using a WAN emulator in conjunction with LoadRunner and add the following factors into the test:

- High latency
- Low bandwidth
• Lower stability (i.e. high packet loss rate)

The above factors should be configured according to a typical end user's location and the mobile network used.

7.7 Clean Up Test Data

Typically, load tests are conducted multiple times with various numbers of Vusers and test cases, and sometimes it is necessary to populate delta data into the backend system before each test run, depending on the test logic. Upsert operations in the test script may also generate a lot of data in the backend system. Overtime the data volume may grow to an extent that the Vusers are synchronizing much more data than that of normal usage scenarios. Therefore it is important to clean up the accumulated test data after a few test runs to ensure accurate test results.

Depending on the application logic and the test data, it may be a good idea to use certain flags to help administrators identify the delta test data. For example, in the SyncOrders Upsert() method, all the script-created sales orders have a very large id (primary key), comparing to the original data in the table. Backend administrators can then delete the accumulated test data based on such flags when necessary.
8. Monitor SMP during Load Tests

Load tests provide a good opportunity for SMP administrators and developers to understand system behavior under load, to pinpoint performance bottlenecks, and to find ways to improve overall performance by removing those bottlenecks.

During load tests, the SMP administrator uses various tools to collect SMP performance metrics, ranging from Operating System performance counters to synchronization-specific metrics. Sometimes a tracing session is necessary to drill down into particular long-running synchronizations.

1. Collect OS Performance Metrics
2. Monitor SMP with MobiLink Monitor
3. Monitor SMP with CA Wily Introscope

8.1 Collect OS Performance Metrics

LoadRunner is able to collect Windows performance counters during the tests and incorporate them into the test results; thereby the SMP synchronization performance charts can be analyzed together with the OS performance metrics on the same timeline. This is configured in the LoadRunner Controller when setting up the test scenario.

1. In LoadRunner Controller, go to the Run tab, double-click on System Resource Graphs -> Windows Resources in the Available Graphs pane
2. Click on Monitors -> Add Measurements… in the main menu. A pop-up window appears which allows you to select the remote Windows server (i.e. SMP server nodes, SMP Data Tier, Relay Server, backend systems if applicable) and choose the OS performance counters from that server.

Figure 40: Collecting SMP server OS performance counters
For SMP server nodes, the following performance counters are usually collected during load tests:

- Processor(_Total)\% Processor Time
- System\Context Switches/sec
- System\Processor Queue Length
- Processor Information\Interrupts/sec
- Memory\Available MBytes
- Process(dbsrv12)\% Processor Time
- Process(dbsrv12)\IO Read Operations/sec
- Process(dbsrv12)\IO Write Operations/sec
- Process(dbsrv12)\Page Faults/sec
- Process(dbsrv12)\Private Bytes
- Process(dbsrv12)\Virtual Bytes
- Process(dbsrv12)\Working Set
- Process(dbsrv12)\Working Set – Private
- Process(JmsBridge)\% Processor Time
- Process(JmsBridge)\IO Read Operations/sec
- Process(JmsBridge)\IO Write Operations/sec
- Process(JmsBridge)\Page Faults/sec
- Process(JmsBridge)\Private Bytes
- Process(JmsBridge)\Virtual Bytes
- Process(JmsBridge)\Working Set
- Process(OBMO)\% Processor Time
- Process(OBMO)\IO Read Operations/sec
- Process(OBMO)\IO Write Operations/sec
- Process(OBMO)\Page Faults/sec
- Process(OBMO)\Private Bytes
- Process(OBMO)\Virtual Bytes
- Process(OBMO)\Working Set
- Process(OBMO)\Working Set – Private
- Process(rsOE)\% Processor Time
- Process(rsOE)\IO Read Operations/sec
- Process(rsOE)\IO Write Operations/sec
- Process(rsOE)\Page Faults/sec
- Process(rsOE)\Private Bytes
- Process(rsOE)\Virtual Bytes
- Process(rsOE)\Working Set
- Process(rsOE)\Working Set – Private
- Process(ObServiceManager)\% Processor Time
- Memory\Demand Zero Faults/sec
- Process(mlsrv12)\% Processor Time
- Process(mlsrv12)\IO Read Operations/sec
- Process(mlsrv12)\IO Write Operations/sec
- Process(mlsrv12)\Page Faults/sec
- Process(mlsrv12)\Private Bytes
- Process(mlsrv12)\Virtual Bytes
- Process(mlsrv12)\Working Set
- Process(mlsrv12)\Working Set – Private

8.2 Monitor SMP with MobiLink Monitor

The OS performance counters, together with LoadRunner test results (mainly synchronization time), provides a good indicator of the overall performance under load, but it does not provide enough information about where the response time is spent. To pinpoint performance bottlenecks, it is necessary to trace and drill down into long-running synchronizations. MobiLink Monitor provides the insight into where the response time is spent on the server side, and provides a good indicator on where the bottleneck may lie in.
MobiLink Monitor can be found in SMP’s installation directory <SMP_Directory>\Servers\SQLAnywhere12\BIN32\mlmon.exe.

The above window is the MobiLink monitor’s login window. After logging in, MobiLink Monitor shows all the synchronization activities and their details, like below:

Figure 41: Login to Mobilink Monitor

The above window is the MobiLink monitor’s login window. After logging in, MobiLink Monitor shows all the synchronization activities and their details, like below:
Figure 42: Monitoring with Mobilink Monitor

For more information about MobiLink Monitor, refer to the product documentation:

8.3 Monitor SMP with CA Wily Introscope

CA Wily Introscope, which is part of SAP Solution Manager, is a powerful monitoring tool based on byte-code instrumentation. It can be used to obtain rich monitoring metrics of SMP servers, such as synchronization time and the duration of the underlying API calls, as well as other useful information such as Java/.Net heap memory utilization.

Below are a few useful SMP monitoring metrics, among many, which Wily Introscope provides:

1. Java VM heap utilization
Figure 43: Monitoring Java heap utilization with Wily Introscope

2. Synchronization response time (categorized into Download, Upload and Prepare)

Figure 44: Monitoring synchronization time with Wily Introscope

3. Time spent in Operation Replay against backend systems
In order to trace a particular long-running synchronization, another useful feature named Transaction Trace can be used to trace long-running synchronizations and pinpoint the most time-consuming API call in the invocation hierarchy. The following screenshot shows a breakdown of API calls for a DownloadData call that ran for 302 milliseconds.

For more information about how to setup Wily Introscope for SMP, refer to Workload Analysis with Wily Introscope Agent and SAP Notes 1634219 and 1126554.
9. Analyze Test Results

After a test has completed, LoadRunner gathers all the test results and makes them available for analysis. In the main menu of LoadRunner Controller, click on Tools -> Analysis to bring up LoadRunner Analysis, where you can analyze various performance metrics collected during the test. Alternatively, you can double-click the .lrr file in the result folder to launch LoadRunner Analysis.

LoadRunner Analysis first shows a summary of the test, such as maximum sync time, average sync time, minimum sync time, standard deviation etc. for each transaction you defined earlier in the script. It also provides the information about the number of concurrent Vusers, how many transactions failed, etc.

Figure 47: Summary of load test result

You can then navigate to individual charts that contain certain performance metrics. The most important metric is synchronization time, which can be found in the chart Average Transaction Response Time.
The Average Transaction Response Time chart comes with a certain granularity; in other words, some outstanding transactions, i.e. synchronizations with extremely long or short durations, are evened out in the default chart. In order to see the response times at a finer granularity, right-click on the chart, and click Set Granularity… in the popup menu. Set the granularity to 1 second for all graphs.

To see the breakdown of response time for the transactions that contain sub-transactions, right-click on the chart Average Transaction Response Time, and click Show Transaction Breakdown Tree. You can then navigate in the Transaction Breakdown Tree and display the corresponding charts. The following chart shows that sub-transaction SalesOrder_BackendReplay contributes most to the parent transaction SMP101_SG_Orders_Upload.
It is useful to understand how consistently the server performs for the same type of synchronization. In the left pane, right-click **Graphs** -> **Add New Item** -> **Add New Graph** -> **Transactions** -> **Transaction Response Time (Percentile)**. This graph helps you understand, for example, how much percentage of synchronizations finish within 1 second.

If you configured OS performance counter collection in the LoadRunner scenario, you can add those charts too.
Figure 52: Chart of OS performance counters

For reporting purpose, all the charts can be customized by clicking the Display Options on the chart’s popup menu, such as renaming the titles of the chart and the axes, changing the ranges of the axes, etc.

For more information about how to use LoadRunner Analysis to analyze the results, refer to HP LoadRunner document.
10. Appendix

This chapter covers the steps to create the sample MBO model that is used in the previous chapters of this document. The source code of the callback handler classes are also provided in this chapter.

1. Create the Sample MBO Model
2. Source Code of ApplicationCallbackHandler.cs
3. Source Code of SMPCallbackHandler.cs

10.1 Create the Sample MBO Model

In this section, we create an MBO model that is used for the load test steps described in this Howto guide. We use sampledb as the backend system, and create three MBOs: Customer, Product and Sales_order. Customer and Product contain relatively static data, or master data; and Sales_order contains transactional data.

1. Launch SAP Mobile SDK, and create a new Mobile Application Project by clicking File -> New -> Mobile Application Project. Name the project SMP101.

2. Create these three MBOs by dragging and dropping the following three tables from sampledb to the Mobile Application Diagram:
   - customer
   - product
   - sales_order

3. Accept the default settings when creating the MBOs. The MBOs should look like the following now:

![Figure 53: MBOs of the SMP101 project](image)
4. Highlight the Sales_order MBO, go to the Properties pane and click on Attributes -> Definition. Edit the MBO’s definition, and add a Load Argument argRegion.

![Image of Mobile Business Object -- Sales_order](image1)

**Figure 54: Load argument for Sales_order**

5. Click on the Synchronization tab, add a Synchronization Parameter paramRegion and map it to region.

![Image of Mobile Business Object -- Sales_order](image2)

**Figure 55: Synchronization Parameter of Sales_order**

6. Click Attributes -> Load Arguments, and map the Synchronization Parameter paramRegion to the Load Argument argRegion.

![Image of Mobile Business Object -- Sales_order](image3)

**Figure 56: Map Synchronization Parameter to Load Argument**

8. Create a Cache Group and name it CG_MasterData. Set its cache policy to Scheduled with a cache interval of 24 Hours.

9. Create another Cache Group and name it CG_Orders. Set its cache policy to On demand with a cache interval of 10 Minutes.
10. Place Customer and Product MBOs in CG_MasterData and Sales_order MBO in CG_Orders, respectively.

11. Save the project. Deploy the project to your Mobile Server. In this HowTo guide, the project is deployed to a domain named test with a nossec Security Configuration. This is reflected in the test script. If you are using a different domain with certain Security Configuration, make sure that it is reflected in your test script. After deployment, the smp101:1.0 package should look like below:
10.2 Source Code of ApplicationCallbackHandler.cs

```csharp
using System;
using Sybase.Mobile;

namespace SMP101
{
    class ApplicationCallbackHandler : DefaultApplicationCallback
    {
        public delegate void ApplicationMessage(string s, CallbackType cbt, CallbackSource cbs, object o);
        public event ApplicationMessage OnApplicationMessage;

        private void RaiseEvent(string s, CallbackType cbt, CallbackSource cbs, object o)
        {
            if (OnApplicationMessage != null)
            {
                OnApplicationMessage(s, cbt, cbs, o);
            }
        }

        public override void OnConnectionStatusChanged(int connectionStatus, int errorCode, string errorMessage)
        {
            object[] objects = new object[4];
            objects[0] = connectionStatus;
            objects[1] = string.Empty;
            objects[2] = errorCode;
            objects[3] = errorMessage;
        }
    }
}
```
CallbackType msgCbkType;
switch (connectionStatus)
{
    case ConnectionStatus.CONNECTED:
        msgCbkType = CallbackType.CONNECTED;
        break;
    case ConnectionStatus.CONNECTING:
        msgCbkType = CallbackType.CONNECTING;
        break;
    case ConnectionStatus.CONNECTION_ERROR:
        msgCbkType = CallbackType.CONNECTION_ERROR;
        break;
    case ConnectionStatus.DISCONNECTED:
        msgCbkType = CallbackType.DISCONNECTED;
        break;
    case ConnectionStatus.DISCONNECTING:
        msgCbkType = CallbackType.DISCONNECTING;
        break;
    case ConnectionStatus.NOTIFICATION_WAIT:
        msgCbkType = CallbackType.NOTIFICATION_WAIT;
        break;
    default:
        msgCbkType = CallbackType.UNINITIALIZED;
        break;
}
RaiseEvent(string.Empty, msgCbkType,
CallbackSource.APPLICATION_CONNECTION_STATUS, objects);
}

public override void OnRegistrationStatusChanged(int registrationStatus, int
errorCode, string errorMessage)
{
    object[] objects = new object[4];
    objects[0] = registrationStatus;
    objects[1] = string.Empty;
    objects[2] = errorCode;
    objects[3] = errorMessage;

    CallbackType msgCbkType;
switch (registrationStatus)
{
    case RegistrationStatus.REGISTERED:
        msgCbkType = CallbackType.REGISTERED;
        break;
    case RegistrationStatus REGISTERING:
        msgCbkType = CallbackType.REGISTERING;
        break;
    case RegistrationStatus.REGISTRATION_ERROR:
        msgCbkType = CallbackType.REGISTRATION_ERROR;
        break;
    default:
        break;
}
break;
            case RegistrationStatus.UNREGISTERED:
                msgCbkType = CallbackType.UNREGISTERED;
                break;
            case RegistrationStatus.UNREGISTERING:
                msgCbkType = CallbackType.UNREGISTERING;
                break;
            default:
                msgCbkType = CallbackType.UNINITIALIZED;
                break;
        }

        RaiseEvent(String.Empty, msgCbkType,
                    CallbackSource.APPLICATION_REGISTRATION_STATUS, objects);
    }
}
}

10.3 Source Code of SMPCallbackHandler.cs

```csharp
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Text;
using Sybase.Persistence;

namespace SMP101
{
    /// <summary>
    /// Represents the type of callback.
    /// </summary>
    public enum CallbackType
    {
        // Synchronization Status
        ASYNC_REPLAY_COMPLETED,
        ASYNC_REPLAY_UPLOADED,
        DOWNLOADING,
        ERROR,
        FINISHING,
        STARTING,
        STARTING_ON_NOTIFICATION,
        UPLOADING,

        // Application connection types
        NO_CONNECTIVITY,
        CONNECTING,
        CONNECTION_ERROR,
        DISCONNECTING,
        NOTIFICATION_WAIT,
    }

```
// Application registration types
REGISTERED,
REGISTERING,
REGISTRATION_ERROR,
UNREGISTERED,
UNREGISTERING,
UNINITIALIZED,

// SUP sync status
INITIAL_SYNC_COMPLETE,
SUBSCRIBE_SUCCESS,
SUBSCRIBE_FAILED,
UNSUBSCRIBE_SUCCESS,
UNSUBSCRIBE_FAILED,
SUBSCRIPTION_ENDED,
BEFORE_IMPORT,
BULK_DOWNLOAD_SUCCESS,
RESUME_SUBSCRIPTION_SUCCESS,
RESET_SUCCESS,

// SMP login status
LOGIN_FAILED,
LOGIN_SUCCESS,
NEW,
UPDATE,
DELETE,
SEARCH_SUCCESS,
SEARCH_FAILED,
NONE,
LOW_STORAGE,
RECOVERED_STORAGE,
IMPORT_DATA,
IMPORT_SUCCESS,
REPLAY_SUCCESS,
REPLAY_FAILED,
REPLAY_ABORTED,
MESSAGE_EXCEPTION,
INFO_ONLY,
SYNC_ENGINE_INIT_COMPLETE,
CONNECTED,
DISCONNECTED,
SYNC_SUCCESS,
SYNC_FAILED,
SYNC_STATUS

/// <summary>
/// Represents the source of the callback.
/// </summary>
public enum CallbackSource
{
}
SAP,
APPLICATION_CONNECTION_STATUS,
APPLICATION_REGISTRATION_STATUS,
SUP,
OTHER
}

class SMPCallbackHandler : DefaultCallbackHandler
{
    public delegate void SyncMessage(string s, CallbackType cbt, CallbackSource cbs, object o);
    public event SyncMessage OnSyncMessage;

    private void RaiseEvent(string s, CallbackType cbt, object o)
    {
        if (OnSyncMessage != null)
        {
            OnSyncMessage(s, cbt, CallbackSource.SUP, o);
        }
    }

    public override void OnReplaySuccess(object mboObj)
    {
        RaiseEvent("Operation replay successful (OnReplaySuccess): " + GetFriendlyMBOName(mboObj), CallbackType.REPLAY_SUCCESS, mboObj);
    }

    public override void OnReplayFailure(object mboObj)
    {
        RaiseEvent("Operation Replay failed (OnReplayFailure): " + GetFriendlyMBOName(mboObj), CallbackType.REPLAY_FAILED, mboObj);
    }

    public override SynchronizationAction OnSynchronize(Sybase.Collections.GenericList<ISynchronizationGroup> groups, SynchronizationContext context)
    {
        StringBuilder sb = new StringBuilder();
        foreach (ISynchronizationGroup synchronizationGroup in groups)
        {
            if (synchronizationGroup != null)
            {
                sb.Append(synchronizationGroup.Name);
                sb.Append(",");
            }
        }
        string status = string.Empty;
        if (context != null)
        {
            status = context.Status.ToString();
        }
    }
public string GetFriendlyMBOName(object mbo) {
    if (mbo != null) {
        // we do not want to log children MBO's
        Type type = mbo.GetType();
        PropertyInfo pi = type.GetProperty("PARENT");
        if (pi != null) {
            return string.Empty;
        }
        return type.Name;
    }
    return "unknown MBO";
}