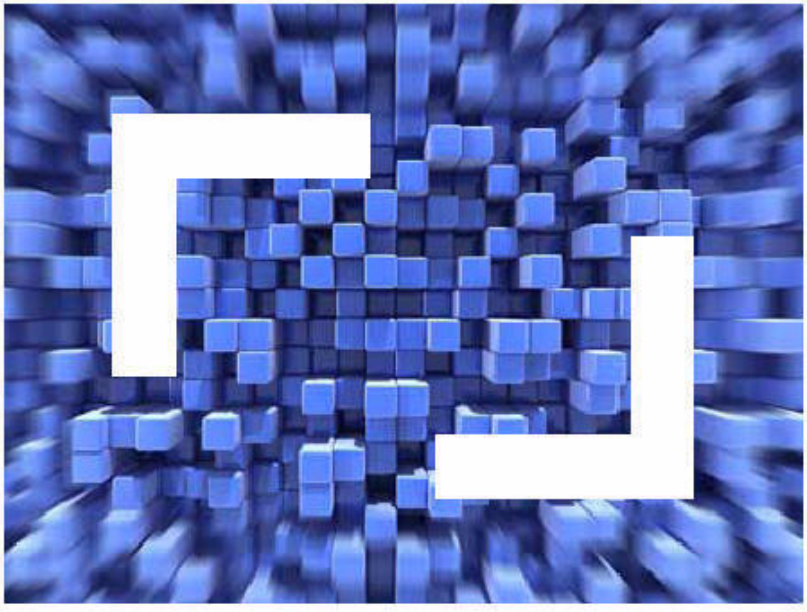


Silk Performer 9.5
Advanced Concepts Reference

Borland®
(A MICRO FOCUS COMPANY)

MICRO FOCUS®
Leading the Evolution™

Micro Focus
575 Anton Blvd., Suite 510
Costa Mesa, CA 92626

2012-11-15

Contents

1

Client IP Address Simulation

Introduction Using Silk Performer's System Configuration Manager

What you will learn This chapter contains the following sections:

Overview

This chapter explores Silk Performer's client IP address simulation feature. Its method of configuring network interface cards for multiple IP addresses is explained, as are the script functions used to set up local IP addresses. A brief introduction describes how to activate newly installed/assigned IP addresses.

Multiple IP Addresses per NIC

When a computer is configured with more than one IP address it is referred to as a multi-homed system. Multi-homed systems are supported in one of three ways:

1 Multiple IP addresses per NIC (logical multi-homed):

Using the Control Panel, five addresses per card may be configured. However, more addresses may be added to the registry. For details, see the next chapter.

- Requirements: NT 4, SP 4
- The total number of IP addresses allowed per NIC depends on the installed NIC. Based on the test results, a 3COM905BTX can handle about 2000 IP addresses.

2 Multiple NICs per physical network (physical multi-homed):

- No restrictions other than hardware.

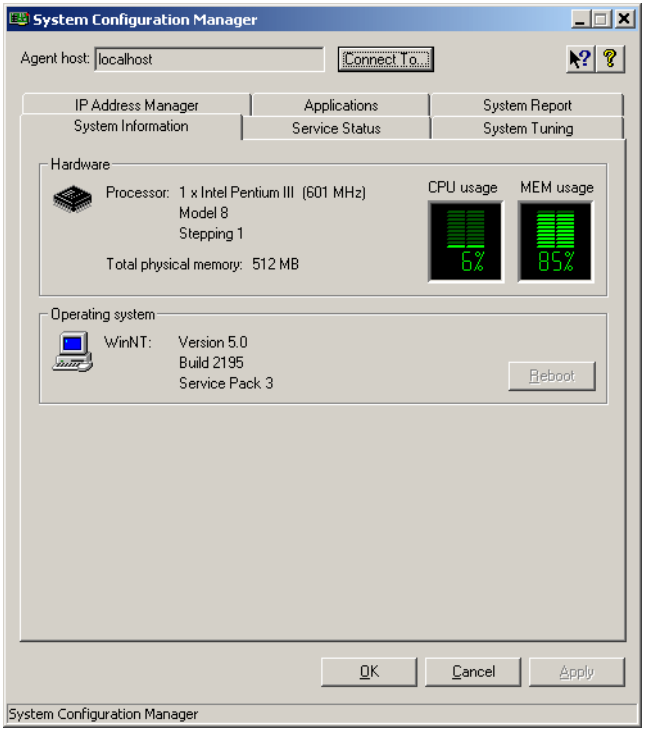
3 Multiple networks and media types:

- No restrictions other than hardware and media support. See the Network Interface Card/Driver section of the Windows NT Server White Paper to learn about supported media types.

Adding IP Addresses via the System Configuration Wizard

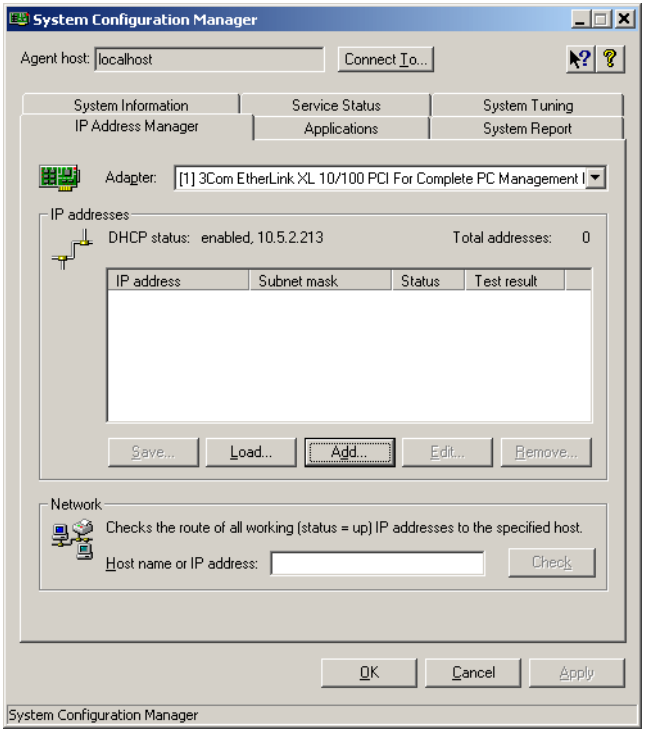
1 Disable DHCP, add gateway and DNS addresses.

2 Launch the Silk Performer System Configuration Wizard. From the Tools menu, select System Configuration Manager.



3 Connect to the machine to which you wish to add new IP addresses.

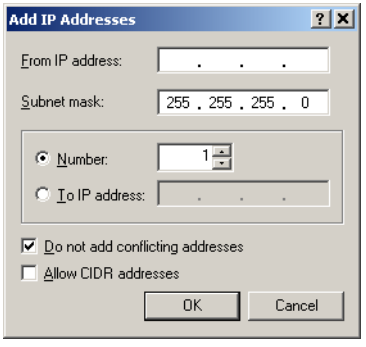
Note By default you will be connected to the localhost. If you aren't able to connect to a particular host, ensure that that host's Silk Launcher Service is running.



4 Select the IP Address Manager tab.

5 Select the network adapter to which you wish to add new IP addresses.

6 Using the Add IP Address dialog, specify the IP addresses you wish to add.



Number - The number of sequential IP addresses to be added subsequent to the one specified in the From IP address entry.

To IP address - Any number of IP addresses will be added until this IP is matched. Addresses must be sequential and can either increase or decrease.

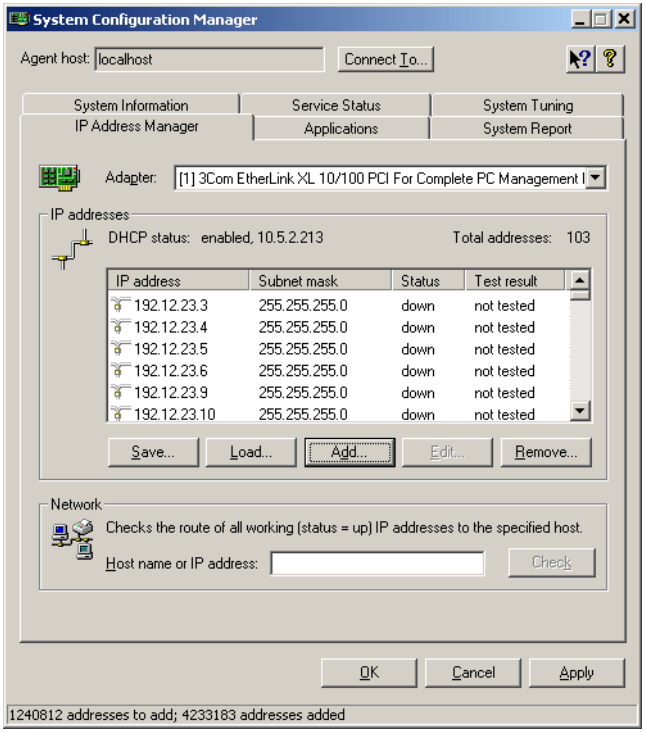
Note When using the From IP address and To IP address fields, ensure that the To IP address is higher than the From IP address (e.g., From: 192.12.23.1, To: 192.12.23.108).

7 Reboot the machine.

8 To verify that the IP addresses you added are valid, either:

a Call ipconfig.exe (within the shell) to verify that all IP addresses are configured properly

b Launch the System Configuration Manager to test that the IP addresses were added correctly.

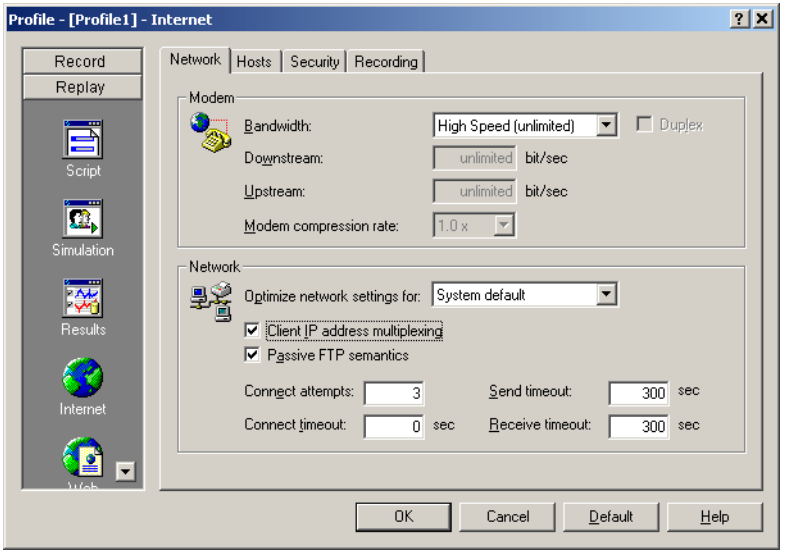


Note By clicking the Check button with the Host name or IP address field blank, the System Configuration Manager checks that all added IP Addresses are functioning correctly. If you enter a particular IP address into the Host name or IP address field, only that address will be checked.

During replay you can use a third-party tool to examine the bindings to the local virtual IP address. TCPView can be downloaded from http://www.sysinternals.com.

Setting Dynamic User IP Addresses Using Automatic Round-Robin

1. From the main Silk Performer window, select the Settings menu and the Active Profile… menu item.
2. On the left side of the Profile Settings dialog, select the Internet icon.
3. Place a check in the Client IP address multiplexing checkbox.
4. Run your test and use a utility such as TCPView to verify your IP address multiplexing.



Setting Dynamic User IP Addresses Using Script Functions

For sample applications, review the WebMultipleClientIpAddresses01.bdf file.

WebSetLocalAddress

Description Sets the local IP address

Include file WebAPI.bdh

Syntax

```
WebSetLocalAddress(in sAddress string) : boolean;
```

`sAddress` IP address

Return value

- **true** if the local IP address has been set successfully
- **false** otherwise

WebTcpipConnect

Include file WebAPI.bdh

Syntax

```
WebTcpipConnect(
            out hWeb      : number,
            in  sHost     : string,
            in  nPort     : number,
            in  nSecurity : number optional
            in  nAddress   : string optional ): boolean;
```

`hWeb` Vriable recieving a handle to the Web connection

`sHost` Domain name or IP address, for example, hostname.com or 192.231.34.1

`nPort`	Port to connect to. The following constants can be used as the standard port: • WEB_PORT_FTP • WEB_PORT_TELNET • WEB_PORT_SMTP • WEB_PORT_GOPHER • WEB_PORT_HTTP • WEB_PORT_HTTPS • WEB_PORT_POP3 • WEB_PORT_NNTP • WEB_PORT_IRC • WEB_PORT_LDAP • WEB_PORT_LDAP_SSL • WEB_PORT_SOCKS
`nSecurity`	Specifies whether to establish a secure connection to the remote server (optional). Possible options are: • WEB_CONNECT_NONE. Establishes an insecure connection to the server (default) • WEB_CONNECT_SSL. Establishes a secure connection using SSL
`nAddress`	binds the IP address to the socket

Return value

- **true** if the function succeeds, and the function stores a valid handle in hWeb
- **false** otherwise

Routing Principles

Routing is one of the most important functions performed by IP. Routing is performed not only by routers, but also by each host that wants to send an IP packet. Fundamentally, routing is a decision made about where a packet should be sent, regardless of the packet's origin (i.e., from its own host or from another host). Each host maintains a routing table, which is processed each time a routing decision is made.

Procedure To do this, Internet Protocol requires that the following steps be performed:

1 Search for a matching host address (netmask: 255.255.255.255)

2 Search for a matching network address

3 Search for a default entry

To compare a given IP address with routing-table entries, look at the *Network Destination / Netmask* pair. The netmask specifies the bits, which you may use or ignore.

For example, the step by step procedure for comparing the 192.200.1.1 IP address with the following entry in the table is:

Network Destination	Netmask	Gateway	Interface	Metric
192.200.1.0	255.255.255.0	192.168.20.126	192.168.20.2	1

1 Convert the addresses to their binary form:

192.200.1.1: 11000000 11001000 00000001 00000001

192.200.1.0: 11000000 11001000 00000001 00000000

255.255.255.0: 11111111 11111111 11111111 00000000

2 Compare only those bits in which the corresponding netmask bit is 1

3 If they are all identical (as shown in the above example), the routing-table entry matches the IP address.

Example

Here is a Windows NT routing-table:

```
(C:\>route print)
===========================================================
Interface List
0x1 ........................MS TCP Loopback interface
0x2 ...00 10 5a 25 1b 08 ..3Com 3C90x Ethernet Adapter
===========================================================
===========================================================
Active Routes:
Network Destination        Netmask          Gateway       Interface  Metric
          0.0.0.0          0.0.0.0    192.168.20.18  192.168.20.126  1
        127.0.0.0        255.0.0.0        127.0.0.1       127.0.0.1  1
     192.168.20.0    255.255.255.0   192.168.20.126  192.168.20.126  1
   192.168.20.126  255.255.255.255        127.0.0.1       127.0.0.1  1
   192.168.20.255  255.255.255.255   192.168.20.126  192.168.20.126  1
        224.0.0.0        224.0.0.0   192.168.20.126  192.168.20.126  1
  255.255.255.255  255.255.255.255   192.168.20.126  192.168.20.126  1
===========================================================================
```

Interpretation of the entries:

0.0.0.0:	**Default Gateway** If IP doesn't find any other matching entry it looks for this line and sends the packet to the specified Gateway (192.168.20.18). If no such line exists, IP generates an error.
127.0.0.0:	Local loopback - Datagrams are sent to the loopback interface
192.168.20.0:	**Subnet address** Any address beginning with 192.168.20. is sent via the interface (192.168.20.126) to the local subnet.
192.168.20.126:	Local IP address - Datagrams sent to this address are sent to 127.0.0.1 (the loopback address)
192.168.20.255:	Subnet broadcast address
224.0.0.0:	IP multicast address
255.255.255.255:	General IP broadcast address

When IP finds the first matching Network Destination / Netmask pair (remember the search order), it looks up the gateway entry and sends the packet to the found gateway address (using the specified interface).

Some special gateway entries include:

- 127.0.0.1 (loopback-address): Sends to the loopback interface
- Its own IP address: IP address is on the local network - just send it.

Routing Problems Due to Client IP Address Simulation

1. Sending packets from multi-homed clients to servers:

   With multi-homed clients, the destination IP address is always that of the selected server. Therefore there aren't differences between network operations and problems don't normally arise when delivering packets from multi-homed clients.

2. Sending packets from servers to multi-homed clients:

   Problems arise when a server attempts to send back a reply to a multi-homed client using the destination IP address of the sending client application. If the IP address is a local subnet-address, the packet will find its way back to the client because there is already a correct entry in

the server's routing-table. If the selected IP address doesn't belong to the local subnet, and the server doesn't find another matching entry in its routing-table (normal behavior when entries haven't been added) it sends the packet to the default-gateway (using the default entry). If it doesn't find a default entry, it generates a network unreachable error.

Solutions

There are several options for configuring servers to send responses to multi-homed clients. Two options for different network configurations are detailed below:

1 Server and agent on the same subnet

If you don't have a router between the agent and server (as shown in Figure 1), you only have to add entries to the server's routing table. If the generated clients' IP addresses begin with the same two numbers (e.g., 192.200.) you'll only need to add one entry to the server's routing table (route add 192.200.0.0 mask 255.255.0.0 <your normal IP address>). The server will then consider the client to be the appropriate router for all such addresses (beginning with 192.200.).

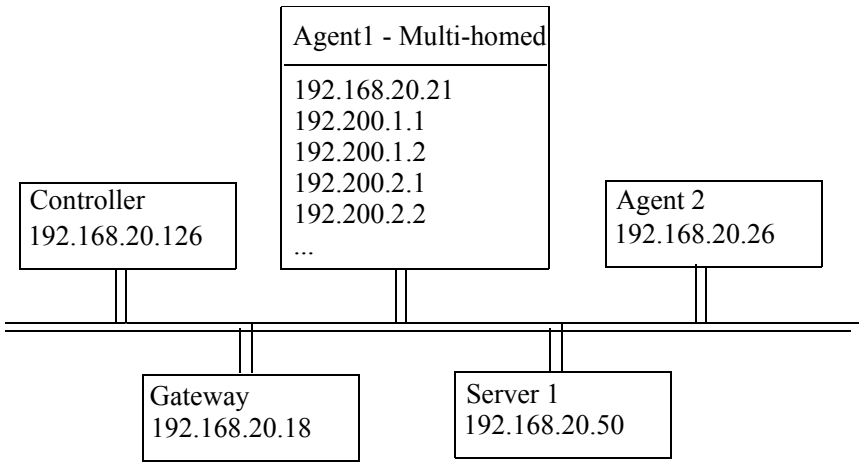


Figure 1

In the example in Figure 1, Server 1 receives a packet from Agent 1 and attempts to send its answer from 192.168.20.50 to 192.200.2.1. Without modifications to the server configuration (the server doesn't know that Agent 1 is multi-homed), Server 1 will send the packet to the default gateway, because there is only one matching routing-table entry:

Network Destination	Netmask	Gateway	Interface	Metric
0.0.0.0	0.0.0.0	192.168.20.18	192.168.20.126	1

If you call (at Server 1),

route add 192.200.0.0 mask 255.255.0.0 192.168.20.21

…a new entry will be added, that looks like this:

Network Destination	Netmask	Gateway	Interface	Metric
192.0.0.0	255.255.0.0	192.168.20.21	192.168.20.50	1

Because this entry has a higher priority than that of the default gateway, Server 1 will send all packets with a destination address type of 192.200.x.x to Agent 1 (believing that this is the correct gateway).

2 A router or load-balancer is positioned between the server and the agent (As shown in Figure 2) In such instances, you need to alter the router's routing-table.

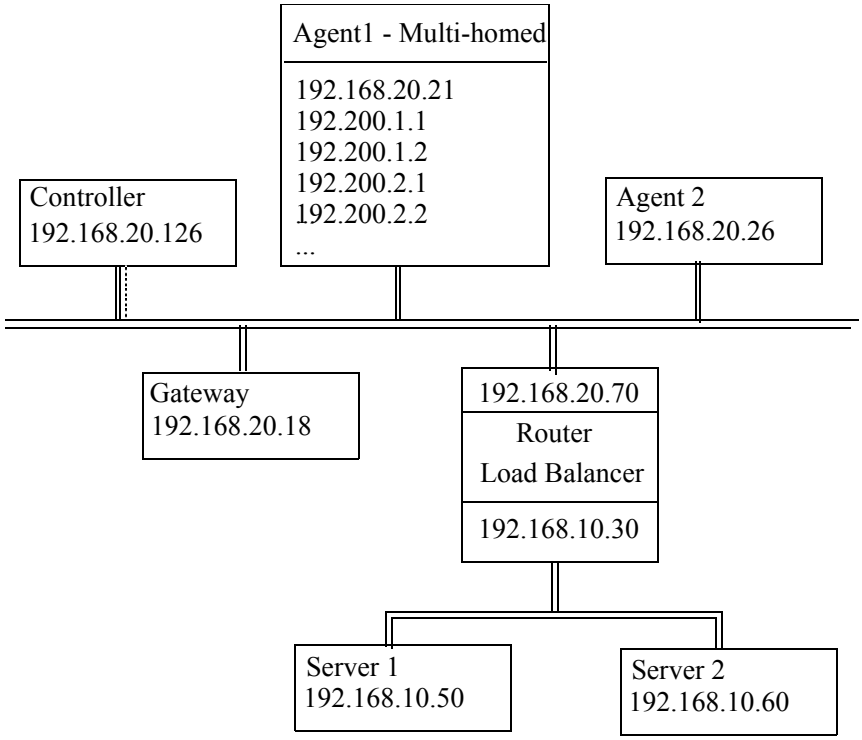


Figure 2

If Agent 1 sends a packet to Server 1 (from 192.200.2.1 to 192.168.10.50) and Agent 1 is configured correctly, i.e it has a routing-table entry such as:

Network Destination	Netmask	Gateway	Interface	Metric
192.168.1.0	255.255.0.0	192.168.20.70	192.168.20.21	1

…Agent 1 will send the packet to the router, which will in turn forward the packet to Server 1.

Now Server 1 wants to send a response to Agent 1's request (from 192.168.10.50 to 192.200.2.1). Because of its default entry in the routing-table, Server 1 sends the packet to the router. Problems arise now however because the router knows nothing of Agent 1's new IP addresses and will use its default route:

Network Destination	Netmask	Gateway	Interface	Metric
0.0.0.0	0.0.0.0	192.168.20.18	192.168.20.70	1

…and send the packet to the gateway.

If you call (at the router):

route add 192.200.0.0 mask 255.255.0.0 192.168.20.21

…a new entry will be added, that looks like this:

Network Destination	Netmask	Gateway	Interface	Metric
192.200.0.0	255.255.0.0	192.168.20.21	192.168.20.70	1

Because this entry has a higher priority than that of the default gateway, the router will send all packets with a destination address type of 192.200.x.x to Agent 1 (believing this to be the correct gateway)

Note It is possible to configure your subnet's default gateway to forward all packets from the server to the multi-homed agent. However, there is a problem in that when a router is forced to send out a packet through the same interface by which the packet is received, the router thinks that the server that originally sent the packet made an incorrect routing decision; the router then generates an ICMP-redirect error. This also increases load on the network and server.

Testing

You can use the System Configuration Manager to determine if newly added IP addresses and routing adaptations work correctly.

- Launch the Silk Performer System Configuration Wizard. Select System Configuration Manager from the Tools menu.
- Connect to the machine to which you added new IP addresses.

Note By default, you will be connected to the localhost. If you cannot connect to a particular host, ensure that that host's Silk Launcher Service is running.

- Select the IP Address Manager tab.

  Select the network adapter to which you added new IP addresses.
- In the network section, enter the host name or IP address of the server you wish to load test.
- Click the Check button and watch for an error window. If an error window doesn't appear, all listed IP addresses will have a route is up mark in the Test result section of the list window.

2

Rule-Based Recording

What you will learn

This chapter contains the following sections:

Overview

This chapter explains how to configure Silk Performer's Recorder (HTTP, TCP/IP) using recording rule files.

> **Note** Manual scripting of recording rules, as outlined in this chapter, is not your only option for creating recording rules. Silk Performer also enables you to create recording rules based on templates via the *Recording Rules* tab *(System Settings/Recorder/Recording Rules)* See Silk Performer Help for full details.

Recording rules allow users to configure the Recorder in a number of ways:

- TCP/IP: By providing protocol descriptions of proprietary TCP/IP based protocols.
- HTTP: By specifying the scenarios in which the Recorder should script parsing functions for dynamically changing values and generate replacements for those values.

This chapter discusses the structure and syntax of recording rules, and offers guided recording rule file design examples.

Recording rules are an advanced Silk Performer concept. To successfully write recording rules, one requires extensive experience using Silk Performer and a thorough understanding of the involved protocols (TCP/IP, HTTP and HTML).

Recording Rule Files

Recording rule files are XML-based files that contain the rules by which Silk Performer's Recorder functions.

Locating Recording Rule Files

Recording rule files carry the file extension .xrl.

Project specific recording rules are stored in the Documents directory of the current project.

Example

```
<my documents>\Silk Performer
9.5\Projects\TestProj\Documents\
```

Global recording rule files are stored in the public or the user's RecordingRules directory.

Example

```
<public user documents>\Silk Performer 9.5\RecordingRules\
```

Formatting Recording Rule Files

Recording rule files are written in XML. Recording rule files may contain any number of recording rules. The root node for all recording rule file node trees is RecordingRuleSet (see Figure 3).

There are three types of recording rules:

- HTTP parsing rules:XML node HttpParsingRule
- TCP rules for WebTcpipRecvProto(Ex):XML node TcpRuleRecvProto
- TCP rules for WebTcpipRecvUntil:XML node TcpRuleRecvUntil

Writing XML Files

Consider the following encoding requirements when writing XML files:

Encoding Special Characters

Some characters and symbols must be encoded for proper XML syntax. See the list below:

Description	Character	XML representation
Less Than	<	<
Greater Than	>	>
Quote	"	"
Ampersand	&	&

Figure 1 - Encoding Special Characters in XML Files

Encoding Binary Data

Binary data must be encoded in hexadecimal notation. See the examples below:

Description	XML representation
CR, (carriage return)
LF, (line feed)
NULL-byte	&#x00;

Figure 2 - Encoding Binary Data in XML Files

Recording Rule File Example

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<RecordingRuleSet>

  <HttpParsingRule>
    <Name>Siebel Session Cookie</Name>
    ...
  </HttpParsingRule>

  <TcpRuleRecvProto>
    <Name>Siebel TCP Protocol</Name>
    ...
```

```
  </TcpRuleRecvProto>

  <TcpRuleRecvUntil>
    <Name>Telnet screen</Name>
    ...
  </TcpRuleRecvUntil>

</RecordingRuleSet>
```

Naming Conventions Used in This Document:

XML nodes are referred to by their full path names.

Example `RecordingRuleSet\HttpParsingRule\Search\LB\RegExpr`

When base paths are clear, the relative paths of XML nodes are used.

Example `Search\LB\RegExpr`

Recording Rule Data Types

Recording rule attributes come in a variety of data types. All valid data types, and associated attribute descriptions, are listed below.

Strings

All string values, including binary data (except NULL-Bytes), are valid.

Please see "Writing XML Files" for information regarding specifying binary data in XML files.

Binary Data

All binary data, including NULL-Bytes, are valid.

Please see "Writing XML Files" for information regarding specifying binary data in XML files.

Numbers

Unsigned numbers in the range of *0* thru *4294967295* are valid.

The value *4294967295* is the *Max* unsigned number.

Signed Numbers

Signed numbers in the range of *-2147483648* thru *2147483647* are valid.

The value *2147483647* is the *Max* signed number.

Numeric Ranges

Numeric ranges are notated using the following syntax:

```
[ MinValue ] [ "-" ] [ MaxValue ]
```

MinValue and *MaxValue* are unsigned numbers.

If *MinValue* is omitted, a default value of 0 is used.

If *MaxValue* is omitted, a default value of Max unsigned is used.

Boolean Values

Boolean values are valid data types. These include "*true*" and "*false*."

Extended Boolean Values

Extended Boolean values are valid data types. These include "*true*", "*unknown*" and "*false*."

Distinct Values

Many distinct values are valid. Valid values are dependent upon, and are listed along with, specific attributes.

Distinct Value Lists

Comma-Separated Value (CSV) lists are valid.

Valid values are dependent upon, and are listed along with, specific attributes below.

Structured Data

Compound data types consisting of nested XML nodes are valid.

General Attributes of Recording Rules

All recording rule types allow users to specify both Name and Active attributes.

Name Attribute

Data type: "Strings"

Default value: *Unnamed Rule*

This attribute specifies a name for a given recording rule. Names need not have special meaning or syntax. They may appear in recorded script comments.

Active Attribute

Data type: "Boolean Values"

Default value: *true*

This attribute specifies whether or not a given recording rule is active. Inactive recording rules are ignored. This attribute allows for the temporary disabling of recording rules without them being deleted from recording rule files.

HTTP Parsing Rules

Introduction

HTTP parsing rules are specified by XML nodes with the name "*HttpParsingRule*".

Purpose

Http parsing rules specify when the Recorder should generate the parsing function *WebParseDataBoundEx*() for dynamically changing values, and then substitute parsing results where appropriate.

This enables the Recorder to directly generate working scripts-and thereby eliminates the need for visual script customization using TrueLog Explorer.

When recording HTTP traffic, the Recorder applies HTTP parsing rules with the following settings:

- Page-based browser-level API
- Browser-level API with/without automatic page detection

The Recorder does not apply HTTP parsing rules when recording HTTP traffic with the *TCP/IP-level API* setting.

How HTTP Parsing Rule Application Works

HTTP parsing rule application involves two main steps:

- Finding possible replacements (called "Rule Hits" or simply "Hits").
- Scripting parsing functions and replacements.

Details regarding both steps can be specified in HTTP parsing rules.

Finding Possible Replacements

During recording, each server response is parsed for rule hits. HTTP parsing rules specify how parsing is to be done and which parsing results (hits) are to be retained for future use.

Parsing functions are not generated in this step; hits are simply retained for future use.

Scripting Parsing Functions

When the Recorder scripts a string value (either a parameter of a function, or a form field value or name), it examines all identified hits and determines a set of hits that are non-overlapping substrings of the string that is to be scripted. The Recorder then scripts the necessary parsing functions and replacements, rather than scripting the original string.

Structuring HTTP Parsing Rules

To keep the two values separate, HTTP parsing rules consist of two sections named *Search* and *ScriptGen*.

Basic structure of a HTTP Parsing Rule

```
<?xml version="1.0" encoding="UTF-8"?>
<RecordingRuleSet>

  <HttpParsingRule>
    <Name>Example Rule</Name>
    <Active>true</Active>

    <Search>
      ...
    </Search>
    <ScriptGen>
      ...
    </ScriptGen>
  </HttpParsingRule>

</RecordingRuleSet>
```

The remainder of this chapter explains HTTP parsing rules through a guided example (“Guided HTTP Parsing Rule Example”) and offers a complete reference of attributes (“Section Search - Finding Rule Hits” and “Scripting Parsing Functions and Replacements”).

Conversion Function

You can write your own custom conversion function and specify this conversion function in a recording rule. The conversion function is contained in a native DLL file (programmed in C/C++). Within the conversion function, the value in the parsing rule is converted using the specified function, while the parsing function only parses the original value.

For example, when values are returned by the server in a double format they are sent by the client in an integer format. Therefore, a simple parsing rule will not

find any matches in the script. However by applying a custom conversion function that converts the integer to a double format, the parsing rule is able to find the expected matches again.

> **Note** A conversion function can only be specified for HTTP Parsing Rules.

See "Creating a Conversion DLL" for more information on creating the conversion DLL.

Sample Recording Rule

Below is an example of a recording rule with the conversion function specified.

```
<RecordingRuleSet>
  <HttpParsingRule>
    <Name>Parse by Boundaries, Convert and Replace (with
custom conversion DLL)</Name>
    <Search>
      <SearchIn type="Select{Body|Header|All}">Body</
SearchIn>
      <LB>
        <Str>&gt;</Str>
      </LB>
      <RB>
        <Str>&lt;/strong&gt;</Str>
      </RB>
      <Conversion>
        <Dll>SampleConversion.dll</Dll>
        <Function>ConvertToUpperCase</Function>
      </Conversion>
    </Search>
    <ScriptGen>
      <ReplaceIn
type="MultiSelect{FormFieldValue|Url|PostedData}">FormFieldV
alue, Url, PostedData</ReplaceIn>
      <VarName>ParsedByBoundary</VarName>
      <GenDebugPrint>true</GenDebugPrint>
      <Conversion>
        <BdlFunction>MyConvertToUpperCase</BdlFunction>
      </Conversion>
    </ScriptGen>
  </HttpParsingRule>
</RecordingRuleSet>
```

Guided HTTP Parsing Rule Example

This chapter steps through the process of designing a HTTP parsing rule for the sample application, ShopIt V 6.0, which ships with Silk Performer.

Recording ShopIt V 6.0 Without Parsing Rules

The Silk Performer sample Web application, ShopIt V 6.0, was deliberately built in such a way that the Recorder has to script the context-less function WebPageUrl() with a form definition that contains a session ID. This was achieved by having JavaScript assemble an URL.

Recording ShopIt V 6.0 without recording rules results in a script with a hard coded session ID, as shown in the following example.

Example

```
WebPageUrl(sParsedUrl, "Unnamed page", SHOPITV60_
KINDOFPAYMENT_ASP002);

// …

dclform
  SHOPITV60_KINDOFPAYMENT_ASP002:
    "choice"                    := "CreditCard",
    "price"                     := "69.9",
    "sid"                       := "793663905";
```

Customizing ShopIt V 6.0 with TrueLog Explorer

In executing a Try Script run, the hard-coded session ID causes a replay error, as shown in Figure 6.

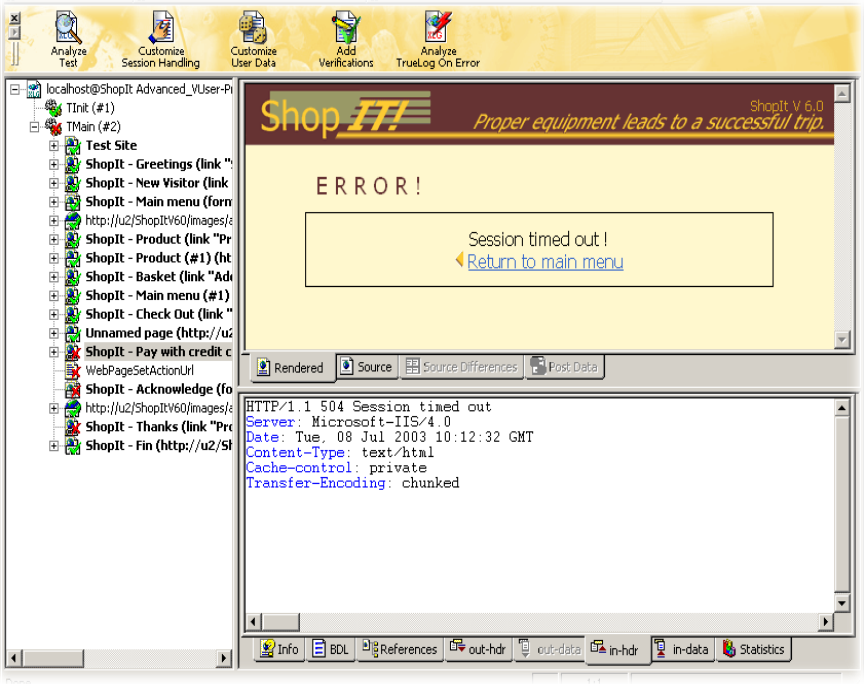


Figure 3 - Replay Error Due to Non-Customized Session ID

The session handling customization feature of TrueLog Explorer solves this problem, modifying the script as shown in the following example.

Example

```
dclparam
  sSessionInfo1     : string;

dcluser
  user
    VUser
  transactions
    TInit           : begin;
    TMain           : 1;

var
  sFormSid1         : string;

// ...

    WebParseDataBoundEx(sSessionInfo1, STRING_COMPLETE,
      "name=\"", 3, "\"", WEB_FLAG_IGNORE_WHITE_SPACE
       | WEB_FLAG_CASE_SENSITIVE, 1);
    WebPageLink("Check out", "ShopIt - Check Out");// Link 3
    Print("sSessionInfo1: " + sSessionInfo1);

    sFormSid1 := sSessionInfo1;
    WebPageUrl("http://u2/ShopItV60/kindofpayment.asp",
      "Unnamed page", SHOPITV60_KINDOFPAYMENT_ASP003);

// ...

dclform
  SHOPITV60_KINDOFPAYMENT_ASP003:
    "choice"                    := "CreditCard",
    "price"                     := "15.9",
//     "sid"                       := "348363999";
    "sid"                       := sFormSid1;
```

A second Try Script run reveals that the customization was successful and that the script now runs correctly.

Transferring Customization Details to the Recorder

The script runs correctly now that it has been customized. However a problem exists in that every script that will be recorded in the future must be also customized.

HTTP parsing rules enable the Recorder to continue this type of customization automatically in the future-so that recorded scripts can be automatically generated without needing manual customization.

To do this, research must be done into how the session ID can be parsed. The customization offered by TrueLog Explorer offers a good place to begin. It reveals the API call where the session ID first occurs, and boundaries that can be used to parse the session ID.

Using TrueLog Explorer, the first occurrence of the session ID can be located in the HTML code, as shown in Figure 8 and the following example.

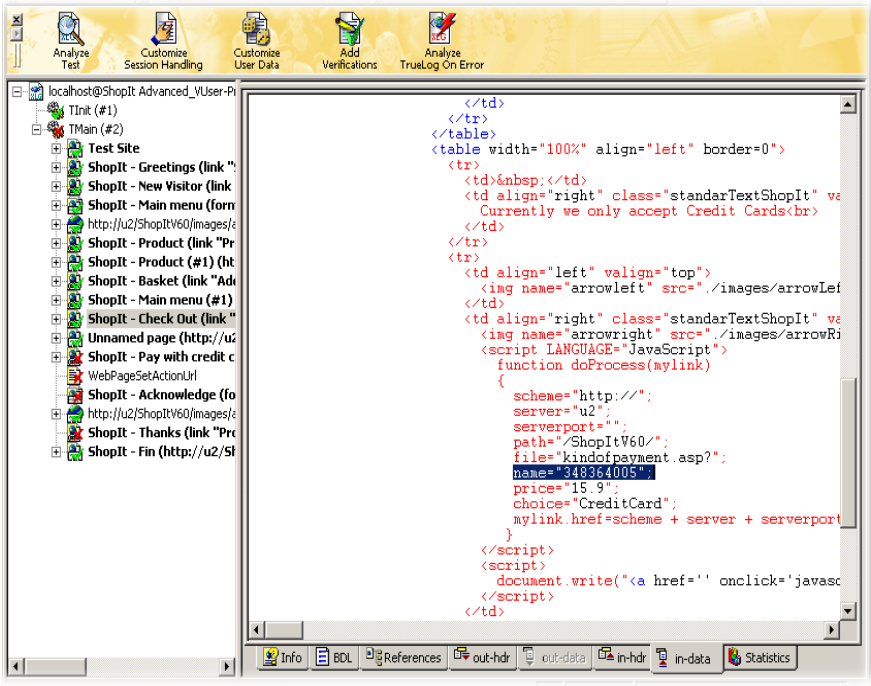


Figure 4 - Locating ShopIt V 6.0 Session ID using TrueLog Explorer

Example

```
<script LANGUAGE="JavaScript">
  function doProcess(mylink)
  {
    scheme="http://";
    server="u2";
    serverport="";
    path="/ShopItV60/";
    file="kindofpayment.asp?";
    name="348364005";
    price="15.9";
    choice="CreditCard";
```

```
    mylink.href=scheme + server + serverport + path
                + file + "choice=" + choice + "&price="
                + price + "&sid=" + name;
  }
</script>
```

The left boundary ("**name=\"**") and the right boundary ("\"") identified by TrueLog Explorer seem to be reasonable choices for parsing the session ID.

Now an initial version of a HTTP parsing rule can be written for the Recorder.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<RecordingRuleSet>

  <HttpParsingRule>
    <Name>ShopIt V6.0 Session Id</Name>

    <Search>
      <SearchIn>Body</SearchIn>
      <LB>
        <Str>name=&quot;</Str>
      </LB>
      <RB>
        <Str>&quot;</Str>
      </RB>
    </Search>

    <ScriptGen>
      <VarName>ShopItSessionId</VarName>
    </ScriptGen>

  </HttpParsingRule>

</RecordingRuleSet>
```

This rule file may be saved to the public RecordingRules directory of Silk Performer-so that the rule will be globally available to all projects. The file name doesn't matter, but the file extension "*.xrl*" must be used. Alternately, if the recording rule is to be used with only one project, the file may be saved to the Documents directory of a Silk Performer project.

ShopIt V 6.0 Session ID's don't appear in HTTP response headers, so it is specified that only response bodies are to be searched (using attribute *Search\SearchIn)*.

The session ID can be found by searching a left boundary. This boundary is specified in the attribute Search\LB\Str. Note that the quote symbol must be encoded in XML using the character sequence "*"*".

A single quote marks the end of session ID's. This is specified in the attribute *Search\RB\Str*. Here again, the quote character must be encoded.

Finally, specifics regarding how the variable for the parsing result should be named need to be defined. Names are specified using the attribute *ScriptGen\VarName*.

Try the Recording Rule

When the rule is used in a recording session, the result is a recorded script with lots of variables.

Script Recorded using a Simple Rule

```
var
  gsShopItSessionId     : string; // Confirm-Button
  gsShopItSessionId_001 : string; // 348364008
  gsShopItSessionId_002 : string; // myForm
  gsShopItSessionId_003 : string; // address
  gsShopItSessionId_004 : string; // city
  gsShopItSessionId_005 : string; // state
  gsShopItSessionId_006 : string; // zip
  gsShopItSessionId_007 : string; // ZipCode
  gsShopItSessionId_008 : string; // cardtype
  gsShopItSessionId_009 : string; // cardnumber
  gsShopItSessionId_010 : string; // expiration
  gsShopItSessionId_011 : string; // sid

MYFORM004:
    gsShopItSessionId_003 := "a",
    gsShopItSessionId_004 := "b",
    gsShopItSessionId_005 := "c",
    gsShopItSessionId_006 := "" <SUPPRESS> ,
    gsShopItSessionId_007 := "d",
    gsShopItSessionId_008 := "Visa",
    gsShopItSessionId_009 := "111-111-111",
    gsShopItSessionId_010 := "07.04",
    gsShopItSessionId_011 := "" <USE_HTML_VAL> ;
```

This is because the rule is too general. The boundaries specified don't simply apply to the parsing of the session ID, they apply to almost all of the form fields. Although this doesn't prevent the script from replaying successfully, it's overkill.

Create a More Specific Rule

A more specific rule that creates a parsing function only for the session ID is needed.

There are several ways of achieving this.

Limit The Number of Rule Hits

The Recorder uses the boundary strings in the Search attribute of parsing rules to extract substrings from each HTTP response. These substrings are called "rule hits" or simple "hits." The Recorder remembers each hit. When scripting a string, the Recorder checks to see if any of the identified rule hits are included in the scripted string. If they are, the Recorder generates a parsing function for the rule hit and substitutes the resulting variable into the scripted strings.

A more specific rule can be created based on the unique characteristics of ShopIt V 6.0 Session ID's.

Session ID properties to consider:

- They consist of digits only.
- Their length is always 9 digits.

Taking this into account, the rule can be extended, as shown in Figure 12.

Limit Number of Rule Hits by Conditions

```
<?xml version="1.0" encoding="UTF-8"?>
<RecordingRuleSet>

  <HttpParsingRule>
    <Name>ShopIt V6.0 Session Id</Name>

    <Search>
      <SearchIn>Body</SearchIn>
      <LB>
        <Str>name=&quot;</Str>
      </LB>
      <RB>
        <Str>&quot;</Str>
      </RB>
      <CondRegExpr>[0-9]+</CondRegExpr>
      <CondResultLen>9-9</CondResultLen>
    </Search>

    <ScriptGen>
      <VarName>ShopItSessionId</VarName>
    </ScriptGen>

  </HttpParsingRule>

</RecordingRuleSet>
```

The attribute *Search\CondRegExpr* specifies a regular expression that is applied to each rule hit. Rule hits that don't match this regular expression are dropped. The regular expression in the example above specifies that only rule hits consisting of digits are relevant.

The attribute *Search\CondResultLen* specifies a range of acceptable length for rule hits. Example above specifies that only hits with exactly nine characters are relevant.

A subsequent recording session using this modified rule is successful: The recorded script contains a parsing function for the session ID only.

Script Recorded using a Modified Rule

```
var
  gsShopItSessionId : string; // 348364011

dclform
    SHOPITV60_KINDOFPAYMENT_ASP003:
    "choice"  := "CreditCard",
    "price"   := "15.9",
    "sid"     := gsShopItSessionId; // value: "348364011"

  MYFORM004:
    "address"    := "a", // changed
    "city"       := "b", // changed
    "state"      := "c", // changed
    "zip"        := "" <SUPPRESS> , // value: ""
    "ZipCode"    := "d", // added
    "cardtype"   := "Visa", // added
    "cardnumber" := "111-111-111", // changed
    "expiration" := "07.04", // changed
    "sid"          := "" <USE_HTML_VAL> ;//value:"348364011"
```

Specify Allowed Usage of Rule Hits

Rather than limiting the number of rule hits, one can be more specific in specifying where in scripts rule hits are to be used.

Consider the following for this example:

The session ID occurs only in form field values where the form field name is "sid".

By extending the *ScriptGen* section of the parsing rule (as shown in the example below), rule hits are used only under these specific criteria.

Be More Specific in Script Generation

```
<?xml version="1.0" encoding="UTF-8"?>
<RecordingRuleSet>

  <HttpParsingRule>
    <Name>ShopIt V6.0 Session Id</Name>

    <Search>
      <SearchIn>Body</SearchIn>
      <LB>
        <Str>name=&amp;quot;</Str>
      </LB>
      <RB>
```

```
        <Str>&amp;quot;</Str>
      </RB>
    </Search>

    <ScriptGen>
      <VarName>ShopItSessionId</VarName>
      <ReplaceIn>FormFieldValue</ReplaceIn>
      <Conditions>
        <CompareData>
          <Data>sid</Data>
          <ApplyTo>FormFieldName</ApplyTo>
        </CompareData>
      </Conditions>
    </ScriptGen>

  </HttpParsingRule>

</RecordingRuleSet>
```

The attribute ScriptGen\ReplaceIn specifies that rule hits may only be used when the Recorder scripts a form field value.

The condition additionally specifies that a replacement is allowed only if the associated form field name is "sid".

Recording with this modified rule generates a script that is identical to the script generated using the original rule ("Specify Allowed Usage of Rule Hits").

Creating a Conversion DLL

Microsoft Visual Studio project

The conversion DLL has to be a native Win32 DLL (no .NET assembly). To create it in Microsoft Visual Studio, you have to create a Win32 project and select the application type *DLL*.

SilkPerformer has provided sample Microsoft Visual Studio projects with header and .cpp files that give you guidance on how to create your own custom conversion DLL. The samples are located at <public user documents>\Silk Performer 9.5\SampleApps\SampleConversion.

After you have created the project, there are some additional project settings to change:

- The *Character Set* option has to be set to **Use Multi-Byte Character Set (MBCS)**. This is a requirement, because the strings to be converted are passed as MBCS strings.

- It is recommended that you use the static version of the C-runtime to remove dependencies to C-runtime DLLs. These dependencies could cause problems if the C-runtime DLLs (of used version) are not installed on the agent or controller machine.

  To use the static C-runtime libraries, change the *Runtime Library* setting in *C++ / Code Generation* to the following:

  - **Multi-threaded (/MT)** in the *Release* configuration
  - **Multi-threaded Debug (/MTd)** in the *Debug* configuration

Recommendation

If you reference the conversion DLL in a recording rule, you have to copy the DLL either into the Silk Performer recording rules directory or into the project directory. When developing the conversion DLL in Microsoft Visual Studio, you could copy the DLL to one of these directories (recording rules or project) in a post-build step or change the output directory path in the *Release* configuration to the recording rules directory.

Exporting the conversion function

You must export the conversion function with the following signature:

```
extern "C"
{
  __declspec( dllexport )
  long MyConversionFunction(
    LPCSTR          sOriginalValue,
    LPSTR           sConvertedValue,
    unsigned long*  psConvertedValueLen,
    void*           pReserved);
}
```

The purpose of using C linkage (extern "C") is to turn off C++ name mangling of the exported function. If using C++ linkage, other information like the types of arguments would be put into the exported name of the function. Use the "pure" function name as the exported function name (this is **`MyConversionFunction`** in the example above).

Function arguments:

sOriginalValue	The MBCS string value that has to be converted.
sConvertedValue	The string buffer that receives the converted value as MBCS string.

psConvertedValueLen	Points to a variable which contains the length of the string buffer *sConvertedValue*. This variable's value has to be set to the number of bytes written to *sConvertedValue* if the conversion succeeded. This variable value has to be set to the number of bytes required for the converted value if the size of the string buffer *sConvertedValue* is too small.
pReserved	Reserved for future usage.
Return value	• ERROR_SUCCESS (0) Returned if the conversion was successful. **psConvertedValueLen* now contains the number of bytes written to *sConvertedValue*. • ERROR_INSUFFICIENT_BUFFER (122) Returned if the conversion failed because the size of of *sConvertedValue* (**psConvertedValueLen*) is too small for the converted value. **psConvertedValueLen* now contains the number of bytes required to store the converted value. The function will be called again with size of s*ConvertedValue* increased to **psConvertedValueLen*. • Return any other value to indicate that the conversion failed.

Section Search - Finding Rule Hits

Details about finding rule hits are specified in the Search section. All XML paths of properties described in the section are relative to HttpParsingRule\Search.

Introduction

Rule hits can be extracted from HTTP responses in two ways:

- By defining boundaries
- By applying a regular expression

Defining Boundaries

When a rule defines boundaries, any occurrence of a left boundary within an HTTP response marks the beginning of a rule hit. From this point onward within the HTTP response, the first occurrence of a right boundary marks the end of the rule hit.

Left boundaries can be defined in three ways:

- Strings: Any occurrence of a given string in an HTTP response marks the beginning of a rule hit.
- Regular Expressions: Any substring of a HTTP response that matches a specified regular expression marks the beginning of a rule hit.
- Offset Calculations: HTTP responses are run through the Offset Calculation to determine the beginning of a rule hit. Offset Calculation is explained in Section "Offset, Length".

Right boundaries can be defined in four ways:

- Strings: The next occurrence of a given string after the left boundary position marks the end of the rule hit.
- Regular Expressions: The next sub string of the HTTP response matching the given regular expression after the left boundary position marks the end of the rule hit.
- Length: The end of a rule hit is determined by running part of an HTTP response (from the beginning of the rule hit through to the end of the response) through a Length Calculation. Length Calculation is explained in Section "Offset, Length".
- Character type: The next character that matches a given set of character types marks the end of the rule hit.

Applying Regular Expressions

If a rule defines a regular expression, any substring of an HTTP response that matches that regular expression yields a rule hit. By default, the entire match is the rule hit. Alternately, the rule can define a tag number so that the tagged sub-expression is the rule hit.

Please see Silk Performer's online help for a description of regular expressions and how to define tagged sub-expressions.

LB\Str Attribute

Type: "Binary Data"

Default: (empty)

This attribute specifies a string for searching rule hits. Each occurrence of this string marks the beginning of a rule hit. When searching, the attributes CaseSensitive and IgnoreWhiteSpaces are used.

Examples:

```
<LB>
  <Str>name=&quot;</Str>
</LB>
```

```
<LB>
  <Str>BV_EngineID=</Str>
</LB>
```

LB\RegExpr Attribute

Type: "Strings"

Default: (empty)

This attribute specifies a regular expression for searching rule hits. Each substring of an HTTP response that matches the regular expression marks the beginning of a rule hit. When searching matching substrings, the attributes CaseSensitive and IgnoreWhiteSpaces are not used.

Example

```
<LB>
  <RegExpr>name *= *[&quot;']</RegExpr>
</LB>
```

LB\Offset Attribute

Type: "Signed Numbers"

Default: (empty)

This attribute specifies an offset value for an Offset/Length calculation, as described in Section "Offset, Length", Offset, Length. This calculation is applied to the entire HTTP response with the given offset and the length of 0. The beginning of the calculation's result marks the beginning of the rule hit.

Example

```
<LB>
  <Offset>17</Offset>
</LB>
```

RB\Str Attribute

Type: "Binary Data"

Default: (empty)

This attribute specifies a string that searches for the end of rule hits. It is used with the beginning of each rule hit identified using LB\Str, LB\RegExpr or LB\Offset. The attributes CaseSensitive and IgnoreWhiteSpaces are used during searches.

Example

```
<RB>
```

```
  <Str>&quot;</Str>
</RB>
```

RB\CharType Attribute

Type: "Distinct Value Lists"

Default: (empty)

This attribute specifies a list of character types and is used to search for the end of each rule hit found using LB\Str, LB\RegExpr or LB\Offset. The ends of rule hits are defined by the first character that matches one of the character types specified below.

Allowed values are:

• UpperCase	Uppercase characters
• LowerCase	Lowercase characters
• NewLine	Newline characters
• Digit	Digits 0-9
• HexDigit	Hexadecimal digits 0-9, a-z, A-Z
• Letter	Letters a-z, A-Z
• White	Whitespaces
• WhiteNoSpace	Whitespaces, excluding blank spaces
• Printable	Printable characters
• NonPrintable	Non-printable characters
• EndOfString	End of Strings (single and double quote)
• NonBase64	Characters not used in Base 64 encoding

Example

```
<RB>
  <CharType>EndOfString, White</CharType>
</RB>
```

RB\RegExpr Attribute

Type: "Strings"

Default: (empty)

This attribute specifies a regular expression that searches for the end of rule hits. It is used with the beginning of rule hits found using LB\Str, LB\RegExpr or LB\Offset. The attributes CaseSensitive and IgnoreWhiteSpaces are not used during searches.

Example

```
<RB>
  <RegExpr>[&quot;' *value]</RegExpr>
</RB>
```

RB\Length Attribute

Type: "Signed Numbers"

Default: (empty)

This attribute specifies a length value for an Offset/Length calculation, as described in section "Offset, Length". The calculation is applied from the portion of an HTTP response where a rule hit begins through to the end of the response; it uses the offset of 0, and the given length. The end of the calculation's result marks the end of the rule hit.

Example

```
<RB>
  <Length>4</Length>
</RB>
```

RegExpr and RegExprTag Attribute

Types: "Strings", "Numbers"

Default: (empty), 0

These two attributes specify a regular expression that is used for searching rule hits. If the attribute RegExprTag is not omitted, it must specify the number of a tagged sub-expression within the given regular expression.

Rule hits are searched for by applying the given regular expression to HTTP responses. Each substring of an HTTP response that matches the regular expression is a rule hit. If the attribute RegExprTag is specified, the rule hit is not the entire match, but the given tagged sub-expression.

Please see Silk Performer's online help for a description of regular expressions and information on defining tagged sub-expressions.

Example

```
<RegExpr>name=&quot;\([0-9]+\)&quot;<RegExpr>
<RegExprTag>1<RegExprTag>
```

SearchIn Attribute

Type:"Distinct Value Lists"

Default: All

Allowed values are:

• All	This is an abbreviation for the complete list of other values
• Header	Search HTTP response headers
• Body	Search HTTP response bodies

This attribute specifies where to search for rule hits, either in response headers, response bodies, or both.

Example

```
<SearchIn>Body</SearchIn>

<SearchIn>Header</SearchIn>
```

CaseSensitive Attribute

Type: "Boolean Values"

Default: false

This attribute specifies whether searches should be case-sensitive or case-insensitive when searching the strings LB\Str and RB\Str.

It also specifies if the option flag *WEB_FLAG_CASE_SENSITIVE* should be scripted when scripting the function *WebParseDataBoundEx*.

IgnoreWhiteSpaces Attribute

Type: "Boolean Values"

Default: true

This attribute specifies whether searches should ignore white spaces when searching the strings LB\Str and RB\Str.

It also specifies whether to script the option flag *WEB_FLAG_IGNORE_WHITE_SPACE* when scripting the function *WebParseDataBoundEx*.

CondContentType Attribute

Type: "Strings"

Default: (empty)

This attribute specifies a string that restricts rule hit searches to server responses that match the specified content type. The comparison is done using a prefix-match.

Example

```
<CondContentType>text/</CondContentType>
```

This restricts the searching for rule hits to server responses with a content type such as "text/html" or "text/plain."

CondRegExpr Attribute

Type: "Strings"

Default: (empty)

This attribute specifies a regular expression that is used to determine if a rule hit should be retained for future use. Each rule hit is checked to see if it matches the regular expression. Matching rule hits are retained, non-matching rule hits are dropped.

Example

```
<CondRegExpr>eCS@Store@[0-9]+-.*</CondRegExpr>
```

CondResultLen Attribute

Type: "Numeric Ranges"

Default: 2-

This attribute specifies a range that is used to determine if a rule hit should be retained for future use. If the number of bytes in the hit doesn't match the given range, the hit is dropped.

Conditions Attribute

Type: "Structured Data"

This attribute specifies conditions that are applied to determine if a rule hit should be retained for future replacement.

The conditions for each rule hit are evaluated within an environment that allows access to the HTTP request/response that included the hit, in addition to other relevant data.

If the conditions aren't determined to be true, the hit is dropped.

See "Conditions" for more information regarding conditions.

See the Section "Condition Evaluation Environment" for more information regarding what may be subject to conditions.

Conversion\Dll Attribute

Type: "Strings"

Default: (empty)

This attribute is the name of the conversion DLL used for modifying a parsed value. Make sure that the conversion DLL specified is added as a data file. The runtime searches for the DLL in the project directory and in the recording rules directory.

Conversion\Function Attribute

Type: "Strings"

Default: (empty)

This attribute is the name of the conversion function exported by the conversion DLL.

Scripting Parsing Functions and Replacements

While the Search section specifies how to find rule hits in HTTP responses, the *ScriptGen* section specifies details regarding script generation.

All XML paths of attributes described here are relative to: *HttpParsingRule\ScriptGen*.

Introduction

The *ScriptGen* section allows users to specify conditions that restrict the usage of rule hits to script locations where replacement is appropriate, and to exclude locations where rule hits appear purely by coincidence. Additionally, some attributes can be used to instruct the Recorder as to which variable names to use and what comments are to be added to scripts to increase their readability.

VarName Attribute

Type: "Strings"

Default: (empty)

This attribute specifies a variable name to script for the result of parsing functions. If omitted, the name of the rule is used as the variable name.

OnlyIfCompleteResult Attribute

Type: "Boolean Values"

Default: false

This attribute specifies that a replacement should be scripted only if a complete string to be scripted can be replaced with a single variable. If false, replacements are scripted when a rule hit is a substring of the string to be scripted.

MaxLbLen Attribute

Type: "Numbers"

Default: Max unsigned

When scripting a parsing function, the Recorder chooses a left boundary. This attribute can be used to specify a maximum length for the left boundary, in cases where there is danger that the left boundary may contain session information.

ReplaceIn Attribute

Type: "Distinct Value Lists"

Default: All

This attribute specifies script locations that may contain hits to be replaced.

Valid values are:

• All	This is an abbreviation for the complete list of other values
• Url	Replace in URL parameters (various functions, e.g. *WebPageUrl*)
• LinkName	Replace in link names (function *WebPageLink*). Link names that result from *WebPageParseUrl* are not replaced.
• FormName	Replace in form names (function *WebPageSubmit*)
• PostedData	Replace in binary posted data (various functions, e.g. *WebPagePost*)
• FormFieldName	Form field name in the dclform section of the script
• FormFieldValue	Form field value in the dclform section of the script
• SetCookie	Parameter of the function *WebCookieSet*

AlwaysNewFunc Attribute

Type: "Boolean Values"

Default: false

This attribute specifies whether or not a rule hit, once parsed, can be reused or if the Recorder should script a new parsing function to parse the most recent occurrence of the string to be replaced.

CommentToVar Attribute

Type: "Boolean Values"

Default: true

This attribute specifies whether or not a comment should be generated for the variable that contains the parsing result. If *true*, a comment that includes the value during recording is generated.

CommentToFunc Attribute

Type: "Boolean Values"

Default: false

This attribute specifies whether or not a comment should be generated for the parsing function *WebParseDataBoundEx*. If *true*, a comment is generated during recording that includes the rule name that triggered the parsing function and the parsing result during recording.

GenBytesReadVar Attribute

Type: "Boolean Values"

Default: false

This attribute specifies whether or not to generate a variable for the number of bytes parsed by a parsing function (see Silk Performer's online help for the function *WebParseDataBoundEx*, parameter nBytesParsed).

GenDebugPrint Attribute

Type: "Boolean Values"

Default: false

This attribute specifies whether or not to generate a diagnostic Print function after the script function where a generated parsing function is in effect.

If true, a *Print* function that prints the parsing result to Silk Performer's controller output window is scripted.

Conditions Attribute

Type: "Structured Data"

This attribute specifies conditions that are applied to determine whether or not a rule hit within a string being scripted should actually be replaced.

The conditions are evaluated within an environment that allows access to the HTTP request/response that is currently being scripted.

If the conditions do not lead to a result of true, no replacement or parsing functions are scripted.

See "Conditions" for more information regarding conditions.

See section "Condition Evaluation Environment" for more information regarding what may be subject to conditions.

Tokenizing of Rule Hits

The *Search* section used both in *HttpParsingRules* and *StringScriptingRules* contains a new feature that allows to extract rule hits by tokenizing the search result.

The idea is that each substring extracted (e.g. using the various left/right boundary options) is not the rule hit itself, but can be "tokenized" to yield several rule hits.

This "tokenizing" can be done in several ways and is specified by the xml tag *Tokenize*.

Valid values for the tag *Tokenize* are:

- *SiebelTokenHtmlSingleQuote*
- *SiebelTokenHtml*
- *SiebelTokenApplet*

The tokenizing methods *SiebelTokenHtmlSingleQuote* and *SiebelTokenHtml* tokenize the search result into individual strings enclosed either in single or in double quotes.

The tokenizing method *SiebelTokenApplet* tokenizes the search result assuming a series of length prefixed strings as used in applet responses in the Siebel 7 web application.

Example for SiebelTokenHtml

The search result

```
["TestName","TestSite","USD","02/21/2003","N","1-2T"]
```

will be tokenized and results in the following rule hits:

- TestName
- TestSite
- USD
- 02/21/2003
- N
- 1-2T

Example for SiebelTokenApplet

The search result

```
19*02/21/2003 08:20:176*SADMIN4*Note5*1-1P5
```

will be tokenized and results in the following rule hits:

- 02/21/2003 08:20:17
- SADMIN

- Note
- 1-1P5

The recorder will use these rule hits in the script by generating one of the tokenizing functions to extract the tokens at runtime.

Example

For a recording rule with tokenizing in the *Search* section:

```
  <HttpParsingRule>
    <Name>Siebel Submit Data Array in HTML (from Javascript
function call)</Name>
    <Active>true</Active>
    <Search>
      <SearchIn>Body</SearchIn>
      <LB>
        <Str>SWESubmitForm</Str>
      </LB>
      <RB>
        <Str>'</Str>
      </RB>
      <Tokenize>SiebelTokenHtml</Tokenize>
      <CondResultLen>1-</CondResultLen>
    </Search>
    <ScriptGen>
      ...
    </ScriptGen>
  </HttpParsingRule>
```

Example

Script fragments from Siebel where tokenizing is used:

```
var
  gsRowValArray_003 : string; // 0*19*02/21/2003 08:20:176*SADMIN4*Note5*1-1P5

// ...


    WebParseDataBoundEx(gsRowValArray_003, sizeof(gsRowValArray_003),
                        "ValueArray`", WEB_OCCURENCE_LAST, "`",
                        WEB_FLAG_IGNORE_WHITE_SPACE, 1);
    WebPageForm("http://lab72/sales_enu/start.swe", SALES_ENU_START_SWE026,
                "Account Note Applet: InvokeMethod: NewRecord");
    Print("Parsed \"RowValArray_003\", result: \"" + gsRowValArray_003 + "\"");
    // Was "0*19*02/21/2003 08:20:176*SADMIN4*Note5*1-1P5" when recording

// ...

dclform

// ...

  SALES_ENU_START_SWE027:
    "SWEMethod"                 := "GetQuickPickInfo",
    "SWEVI"                     := "",
    "SWEView"                   := "Account Note View",
    "SWEApplet"                 := "Account Note Applet",
    "SWEField"                  := "s_2_2_24_0",
    "SWER"                      := "0",
    "SWEReqRowId"               := "1",
    "s_2_2_26_0"                := "2/21/2003 08:20:17 AM",
```

```
"s_2_2_27_0"                := SiebelTokenApplet(gsRowValArray_003, 2), // value: "SADMIN"
"s_2_2_24_0"                := SiebelTokenApplet(gsRowValArray_003, 3), // value: "Note"
"s_2_2_25_0"                := "",
"SWERPC"                    := "1",
"SWEC"                      := "11",
"SWEActiveApplet"           := "Account Note Applet",
"SWEActiveView"             := "Account Note View",
"SWECmd"                    := "InvokeMethod",
"SWERowId"                  := SiebelTokenApplet(gsRowValArray_003, 4),
                               // value: "1-1P5"
"SWERowIds"                 := "SWERowId0=" + SiebelTokenHtml(gsRowValArray_002, 18),
                               // value: "SWERowId0=1-2T"
"SWEP"                      := "",
"SWEJI"                     := "false",
"SWETS"                     := GetTimeStamp(); // value: "10458444419057"
```

Section ScriptGen

There are new options in the ScriptGen section.

MinRbLen, MinLbLen

This option allows to specify a minimum length for the right / left boundary string which will be determined by the recorder and scripted as a parameter of the function call *WebParseDataBoundEx*.

Type: "Numbers"

Default: *1*

LastOccurence

If this option is set to *true*, the recorder will script the constant *WEB_OCCURENCE_LAST* instead of the actual occurence of the left boundary, **if** the found occurence indeed was the last one during recording.

Type: "Boolean Values"

Default: *false*

ExpectBinaryData

If this option is set to *true*, the recorder will generate a "bin-cast" for every use of the variable which is generated by this rule. This is necessary with the introduction of dynamic strings, if it is expected that the parsing result will contain binary data.

Example

Instead of

```
"SomeText" + gsParsedValue + "some other text"
```

the recorder will generate

```
"SomeText" + bin(gsParsedValue) + "some other text"
```

Default: *false*

Conversion\Bdl Attribute

This attribute is optional. It is the name of the BDL function wrapping the call of the conversion DLL function. If missing or left empty, the BDL wrapper function will have the same name as the "Conversion\Dll Attribute" function.

New Script Locations

HTTP parsing rules, as well as *StringScriptingRule*s, are now also applied for the parameters of the function *WebHeaderAdd*.

Therefore, there are two new script locations allowed in the *ReplaceIn* tag of a *HttpParsingRule* and in the *SearchIn* tag of a *StringScriptingRule*:

- *HeaderName*
  Refers to the 1st parameter of the function *WebHeaderAdd*.
- *HeaderValue*
  Refers to the 2nd parameter of the function *WebHeaderAdd*.

TCP/IP Protocol Rules

Introduction

The Recorder uses TCP/IP protocol rules to detect proprietary TCP/IP protocols. If detected, the Recorder generates functions (depending on the protocol detected) that are better suited to handling dynamic server responses than is the WebTcpipRecvExact function, which is automatically scripted for unknown protocols.

Types of TCP/IP Protocol Rules

There are two types of TCP/IP protocol rules.

TcpRuleRecvProto

Use *TcpRuleRecvProto* rules to describe length-based protocols. Length-based protocols are protocols in which the number of bytes to be received from the server can be extracted from a fixed location in a protocol header.

This rule type is specified by XML nodes with the name TcpRuleRecvProto.

TcpRuleRecvUntil

Use *TcpRuleRecvUntil* rules to describe protocols in which the end of a server response can be detected by a terminating sequence of bytes.

This rule type is specified by XML nodes with the name *TcpRuleRecvUntil*.

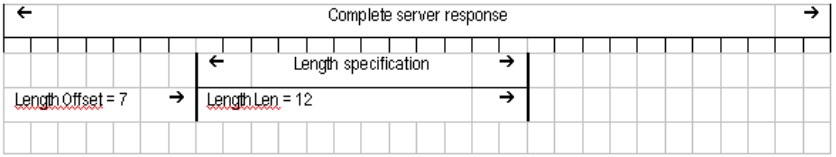
Structure of TCP/IP Protocol Rules

Both types of TCP/IP protocol rules share the same basic structure. The Identify section contains attributes specific to the rule type. The *Conditions* section contains additional conditions that can be specified to avoid "detecting" protocols where they should not be detected (in cases where server responses coincidentally resemble protocols).

```
<TcpRuleRecvProto>
  <Name>Sample TCP RecvProto Rule</Name>
  <Active>true</Active>
  <Identify>
    …
  </Identify>
  <Conditions>
    …
  </Conditions>
</TcpRuleRecvProto>

<TcpRuleRecvUntil>
  <Name>Sample TCP RecvUntil Rule</Name>
  <Active>true</Active>
  <Identify>
    …
  </Identify>
  <Conditions>
    …
  </Conditions>
</TcpRuleRecvUntil>
```

TcpRuleRecvProto

The *TcpRuleRecvProto* rule type describes protocols with the following basic structure:



The number of bytes to be received can be extracted from the protocol header at offset *LengthOffset* using *LengthLen* number of bytes. This can be interpreted either in *big endian* or *little endian* representation. Additionally, the obtained value may be multiplied by a value (attribute *LengthFactor*), and/or a constant value may be added (attribute *LengthAdd*).

Identify\LengthOffset Attribute

Type: "Numbers"

Default: 0

This attribute specifies the offset of the length specification within the protocol header.

This corresponds to the parameter *nProtoStart* of the functions *WebTcpipRecvProto(Ex)*.

Identify\LengthLen Attribute

Type: "Numbers"

Default: 4

This attribute specifies the number of bytes for the length specification within the protocol header.

This corresponds to the parameter nProtoLength of the functions *WebTcpipRecvProto(Ex)*.

Identify\OptionFlags Attribute

Type: "Distinct Value Lists"

Default: (empty)

This attribute specifies how to interpret the length specification. Valid values correspond to the options available for the parameter *nOption* of the functions *WebTcpipRecvProto(Ex)*. See Silk Performer's online help for a description of these option flags. The empty default value means: big endian.

Valid values are:

- ·LittleEndian Option TCP_FLAG_LITTLE_ENDIAN
- ·ProtoIncluded Option TCP_FLAG_INCLUDE_PROTO

Identify\LengthFactor Attribute

Type: "Numbers"

Default: 1

This attribute specifies a factor. The protocol length is multiplied by this factor.

This corresponds to the parameter *nMultiply* of the function *WebTcpipRecvProtoEx*.

Identify\LengthAdd Attribute

Type: "Signed Numbers"

Default: 0

This property specifies a constant value. This value is added to the protocol length.

This corresponds to the parameter *nSum* of the function *WebTcpipRecvProtoEx*.

Conditions

Additional conditions can be specified to exclude the detection of protocols in situations where server responses coincidentally resemble protocol specifications.

See "Conditions" for more information regarding conditions.

See section "Condition Evaluation Environment" for more information regarding what may be subject to conditions.

GenVerify Attribute Of Conditions

Type: "Boolean Values"

Default: false

In the course of specifying basic conditions, the attribute *GenVerify* can be used to specify scripting for the parameters *sVerify*, *nVerifyStart* and *nVerifyLength* of the function *WebTcpipRecvProtoEx*.

When the conditions are evaluated, the first basic condition that results in an evaluation of true and specifies the attribute *GenVerify*, determines that the data to which this condition has been applied should be used to script the verification part of the function *WebTcpipRecvProtoEx*.

Guided TcpRuleRecvProto Example

This example illustrates how to write a recording rule for the Siebel 6 Thin Client.

This client is an ActiveX control that runs in a Web browser. A recorded script consists of one *WebPageUrl*() call followed by TCP/IP traffic from the ActiveX control. Without recording rules, the server responses are scripted by *WebTcpipRecvExact*() calls.

Portion of a Recorded Siebel 6 TCP/IP Script

```
WebTcpipSendBin(hWeb0,
  "\h00000030000000000000000000000001" // ···0·····------ 00000000
  "\h0000000C0000001C0000019100000000" // ·····------····· 00000010
  "\h00000002000000258000000C00000000" // ···%···X------- 00000020
  "\h00000000", 52);                   // ····            00000030
WebTcpipRecvExact(hWeb0, NULL, 40);
```

Server responses can be analyzed with the help of TrueLog Explorer.

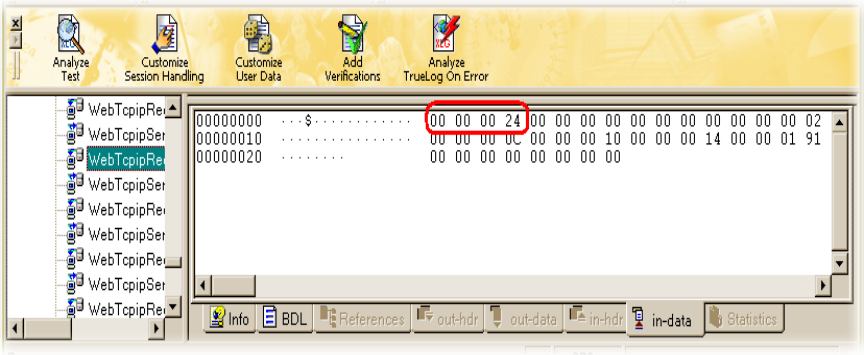


Figure 5 - Analyzing Server Responses with TrueLog Explorer

Each response contains a protocol header consisting of 4 bytes that specify the number of bytes following the protocol header. This length specification is in big endian notation (most significant byte first).

With these findings a recording rule can be written, as shown below.

Recording Rule for Siebel 6 Thin Client TCP/IP Traffic

```
<?xml version="1.0" encoding="UTF-8"?>
<RecordingRuleSet>

  <TcpRuleRecvProto>
    <Name>Siebel TCP Protocol</Name>
    <Identify>
      <LengthOffset>0</LengthOffset>
      <LengthLen>4</LengthLen>
    </Identify>
  </TcpRuleRecvProto>
```

```
</RecordingRuleSet>
```

This rule specifies that the protocol header contains the length of the data block at offset 0 using 4 bytes.

Using this rule, the Recorder generates scripts that use the function WebTcpipRecvProto() for the server responses, as shown below.

Portion of a Recorded Script using Recording Rules

```
WebTcpipSendBin(hWeb0,
  "\h00000030000000000000000000000001" // ···0·····------ 00000000
  "\h0000000C0000001C0000019100000000" // ·····---------- 00000010
  "\h0000001F00000258000000C00000000" // ·------X-------- 00000020
  "\h00000000", 52);                  // ····            00000030
WebTcpipRecvProto(hWeb0, 0, 4);
```

Scripts recorded with this rule replay correctly even when the number of bytes to be received differs from the number of bytes received during recording.

TcpRuleRecvUntil

The *TcpRuleRecvUntil* rule type specifies protocols whereby the end of a server response can be detected by searching for a terminating byte sequence.

Identify\TermData Attribute

Type: "Binary Data"

Default: (empty)

This attribute specifies the terminating byte sequence of the protocol.

Identify\IgnoreWhiteSpaces Attribute

Type: "Boolean Values"

Default: false

This attribute specifies whether or not the Recorder should ignore white spaces while searching for terminating data.

Scripted terminating data (parameter *sPattern* of the function *WebTcpipRecvUntil*) exactly reflects what was seen during recording and therefore, with respect to white spaces, may differ from what is specified in the property *Identify\TermData*.

Conditions

Additional conditions can be specified to exclude the "detection" of protocols in situations where server responses coincidentally resemble protocol specifications.

See "Conditions" for more information regarding conditions.

See section "Condition Evaluation Environment" for more information regarding what may be subject to conditions.

Guided TcpRuleRecvUntil Example

This example shows how to improve recorded scripts for telnet sessions.

Recording a telnet session without recording rules results in scripts that have lots of **WebTcpipRecvExact**() function calls, as shown below.

Portion of a Recorded Telnet Script Without Recording Rules

```
WebTcpipSend(hWeb0, "l");
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipSend(hWeb0, "s\r\n");
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipRecvExact(hWeb0, NULL, 2);
WebTcpipRecvExact(hWeb0, NULL, 1220);
```

TrueLog Explorer is used to analyze server responses. Each server response ends with a command prompt such as shown in Figure 6.

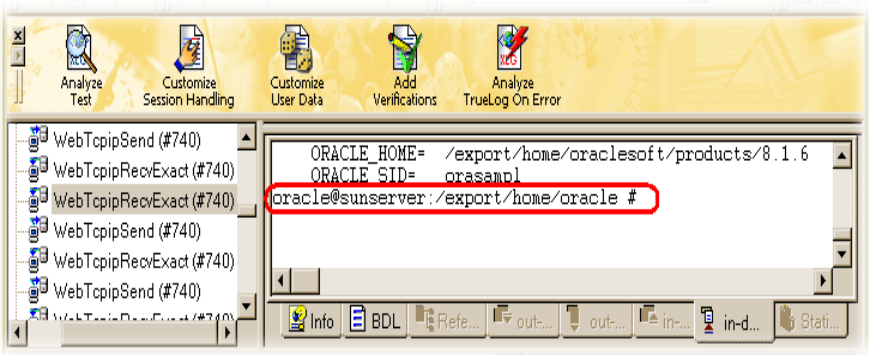


Figure 6 - Server Response in a Telnet Session

The command prompt ends with the three-character sequence *blank*, *hash*, *blank*, which seems a reasonable choice for the terminating data of a server response.

A recording rule can be written for this kind of server response. The rule instructs the Recorder to watch for server responses that end with the special terminating sequence, and to script the function *WebTcpipRecvUntil*() instead of *WebTcpipRecvExact*() for those server responses.

TCP/IP Recording Rule for Telnet

```
<?xml version="1.0" encoding="UTF-8"?>
<RecordingRuleSet>

  <TcpRuleRecvUntil>
    <Name>Telnet Command Prompt</Name>
    <Active>true</Active>
    <Identify>
      <TermData> # </TermData>
      <IgnoreWhiteSpaces>false</IgnoreWhiteSpaces>
    </Identify>
    <Conditions>
      <NoBlockSplit>true</NoBlockSplit>
    </Conditions>
  </TcpRuleRecvUntil>

</RecordingRuleSet>
```

The terminating data is specified in the attribute *Identify\TermData*.

Since the terminating data contains significant white spaces, not to ignore white spaces must be specified in the attribute *Identify\IgnoreWhiteSpaces*.

The rule should not be applied if a server response coincidentally contains the same terminating sequence elsewhere in the response. Therefore the condition *NoBlockSplit* should be specified. This means that the end of the terminating data must be aligned with the end of a TCP/IP packet received from the telnet server; otherwise the rule won't be applied.

Portion of a Recorded Telnet Script using a Recording Rule

```
WebTcpipSend(hWeb0, "l");
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipSend(hWeb0, "s");
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipSend(hWeb0, "\r\n");
WebTcpipRecvUntil(hWeb0, NULL, 0, NULL, " # ");
```

This example shows that the rule works. This script is better equipped to handle varying server responses during replay.

Note that the terminating sequence may be different for other Telnet servers and may depend on operating system and user settings. Therefore the rule shown here is not a general rule that works for all Telnet servers. It can however easily be adapted by changing the terminating character sequence. With the help of TrueLog Explorer, it's easy to determine suitable terminating byte sequences for other Telnet servers.

StringScriptingRule

The purpose of the *StringScriptingRule* is to hook into the process of scripting strings.

Whenever the Web recorder generates a string into the script (no matter where, this can be any parameter of an API function or a form field name or value), rules of this type are evaluated and may result in special actions, so that the string is not generated "as is", but processed in some way.

Structure

The basic structure of a *StringScriptingRule* is very similar to the structure of a *HttpParsingRule*. It also consists of two sections named *Search* and *ScriptGen* (see example below).

Example

```
<StringScriptingRule>
  <Name>Replace TimeStamp</Name>
  <Active>true</Active>
  <Search>
    <SearchIn>FormFieldValue</SearchIn>
    <LB>
      <Offset>0</Offset>
    </LB>
    <RB>
      <Length>0</Length>
    </RB>
  </Search>
  <ScriptGen>
    <Action>CheckTimeStamp</Action>
  </ScriptGen>
</StringScriptingRule>
```

Section Search

While the details of the *Search* section of a *HttpParsingRule* are applied to any HTTP response (with the goal to extract rule hits which are candidates for future replacements), the *Search* section of a *StringScriptingRule* is applied to the string being scripted, with the goal to extract substrings that should be treated in a special way. All techniques how substrings can be extracted (see "Section Search - Finding Rule Hits") can also be used in the *Search* section of a *StringScriptingRule*.

The only difference is the meaning of the attribute *SearchIn*:

For a *HttpParsingRule*, the attribute *SearchIn* specifies where to look for rule hits. Allowed values are: *Header*, *Body*, *All*.

For a *StringScriptingRule*, the attribute *SearchIn* specifies a list of script locations where this rule should be applied. This can be a list of script locations for the condition type *Scripting*, as described in "Scripting Condition".

Additionally, there are two new script locations *HeaderName* and *HeaderValue*, which identify the two parameters of the function *WebHeaderAdd*. See "Additional ApplyTo values" for detailed information.

There is one additional possibility how to search for substrings (which is also applicable in the *Search* section of a *HttpParsingRule*):

The attribute *Special* allows to specify a special search algorithm. The only value currently supported is *SiebelParam*. This will search the string for any substring that looks like a length-prefixed value as it is used in the Siebel 7 Web application.

Example

```
<StringScriptingRule>
  <Name>SiebelParam Function Call</Name>
  <Active>true</Active>
  <Search>
    <SearchIn>FormFieldValue</SearchIn>
    <Special>SiebelParam</Special>
    <Conditions>
    </Conditions>
  </Search>
  <ScriptGen>
    <Action>CheckSiebelParam</Action>
  </ScriptGen>
</StringScriptingRule>
```

Section ScriptGen

The section *ScriptGen* of a *StringScriptingRule* is different from the section *ScriptGen* of a *HttpParsingRule*.

The following attributes are allowed.

Attribute Action

This attribute specifies which action should be taken for the substring identified by the Search section.

The following values are allowed:

CreateVariable

This will create a variable initialized to the value of the string, and the variable will be substituted in the script instead of the original value.

CheckTimeStamp This will create the function call *GetTimeStamp()* instead of the original string, **if** the original string indeed looks like a plausible timestamp value.

CheckSiebelParam Checks if the original string is structured like a length prefixed value like used in the Siebel 7 Web application. If so, it will replace the original value with the function call *Siebel7Param(...)*.

Example: "5*Value" will become: Siebel7Param("Value")

DecodeSiebelJavaScriptString Checks if the original string contains special characters that are usually escaped with a backslash within JavaScript code, modifies the string in a way it would be if it was embedded in JavaScript code, and creates the function *SiebelDecodeJsString(...)* with the encoded string as parameter. The function *SiebelDecodeJsString* will reverse this encoding during script replay, so that the same network traffic is generated. The purpose of this is that the modified parameter may now be parseable by *HttpParsingRules*, which might not be possible without this substitution.

CheckSiebelDateTime Checks if the original string looks like a date/time combination in the format which is sent by the Siebel Web client, and transforms it to an equivalent date/time combination in the format which appears in server responses to the Siebel Web client. If this is the case, the wrapper function *SiebelDateTime* is recorded which undoes this transformation during script replay. The purpose of this is that date/time combinations can then be parsed by the other parsing rules, because they appear in the script in the same format as they appear in server responses.

CheckSiebelDate The same as *CheckSiebelDateTime*, but for dates only. Records the wrapper function *SiebelDate*.

CheckSiebelTime The same as *CheckSiebelDateTime*, but for times only. Records the wrapper function *SiebelTime*.

CheckSiebelPhone The same as *CheckSiebelDateTime*, but for phone numbers. Records the wrapper function *SiebelPhone*.

For examples, see the recording rules of the Siebel 7 Web SilkEssential.

Example (create variable for the quantity in an online shop):

This example assumes that the quantity is sent in a form field named *qty*.

```
<?xml version="1.0" encoding="UTF-8"?>
<RecordingRuleSet>

  <StringScriptingRule>
    <Name>Parameterize item quantity</Name>
    <Active>true</Active>
    <Search>
      <SearchIn>FormFieldValue </SearchIn>
      <LB>
        <Offset>0</Offset>
      </LB>
      <RB>
```

```
        <Length>0</Length>
      </RB>
      <Conditions>
        <CompareData>
          <ApplyTo>FormFieldName</ApplyTo>
          <Length>0</Length>
          <Data>qty</Data>
        </CompareData>
      </Conditions>
    </Search>
    <ScriptGen>
      <Action>CreateVariable</Action>
      <VarName>Quantity</VarName>
    </ScriptGen>
  </StringScriptingRule>

</RecordingRuleSet>
```

Attributes VarName, VarNamePrefix, IsExternalVar

These attributes are only applicable if the *Action* attribute is *CreateVariable*. This allows to specifiy eiter a variable name (attribute *VarName*) or a prefix for the variable name (attribute *VarNamePrefix*). In the latter case, the actual variable name will be built from the given prefix and the actual string value during recording.

The attribute *IsExternalVar* (boolean, Default: *false*) can be used to suppress the scripting of a declaration for the variable. This is useful if it is known that this variable is already declared in some bdh-file within a SilkEssential.

HttpScriptingRule

The rule type *HttpScriptingRule* allows to hook into script generation decisions the recorder has to make. It allows to override the default heuristics the recorder employed prior to this rule type and still employs in the absence of such rules.

Structure

The basic structure of a *HttpScriptingRule* is quite simple. The only tag allowed, additional to the commen tags *Name* and *Active*, is the tag *Action*. The *Action* tag specifies which decision is to be hooked.

The actual decision is implemented with conditions. The result of evaluating the conditions is the return value of such a rule.

Conditions have access to the HTTP request / response which is being scripted.

Example

```
<HttpScriptingRule>
  <Name>Suppress some cookie</Name>
  <Active>true</Active>
  <Action>SuppressCookie</Action>
  <Conditions>
    ...
  </Conditions>
</HttpScriptingRule>
```

HttpScriptingRule Actions

NoHtml, ForceHtml

The recorder will evaluate *HttpScriptingRules* with Action *NoHtml* or *ForceHtml* whenever it needs a decision if a HTTP response body is HTML or not. In the absence of such rules or if no such rules return true, the recorder inspects the content-type header of the HTTP response.

This rule type is useful to suppress or force the scripting of a page-level function in cases where the default recording result is not satisfactory.

Example

Often a "404 Not Found" response comes back with an HTML error description, which will cause the scripting of a *WebPageUrl* or *WebPageLink*, even when a *WebPageAddUrl* would be more appropriate.

```
<HttpScriptingRule>
  <Name>No HTML for zip files</Name>
  <Active>true</Active>
  <Action>NoHtml</Action>
  <Conditions>
    <CompareData>
      <ApplyTo>Http.Initial.Request.Url.Ext</ApplyTo>
      <Data>zip</Data>
    </CompareData>
  </Conditions>
</HttpScriptingRule>
```

NoFuzzyFormDetection, ForceFuzzyFormDetection

These Actions of a *HttpScriptingRule* can be used to override the setting "Fuzzy form detection" from the profile settings for individual HTTP requests.

Example

This rule allows fuzzy form detection (provided this is enabled in the profile settings) only for forms which contain a form field named *sid*.

```
<HttpScriptingRule>
  <Name>No fuzzy form detection except for forms with
field sid</Name>
  <Active>true</Active>
  <Action>NoFuzzyFormDetection</Action>
```

```
    <Conditions>
      <Not>
        <Exists>
          <ApplyTo>Form.Field(Name:sid)</ApplyTo>
        </Exists>
      </Not>
    </Conditions>
  </HttpScriptingRule>
```

NoDynLinkParsing, ForceDynLinkParsing

These *Actions* of a *HttpScriptingRule* can be used to override the setting "Dynamic link parsing" from the profile settings for individual HTTP requests.

Example

This example forces dynamic link parsing (even if it turned off in the profile settings) for HTTP requests where the query string contains the string **sid=**.

```
  <HttpScriptingRule>
    <Name>Force dynamic link parsing for some query
strings</Name>
    <Active>true</Active>
    <Action>ForceDynLinkParsing</Action>
    <Conditions>
      <FindData>
        <ApplyTo>Http.Initial.Request.Url.QueryData</
ApplyTo>
        <Data>sid=</Data>
      </FindData>
    </Conditions>
  </HttpScriptingRule>
```

NoDynUrlParsing, ForceDynUrlParsing

These *Actions* of a *HttpScriptingRule* can be used to override the setting "Dynamic URL parsing" from the profile settings for individual HTTP requests.

NoUrlEncoded, ForceUrlEncoded

With HTTP POST requests, the recorder needs to decide if the request body should be scripted as a form in the *dclform* section or if it is better to use a *Bin function (*WebPagePostBin* and friends) instead.

These *Actions* can be used to override the default decision of the recorder. By default, the recorder looks at the content-type request header. The recorder will script a form in the *dclform* section if the content-type is *application/x-www-form-urlencoded* and will script a *Bin function otherwise.

This can be used to get better recording results for ill-behaved applications. A common example are applications which send XML data with a POST request,

but incorrectly send the content-type header *application/x-www-form-urlencoded*.

Example

The following rule is an example for handling this case.

```
<HttpScriptingRule>
  <Name>Suppress XML to form conversion</Name>
  <Active>true</Active>
  <Action>NoUrlEncoded</Action>
  <Conditions>
    <CompareData>
      <ApplyTo>Http.Initial.Request.Body</ApplyTo>
      <Data>&lt;?xml</Data>
    </CompareData>
  </Conditions>
</HttpScriptingRule>
```

DefinePageName

This action allows defining an alternate page name for a HTML page. The recorder generates various strings (page timer name, name of stored context variable, ...) based on the name of a HTML page. By default the recorder uses the title of an HTML page for the page name, or "Unnamed page" if no title exists.

Such a rule must have conditions which:

- return *true*
- save a non-empty string to the variable *PageName* (by means of the tag *SaveAs* of a condition)

The conditions have access to the default page name the recorder would use through the variable *DefaultPageName*.

The following example checks if the default page name does not exist (the recorder would use "Unnamed page" then), and defines the page name to be the URL of the HTTP document instead, if it is at least 3 characters long.

Example

```
<HttpScriptingRule>
  <Name>Define Page Name</Name>
  <Active>true</Active>
  <Action>DefinePageName</Action>
  <Conditions>
    <Not>
      <Exists>
        <ApplyTo>DefaultPageName</ApplyTo>
      </Exists>
    </Not>
    <CheckRange>
      <ApplyTo>Http.Initial.Request.Url</ApplyTo>
      <Range>3-</Range>
```

```
      <SaveAs>PageName</SaveAs>
      <SaveMode>Replace</SaveMode>
    </CheckRange>
  </Conditions>
</HttpScriptingRule>
```

SuppressCookie, CommentCookie

In some cases the function *WebCookieSet* may be recorded where this is not useful or even incorrect.

This may happen when there is a tight succession of client side "Cookie" and server side "Set-Cookie" headers in embedded objects of a page, with tight timing. In such a case it is possible that a browser does not send a cookie or sends a cookie with an older value, although the browser already received a "Set-Cookie" header which should cause it to send a different cookie value.

In such cases, these *Actions* can be used to script *WebCookieSet* function calls commented or suppress the recording completely.

Example

This rule suppresses scripting of any *WebCookieSet* function call if the cookie name is *PS_TOKENEXPIRE* (from the Peoplesoft SilkEssential).

```
<HttpScriptingRule>
  <Name>Suppress Cookie PS_TOKENEXPIRE</Name>
  <Active>true</Active>
  <Action>SuppressCookie</Action>
  <Conditions>
    <CompareData>
      <ApplyTo>Cookie</ApplyTo>
      <Data>PS_TOKENEXPIRE</Data>
    </CompareData>
  </Conditions>
</HttpScriptingRule>
```

SuppressFrameName, SuppressLinkName, SuppressFormName, SuppressCustomUrlName

Functions like *WebPageLink*, *WebPageSubmit* and *WebPageSetActionUrl* use a name (of a link, form or parsed URL resp.), a frame name, and an ordinal number to reference a link, form, or parsed URL.

In cases where e.g. a link or form name changes dynamically, it is common practice to do the so-called "name to number customization", which means one does not specify the name (of the link or form), but instead specifies the overall ordinal number within the page or within the frame.

These *Actions* can be used to let this customization be done already by the recorder.

Example

Assume a shop application which uses form names which are built by adding the numeric session ID to the string "cart_".

This rule will not record the form name within the *WebPageSubmit* calls, but will record NULL instead, with an ordinal number which references the form in the entire page.

```
<HttpScriptingRule>
  <Name>Suppress XML to form conversion</Name>
  <Active>true</Active>
  <Action>SuppressFormName</Action>
  <Conditions>
    <RegExpr>
      <ApplyTo>FormName</ApplyTo>
      <Data>cart_[0-9]*</Data>
      <ExpectMatch>Complete</ExpectMatch>
    </RegExpr>
  </Conditions>
</HttpScriptingRule>
```

MapFunctionName

This *Action* can be used to define a function name mapping, so that the recorder scripts a wrapper function (which needs to be present, e.g. in a BDH file, for the script to be compileable) instead of the original function. Such a wrapper function must have the same parameter list as the original function, because this rule does not modify the function parameters in any way.

The recorder scripts the wrapper function instead of the original function, if the condition returns *true* and in the course of the evaluation of the condition something was saved to the variable *FunctionName*.

If the conditions also save a non-empty string to the variable *BdhFileName*, the recorder will also generate a *use* statement to include the given BDH file into the script.

Example

This rule replaces each *WebPageLink* function call with a *MyWebPageLink* function call.

```
<HttpScriptingRule>
  <Name>Replace WebPageLink with my wrapper function</
Name>
  <Active>true</Active>
  <Action>MapFunctionName</Action>
  <Conditions>
    <CompareData>
      <ApplyTo>DefaultFunctionName</ApplyTo>
      <Data>WebPageLink</Data>
    </CompareData>
    <Exists>
      <ApplyTo>Literal:MyWebPageLink</ApplyTo>
      <SaveAs>FunctionName</SaveAs>
```

```
      </Exists>
      <Exists>
        <ApplyTo>Literal:MyFunctions.bdh</ApplyTo>
        <SaveAs>BdhFileName</SaveAs>
      </Exists>
    </Conditions>
  </HttpScriptingRule>
```

AddInitFunction

This *Action* can be used to add function calls to the *TInit* or *TMain* transaction. This is the only *Action* which allows additional tags, besides the *Conditions* tag.

While conditions are allowed, they are not useful because they can not reference any data.

There can be any number of functions specified, each one with its own *Function* tag.

Within the *Function* tag, there must be the tag *FunctionName* and optionally the tag *BdhFileName*.

There can be any number of *Param* tags to specify parameters. Each parameter must have a *Value* tag and a *Type* tag. The *Type* tag can be either *String* or *Const*. *String* will put the parameter in quotes, *Const* won't.

Example

```
<HttpScriptingRule>
  <Name>Automatically insert my useful function</Name>
  <Active>true</Active>
  <Action>AddInitFunction</Action>
  <Function>
    <FunctionName>InitMyLogLibrary</FunctionName>
    <BdhFileName>Log.bdh</BdhFileName>
    <Param>
      <Value>C:\Logs\xx.log</Value>
      <Type>String</Type>
    </Param>
    <Param>
      <Value>true</Value>
      <Type>Const</Type>
    </Param>
  </Function>
</HttpScriptingRule>
```

This will generate the following script fragments:

```
use "Log.bdh"

  transaction TInit
  begin
    // ...
    InitMyLogLibrary("C:\\Logs\\xx.log", true);
```

```
end TInit;
```

Silk Performer allows the tag *Location* with the following values:

- TInit
- TMainBegin
- TMainEnd

NoAddUrl

This *Action* can be used to suppress the generation of a *WebPageAddUrl* function for particular HTTP requests. The recorder will then choose the next best way to record this request, usually a *WebUrl* function will be scripted instead, but it might also be another function (e.g. *WebPageLink*, *WebPageUrl*, ...) depending on circumstances.

This can be useful, because parsing rules never parse data from embedded objects of a web page. If a HTTP request is scripted by a *WebUrl* function call instead, it is possible to apply HTTP parsing rules to this request.

SuppressRecording

This *Action* can be used to suppress the recording of individual HTTP requests. It is used by Silk Performer's Oracle Forms support to suppress the recording of HTTP requests which come from an Oracle Forms applet which is recorded at a different API level.

ForceContextless

This *Action* can be used to suppress the recording of HTTP requests with a context-full function like *WebPageLink* or *WebPageSubmit*. Instead, the recorder will record an appropriate context-less function instead.

DontUseHtmlVal

This *Action* can be used to suppress the recording of the *<USE_HTML_VAL>* tag for individual form fields, even if the recorder would do so otherwise.

This can be useful if the form field value is required in the script, e.g. because another recording rule should be applied to the value.

Example

In the *PeopleSoft* project type, the user name in the login form is forced to be in the script with this rule (even if the user name input field is pre-populated), and then another rule of type *StringScriptingRule* with the action *CreateVariable* creates a variable for it. In this way the script is better prepared for randomization of the login data.

NoQueryToForm, ForceQueryToForm

These *Actions* can be used to override the profile setting "Convert URL query strings to forms" on a per HTTP request basis.

ProxyEngineRule

The rule type *ProxyEngineRule* allows to control some aspects of the ProxyEngine's operation.

Structure

The basic structure of a *ProxyEngineRule* is quite simple. The only tag allowed, additional to the common tags *Name* and *Active*, is the tag *Action*. The *Action* tag specifies what to do.

The actual decision is implemented with conditions. The result of evaluating the conditions is the return value of such a rule.

Example

```
<ProxyEngineRule>
  <Name>Switch on GUI recording for Jacada</Name>
  <Active>true</Active>
  <Action>AddAppletParam</Action>
  <Conditions>
    <FindData>
      <ApplyTo>Attribute.ARCHIVE</ApplyTo>
      <Data>clbase.jar</Data>
    </FindData>
    <Exists>
      <ApplyTo>Literal:GUIMode</ApplyTo>
      <SaveAs>Param.Name</SaveAs>
      <SaveMode>Replace</SaveMode>
    </Exists>
    <Exists>
      <ApplyTo>Literal:GUIRecorder</ApplyTo>
      <SaveAs>Param.Value</SaveAs>
      <SaveMode>Replace</SaveMode>
    </Exists>
  </Conditions>
</ProxyEngineRule>
```

ProxyEngineRule Actions

AddAppletParam

This *Action* allows modifying applets contained in HTML before the HTML is forwarded to the client. The modification will not be visible in the log files. If the conditions evaluate to *true*, and the conditions save values to "Param.Name" and "Param.Value", this parameter will be added to the applet tag.

Any *ProxyEngineRule* with Action *AddAppletParam* will be evaluated once per applet.

The conditions have access to the attributes of the applet using the following syntax:

```
Attribute.attr
```

attr may be any attribute of the applet tag.

RemoveAppletParam, ChangeAppletParam

This *Action* allows modifying applets contained in HTML before the HTML is forwarded to the client. The modification will not be visible in the log files. If the conditions evaluate to *true*, the parameter will be removed from the applet (*RemoveAppletParam*), or its value will be changed to whatever was saved by the conditions to "Param.Value" (*ChangeAppletParam*).

Any *ProxyEngineRule* with one of these *Actions* will be evaluated once per applet parameter.

The conditions can access the name and value of the applet parameter by "Param.Name" and "Param.Value".

Example

```
<ProxyEngineRule>
  <Name>Remove Cabbase applet param</Name>
  <Active>true</Active>
  <Action>RemoveAppletParam</Action>
  <Conditions>
    <Or>
      <CompareData>
        <ApplyTo>Param.Name</ApplyTo>
        <Data>Cabbase</Data>
        <Length>0</Length>
      </CompareData>
      <CompareData>
        <ApplyTo>Param.Value</ApplyTo>
        <Data>.cab</Data>
        <Offset>-4</Offset>
      </CompareData>
    </Or>
```

```
    </Conditions>
  </ProxyEngineRule>
```

DetectProtoFtp, DetectProtoSmtp

In the course of protocol detection, it is sometimes hard do distinguish between FTP and SMTP, since these protocols start with very similar traffic.

The Actions *DetectProtoFtp* and *DetectProtoSmtp* can be used to force detecting FTP or SMTP in cases where the recorder would misdetect the protocol otherwise.

The conditions can assess the following pieces of information in the *ApplyTo* tag:

- *WelcomeMsg*
  The welcome message of the server
- *NoopResponse*
  The server response to a NOOP command which is sent by the ProxyEngine
- *TargetPort*
  The target port

DontModifyRequestHeader, DontModifyResponseHeader

The recorder routinely modifies some request/response headers of HTTP traffic to ensure best recording results with common browsers. However, some not so common user agents may misbehave when recorded.

These *Actions* allow to suppress the modification of request/response HTTP headers for individual requests, and thus allow to solve such recording problems.

Conditions can reference the header name and value which are subject to be modified in the *ApplyTo* tag with the values *HeaderName*, respectively *HeaderValue*.

Example

This example is from the Flex/AMF3 project type.

This rule suppresses the modification of the *Pragma* and *Cache-Control* response header, if there is no *Accept-Language* request header.

```
  <ProxyEngineRule>
    <Name>Suppress modification of some server response
headers for HTTP requests coming from Shockwave/Flash</Name>
    <Active>true</Active>
    <Action>DontModifyResponseHeader</Action>
    <Conditions>
      <Not>
        <Exists>
```

```
          <ApplyTo>Http.Initial.Request.Header.Accept-
Language</ApplyTo>
        </Exists>
      </Not>
      <Or>
        <CompareData>
          <ApplyTo>HeaderName</ApplyTo>
          <Data>Pragma</Data>
          <Length>0</Length>
        </CompareData>
        <CompareData>
          <ApplyTo>HeaderName</ApplyTo>
          <Data>Cache-Control</Data>
          <Length>0</Length>
        </CompareData>
      </Or>
    </Conditions>
  </ProxyEngineRule>
```

DontDetectProtoHttp

Sometimes it is necessary to record a TCP connection on TCP/IP level, although the automatic protocol detection would detect HTTP.

The Actions *DontDetectProtoHttp* can be used to supress detecting HTTP in such cases.

The conditions can assess the following pieces of information in the *ApplyTo* tag:

- PeekData
  The first chunk of data sent by the client.

Conditions

Conditions are an important aspect of *HttpParsingRule*, *TcpRuleRecvProto* and *TcpRuleRecvUntil* rule types.

They are a powerful means of specifying exactly when rules should be applied.

Introduction

Compound Conditions

Complex conditions can be created using Boolean operators *And*, *Or* and *Not*.

Example

```
<And>
```

```
<Or>
  <SomeBasicCondition> …     </SomeBasicCondition>
  <SomeBasicCondition> …     </SomeBasicCondition>
  <Not>
    <SomeBasicCondition> …     </SomeBasicCondition>
  </Not>
</Or>
<Not>
  <And>
    <SomeBasicCondition> …     </SomeBasicCondition>
    <SomeBasicCondition> …     </SomeBasicCondition>
  </And>
</Not>
</And>
```

Extended Boolean Values

The result of a condition, once evaluated, is an extended Boolean value. Extended Boolean values have the values true, unknown or false. Think of the value unknown as: *"Cannot be evaluated now, but may be evaluated when more data becomes available"*.

This is important for *TcpRuleRecvProto* and *TcpRuleRecvUntil* type rules. If conditions in a TCP rule result in a value of unknown, the Recorder defers scripting and reevaluates conditions when more data arrives from the server.

Basic Conditions

There are a number of basic condition types that execute checks and can be combined (using the Boolean conditions *And*, *Or* and *Not*) to build complex compound conditions.

Basic condition types include:

- CheckRange — Checks to see if a numeric value lies within a given range.
- ResultLen — A special form of the condition CheckRange.
- CompareData — Compares data.
- FindData — Searches data.
- Verify — A special form of the condition CompareData.
- RegExpr — Applies a regular expression.
- NoBlockSplit — Checks block boundaries.
- Scripting — Checks for the type of string being scripted.

Condition evaluation environment

A condition is evaluated within an *environment*. Through the environment, the condition has access to a number of strings to which the condition can be applied. Environment configuration differs with each rule type.

See section "Condition Evaluation Environment" for details.

Conditions operate on data

Most conditions (except the *Scripting* condition) apply specific checks on specific blocks of data. There are flexible means of specifying what data is to be checked. See section "Specifying Data for Conditions" for more information.

Specifying Data for Conditions

Most conditions (except the *Scripting* condition) operate on specific blocks of data.

Conditions have sets of attributes that specify what data is to be used and what should be done if required data is not available.

These attributes are:

- ApplyTo
- Offset
- Length
- IfNull
- UseDataAvail

The basic idea is:

1. ApplyTo specifies a block of data (e.g., a request URL, a response body, the string to be scripted, etc.).
2. Offset and Length are optionally used to target data subsets.

ApplyTo

Type: "Strings"

Default: Self

ApplyTo specifies the data that the condition operates on.

ApplyTo is resolved within the current environment. The valid values for *ApplyTo* are therefore dependent on the environment configuration, which is different for each rule type. See section "Condition Evaluation Environment" for details.

Literal

With the *Literal* value in the *ApplyTo* tag it is possible to specify a literal as data for conditions. This is useful in conjunction with the *Exists* condition and the *SaveAs* attribute.

Example:

```
<Conditions>
  <Exists>
    <ApplyTo>Literal: - </ApplyTo>
    <SaveAs>PageName</SaveAs>
    <SaveMode>AppendSeparator</SaveMode>
  </Exists>
  <Exists>
    <ApplyTo>Form.Submit.Field(Name:SWECmd).Value</ApplyTo>
    <SaveAs>PageName</SaveAs>
    <SaveMode>Append</SaveMode>
  </Exists>
</Conditions>
```

Additional ApplyTo values

Additional *ApplyTo* values in the ScriptGen Section of a HttpParsingRule

In addition to the possible values *LinkName*, *FormName*, *FormFieldName*, *FormFieldValue* and *TargetFrame* in the ScriptGen section of a *HttpParsingRule* (see "Access to special strings"), there are four new possibilities:

CustomUrlName	The current name of a custom URL, if used in a function *WebPageLink*, *WebPageSetActionUrl* or *WebPageQueryParsedUrl*.
PostedData	The value of (possibly binary) posted data that appears as a parameter to *WebPagePostBin* and similar functions.
HeaderName	The name of a HTTP header, when the function *WebHeaderAdd* is recorded (1st parameter of this function).
HeaderValue	The value of a HTTP header, when the function *WebHeaderAdd* is recorded (2nd parameter of this function).

Access to forms

Both rules of type *HttpParsingRule* (ScriptGen section), *HttpScriptingRule* and *StringScriptingRule* (Search section) also have convenient access functions for form data according to the following syntax:

```
"Form" [ ".Query" | ".Body" | ".Submit" ]
```

This allows specifying the form in the query string ("Query") or in the request body ("Body") of the current HTTP request.

The value *Submit*, which is the default, is automatically equivalent to *Query* if the HTTP request uses the method *GET*, and to *Body* if the HTTP request uses the method *POST*.

Once a form is specified according to the syntax above, it is possible to extract details about this form:

- *ActionUrl*
  Specifies the action URL of a form. It is possible to get more details of the action URL (see "Access to HTTP request/response pairs" for detailed information).
- *Encoding*
  Specifies the encoding to be scripted (e.g. *ENCODE_BLANKS*)

Examples:

```
<ApplyTo>Form.ActionUrl</ApplyTo>
<ApplyTo>Form.ActionUrl.Host</ApplyTo>
<ApplyTo>Form.ActionUrl.Coords</ApplyTo>
<ApplyTo>Form.Query.Encoding</ApplyTo>
<ApplyTo>Form.Body.Encoding</ApplyTo>
<ApplyTo>Form.Submit.Encoding</ApplyTo>
```

Moreover, it is possible to specify the individual form fields according to the following syntax:

```
".Field(" ( "Name" | "Value" | "Index" ) ":" Selector ")" "."
    ( "Name" | "Value" | "Encoding" | "Usage" )
```

So it is possible to reference a form field by name, value, or index (zero-based), and extract the name, value, encoding, or usage of such a form field.

Encoding means one of the following: "ENCODE_FORM" etc. (see online help for encodings).

Usage means one of the following: "SUPPRESS", "USE_HTML_VAL", "USE_SCRIPT_VAL".

Examples:

```
<ApplyTo>Form.Submit.Field(Name:SWECmd).Value</ApplyTo>
<ApplyTo>Form.Field(Name:sid).Usage</ApplyTo>
<ApplyTo>Form.Field(Name:sid).Encoding</ApplyTo>
<ApplyTo>Form.Field(Index:0).Name</ApplyTo>
<ApplyTo>Form.Field(Index:2).Value</ApplyTo>
<ApplyTo>Form.Field(Value:Submit with GET).Name</ApplyTo>
```

Offset, Length

Type: "Signed Numbers"

Default value: 0

Offset and *Length* can be used to target subsets of data referenced with *ApplyTo*.

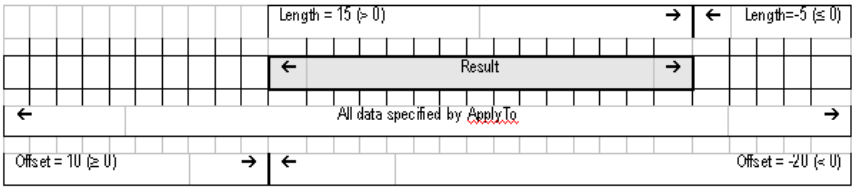
First, *Offset* is applied: If *Offset* >= 0, *Offset* specifies the number of bytes that are to be removed from the beginning of the data.

If *Offset* < 0, *-Offset* specifies the number of bytes that are to be kept, counting backward from the last byte.

Second, *Length* is applied:

If *Length* > 0, *Length* specifies the number of bytes that are to be used, counting forward from the first byte.

If *Length* <= 0, *-Length* specifies the number of bytes that are to be removed, counting backward from the last byte.



Offset and *Length* calculations may not be possible, or may be only partially possible.

Example 1:

ApplyTo returns 30 Bytes of data, *Offset* specifies "40" => *Offset* calculation is not possible.

In this case the resulting data has the value Null.

Example 2:

ApplyTo returns 30 Bytes of data, *Length* specifies "35" => *Length* calculation is possible, however the length must be adjusted to the maximum available value of "30."

In this case the resulting data may have the special value Null or it may equal the maximum data available, depending on the condition attribute *UseDataAvail*.

UseDataAvail

Type: "Boolean Values"

Default value: true

This attribute specifies what data is to be used when an *Offset*/*Length* calculation is only partially possible.

If *true*, the maximum data available is used.

If *false*, the result of the *Offset*/*Length* calculation is *Null*.

IfNull

Type: "Extended Boolean Values"

Default value: unknown

This attribute specifies the return value of the condition if there is no data to operate on (i.e., if calculations with *ApplyTo*, *Offset*, *Length* and *UseDataAvail* return *Null*).

Saving Temporary Variables

Conditions which apply to a block of data can specify this data as described in "Specifying Data for Conditions".

This block of data can now be saved to a "temporary variable" so that it can be referenced by the *ApplyTo* attribute of a subsequent condition.

The new attributes of such conditions are:

- SaveAs
- SaveIfTrue
- SaveIfUnknown
- SaveIfFalse
- SaveMode
- SaveTag

SaveAs

Type: "Strings"

This attribute causes the data to be saved and specifies the name of the temporary variable to which the data is saved to.

SaveIfTrue, SaveIfUnknown, SaveIfFalse

Type: "Boolean Values"

These attributes specify if the data should be saved in case the result of the condition is *true*, *unknown* or *false*.

Default Values:

- *SaveIfTrue: true*
- *SaveIfUnknown: false*
- *SaveIfFalse: false*

SaveMode

Type: "Distinct Values"

Default value: *IfNew*

This attribute specifies how to save the data to the temporary variable.

Allowed values:

- *IfNew*
  Saves the data only if there is no value associated with the given variable name yet.
- *Replace*
  Always saves the data, replaces an existing value.
- *AppendSeparator*
  Appends the data to the existing value if the existing value is not empty, no action otherwise.
- *Append*
  Appends the data to the existing value.
- *PrependSeparator*
  Prepends the data to the existing value if the existing value is not empty, no action otherwise.
- *Prepend*
  Prepends the data to the existing value.

SaveTag

Type: "Numbers"

Default value: 0

This tag is only applicable to the *RegExpr* condition and allows to specify a tag number referencing a tagged sub-expression of the regular expression. Only the portion of the entire data that corresponds to the tagged sub-expression is saved.

SaveWhat

Type: "Distinct Values"

Default value: *Self*

A condition extracts a fragment of the entire block of data which is identified by the *ApplyTo* tag. The *SaveWhat* tag allows to specify if the extracted fragment should be saved, or the fragment left or right of the extracted fragment.

← complete data specified by **ApplyTo** -→		
← Left -→	← data extracted by the condition -→	← Right -→

Allowed values:

- *Self*
  Saves the fragment identified by the condition.

- *Left*
  Saves fragment left of *Self*.
- *Right*
  Saves the fragment right of *Self*.

Example

```
<Conditions>
  <RegExpr>
    <ApplyTo>Http.Final.Response.Body</ApplyTo>
    <Data>Instructions:\([^&lt;]+\)&lt;</Data>
    <SaveAs>Instructions</SaveAs>
    <SaveTag>1</SaveTag>
  </RegExpr>
  <FindData>
    <ApplyTo>Instructions</ApplyTo>
    <Data>Download</Data>
  </FindData>
</Conditions>
```

Condition Evaluation Environment

The condition attribute *ApplyTo* is resolved within an environment that is configured based on the rule type in which the condition is used.

This chapter explains which attribute *ApplyTo* values are allowed for conditions, depending on where conditions are used.

Access to special strings

HttpParsingRule, *TcpRuleRecvProto* and *TcpRuleRecvUntil* rule types can reference the strings *All*, *Self*, *Left*, *Right* and *Rest*.

RuleType / ApplyTo	HttpParsingRule (Search)	HttpParsingRule (ScriptGen)	TcpRuleRecvProto	TcpRuleRecvUntil
All	String being parsed (response header or body)	String being scripted	Data identified by rule	
Self	Substring of **All** that is a possible rule hit	Substring of **All** that can be replaced by a parsed value	Substring of **All** that is the protocol length specification	Substring of **All** that is the terminating data
Left	Substring of **All** that is left of **Self**			
Right	Substring of **All** that is right of **Self**			
Rest	N / A		Additional data already received, right of **All**	

Figure 7 - Common Condition Environment Configuration

Conditions in the *ScriptGen* sections of HTTP parsing rules have additional options for referencing specific data using the *ApplyTo* property:

- LinkName — The current link name (only available when scripting a *WebPageLink* function)
- FormName — The current form name (only available when scripting a *WebPageSubmit* function)
- FormFieldName — The current form field name (only available when scripting a form field name or form field value)
- FormFieldValue — The current form field value (only available when scripting a form field name or form field value)
- TargetFrame — The current target frame name (only available when scripting a *WebPageLink* or *WebPageSubmit*(*Bin*) function)

Examples:

```
<ApplyTo>All</ApplyTo>
<ApplyTo>Self</ApplyTo>
<ApplyTo>FormFieldName</ApplyTo>
```

Access to HTTP request/response pairs

Conditions used within *HttpParsingRule* rules can access details of HTTP requests / responses with the *ApplyTo* property.

Syntax:

```
"Http" [ ".Initial" | ".Final" ] ( ".Request" | ".Response"
) [ "." Component ]
```

The optional component Initial or Final is only significant in cases where HTTP requests are part of a redirection chain. In such cases, Initial returns the first HTTP request in the chain; Final returns the last request in the chain.

Final is the default for conditions in the *HttpParsingRule\Search* section.

Initial is the default for conditions in the *HttpParsingRule\ScriptGen* section.

Valid Component values for HTTP requests, plus return values:

• (empty)	Equal to *Header*
• Body	Request body
• RequestLine	Request line (i.e., Method + URL + HTTP version)
• Method	HTTP method
• Version	HTTP version
• Header	Complete request header, including request line
• Header.*	Use this to reference any HTTP request header
• Url	Complete request URL
• Url.Complete	Complete request URL
• Url.BaseUrl	URL without query string
• Url.DirectUrl	Relative request URL (without scheme and host)
• Url.BaseDirOnlyUrl	URL without query string and file name
• Url.Scheme	URL scheme (HTTP, HTTPS or FTP)
• Url.Host	Host name in URL
• Url.Port	Port in URL
• Url.Path	Path (directory plus file name)
• Url.Dir	Directory
• Url.File	File name
• Url.Ext	File extension
• Url.Username	User name
• Url.Password	Password
• Url.Query	Query string, including "?"
• Url.QueryData	Query string, excluding "?"
• Url.Coords	Image coordinates

Valid Component values for HTTP responses:

- (empty) Equal to Header
- Body Response body
- StatusLine Response status line (i.e., HTTP version plus status code and status phrase)
- Version HTTP version
- StatusCode HTTP response status code
- StatusPhrase HTTP response status phrase
- Header Complete response header, including status line
- Header.* Use this to reference any HTTP response header

Examples:

```
<ApplyTo>Http.Response.Header.Content-Type</ApplyTo>
<ApplyTo>Http.Response.Header</ApplyTo>
<ApplyTo>Http.Request.Url.QueryData</ApplyTo>
<ApplyTo>Http.Request.Body</ApplyTo>
<ApplyTo>Http.Response.StatusCode</ApplyTo>
```

CompareData Condition

The *CompareData* condition checks to see if certain data has a specific value.

It uses the attributes *ApplyTo*, *Offset*, *Length*, *UseDataAvail* and *IfNull* to determine what data is subject to the test and what the return value should be if the data isn't available.

Also see section "Specifying Data for Conditions".

In contrast to other condition types, if the *Length* property is omitted, the number of bytes specified in the *Data* property is used as the *Length* property. By omitting the *Length* property, this allows for a prefix match rather than an exact match.

The *Verify* condition is an alias for *CompareData*.

Data Attribute

Type: "Binary Data"

Default: (empty)

The *Data* attribute specifies what data is to be compared to the data referenced by *ApplyTo*, *Offset* and *Length*.

CaseSensitive Attribute

Type: "Boolean Values"

Default: false

Specifies whether to search for the data in case-sensitive or case-insensitive format.

GenVerify Attribute

This attribute is only applicable if the condition is used within a *TcpRuleRecvProto* rule.

See section "GenVerify Attribute Of Conditions" for details.

Example

```
<CompareData>
  <ApplyTo>Http.Response.Header.Content-Type</ApplyTo>
  <Data>text/css</Data>
  <IfNull>true</IfNull>
  <CaseSensitive>false</CaseSensitive>
</CompareData>
```

FindData Condition

The *FindData* condition checks to see if certain data contains a specific value.

It uses the attributes *ApplyTo*, *Offset*, *Length*, *UseDataAvail* and *IfNull* to determine what data is to be subject to the test and what the return value should be if the data is not available.

Also see section "Specifying Data for Conditions".

Data Attribute

Type: "Binary Data"

Default: (empty)

The *Data* attribute specifies what data is to be found in the data referenced by *ApplyTo*, *Offset* and *Length*.

CaseSensitive Attribute

Type: "Boolean Values"

Default: false

Specifies whether to search for the data in case-sensitive or case-insensitive format.

IgnoreWhiteSpaces Attribute

Type: "Boolean Values"

Default: false

Specifies whether or not to ignore white spaces while searching.

GenVerify Attribute

This attribute is only applicable if the condition is used within a *TcpRuleRecvProto* rule.

See section "GenVerify Attribute Of Conditions" for details.

Example

```
<FindData>
  <ApplyTo>Http.Request.Url</ApplyTo>
  <Data>/images/</Data>
  <CaseSensitive>true</CaseSensitive>
</FindData>
```

RegExpr Condition

The *RegExpr* condition applies a regular expression to certain data.

It uses the properties *ApplyTo*, *Offset*, *Length*, UseDataAvail and *IfNull* to determine what data will be subject to the regular expression test and what the return value will be if the data isn't available.

Also see section "Specifying Data for Conditions".

See Silk Performer's online help for a syntax description of regular expressions.

Data Attribute

Type: "Binary Data"

Default: (empty)

The *Data* attribute specifies the regular expression to be applied to the data referenced by *ApplyTo*, *Offset* and *Length*.

ExpectMatch Attribute

Type: "Distinct Values"

Default: Any

Specifies if the regular expression must match the complete data or if any substring matching the regular expression is sufficient.

Allowed values are:

• Any	Matching any substring is sufficient.
• Complete	Complete data must match the regular expression.

GenVerify Attribute

This attribute is only applicable if the condition is used within a *TcpRuleRecvProto* rule.

See section "GenVerify Attribute Of Conditions" for details.

Example

```
<RegExpr>
  <ApplyTo>Http.Response.Body</ApplyTo>
  <Data>&lt;html&gt;.*&lt/html&gt;</Data>
</RegExpr>
```

CheckRange Condition

The *CheckRange* condition checks to see if a numeric value lies within a given range.

It uses the properties *ApplyTo*, *Offset*, *Length*, *UseDataAvail* and *IfNull* to determine what data is to be converted to a numeric value and what the return value should be if the data isn't available or the conversion fails.

Also see section "Specifying Data for Conditions".

Range Attribute

Type: "Numeric Ranges"

Default: -

Specifies the allowed range for the numeric value.

Convert Attribute

Type: "Distinct Values"

Default: Length

Specifies how to convert the data (obtained through the properties *ApplyTo*, *Offset*, *Length*, and *UseDataAvail*) into a number.

Allowed values are:

- Length (default): The data is converted to a number by measuring the length (number of bytes) of the data.
- Parse: The data is converted to a number by assuming a textual numeric value and parsing it. If parsing fails, no range check is performed and the attribute IfNull determines the result of the condition.
- LittleEndian: The data is treated as a binary representation of a number in little endian format. If the data is empty or is longer than 4 bytes, this fails and the attribute IfNull determines the result of the condition.
- BigEndian: The data is treated as a binary representation of a number in big endian format. If the data is empty or is longer than 4 bytes, this leads to a failure and the attribute IfNull determines the result of the condition.

Example

```
<CheckRange>
  <ApplyTo>Http.Response.StatusCode</ApplyTo>
  <Range>200-299</Range>
  <Convert>Parse</Convert>
</CheckRange>
```

ResultLen Condition

The *ResultLen* condition is a special form of the *CheckRange* condition, with two differences:

1. The default value for the *ApplyTo* attribute is not *Self*, but *All*.
2. The accepted range isn't specified in the *Range* attribute, but directly within the *ResultLen* - tag.

Example

```
<ResultLen>5-15</ResultLen>
```

NoBlockSplit Condition

The *NoBlockSplit* condition is only applicable within *TcpRuleRecvProto* and *TcpRuleRecvUntil* rules.

It uses the attributes *ApplyTo*, *Offset*, *Length*, *UseDataAvail* and *IfNull* to determine what data is to be subject to the test and what the return value should be if the data isn't available.

Also see section "Specifying Data for Conditions".

In contrast to other conditions, the default value for the *ApplyTo* attribute is *All*, rather than *Self*.

Semantics

When recording TCP/IP traffic, servers may send responses in multiple TCP/IP packets (blocks). The *NoBlockSplit* condition tests to see if the end of the data obtained by *ApplyTo*, *Offset* and *Length* is identical to the end of one of the packets.

The result is *false* if the end of the tested data is not on a block boundary; otherwise the result is *true*.

Examples

```
<NoBlockSplit></NoBlockSplit>

<NoBlockSplit>
  <ApplyTo>Left<ApplyTo>
</NoBlockSplit>
```

Scripting Condition

The *Scripting* condition doesn't operate on specific data, therefore the attributes *ApplyTo*, *Offset*, *Length*, *UseDataAvail* and *IfNull* aren't available.

Attribute

Type: "Distinct Value Lists"

Default: All

The attribute value is specified directly within the *Scripting* tag, not within a sub node of the condition.

Allowed values are:

• All	An abbreviation for the complete list of other values
• Url	Scripting any URL parameter
• LinkName	Scripting a link name of *WebPageLink*
• FormName	Scripting a form name of *WebPageSubmit*
• PostedData	Scripting binary posted data (e.g., *WebPagePost*)
• FormFieldName	Scripting a form field name in the dclform section
• FormFieldValue	Scripting a form field value in the dclform section
• SetCookie	Scripting the first parameter of the function *WebCookieSet*

True is returned if the condition is evaluated while scripting a string parameter in one of the specified script locations; otherwise it's *false*.

Examples:

```
<Scripting>Url, FormFieldValue</Scripting>
<Scripting>All</Scripting>
<Scripting>LinkName, FormName, PostedData</Scripting>
```

Exists Condition

The condition type *Exists* checks if the data specified in the *ApplyTo* tag is not null. It is especially useful in conjunction with the *SaveAs* tag of conditions.

Example

```
<Conditions>
  <Or>
    <Exists>
      <ApplyTo>Form.Submit.Field(Name:SWECmd).Value</
ApplyTo>
      <SaveAs>Command</SaveAs>
    </Exists>
    <Exists>
      <ApplyTo>Form.Submit.Field(Name:SWEMethod).Value</
ApplyTo>
      <SaveAs>Command</SaveAs>
    </Exists>
  </Or>
  <Exists>
    <ApplyTo>Command</ApplyTo>
    <SaveAs>PageName</SaveAs>
    <SaveMode>Append</SaveMode>
  </Exists>
</Conditions>
```

Loop Condition

The new compound condition *Loop* is similar to the condition *And* in that it evaluates its sub-conditions as long as they evaluate to *true*. However, unlike the *And* condition, it doesn't stop once all conditions have been evaluated. Instead, the *Loop* condition starts over again and repeatedly evaluates its sub-conditions and stops only when a condition evaluates to *false*. Of course this only makes sense if at least one sub-condition has side-effects by using the *SaveAs* tag, otherwise it would loop forever once all sub-conditions have returned *true* on the first pass.

Example

This example is taken from the *Flex/AMF3* project type and shows how the page timer name is assembled from the response body.

```
<Conditions>
  <Exists>
    <ApplyTo>Http.Initial.Request.Body</ApplyTo>
    <SaveAs>RestOfBody</SaveAs>
    <SaveMode>Replace</SaveMode>
  </Exists>
  <Loop>
    <RegExpr>
      <ApplyTo>RestOfBody</ApplyTo>
      <Data>operation=&amp;quot;\([^&amp;quot;]*\)</Data>
      <SaveAs>operation</SaveAs>
      <SaveMode>Replace</SaveMode>
      <SaveTag>1</SaveTag>
    </RegExpr>
    <RegExpr>
      <ApplyTo>RestOfBody</ApplyTo>
      <Data>operation=&amp;quot;\([^&amp;quot;]*\)</Data>
      <SaveAs>RestOfBody</SaveAs>
      <SaveMode>Replace</SaveMode>
      <SaveWhat>Right</SaveWhat>
    </RegExpr>
    <Exists>
      <ApplyTo>Literal:, </ApplyTo>
      <SaveAs>OperationList</SaveAs>
      <SaveMode>AppendSeparator</SaveMode>
    </Exists>
    <Loop>
      <FindData>
        <ApplyTo>operation</ApplyTo>
        <Data>.</Data>
        <SaveAs>operation</SaveAs>
        <SaveMode>Replace</SaveMode>
        <SaveWhat>Right</SaveWhat>
      </FindData>
    </Loop>
```

```
    <Exists>
      <ApplyTo>operation</ApplyTo>
      <SaveAs>OperationList</SaveAs>
      <SaveMode>Append</SaveMode>
    </Exists>
  </Loop>
  <Exists>
    <ApplyTo>OperationList</ApplyTo>
    <SaveAs>PageName</SaveAs>
    <SaveMode>Replace</SaveMode>
  </Exists>
</Conditions>
```

Troubleshooting

This chapter offers some recommendations for troubleshooting recording rules that don't operate as expected.

The Recorder outputs diagnostic information to the "Log" tab window, as shown in Figure 8.

This output contains:

- The recording rule files that have been read by the Recorder.
- The recording rules contained in the recording rule files.
- "+" signs preceding rules that are correct and active.
- "-" signs preceding rules that are incorrect or inactive.
- Reasons why rules aren't used ("not active" or an error description).

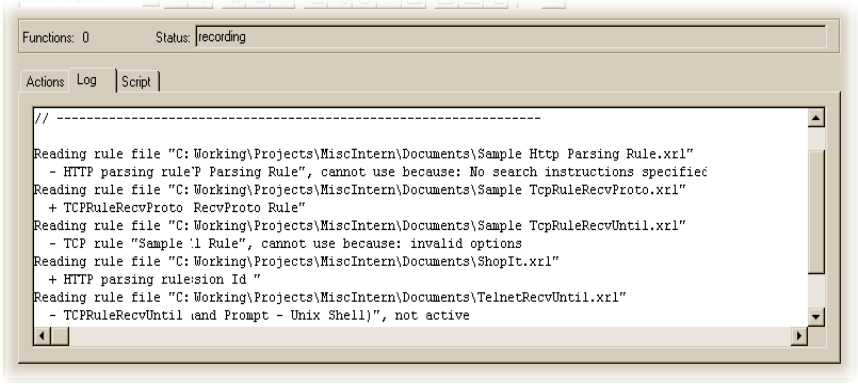


Figure 8 - Diagnostic Output of the Recorder

If a rule file doesn't appear in this diagnostic output, check the following:

- Is the rule file in the *Documents* directory of the current project or the public or user's *RecordingRules* directory?
- Does the file have the extension "*.xrl*"?

Ensure that the setting "*Hide file extensions for known file types*" is not checked in the "*Folder Options*" Windows Explorer settings dialog box, as shown in Figure 9. Otherwise windows may show a file name such as "*Rule.xrl*" when the file name is actually "*Rule.xrl.txt*".

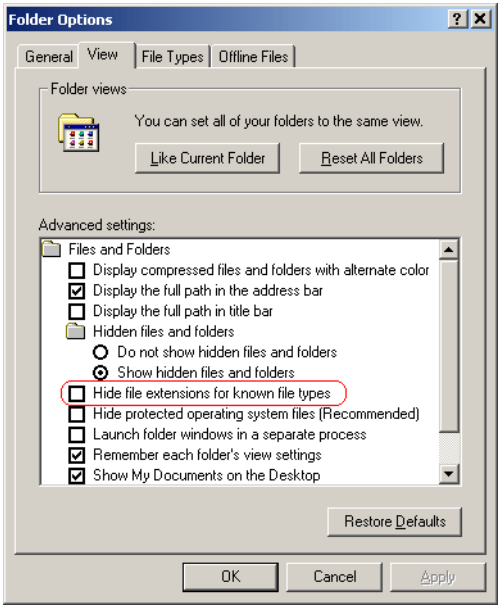


Figure 9 - Windows Explorer Folder Options

5

Enhanced SNMP Support for Silk Performer

What you will learn

This chapter contains the following sections:

Overview

Standard Silk Performer SNMP support includes a generic interface for all SNMP data sources in which MIBs are present. Specific measures are accessed using a combination of *object ID (OID)* and *instance ID* (when objects have more than one instance). The challenge with this approach is that you must know exactly which counter instance is to be applied—and instances are only distinguished by instance ID, not by instance name. Compounding this challenge is the fact that when servers reboot certain instance numbers can change due to dynamic monitoring-object creation. Therefore monitoring scripts must be able to cope with dynamic monitoring targets.

The value of Silk Performer's enhanced SNMP support is that it allows you to access values not by OID and instance ID, but by OID and instance name.

The following example illustrates the enhanced SNMP support that is available with Silk Performer. This example shows how predefined ESNMP measures are defined, how they are used, and how additional measures can be added to the list of pre-defined measures.

Identifying instance IDs and OIDs

The first step in working with a predefined ESNMP measure is to identify the OID, instance ID, and instance name of a WebLogic counter. The *JDBC Connection Pool Runtime Active Connections Current Count* for a certain instance is accessible with the OID:
1.3.6.1.4.1.140.625.190.1.25 + <Instance ID>

Without enhanced SNMP support, you have to provide:

- OID of the table containing the values
- Correct Instance ID.

JDBC Connection Pool Runtime Object Name and *JDBC Connection Pool Runtime Name* contain the human readable names of the instances and are accessible with the OIDs:

1.3.6.1.4.1.140.625.190.1.5 + <Instance ID> for "Object Names"

and

1.3.6.1.4.1.140.625.190.1.15 + <Instance ID> for "Names"

ESNMP support provides you with the option of specifying an instance name, retrieving an instance ID using the *(object-) name* table, and then getting the value for the correct counter. With ESNMP support, the following must be provided:

- OID of the table containing the values
- Instance name
- OID of the table containing the instance names

Data Source Definition

> **Note** For Silk Performer 7.0, this is not part of the *REALTIME.INI* because the file is split and there is a separate file for each data source. As this advanced concept paper is written for 7.0, it must be adapted once the new structure has been defined.

By searching for "WebLogic" and "ESNMP" in the *REALTIME.INI* file, you'll find the following entries:

```
S= Application Server\BEA WebLogic preconfigured\SNMP, ESNMP:BEA-
WEBLOGIC-MIB,

 M= ESNMP:1.3.6.1.4.1.140.625.340.1.25, BEA WebLogic\JVM
Runtime\HeapFreeCurrent, eAvgOnlyCounter, 0, kbytes, 0, 1, 2, 3, 4, 5,
Current Heap Free, measureMatchName=Runtime Object
Name;measureMatchOID=1.3.6.1.4.1.140.625.340.1.5;

 M= ESNMP:1.3.6.1.4.1.140.625.340.1.30, BEA WebLogic\JVM
Runtime\HeapSizeCurrent, eAvgOnlyCounter, 0, kbytes, 0, 1, 2, 3, 4, 5,
Current Heap Size, measureMatchName=Runtime Object
Name;measureMatchOID=1.3.6.1.4.1.140.625.340.1.5;
```

```
 M= ESNMP:1.3.6.1.4.1.140.625.180.1.25, BEA WebLogic\Queue
Runtime\ExecuteThreadCurrentIdleCount, eAvgOnlyCounter, 0, , 0, 1, 2, 3,
4, 5, Thread Idle Count, measureMatchName=Queue Runtime Object
Name;measureMatchOID=1.3.6.1.4.1.140.625.180.1.5;

 M= ESNMP:1.3.6.1.4.1.140.625.180.1.35, BEA WebLogic\Queue
Runtime\PendingRequestCurrentCount, eAvgOnlyCounter, 0, , 0, 1, 2, 3, 4,
5, Current Requests, measureMatchName=Queue Runtime Object
Name;measureMatchOID=1.3.6.1.4.1.140.625.180.1.5;

 M= ESNMP:1.3.6.1.4.1.140.625.180.1.40, BEA WebLogic\Queue
Runtime\ServicedRequestTotalCount, eAvgOnlyCounter, 0, , 0, 1, 2, 3, 4,
5, Total Requests, measureMatchName=Queue Runtime Object
Name;measureMatchOID=1.3.6.1.4.1.140.625.180.1.5;

 M= ESNMP:1.3.6.1.4.1.140.625.430.1.50, BEA WebLogic\WebApp Component
Runtime\OpenSessionsCurrentCount, eAvgOnlyCounter, 0, , 0, 1, 2, 3, 4,
5, Open sessions, measureMatchName=Runtime Object
Name;measureMatchOID=1.3.6.1.4.1.140.625.430.1.25;

 M= ESNMP:1.3.6.1.4.1.140.625.430.1.55, BEA WebLogic\WebApp Component
Runtime\OpenSessionsHighCount, eAvgOnlyCounter, 0, , 0, 1, 2, 3, 4, 5,
Session High Count, measureMatchName=Runtime Object
Name;measureMatchOID=1.3.6.1.4.1.140.625.430.1.25;

 M= ESNMP:1.3.6.1.4.1.140.625.550.1.25, BEA WebLogic\Execute
Queue\ThreadCount, eAvgOnlyCounter, 0, , 0, 1, 2, 3, 4, 5, Thread Count,
measureMatchName=Queue Object
Name;measureMatchOID=1.3.6.1.4.1.140.625.550.1.5;
```

The first line, which begins with "S=", is the data source definition header. You only need one definition header per data source. So if you want to enhance the existing WebLogic ESNMP data source, you can leave the data source definition header as is.

If you want to define a new data source, you should not edit the REALTIME.INI file, because you would loose the changes when you upgrade your Silk Performer version. To avoid that, create a new file in the sub-folder *C:/Program Files\Silk\Silk Performer 9.5\Include\DataSrcWzd*, for example **myESNMP_realtime.ini**. This file has to contain the "S=" and the "M=" lines of your data source definition.

Subsequent lines are for the individual measures:

- Following the "M=" is the OID that is to be queried, without the instance ID (e.g., *1.3.6.1.4.1.140.625.190.1.25)*.
- Following that is the name of the measure as it will appear in the GUI. Note that the back slashes are used to create the hierarchy (e.g., *BEA WebLogic\JDBC Connection Pool\Active Connections)*.
- Following that are standard entries: *measure class, scale, unit, scale factor,* and five unused fields (e.g., *eAvgOnlyCounter, 0, , 0, 1, 2, 3, 4, 5)*.
- The next field is a description for tool-tip help in Performance Explorer. You might write, "Current count of active connections in the JDBC connection pool runtime." The measure name itself can be shorter.
- The last entry is divided into two parts:

- ○*measureMatchName*= This is used in the data source wizard to describe the instances (e.g., *JDBC Connection Pool Runtime Object Name)*.
- ○*measureMatchOID*= This is the OID of the table that contains the instance names *(e.g., 1.3.6.1.4.1.140.625.190.1.15)*.

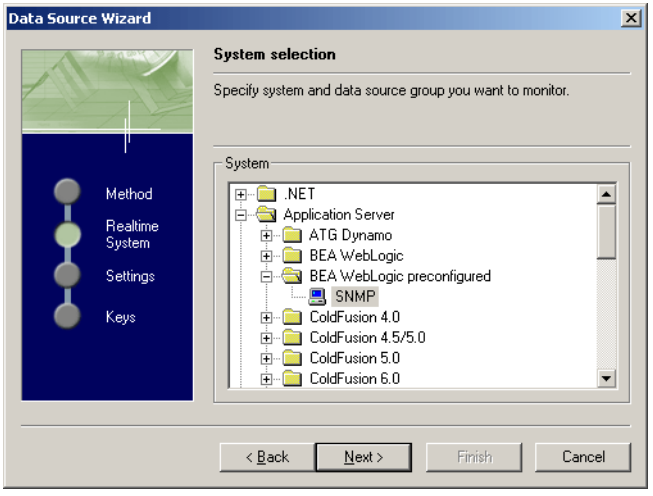
Note that the final measure name consists of the measure name given here *(BEA WebLogic\JDBC Connection Pool\Active Connections)* and the instance name in parenthesis.

So the key for the measure should look like this:

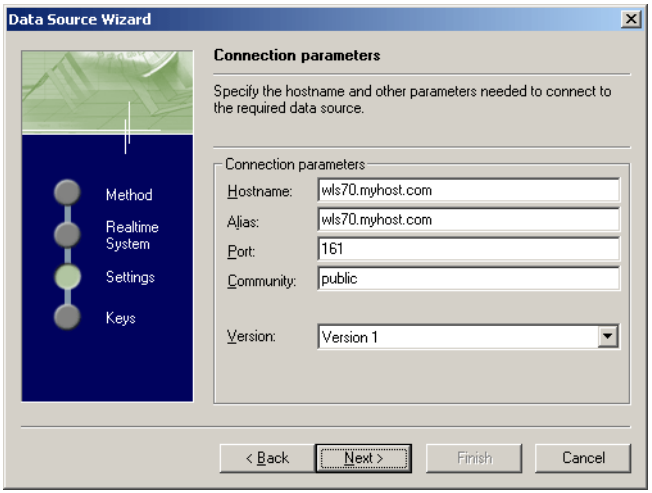
```
M= ESNMP:1.3.6.1.4.1.140.625.190.1.25, BEA WebLogic\JDBC Connection
Pool\Active Connections, eAvgOnlyCounter, 0, , 0, 1, 2, 3, 4, 5, Current
count of active connection in the JDBC connection pool runtime,
measureMatchName=JDBC Connection Pool Runtime Object
Name;measureMatchOID=1.3.6.1.4.1.140.625.190.1.15;
```

Using Enhanced SNMP in Performance Explorer

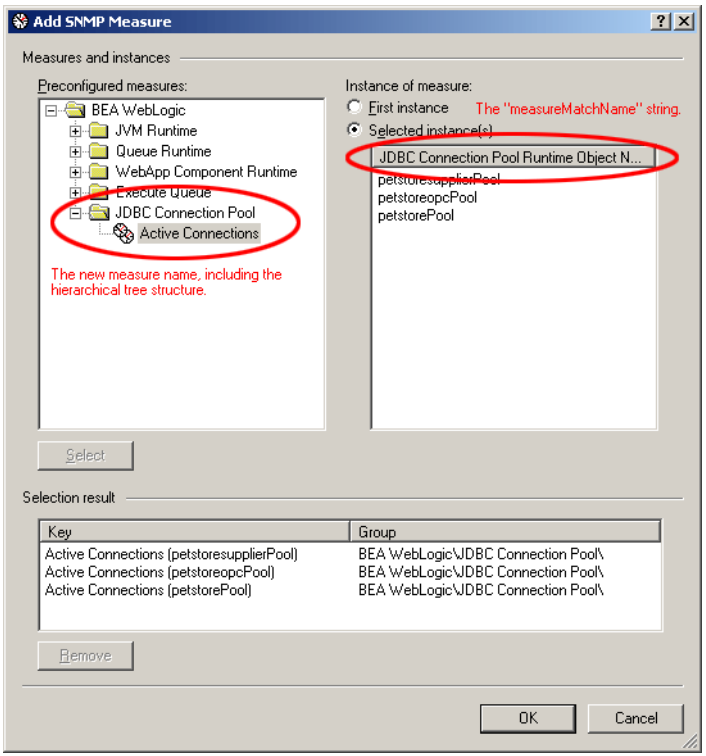
After adding the key (see previous section) to the REALTIME.INI file below the other WebLogic ESNMP keys, open Performance Explorer and add a data source. Select *SNMP* of the *BEA WebLogic preconfigured* data source.



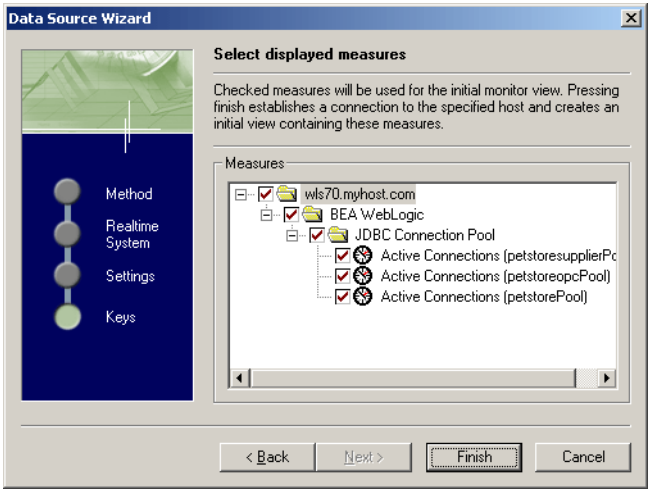
Fill in the *Hostname, Port,* and *Community* fields.



Here you can see the effects of the strings in the ESNMP key. Select your newly created measure and select the *Selected instance(s)* radio button to query a list of instance names. Choose one or more instance names and add them.



Select all measures and click *Finish* to view the measures in Performance Explorer.



4

Generating BDL Monitoring Projects for Performance Explorer

What you will learn

This chapter contains the following sections:

Overview

Silk Performer projects, or more precisely *BDL scripts*, can be used to collect performance data for monitored systems. This chapter outlines the steps that are required to assemble BDL monitoring projects for use with Performance Explorer and thereby enable real-time display of collected data.

Concepts

BDL Monitoring Projects

The *Performance Data Collection Engine (PDCE)* processes performance data collected by BDL scripts. Performance data collected with such scripts can be displayed in Performance Explorer in real-time. Entities collected by the PDCE are called measures, which may be specified in BDL using the following functions:

- MeasureInc
- MeasureIncFloat
- MeasureStart & MeasureStop
- MeasureSetTimer
- MeasureSet

These functions are called within transactions. We allow one to many measures within one transaction. Furthermore a monitoring project may use exactly one script defining exactly one usergroup. This usergroup may define one init transaction, one end transaction and one main transaction.

In practice, we will use special wrapper functions defined in "bdlMonitor.bdh" which covers some additional functionality absolutely necessary for BDL realtime monitoring. See the tutorials on how to use these functions:

- MonitorInc
- MonitorIncFloat
- MonitorStart and MonitorStop
- MonitorSet
- MonitorSetFloat
- MonitorSetTimer

Workflow

This section introduces the workflow required to generate BDL monitoring projects for Performance Explorer via Silk Performer.

1 Launch Silk Performer and create a BDL monitoring project

a Click the **Outline Project** button on the Workflow bar.
The ***Workflow - Outline Project*** dialog then opens.

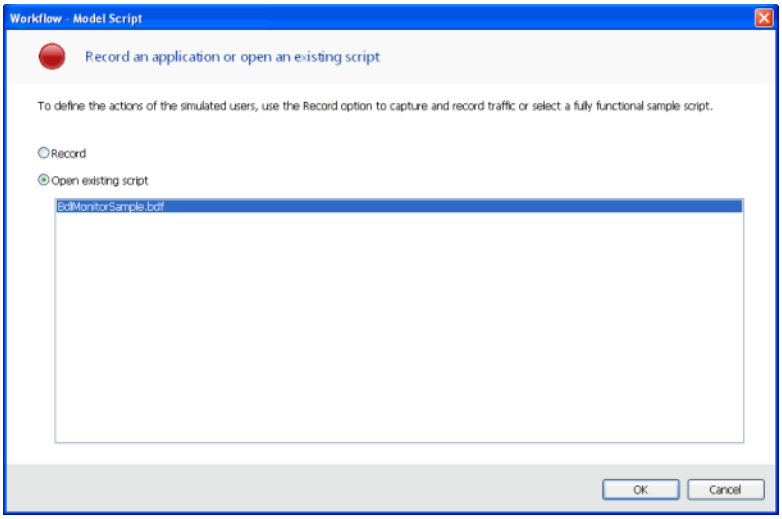
b Enter a name for the project in the *Name* text field.

c **Select** "*Bdl Monitor for Performance Explorer*" (under "*Monitoring*") from the application *Type* list.

d Click **Next**.

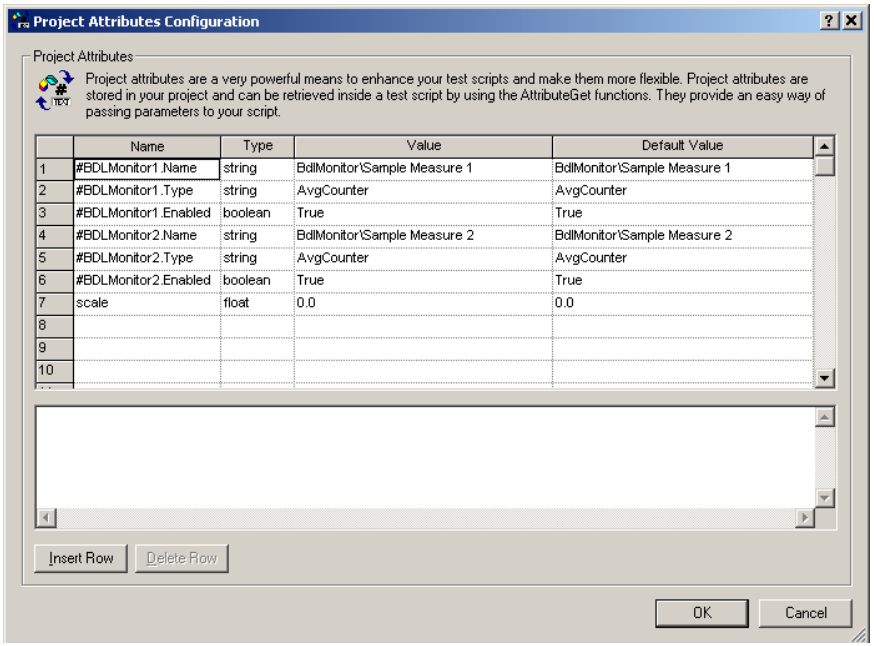
2 Create a monitoring script

On the *Model Script* dialog box, choose *Open existing scrip*t and click **OK**. This brings up a pre-configured monitoring script that can be used as a template.



3 Define project attributes

Each measure exported to Performance Explorer requires a fixed set of project attributes. To open the project attributes go to "*Project/Project Attributes*".



To be viewed with Performance Explorer, each measure requires at least these three project attributes:

Name: Name to be shown in Performance Explorer

Type: An average value or a cummulative value.

Enabled: Reserved. Always set to true.

4 Sample Monitor Script

As specified in the project attributes, the project exports two measures that can be viewed in Performance Explorer in real-time.

Look for *MonitorInc* and *MonitorIncFloat* in the *TMon* transaction. This is where the last snapshot is handed over to Performance Explorer.

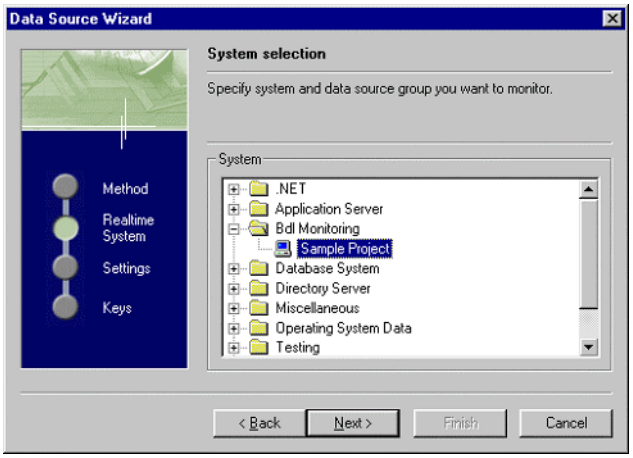
```
use "bdlMonitor.bdh"

const
  nMeasure1 := 1;
  nMeasure2 := 2;

dclrand
  rRand    : RndExpN  (1..10000: 100.000000);
  rRandf   : RndExpF  (5.500000);

var
  fScale : float;
```

```
dcluser
  user
    VMon
  transactions
    TInit           : begin;
    TMon            : 1;
    TEnd            : end;

dcltrans
  transaction TInit
  begin
    fScale := AttributeGetDouble("scale");
  end TInit;

  transaction TMon
  var
    n : number;
    f : float;
  begin
    n := rRand;
    MonitorInc(1, n);

    f := rRandf;
    f := f * fScale;
    MonitorIncFloat(2, f);
  end TMon;
```

5 Edit the .conf File

The sample .conf file is available in the project panel under *Data Files*, just below the *Script* node. Change the value of the *Type* entry. This indicates where in the Performance Explorer hierarchy the project will be

located. For example, use type *Bdl Monitoring\Sample Project* to locate the monitoring project in Performance Explorer under Monitor/Add *DataSource/Predefined Data Sources.*



6 Exporting Monitoring Projects

Export the project to a single zip file. Go to "**File\Export Project**". Check "*zip to single archive*" and export your file to C:\Program Files\Silk\Silk Performer 9.5\Monitors\BdlMonitorSample.sep".

Note Save the file with an ".sep" extension.

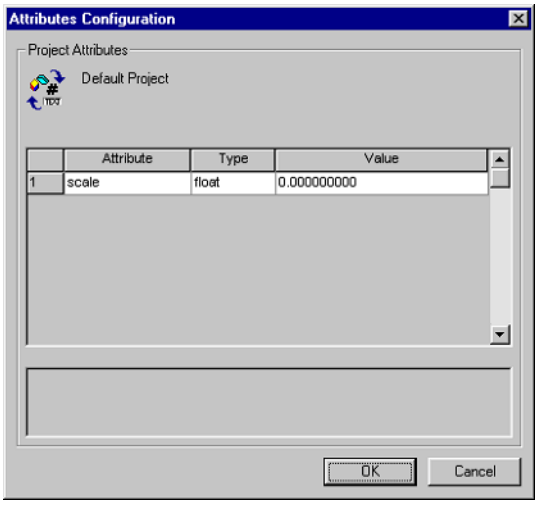
7 Start a Realtime BDL Monitoring Project

Launch Performance Explorer and go to *Monitor\Add Data Source.* **Select** the *Select from predefined Data Sources* radio button and click **Next**.

Select the newly created monitoring project and click **Next**.

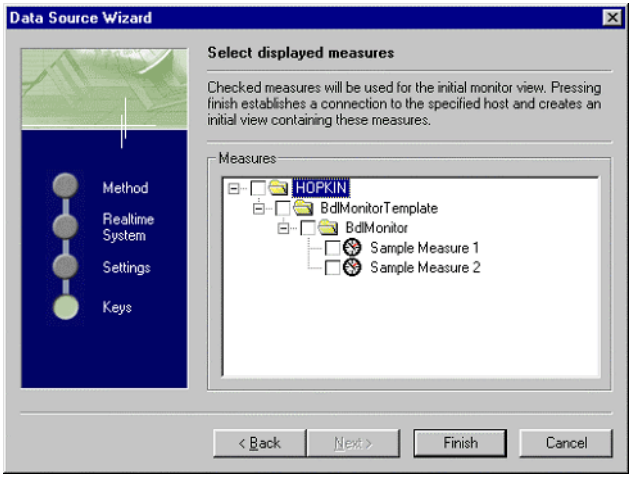
Enter the name of the host that is to be monitored (select *localhost* or the host recommended by Performance Explorer and click **Next**).

This brings up a dialog on which project attributes can be modified (only one attribute can be modified in this instance).

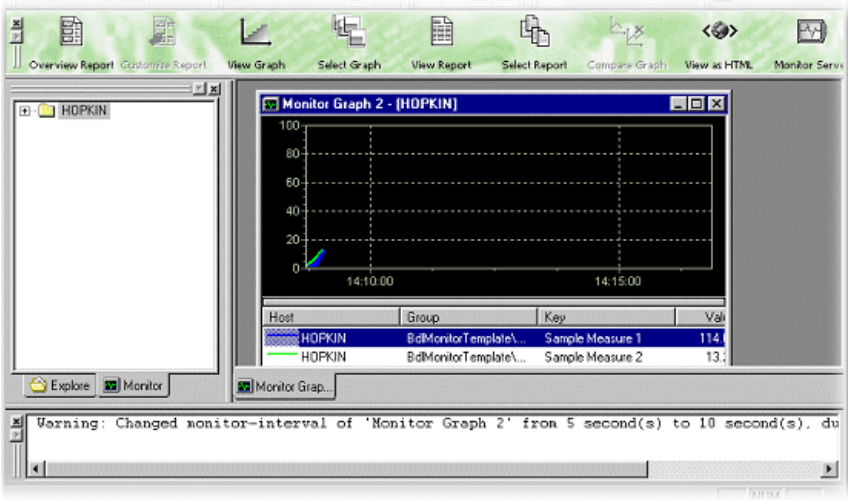


Enter a value *greater than 0*. This value will be used in the monitoring project. See the sample script for details.

Click **OK**. This brings up a choice of measures specified in the BDL monitoring project.



Select both measures and click **Finish**. This launches the monitoring project. The values can be seen in real-time using a monitoring graph.



Tutorial

This tutorial consists of two use cases: (1) a simple use case to get you started and (2) an advanced use case. The first use case involves a monitor project that consists of one user group with a single transaction and a single measure.

The second use case illustrates how to have several user groups, transactions, and measures in a single project.

Use Case 1

In this use case we'll create a project that keeps track of the number of processes running on a SunOs. SunOs can usually be accessed through remote execution. Compare with the Win2000 command line tool rexec. To count the number of processes running on a SunOs, execute 'ps -ef | egrep -c ".*"' within an X-Terminal, Telnet session, rexec, or other preferred tool. For example, at a DOS prompt type:

```
c:\>rexec yourSunHost -l username "ps -ef | egrep -c \".*\""
```

This returns the number of processes running on yourSunHost. The goal here is to continuously track and display this value in a Performance Explorer real-time chart.

Create a new Silk Performer Project

Enter a simple project description that reflects the overall purpose of the monitoring project to be displayed in Performance Explorer - for example, "A

powerful project used to collect the number of running processes on SunOs systems."

Planning Project Attributes

Begin by entering a name for the measure in the project attributes, for example *CountNrOfProcesses*. This name will also be used in the Performance Explorer hierarchy. A hierarchy is introduced with '\'.

For example, to create this hierarchy:

OS

Processes

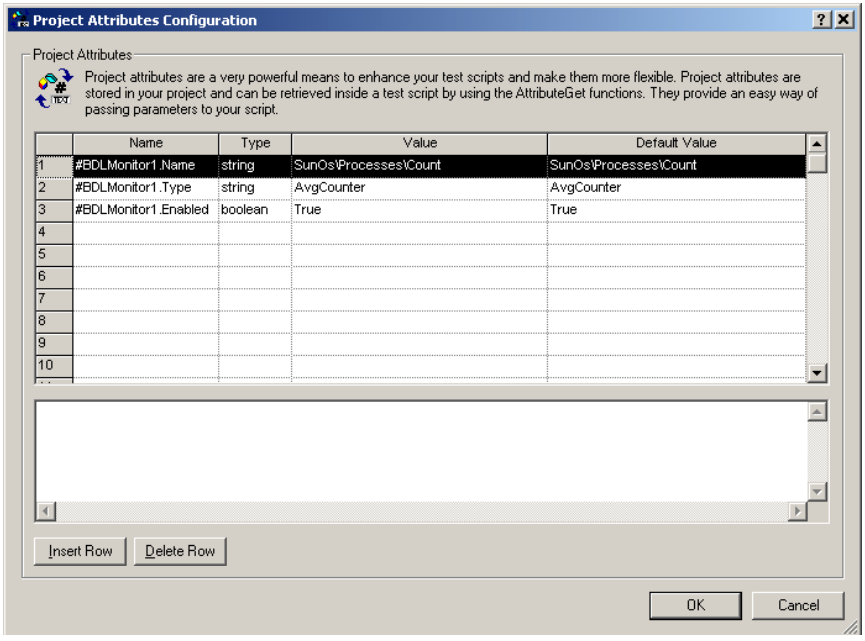
CountNrOfProcesses

AnotherCounter

Specify the name of the measure as "*SunOs\Processes\CountNrOfProcesses*".

This will be an *AvgCounter* (for average) counter and it should be *collected* by *default*.

The values in italics must be transferred into the project's attributes. Open the attribute editor under *Project\Project Attributes*.



The values in the project attributes identify a single measure. Several attributes (lines in the dialog) may be required to create a measure however - for example all entries beginning with *#BDLMonitor1* in the above example. This is how settings are defined for a measure.

The measure is now defined with a name, a type, and the setting Enabled. A second measure will begin with *#BDLMonitor2* (covered in Use Case #2).

Creating a BDL Monitoring Script

To remotely execute certain command line tools in BDL, for example "ps" on a SunOs, three functions are required:

- Connect to a remote machine's "exec" server.
- Send the command the has to be executed on the remoe machine.
- Close the connection

```
// hCon will be assigned the connection handle
function Connect(/*inout*/hCon : number; sHost : string)
begin
  WebTcpipConnect(hCon, sHost, 512);
end Connect;

// Send a request to the remote execution server
// remote execution protocol:
// What does a request look like in binary:
// 00username00password00shellCommandToExecute00
// What does the response look like
// 00responseData
// sample request:
//   00root00labpass00ps -ef | egrep -c ".*"00
function Request(hCon: number; sUser: string; sPwd: string;
                 sCmd: string):number
var
  sSend : string;
  sBuf  : string;
  nSize : number;
  nRec  : number;
begin
  sSend := "\h00";
  SetString(sSend, 2, sUser);
  SetString(sSend, Strlen(sUser) + 3, sPwd);
  SetString(sSend, Strlen(sUser) + Strlen(sPwd) + 4, sCmd);

  nSize := 3 + Strlen(sUser) + Strlen(sPwd)
             + Strlen(sCmd) + 1;

  WebTcpipSendBin(hCon, sSend, nSize);
```

```
  WebTcpipRecvExact(hCon, NULL, 1);
  WebTcpipRecv(hCon, sBuf, sizeof(sBuf), nRec);
  Request := number(sBuf);
end Request;

// Closes the connection to the remote exec server
function Close(hCon : number)
begin
  WebTcpipShutdown(hCon);
end Close;
```

A function wrapper is needed around Silk Performer's *MeasureInc* functions. This function can be used in all monitoring projects. A function named *MonitorInc* is created to access project attributes. This function accesses the attributes that were specified in section "Planning Project Attributes".

The *MonitorInc* function can also be imported from an existing bdh, *bdlMonitor.bdh*.

```
function MonitorInc(nMon : number; nVal : number)
var
  sMeasure : string;
begin
  // This will check whether the attribute
  // "#BDLMonitor1.Enabled" was set to true
  if AttributeGetBoolean("#BDLMonitor" + string(nMon)
                         + ".Enabled") then
      // If yes then let's read the name of the measure.
      // To do this we read the the project attribute
      // "#BDLMonitor1.Name" and store it
      // to a local variable named sMeasure.
      // sMeasure will have the value:
      // "SunOs\Processes\CountNrOfProcesses"
      AttributeGetString("#BDLMonitor" + string(nMon)
              + ".Name", sMeasure, sizeof(sMeasure));

      //   Set a new value for
      // "SunOs\Processes\CountNrOfProcesses"
      MeasureInc(sMeasure, nVal, MEASURE_KIND_AVERAGE);
    end;
end MonitorInc;
```

Now the transaction that will take the snapshot using all the functions that have been defined can be coded. This transaction also accesses the project file attributes. The goal is to later have these attributes set in Performance Explorer. For now however, to ensure that the script works, four attributes need to be added to the project attributes.

Note that attribute names are case sensitive.

- *host*: Assign it a sample value for, example "*sunserver*"
- *command*: Assign it a sample value, for example "*ps -ef | egrep -c ".*'"*"
- *user*: Assign a sample value, for example "*root*".
- *password*: Assign a sample value for testing purposes.

Open the project attributes editor under *Project\Project Attributes* and add these additional attributes. All are of type "string" except for the attribute password which is of type "password". Assign values to the attributes for testing purposes. Choose a description for each attribute that conveys the purpose of the attribute.

```
const
  nMeasure := 1;

dcluser
  user
    VMonitor
  transactions
    TSnap : 1;

dclfunc
.... // your functions here

dcltrans
  transaction TSnap
  var
    hCon  : number init 0;
    sHost : string;
    sCmd  : string;
    sUser : string;
    sPwd  : string;
    nVal  : number;
  begin
    AttributeGetString("host", sHost, sizeof(sHost));
    AttributeGetString("command", sCmd, sizeof(sCmd));
    AttributeGetString("user", sUser, sizeof(sUser));
    AttributeGetString("password", sPwd, sizeof(sPwd));

    Connect(hCon, sHost);
    nVal := Request(hCon, sUser, sPwd, sCmd);
    MonitorInc(nMeasure, nVal);
    Close(hCon);
  end TSnap;
```

The project now consists of the newly created script. Save the project and verify that it works by initiating a TryScript run. Have "nVal" printed or written to a log file to verify that the script works. If the script works, save and close the project.

Packaging the Project

To export the project as a single zip file, go to *File\Export Project*. Check *zip to single archive* and export the file to *C:\Program Files\Silk\Silk Performer 9.5\Monitors\CountProcess.sep. (note the .sep extension)*.

Before the sep file can be used with Performance Explorer the *.conf* file for the sep must be modified. The *.conf* file looks like:

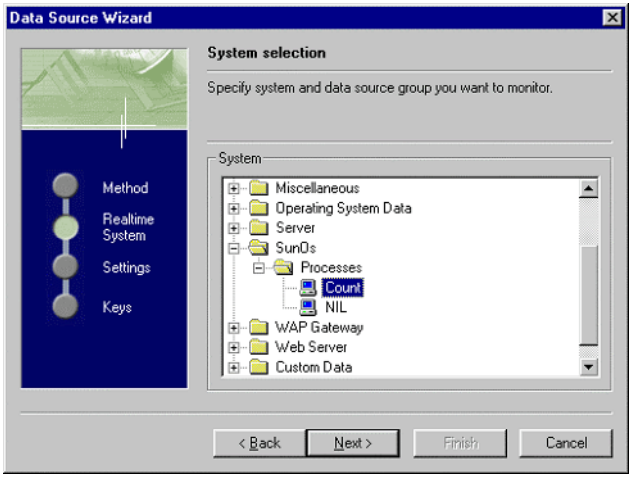
```
<?xml version='1.0' encoding='UTF-8'?>
<Project>
  <Type>Sample\Remote</Type>
  <Copyright>Borland</Copyright>
  <Author>Borland</Author>
  <Version>5.1</Version>
</Project>
```

For the type setting, specify a hierarchy separated by "\". This hierarchy will be reflected in Performance Explorer.

Using the Montoring Project inside Performance Explorer

Launch Performance Explorer and go to *Monitor\Add Data Source*. **Select** the *Select from predefined Data Sources* radio button and click **Next**.

Select the newly created monitoring project and click **Next**.

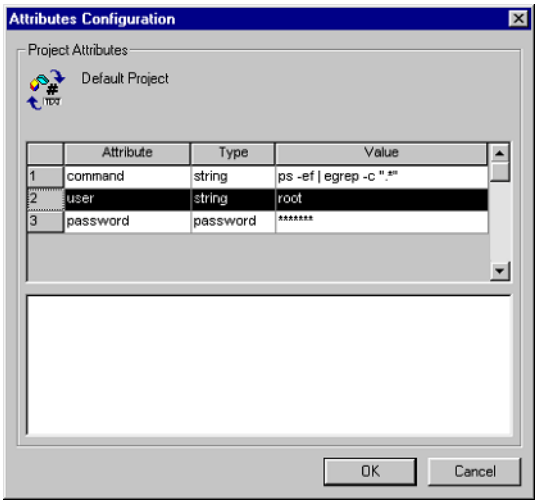


Specify the host that is to be monitored, preferably a machine running SunOs. Click **Next** to bring up a dialog for setting project attributes. This dialog displays the attributes that were set in section "Creating a BDL Monitoring Script" (except for the host which was specified previously).

Therefore, the dialog will display following attributes:

- user

- password
- command



Select the measures (in this case only a single measure) and click **OK**. The measures are then immediately handed over to the data collection mechanism.

Use Case #2

This use case involves a project with two user groups - the user group from the previous use case and a new user group that measures HTTP requests. Two URI's and the host are configurable through project attributes.

Creating a Silk Performer Project

Enter a simple project description that is to be displayed in Performance Explorer. The description should describe the overall purpose of the monitoring project - for example, "A powerful project used for collecting the number of running processes on SunOs systems and measuring two configurable HTTP requests."

Planning Project Attributes

Begin by defining three measures. Pick names that indicate the hierarchy and purpose of the measures. In this example a measure counts the number of processes on a SunOs system and includes two measures that measure HTTP requests (Performed via WebUrl):

- SunOs\Processes\CountNrOfProcesses
- SunOs\Http Server\URI1
- SunOs\Http Server\URI2

'\' reflects the intended hierarchy which will look like:

SunOs

Processes

CountNrOfProcesses

Http Server

URI1

URI2

A BDL script with two user groups and two transactions is written. See the table below to see how measures are assigned to transactions and how transactions relate to user groups.

User Group	Transaction	Measure
VMonitor	TSnap	CountNrOfProcesses
VWebMon	TWebSnap	URI1
VWebMon	TWebSnap	URI2

With projects that include numerous transactions and/or user groups, measures must be specified with the corresponding user group and assigned measure. This differs from Use Case #1, which involved only one transaction and one user group.

Open the project attributes editor at *Projects\Project Attribute* and enter the following data:

Name	Type	Value
#BDLMonitor1.Name	string	SunOs\Processes\CountNrOf Processes
#BDLMonitor1.Type	string	AvgCounter

Name	Type	Value
#BDLMonitor1.Enabled	boolean	True
#BDLMonitor1.Script	string	remote.bdf
#BDLMonitor1.Usergroup	string	VMonitor
#BDLMonitor1.Transaction	string	TSnap
#BDLMonitor2.Name	string	SunOs\Http Server\URI1
#BDLMonitor2.Type	string	AvgCounter
#BDLMonitor2.Enabled	boolean	True
#BDLMonitor2.Script	string	remote.bdf
#BDLMonitor2.Usergroup	string	VWebMon
#BDLMonitor2.Transaction	string	TWebSnap
#BDLMonitor3.Name	string	SunOs\Http Server\URI2
#BDLMonitor3.Type	string	AvgCounter
#BDLMonitor3.Enabled	boolean	True
#BDLMonitor3.Script	string	remote.bdf
#BDLMonitor3.Usergroup	string	VWebMon
#BDLMonitor3.Transaction	string	TWebSnap
host	sunserver	sunserver
command	string	ps -ef | egrep -c ".*"
user	string	root
password	password	**********
URI1	string	/
URI2	string	/manual/ibm/index.html

Create your BDL Monitor Script

Make a copy of Use Case #1's BDL script and add functions and transactions as described in this section.

Begin with the function that takes care of the HTTP hit:

```
// self explanatory
function HttpHit(sURL : string)
begin
  WebUrl(sURL);
```

```
end HttpHit;
```

Wrapper functions for Silk Performer's *MeasureStart* and *MeasureStop* functions are built to read measure names from project attributes.

These wrapper functions can also be imported from Silk Performer's bdh file, *bdlMonitor.bdh*.

```
// MonitorStart will be passed a number such identifying
// a measure as defined in
// the project attributes:
// Example MonitorStart(1) is #BDLMonitor1,
// the name of the measure is
// "SunOs\Processes\CountNrOfProcesses"
function MonitorStart(nMon : number)
var
  sMeasure : string;
begin
  if AttributeGetBoolean("#BDLMonitor" + string(nMon)
                                      + ".Enabled") then
    AttributeGetString("#BDLMonitor" + string(nMon)
        + ".Name", sMeasure, sizeof(sMeasure));

    MeasureStart(sMeasure);
  end;
end MonitorStart;

// MonitorStop does the corresponding MeasureStop
function MonitorStop(nMon : number)
var
  sMeasure : string;
begin
  if AttributeGetBoolean("#BDLMonitor" + string(nMon)
                                      + ".Enabled") then
    AttributeGetString("#BDLMonitor" + string(nMon)
        + ".Name", sMeasure, sizeof(sMeasure));

    MeasureStop(sMeasure);
  end;
end MonitorStop
```

Equipped with these new functions you can move on to the new transaction. There are three measures. Two of them can be found in the *TWebSnap* transaction. The third measure in the transaction should be copied from Use Case #1.

In transaction *TWebSnap* you'll see project attribute values that will be set later be Performance Explorer; for now simply note the sample values.

```
use "bdlMonitor.bdh"
use "webapi.bdh"
```

```
const
  nMeasure1 := 1;
  nMeasure2 := 2;
  nMeasure3 := 3;


dcluser
  user
    VMonitor
  transactions
    TSnap : 1;

  user
    VWebMon
  transactions
    TWebSnap : 1;

dclfunc
  // Your function HttpHit

transaction TWebSnap
var
  sHost : string;
  sURI1 : string;
  sURI2 : string;
  sURL1 : string;
  sURL2 : string;
begin
  AttributeGetString("host", sHost, sizeof(sHost));
  AttributeGetString("URI1", sURI1, sizeof(sURI1));
  AttributeGetString("URI2", sURI2, sizeof(sURI2));

  sURL1 := "http://" + sHost + sURI1;
  sURL2 := "http://" + sHost + sURI2;

  MonitorStart(nMeasure2);
  HttpHit(sURL1);
  MonitorStop(nMeasure2);

  MonitorStart(nMeasure3);
  HttpHit(sURL2);
  MonitorStop(nMeasure3);
end TWebSnap;
```

Execute a "**Try script**" run to verify that the script runs accurately.

Packaging it all up

Export the project to a single zip file. Go to *File\Export Project*. Check zip to single archive and export the file to *C:\Program Files\Silk\ Silk Performer 9.5\Monitors\Advanced.sep*.

Note Note the .sep file extension.

Before the sep file can be used with Performance Explorer the *.conf* file for the sep must be modified. The *.conf* file appears as:

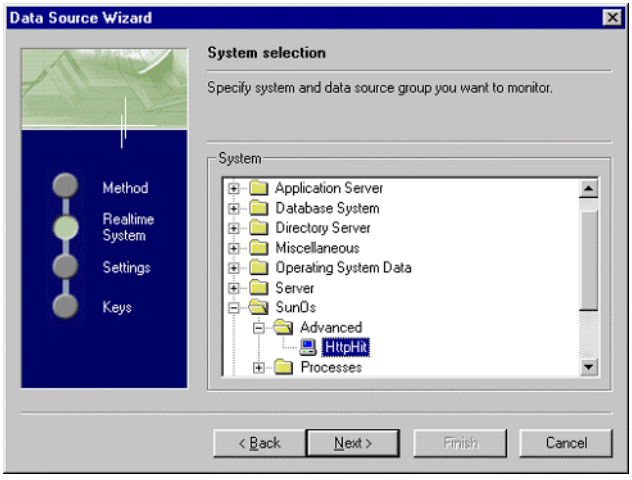
```
<?xml version='1.0' encoding='UTF-8'?>
<Project>
  <Type>Sample\Advanced\HttpHit</Type>
  <Copyright>Borland</Copyright>
  <Author>Borland</Author>
  <Version>5.1</Version>
</Project>
```

For the type setting, specify a hierarchy that is to be reflected in Performance Explorer, separated by backward slashes ("\").

Using the Monitoring Project within Performance Explorer

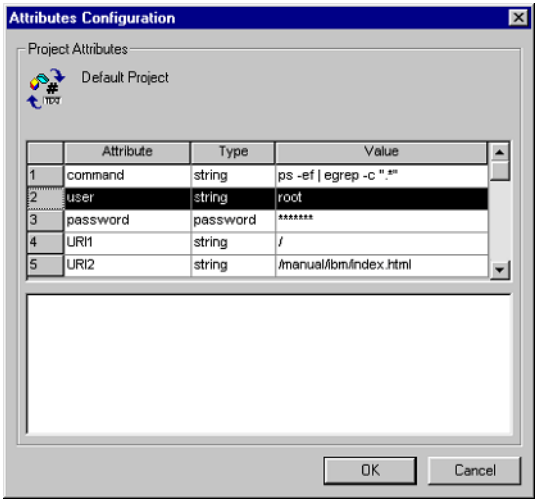
Launch Performance Explorer and go to *Monitor\Add Data Source*. **Choose** *Select from predefined Data Sources*.

Select the newly created monitoring project and click **Next**.

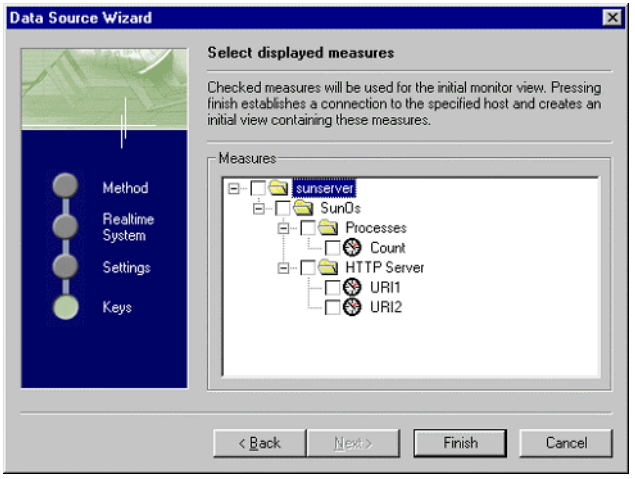


Enter the name of the server that is to be monitored. For example, enter "sunserver" and click Next. A dialog on which project attributes must be set appears. The dialog contains the following attributes, which must be set by the user.

- *command*: As with Use Case #1, a command that counts the number of processes.
- *user*: As with Use Case #1, a valid user account to access the remote machine.
- *password*: The password.
- *URI1*: The URI where response time is to be measured.
- *URI2*: A second URI where response time is to be measured.



Select the measures (in this case there is only one) and click **OK**. The measures are immediately handed over to the data collection mechanism. The choice corresponds to the hierarchy defined in the project attributes.



Best Practices

This section outlines suggestions for working with real-time monitoring projects as they pertain to the writing of monitoring scripts.

Loops

Performance Explorer executes transactions that run snapshots at equidistant intervals. For example a snapshot transaction might be executed every ten seconds. In such a case, Performance Explorer assumes that the entire transaction is executed within this one interval. Otherwise Performance Explorer will return a warning stating that it can't collect the interval's snapshot.

Example

Do not use endless loops like

```
// Forbidden
transaction TSnap
begin
  while true do
    Snap();
  end;
end TSnap;
```

ThinkTimes and Wait

Be careful with wait statements. Think-times are ignored. A monitoring transaction must be executed within a given interval. Wait statements block execution and transactions may not return on time.

Example

```
// Forbidden
transaction TSnap
begin
  wait 500000.0;
end TSnap;
```

Initialization

Don't place function calls in TInit transactions that may halt runtime. For example *OraLogon* may fire a ProcessExit on a failed logon. Performance Explorer won't catch this message and will assume that the project is still running. Of course, Performance Explorer will report that it can't gather data, but it won't be able to report why this is happening.

Name Length

Attribute value lengths must not exceed 79 characters.

Performing a Try Script run for the Project

Execute a **TryScript** run to verify that the project is working before you create a sep file.

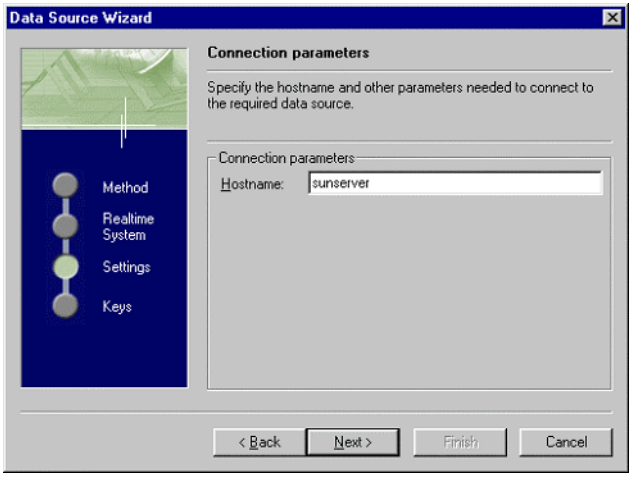
Special Project Attributes

A special monitoring project attribute is "*host*". This attribute can be accessed within BDL scripts using the following function call:

```
AttributeGetString("host", sHost, sizeof(sHost));
```

Instead of "*host*" an attribute named '#MonitorHost' may also be queried. Those two attributes are treated equally.

In Performance Explorer the value for this attribute is set using the following dialog:



8

Load Testing Legacy and Custom TCP/IP Based Protocols

What you will learn

This chapter contains the following sections:

Overview

This chapter explains how to develop load test scripts for applications that use legacy or custom TCP/IP based protocols.

The first half of this chapter explores the challenges that are commonly encountered in this type of testing: dealing with dynamic server responses, translating between character code tables, string manipulation of binary data, and more.

In the second half of this chapter, three protocols are examined in detail: Telnet, TN3270e, and a custom TCP/IP based protocol. Using recorded scripts from actual load test projects as examples, the reader is guided through the process of

recording an application, analyzing traffic, configuring recording rules, and customizing a script.

Memory Usage of TCP/IP Virtual Users

Each virtual user running in its own process requires approximately 1.6 MB system memory.

Running more than one user per process results in the following memory consumption:

- 25 vusers: Approx. 0.2 MB per vuser
- 50 vusers: Approx. 0.1 MB per vuser

Introduction

Though Silk Performer supports the testing of numerous protocols and technologies directly out of the box, there are some protocols - many of them so-called legacy systems - for which Silk Performer requires some customization.

> **Note** Since Silk Performer 7.3, the following protocols are directly supported and no longer need the customizations which are described in this chapter:
>
> - TN3270
> - TN5250
> - VT100
> - VT200+

All of the protocols discussed in this article are based on TCP/IP. With Silk Performer, you can record protocols at the TCP level and customize the resulting scripts using WebTcpip functions. With Silk Performer's enhanced TCP/IP support, the new TCP/IP TrueLog Explorer, and rule-based recording, this process is easier than ever.

This article has two main goals:

- To take you through the process of analyzing recorded custom TCP/IP traffic and give you the generic tools and methodology you need to customize such traffic.
- To analyze two protocols in-depth: Telnet and TN3270e.

In section, "Script Customization - General Tasks", the challenges of load testing at the TCP level are discussed in general terms. This chapter gives you some ideas on approaching script customization.

The section, "Telnet, TN3270e, and Custom Protocols", contains three examples of TCP/IP based traffic:

- An application based on the Telnet protocol
- The TN3270e protocol (emulation of 3270 devices)
- An entirely customized, application-specific TCP based protocol

The first two protocols are commonly used. The third protocol is an example of a true custom protocol.

Finally, this chapter explores recorder settings for TCP-based protocols.

All reusable functions introduced and explained in this document are contained in the Silk Performer file telnet.bdh.

Script Customization - General Tasks

This section covers the challenges that are typically faced when customizing TCP-based load test scripts: dynamically receiving and verifying server responses, character conversion, and string manipulation of binary data.

Character Conversion

Following is a sampling of TCP/IP traffic recorded by Silk Performer:

```
WebTcpipSendBin(hWeb0,
    "\h00000000007DD7D311D57CA2899392F0"        // ·····}×Ó·Õ|¢···ð
    "\hF0F14011D74CA2899392F0F0F140FFEF", 32);  // ðñ@·×L¢···ððñ@·ï
WebTcpipRecvExact(hWeb0, NULL, 645);
```

Customization of such a script may at first appear daunting. The server responses (not shown in the example) look much like the client request - they don't include a single readable character.

The reality is that customization of such a script is fairly simple - the server is an IBM mainframe that uses the EBCDIC (Extended Binary Coded Decimal Interchange Code) character set rather than the more familiar ASCII character set. There is a simple one-to-one mapping between these two character sets, which can be implemented as a Silk Performer function (see section "EBCDIC to ASCII Character Code Conversion" for details).

Other possible causes for traffic that contains unreadable strings:

Reason	Consequence
Traffic uses a character set other than ASCII.	Character mapping functions can easily be implemented in Silk Performer's BDL language.

Reason	Consequence
Traffic is encrypted with a custom encryption algorithm (i.e., one that doesn't rely on SSL, which would automatically be supported by Silk Performer).	The encryption/decryption routines should be re-implemented in BDL, or made accessible to Silk Performer via a DLL or a Silk Performer framework (e.g., Java Framework).
Traffic doesn't contain any strings - only numbers, sent as binaries.	In such instances you don't have to worry about strings.

Dynamically Receiving Server Responses

Full control of client requests is easily achieved, though anticipating appropriate server responses can be challenging.

Most significantly, virtual users must know when server responses are complete so that they can proceed with subsequent requests, or raise errors if server responses contain errors or aren't complete.

Silk Performer's recorder doesn't know in advance about the semantics of the TCP based protocols it records. Hence, all server responses are recorded like this:

```
WebTcpipRecvExact(hWeb0, NULL, 26);
```

During replay, the virtual user expects to receive exactly 26 bytes from the server - it then continues on with the next statement in the script. If the server sends more then 26 bytes, the bytes will be received by the next *WebTcpipRecvExact* statement (making all further script execution unpredictable). If the server sends fewer bytes, *WebTcpipRecvExact* will report a timeout error.

Therefore this line can remain unchanged only if the number of response bytes won't change under any circumstances. If response length cannot be guaranteed, scripts must be customized so that they can handle server responses of varying length.

Appropriate customization depends on the semantics of the protocol. Three common scenarios are reflected by Silk Performer's API functions, which simplify the customization process.

Case I: Packet Length Contained in the Packet

The length of request and response data may be encoded into packets at a defined position (typically in the packet header). The Silk Performer function *WebTcpipRecvProto* and its sibling, *WebTcpipRecvProtoEx*, can adequately

handle such situations. They allow for definition of the position and length of packet-length information in response data. For example, the BDL code line:

```
WebTcpipRecvProto(hWeb, 0, 2, TCP_FLAG_INCLUDE_PROTO,
                  sData, sizeof(sData), nReceived);
```

…will be used if the length of the response data remains a two-byte sequence in the first two bytes (i.e., position 0) of the server response.

See section "A Custom TCP/IP Based Protocol" for an example taken from an actual load test project.

Case II: Termination Byte Sequence

In this scenario data packets are terminated by a constant byte sequence. The corresponding Silk Performer function is WebTcpipRecvUntil. If, for example, the end sequence is defined by the two-byte sequence 0xFF00, the line:

```
WebTcpipRecvUntil(hWeb, sResp, sizeof(sResp), nRecv,
                  "\hff00", 2);
```

…will be appropriate.

See section "The TN3270e Protocol" for an example of this type based on the TN3270e protocol.

Case III: No Information on Response Packet Size

This is the trickiest of the three scenarios. You can use a combination of the functions *WebTcpipRecv*, which receives a buffer of unknown length from the server, and *WebTcpipSelect*, which checks whether or not a subsequent *WebTcpipRecv* operation on the provided connection handle will block or succeed immediately.

See section "The Telnet Protocol" for an example of this type based on the Telnet protocol.

Rule-Based Recording

An advanced feature introduced in Silk Performer is rule-based recording. You can configure the TCP/IP recorder to be aware of the semantics of proprietary TCP/IP protocols. In particular, the recorder can be configured for two of the scenarios discussed earlier (Packet length contained in the packet and Termination byte sequence).

As a result, the recorder can automatically generate correct *WebTcpipRecvProto(Ex)* and *WebTcpipRecvUntil* functions so that further customization for this part of the script isn't required.

When configuring recording rules, you write a configuration file encoding the recording rules in XML and save them to the project's Documents folder

(project specific rules) or in the public or the user's RecordingRules directory (global rules). Recording rule files carry the file extension .xrl.

Rule-based recording exists for TCP/IP and HTTP, however only recording rules for TCP/IP are discussed in this chapter. Examples of recording rule configuration files can be found in sections "Recording Rule Configuration Files" and "Recording Rule Configuration Files"

Rule-based recording is also well documented in "Rule-Based Recording".

String Manipulation of Binary Data

Request Parameterization

Example code contained in a recording of a TCP session:

```
WebTcpipSendBin(hWeb0, "\h0000104F0000005065746572FFFF",
                14);   // ···O···Peter
```

The actual business data sent here - the data that should be parameterized - is that the name "Peter" = 0x50 65 74 65 72.

won't work: "\h0000104F000000" + sName + "\hFFFF" won't yield the results you might expect. The problem is caused by the zero (0x00) bytes in this string, which Silk Performer (like C) uses for string termination. Therefore all bytes after the first zero byte in a string are ignored when performing such string concatenation.

For manipulation of binary data that may contain zero bytes, you should use the function *SetMem*. Using the function library detailed below, the above request can be broken into the following lines:

```
ResetRequestData();
AppendRequestData("\h0000104f000000", 7);
AppendRequestData(sName);      // e.g., "Peter"
AppendRequestData("\hFFFF");
SendTcpipRequest(hSend);
```

The function library is included via an include file (*.bdh). It uses two global variables for the request data and request data length. The function *AppendRequestData* simply appends a string (which may contain zero bytes) to the request buffer, and *SendTcpipRequest* sends the request data using an open TCP connection.

Here are the contents of the include file, consisting of three functions. The most important line, containing the data concatenation, is in bold:

Sample

```
const
  BUFSIZE := 10000; // maximal size for request data
var
  // global variables for Request data contents and length
```

```
  gsRequestData : string(BUFSIZE);
  gnRequestLen  : number init 0;

dclfunc
  // start a new request string
  function ResetRequestData
  begin
    gnRequestLen := 0;
  end ResetRequestData;

  // append data (of length nLen) to the request string
  // if nLen=0, use strlen(sData) instead
  function AppendRequestData(sData: string;
                             nLen: number optional): number
  begin
    if nLen = 0 then nLen := strlen(sData); end;
    if nLen + gnRequestLen <= BUFSIZE then
      // append sData to gsRequestData
      SetMem(gsRequestData, gnRequestLen + 1, sData, nLen);
      // the request length has grown by nLen bytes:
      gnRequestLen := gnRequestLen + nLen;
    else
      RepMessage("Request buffer too small!",
                 SEVERITY_ERROR);
    end;
    AppendRequestData := gnRequestLen;
  end AppendRequestData;

  // Send the request buffer
  // (TCP-connection identified by hTcp)
  function SendTcpipRequest (hTcp: number): boolean
  begin
    SendTcpipRequest :=
      WebTcpipSendBin(hTcp, gsRequestData, gnRequestLen);
  end SendTcpipRequest;
```

Searching in Response Data

When zero bytes are contained in response data, it makes it difficult to search for strings. The reason is that the *StrSearch* and *StrSearchDelimited* functions search in a source string only until they hit the first zero byte, which is interpreted as the string terminator.

Silk Performer offers the function *BinSearch*, which works on binary data exactly as *StrSearch* does on strings. You can search for a string (or a sequence of binary data) in response data as follows:

nPos := BinSearch(sResponseData, nResponseLength, sSearch);

There is no binary counterpart yet for the *StrSearchDelimited* function. If you only want to search for strings (as opposed to binary data containing zero bytes), a workaround is to introduce a function that first eliminates all zero bytes from the response data by mapping them (e.g., to the byte 0xFF).

Here is a simple version of such a function. Note that it works only if *sLeftVal*, *sRightVal* and the string you are searching are strings (i.e., they don't contain zero bytes).

Sample

```
function BinSearchDelimited(sSource   : string;
                            nSrcLength: number;
                            sLeftVal  : string;
                            sRightVal : string)
                           :string(BUFSIZE)
var
  i : number;
begin
  // eliminate zero bytes in the source string
  for i:=1 to nSrcLength do
    if ord(sSource[i]) = 0 then
      sSource[i] := chr(255);
    end;
  end;
  StrSearchDelimited(BinSearchDelimited, BUFSIZE, sSource,
                     sLeftVal, 1, sRightVal, 1,
                     STR_SEARCH_FIRST);
end BinSearchDelimited;
```

Session ID's

Since many TCP/IP based protocols rely on TCP/IP connections between server and client to remaining open and active during sessions, they don't contain session ID's. Therefore the problems that can arise when customizing load test scripts for stateless protocols such as HTML often don't exist.

There are of course exceptions to this rule, so keep watch for session ID's. The TCP support that was added to TrueLog Explorer with Silk Performer can be helpful in this regard.

Finding Information

Standard protocols are typically defined in Requests for comments (RFC's). You'll find samples of these at http://www.rfc-editor.org/rfcxx00.html. [RFC854], [RFC2355] and others are relevant to the examples included in the following sections of this chapter.

Telnet, TN3270e, and Custom Protocols

The following sections apply the theory discussed in the previous sections to specific real-world examples.

The Telnet Protocol

The introduction of RFC 854, the Telnet protocol specification, states:

> The purpose of the TELNET Protocol is to provide a fairly general, bi-directional, eight-bit byte oriented communications facility. Its primary goal is to allow a standard method of interfacing terminal devices and terminal-oriented processes to each other.

A Telnet session consists of two main parts:

1. A connection is established, and session details and options are negotiated between client and server.
2. The application launches. From that point forward, generated traffic is user driven.

The following Telnet Silk Performer script comes from a project in which a script was recorded and customized from a Telnet session with VT220 terminal emulation. The client (i.e., the terminal emulation software) is NT based and the server application resides on a Unix box.

Part I: Establishing a Connection

Here is the first part of the recording of the Telnet session:

```
WebTcpipConnect(hWeb0, "myserver", 23);

WebTcpipRecvExact(hWeb0, NULL, 3);
WebTcpipSendBin(hWeb0, "\hFFFC24", 3); // ·ü$

WebTcpipRecvExact(hWeb0, NULL, 3);
WebTcpipSendBin(hWeb0, "\hFFFB18", 3); // ·û·

WebTcpipRecvExact(hWeb0, NULL, 6);
WebTcpipSendBin(hWeb0, "\hFFFA18005654323230FFF0", 11);
                                    // ·ú··VT220·ð

WebTcpipRecvExact(hWeb0, NULL, 3);
WebTcpipSendBin(hWeb0, "\hFFFC20", 3); // ·ü

WebTcpipRecvExact(hWeb0, NULL, 60);
```

There is only one readable string in the inline comments: "VT220", the terminal type. This is a first hint at the semantics of this Telnet traffic - server and client are negotiating terminal type and various other session settings.

In terms of script customization, nothing needs to be changed here. Silk Performer virtual users should negotiate session settings exactly as the terminal emulation software has done here.

Nevertheless, an analysis of the *record.log* file, along with the information from RFC 854 (Telnet protocol specification), reveals how the conversation was achieved. The first part of the log is translated into TELNET codes in the following table:

Red (italic) = Server to Client - **Green (bold) = Client to Server**	
FF FD 24	*IAC DO ENVIRONMENT VARIABLES*
FF FC 24	**IAC WON'T ENVIRONMENT VARIABLES**
FF FD 18	*IAC DO TERMINAL-TYPE*
FF FB 18	**IAC WILL TERMINAL-TYPE**
FF FA 18 01 FF F0	*IAC SB TERMINAL-TYPE SEND IAC SE*
FF FA 18 00 56 54 32 32 30 FF F0	**IAC SB TERMINAL-TYPE IS "VT220" IAC SE**
FF FD 20	*IAC DO TERMINAL-SPEED*
FF FC 20	**IAC WON'T TERMINAL-SPEED**
...	...

Each command begins with an "Interpret as Command" (IAC) escape character 0xFF - server and client agree on a terminal type (VT220) and a number of other session settings. Telnet session settings that can be negotiated between client and server include terminal speed, echo, "suppress go ahead," window size, remote flow control, and more.

Part II: User Interaction

The second part of the script contains the user interaction. The script generated from the recording session looks like this:

```
// …
// login: send username
WebTcpipSend(hWeb0, "t");
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipSend(hWeb0, "e");
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipSend(hWeb0, "s");
```

```
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipSend(hWeb0, "t");
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipSend(hWeb0, "u");
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipSend(hWeb0, "se");
WebTcpipRecvExact(hWeb0, NULL, 2);
WebTcpipSend(hWeb0, "r");
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipSend(hWeb0, "\r");
WebTcpipRecvExact(hWeb0, NULL, 2);
WebTcpipRecvExact(hWeb0, NULL, 10);
// login: send password
WebTcpipSend(hWeb0, "password\r");
WebTcpipRecvExact(hWeb0, NULL, 2);
WebTcpipRecvExact(hWeb0, NULL, 230);
WebTcpipRecvExact(hWeb0, NULL, 47);
WebTcpipRecvExact(hWeb0, NULL, 58);
WebTcpipRecvExact(hWeb0, NULL, 40);
WebTcpipRecvExact(hWeb0, NULL, 594);
WebTcpipRecvExact(hWeb0, NULL, 340);
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipRecvExact(hWeb0, NULL, 188);
WebTcpipRecvExact(hWeb0, NULL, 147);
// Choose "1" from main menu and hit RETURN
ThinkTime(7.0);
WebTcpipSend(hWeb0, "1");
WebTcpipRecvExact(hWeb0, NULL, 1);
WebTcpipSend(hWeb0, "\r");
WebTcpipRecvExact(hWeb0, NULL, 2);
WebTcpipRecvExact(hWeb0, NULL, 7);
WebTcpipRecvExact(hWeb0, NULL, 21);
WebTcpipRecvExact(hWeb0, NULL, 691);
// …
```

As you can see from the inline comments, this part of the script contains a login process (account name and password), followed by the selection of an item from a main menu (by hitting the *1* and <*RETURN*> keys).

If you leave a recorded script unchanged, it will typically play back without problems. However changes do have to be applied to scripts for data parameterization, response verification, etc.

Each keystroke is sent to the server as a single byte (without header or footer). The log file reveals that the server sends back the same byte as an echo:

```
WebTcpipSend(hWeb0, "s");
WebTcpipRecvExact(hWeb0, NULL, 1);
TcpipServerToClient(#432, 1 bytes)
{
  s
```

```
}
```

Looking back at the recorded script, you'll find that sending the password - as opposed to sending the user login name - doesn't trigger a sequence of echoes. This is because the password isn't supposed to appear on the terminal screen.

Note that the communication is full duplex. This means that both server and client can send simultaneously; they don't have to wait for each other. In the above example, you can see the result of this in the lines:

```
WebTcpipSend(hWeb0, "se");
WebTcpipRecvExact(hWeb0, NULL, 2);
```

Here, because of rapid typing during the recording session, the echo "s" came back only after the "e" keystroke had been sent to the server.

Once the password is sent, a number of WebTcpipRecvExact statements follow in the above code. In the recording's log file, one of these statements looks like this:

```
TcpipServerToClientBin(#432, 59 bytes)
{
  00000000 [8;19H·[;7m+---  1B 5B 38 3B 31 39 48 1B 5B
3B 37 6D 2B 2D 2D 2D
  00000010 ---Hinweis: Kein 2D 2D 2D 48 69 6E 77 65 69
73 3A 20 4B 65 69 6E
  00000020 Kunde gefunden-  20 4B 75 6E 64 65 20 67 65
66 75 6E 64 65 6E 2D
  00000030 ------+·[m·      2D 2D 2D 2D 2D 2D 2B 1B 5B
6D 0A
}
```

The server response contains plain text as well as meta information, such as text position and format (note that in German, ""Hinweis: Kein Kunde gefunden" means "Message: No customer found").

In this Telnet example, determining when the server response is complete is challenging. First, the packet length is not included in the response data (Case I). Second, there is no termination byte sequence (Case II). Therefore this example represents Case III: No information on response packet size from section "Dynamically Receiving Server Responses".

> **Note** In other Telnet based projects, you may find terminator strings in server response data (for example, the command prompt character). This depends entirely on the application under test.

To solve this problem generically, a *TelnetReceive*-Response function that accepts incoming server responses of unspecified length in a loop is written. The loop is terminated when the client waits for a new response packet for more than a specified number of seconds. The corresponding function code is included later in this section.

Looking at the same part of the script after customization, the structure is more visible, and the server responses are handled by the new function. Note that complete strings can be sent to the server - as opposed to sending each key stroke as a single packet: you then don't have to wait for each echo character individually, because they can be read asynchronously (due to the full-duplex nature of the Telnet protocol) after sending the complete request.

```
// Login: Send Username
WebTcpipSend(hWeb0, "testuser\r");
TelnetReceiveResponse(hWeb0, 1.0, "Login: Username");

// Login: Send Password
WebTcpipSend(hWeb0, "password\r");
TelnetReceiveResponse(hWeb0, 5.0, "Login: Passwort");

// Choose "1" from main menu and hit RETURN
WebTcpipSend(hWeb0, "1\r");
TelnetReceiveResponse(hWeb0, 5.0, "Choose 1 from menu");
```

The *TelnetReceiveResponse* function eliminates the need to wait for incoming server data for appropriate periods of time. It takes three parameters:

- *hWeb0*: Is the handle of the open TCP connection
- *fTimeout*: Defines how to decide when the server response is complete: If after fTimeout seconds, no further server response is available, the function returns.
- *sAction*: This string is used for appropriate naming of the custom timer, and can be used for logging and debugging purposes.

Sample

```
function TelnetReceiveResponse(hWeb0: number;
                               fTimeout: float;
                               sAction: string): number
var
  sData:      string(4096);
  nRecv:      number;
  nRecvSum:   number;
  fTime:      float;

begin
  gsResponse := "";
  nRecvSum   := 0;

  MeasureStart(sAction);

  while WebTcpipSelect(hWeb0, fTimeout) do
    if NOT WebTcpipRecv(hWeb0, sData, sizeof(sData), nRecv)
    then
      exit;
    end;
```

```
    if nRecv = 0 then
      exit;
    end;
    SetMem(gsResponse, nRecvSum + 1, sData, nRecv);
    nRecvSum   := nRecvSum + nRecv;
  end;

  MeasureStop(sAction);
  MeasureGet(sAction, MEASURE_TIMER_RESPONSETIME,
             MEASURE_KIND_LAST, fTime);

  if fTime > fTimeout then
    fTime := fTime - fTimeout;
  end;
  MeasureIncFloat("RespTime: " + sAction, fTime, "sec",
                  MEASURE_USAGE_TIMER);

  TelnetReceiveResponse := nRecvSum;

end TelnetReceiveResponse;
```

The function works as follows: It waits until a server response is ready to be read in under *fTimeout* seconds (*WebTcpipSelect*). As soon as a server response is available, it is received and appended to the global string variable *gsResponse*. The loop terminates when the server response is empty, when the timeout is exceeded, or if *WebTcpipRecv* fails for any reason.

This loop structure is necessary because often a Telnet server will send a line of characters, nothing will happen for a couple of seconds, and then suddenly more lines come in.

Some care must be taken when looking at these time measurements. Because of the nature of the timeout, the time measurements usually include a final timeout period of *fTimeout* seconds. This has to be subtracted from the measured time to get the true roundtrip time measurement. This corrected time measurement is made available as a custom measurement with the name "RespTime: " + *sAction* and dimension "seconds."

The TN3270e Protocol

The TN3270e protocol is a method of emulating 3270 terminal and printer devices via Telnet. It is used by terminal emulation software such as NetManage's Rumba® and Hummingbird's HostExplorer® for direct connections to mainframes.

Similar to Telnet, a typical TN3270e session consists of two main parts:

1 A connection is established; session details and options are negotiated.

2 The application starts, and from that point forward generated traffic is user driven.

Part I: Establishing a Connection

The first part of a typical recorded script of a TN3270e session looks like this:

```
WebTcpipConnect(hWeb0, "10.19.111.201", 7230);
WebTcpipRecvExact(hWeb0, NULL, 3);
WebTcpipSendBin(hWeb0, "\hFFFB28", 3);     // ·û(
WebTcpipRecvExact(hWeb0, NULL, 7);
WebTcpipSendBin(hWeb0,
  "\hFFFA28020749424D2D333237382D342D"
    // ·ú(··IBM-3278-4-
  "\h45FFF0", 19);                        // E·ð
WebTcpipRecvExact(hWeb0, NULL, 28);
WebTcpipSendBin(hWeb0, "\hFFFA2803070004FFF0", 9);
  // ·ú(·····ð
WebTcpipRecvExact(hWeb0, NULL, 9);
WebTcpipRecvExact(hWeb0, NULL, 347);
```

This looks similar to the first part of the Telnet session from the previous section because the TN3270e protocol is based on the Telnet protocol. Again, nothing needs to be changed here. Because this is a stateful, connection-oriented protocol, session handling is not an issue.

The traffic from the first part of the session is translated into TELNET code in the following table:

Red (italic) = Server to Client - **Green (bold) = Client to Server**	
FF FD 28	*IAC DO TN3270E*
FF FB 28	**IAC WILL TN3270E**
FF FA 28 08 02 FF F0	*IAC SB TN3270E SEND DEVICE-TYPE IAC SE*
FF FA 28 02 07 49 42 4D 2D 33 32 37 38 2D 34 2D 45 FF F0	**IAC SB TN3270E DEVICE-TYPE REQUEST "IBM-3278-4-E" IAC SE**
FF FA 28 02 04 49 42 4D 2D 33 32 37 38 2D 34 2D 45 01 54 39 35 49 54 51 4D 55 FF F0	*IAC SB TN3270E DEVICE-TYPE IS "IBM-3278-4-E" CONNECT "T95ITQMU" IAC SE*
FF FA 28 03 07 00 04 FF F0	**IAC SB TN3270E FUNCTIONS REQUEST (BIND-IMAGE SYSREQ) IAC SE**
FF FA 28 03 04 00 04 FF F0	*IAC SB TN3270E FUNCTIONS IS (BIND-IMAGE SYSREQ) IAC SE*

In summary, server and client agree on a protocol, a device type (IBM-3278-4-E), and on whether or not to use certain protocol features. Note that the terminal name (T95ITQMU) is assigned by the server, not the client. This is because the server system holds a database that handles the mapping of client IP addresses to terminal names.

Part II: User Interaction

The second part of the script contains the user interaction. The following request-response pair example represents an interaction where the user enters an account number ("238729") into a text field and then hits the RETURN key:

```
WebTcpipSendBin(hWeb0,
  "\h00000000007DC6C61140C4F311C640F2"
     // ·····}ÆÆ·@Äó·Æ@ò
  "\hF3F8F7F2F9FFEF", 23);              // óø÷òù·ï
WebTcpipRecvExact(hWeb0, NULL, 199);
```

Knowing that the traffic is encoded in EBCDIC (rather than ASCII), the account number can be found in the request string using an EBCDIC code table: "238729" is represented as the binary byte sequence 0xF2 F3 F8 F7 F2 F9 in EBCDIC. This is the part that is relevant to customization; the rest may be left unchanged. An EBCDIC code map is included in section "EBCDIC to ASCII Character Code Conversion" - there, an explanation of conversion between EBCDIC and ASCII using Silk Performer's script language (BDL) is discussed.

Here's an account of the other parts of this message. The first five bytes represent the TN3270e message header (cf. RFC 2355):

Field	Length	Value in our example	
Data type	1 byte	0x00	3270-DATA
Request flag	1 byte	0x00	ERR-COND-CLEARED
Response flag	1 byte	0x00	NO-RESPONSE
Sequence number	2 bytes	0x0000	Sequence numbers may or may not be used. In this case, they aren't.

Each client request is terminated by a two-byte sequence: 0xFF EF.

The remainder of the request data (bytes #6 - #21 in this example) is application data containing screen positions, key codes, and text.

The server response can be analyzed in the corresponding log or TrueLog files from the recording session. In this example, the response is a 199-byte data block beginning with five zero bytes (0x00) and ending with 0xFF EF, just like the request data. The data content in between is similar to the request data: cursor positions, formatting, and text content (encoded in EBCDIC).

A simple customization of the script extract above, using the function library introduced in section "Request Parameterization", would look like this:

```
ResetRequestData();
AppendRequestData("\h00000000007DC6C61140C4F311C640",
                  15);
AppendRequestData(ASC2EBCDIC(sAccountNr) + "\hFFEF");
SendTcpipRequest(hWeb0);
WebTcpipRecvUntil(hWeb, sResp, sizeof(sResp), nRecv,
                    "\hFFEF", 2);
```

Here a string variable *sAccountNr* has been introduced for the account number (where "238729" was used during recording). The function ASC2EBCDIC that converts between ASCII and EBCDIC is explained in the following section.

Finally, the inflexible *WebTcpipRecvExact* has been replaced with *WebTcpipRecvUntil*, which is appropriate here because the trailing byte sequence is known. The response data is stored in the string variable *sResp*, and *nRecv* contains the number of received bytes.

For logging and verification purposes, the server response should be translated from EBCDIC to ASCII. For example:

```
Writeln("Response: " + EBCDIC2ASC(sResp, nRecv));
```

When necessary, single response data (such as a new ID that's needed as input data for subsequent requests) can be extracted from this response using the *StrSearchDelimited* function.

Taking customization a step further, it's good practice to replace the *WebTcpipRecvUntil* call in the script above with a *MyWebTcpipRecv* function that encapsulates all the necessary actions on server responses:

- Call *WebTcpipRecvUntil* with the appropriate end byte sequence.
- Convert the response from EBCDIC to ASCII.
- Log the response to the output file (in ASCII).
- Do generic error checking on the response by searching for common error messages.
- Make the response data available as a return value or global variable for further verification or data extraction.

In the Silk Performer script, each client request should be followed by a call to this new function, replacing the *WebTcpipRecvExact* function calls from the recording session.

EBCDIC to ASCII Character Code Conversion

EBCDIC (Extended Binary Coded Decimal Interchange Code) is the data alphabet used in all IBM computers, except personal computers. A conversion routine that translates server responses from the EBCDIC character set to ASCII is easy to implement.

The function *EBCDIC2ASC* uses the code map defined by the array as *EBCDIC_2_ASCII*, which maps each EBCDIC character to its ASCII equivalent. The function ASC2EBCDIC - required for translating client requests from ASCII to EBCDIC - works in a similar manner.

Sample

```
var
  asEBCDIC_2_ASCII : array[256] of string(1) INIT
    "","","","","","","","","","","","","","","","","","","","",
    "","","","","","","","","","","","","","","","","","","","",
    "","","","","","","","","","","","","","","","","","","","",
    "","","","","","","","","","","","","","","","","","","","",
    "","","","","","","","","","","","","","","","","","","","",
    "","","","","","","","","","","","","","","","","","","","",
    "","","","","","","","","","","","","","","","","","","","",
    "","","","","","","","","","","","","","","","","","","","",
    "","","","","","","","","","","","","","","","","","","","",
    "","a","b","c","d","e","f","g","h","i","","","","","","","",
    "","j","k","l","m","n","o","p","q","r","","","","","","","",
    "","~","s","t","u","v","w","x","y","z","","","","","","","",
    "","","","","","","","","","","","","","","","","","","","",
    "","A","B","C","D","E","F","G","H","I","","","","","","","",
    "","J","K","L","M","N","O","P","Q","R","","","","","","","",
    "","","S","T","U","V","W","X","Y","Z","","","","","","","",
    "0","1","2","3","4","5","6","7","8","9","","","","","","","";

function EBCDIC2ASC(pEBCDIC: string; pMaxLen: number
optional)
  var
    i: number;
  begin
  //
   if pMaxLen = 0 then
      pMaxLen := Strlen(pEBCDIC);
      //writeln("length of string : " + string(pMaxLen));
   end;
   //
   // writeln("EBCDIC STRING " + pEBCDIC);
   //
  for i := 1 to pMaxLen do
  //
  //writeln("Ordinal Value:" + String(ord(pEBCDIC[i])));
  //write("Value From Array:" + asASCII_2_
EBCDIC[ord(pEBCDIC[i]) + 1]);
  write(asEBCDIC_2_ASCII[ord(pEBCDIC[i]) + 1]);
  // writeln;
```

```
//
end;
 writeln;
end EBCDIC2ASC;
```

The table below is the standard EBCDIC table for the 2780/3780 Protocol code map (taken from [EBCDIC_CTI]). As an example, to decode the EBCDIC byte 0x83, choose row "8" and column "3". You'll find that 0x83 maps to the letter "c" in ASCII.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX		PT			GE				FF	CR		
1	DLE	SBA	EUA	1C		NL				EM			DUP	SF	FM	ITB
2							ETB	ESC						ENQ		
3			SYN					EOT					RA	NAK		
4	SP										¢	.	<	(	+	
5	&										!	$	*	)	;	¬
6	-	/									|	,	%	_	>	?
7											:	#	@	'	=	"
8		a	B	c	d	e	f	g	h	I						
9		j	K	l	m	n	o	p	q	r						
A		~	S	t	u	v	w	x	y	z						
B																
C	{	A	B	C	D	E	F	G	H	I						
D	}	J	K	L	M	N	O	P	Q	R						
E	\		S	T	U	V	W	X	Y	Z						
F	0	1	2	3	4	5	6	7	8	9						

Figure 10 - EBCDIC code map

[EBCDIC_UNI] is a reference that presents the specifications of the UTF-EBCDIC - EBCDIC Friendly Unicode (or UCS) Transformation Format.

Issues with "Keep Alive" Mechanisms

Note The sequence in the load test script file below that was taken from a TN3270e traffic recording:

```
WebTcpipRecvExact(hWeb0, NULL, 3);
WebTcpipSendBin(hWeb0, "\hFFFB06", 3);  // IAC WILL
TIMING-MARK
WebTcpipRecvExact(hWeb0, NULL, 3);
WebTcpipSendBin(hWeb0, "\hFFFC06", 3);  // IAC WON'T
TIMING-MARK
```

In this example, the server challenges the client with a three-byte code (first line of the code above) whenever the client is inactive for a period of time. The client responds with three-byte sequence of its own. Such "ping pong" activity, initiated by the server, can serve as a control mechanism for detecting whether or not a client is still active.

These lines cause problems during replay because they aren't predictable or reproducible. To address such a situation, rather than incorporating the appropriate intelligence into a Silk Performer script - which would be a daunting task - simply disable this feature on the server.

Some background info from RFC 860 (Telnet timing mark option):

> It is sometimes useful for a user or process at one end of a TELNET connection to be sure that previously transmitted data has been completely processed, printed, discarded, or otherwise disposed of. This option provides a mechanism for doing this. In addition, even if the option request (DO TIMING-MARK) is refused (by WON'T TIMING-MARK) the requester is at least assured that the refuser has received (if not processed) all previous data.

IP Spoofing

For successful replay of such a script with parallel virtual users, each user must use a unique IP address.

Check the option Client IP address multiplexing in Silk Performer's Active Profile/Internet/Network tab page, and configure enough IP addresses on Silk Performer agents so that each virtual user can have a unique IP address ("IP Spoofing"). IP addresses can be configured using the IP Address Manager in the System Configuration Manager that ships with Silk Performer.

Recording Rule Configuration Files

Silk Performer's recorder can be configured to generate correct WebTcpipRecvUntil calls for the TN3270e protocol automatically using a recording rule configuration file.

Two different rules must be specified: in the first part of the session where the session parameters are negotiated, 0xFF F0 is used as the termination sequence, while 0xFF EF serves as the termination sequence in the 3270-DATA part of the session.

These two parts can be distinguished by checking the first byte of the response data. In the first part, it is equal to 0xFF (Telnet code IAC - "Interpret as Command"), while it is equal to 0x00 (Code "3270-DATA", cf. the TN3270e response header above) in the second part.

The resulting recording rule XML file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<RecordingRuleSet>
  <TcpRuleRecvUntil>
    <Name>Telnet session start</Name>
    <Identify>
      <TermData>&#xFF;&#xF0;</TermData>
      <IgnoreWhiteSpaces>false</IgnoreWhiteSpaces>
    </Identify>
    <Conditions>
      <CompareData>
        <Data>&#xFF;</Data>
        <ApplyTo>Left</ApplyTo>
        <Offset>0</Offset>
      </CompareData>
    </Conditions>
  </TcpRuleRecvUntil>
  <TcpRuleRecvUntil>
    <Name>3270-DATA</Name>
    <Identify>
      <TermData>&#xFF;&#xEF;</TermData>
      <IgnoreWhiteSpaces>false</IgnoreWhiteSpaces>
    </Identify>
    <Conditions>
      <CompareData>
        <Data>&#x00;</Data>
        <ApplyTo>Left</ApplyTo>
        <Offset>0</Offset>
      </CompareData>
    </Conditions>
  </TcpRuleRecvUntil>
</RecordingRuleSet>
```

A Custom TCP/IP Based Protocol

The following example comes from an application that used an entirely custom TCP/IP based protocol. In this protocol, both the client request and server response data obey a set of predefined message telegram structures consisting of fixed-length fields. Some of the fields are binary (and may therefore contain zero bytes) some are string type.

Here is a typical request-response pair from the recorded traffic:

```
WebTcpipConnect(hWeb0, "10.9.96.81", 3311);
WebTcpipSendBin(hWeb0,
"\h00000280000000000000000000000000" // ................
"\h00010000000000000004F4B36302020" // ..........OK60
...
"\h303030302B303030303030303030303030" // 0000+00000000000
"\h30303030000000000000000000000000" // 0000............
, 640);
```

```
WebTcpipRecvExact(hWeb0, NULL, 80);
WebTcpipShutdown(hWeb0);
```

Here is the server response from the record log. Based on the third argument of the *WebTcpipRecvExact* function in the above script, we already know that it contains 80 bytes:

```
00 00 00 00  ···P·····------  00 00 00 50 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 10  ··········OK60  00 01 00 00 00 00 00 00 00 00 4F 4B 36 30 20 20
00 00 00 20    ··U60OKS1 SILK  20 20 00 00 55 36 30 4F 4B 53 31 20 53 49 4C 4B
00 00 00 30  01  ------------  30 31 20 20 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 40  ----------------  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Note The first four bytes: 0x00 00 00 50. They are a representation of a four-byte integer (a long variable) with the decimal value 80 (5 * 16 = 80). Note that the length of the entire packet is 80 bytes, hence the 4 initial bytes are included in the "packet size" variable.

If available, consult protocol documentation regarding the custom application under test - otherwise you will have to experiment to discover such protocol behavior on your own.

Note That the client request telegram contains the same information. The first four bytes are 0x00 00 02 80, equal to 2*256 + 8*16 = 640 - this is the number of bytes sent to the server.

Here is the script after customization. Note that:

- An added pair of MeasureStart/MeasureStop functions has been added to measure the time for the request.
- The request has been split into several pieces for parameterization, using the function library introduced in section "Request Parameterization".
- The *WebTcpipRecvExact* function has been replaced by a call to *WebTcpipRecvProto*

Here is the result:

```
ResetRequestData();
AppendRequestData("\h00000280000000000000000000000000"
                  "\h00010000000000000000", 26);
AppendRequestData("OK60    ");
AppendRequestData("\h0000", 2);
AppendRequestData(sUser + sPassword);
// ... (some more lines not displayed here)

MeasureStart("Write Journal");
WebTcpipConnect(hWeb0, "10.9.96.81", 3311);
SendTcpipRequest(hWeb0);
WebTcpipRecvProto(hWeb0, 0, 4, TCP_FLAG_INCLUDE_PROTO,
                  sResponse, sizeof(sResponse), nReceived);
// ... (response verification not displayed here)
WebTcpipShutdown(hWeb0);
```

```
  MeasureStop("Write Journal");
```

Recording Rule Configuration Files

The following recording rule XML file configures Silk Performer's recorder so that it generates the correct WebTcpipRecvProto calls in place of WebTcpipRecvExact. The settings in the "Identify" node map to the arguments of WebTcpipRecvProto in the script extract above.

```
<?xml version="1.0" encoding="UTF-8"?>
<RecordingRuleSet>
  <TcpRuleRecvProto>
    <Name>My custom TCP protocol</Name>
    <Identify>
      <LengthOffset>0</LengthOffset>
      <LengthLen>4</LengthLen>
      <OptionFlags>ProtoIncluded</OptionFlags>
    </Identify>
  </TcpRuleRecvProto>
</RecordingRuleSet>
```

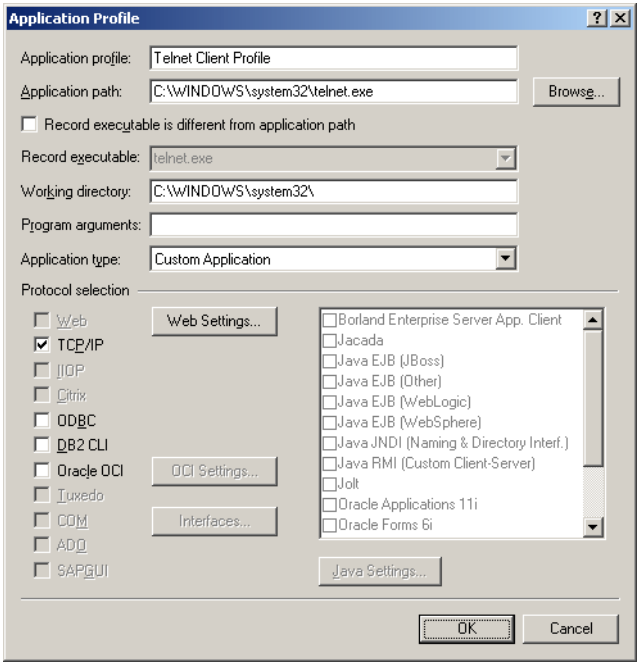
Recorder Settings

Silk Performer offers two mechanisms for recording TCP/IP based protocol traffic. The following sections explain how to configure them and how to select an appropriate configuration.

"Socksifying" an Application

If you know the path of the client application executable that's connecting to the server, you can set up an application profile for the application in *Settings/*

System/Recorder/Application Profiles. Here is an example using the application profile for the Windows Telnet client:

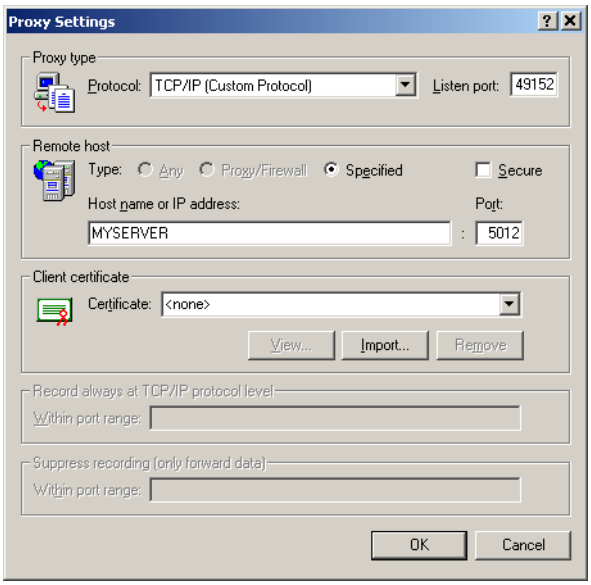


TCP Proxy Recorder

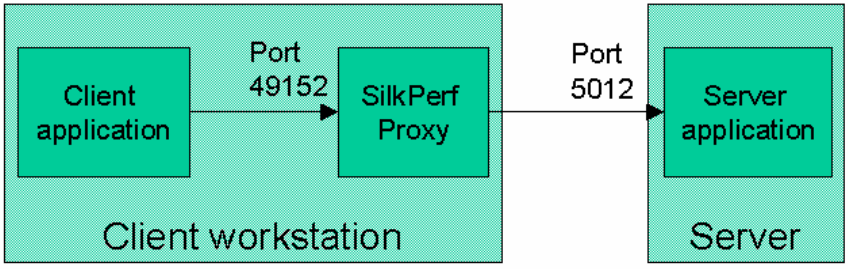
An alternate solution involves using a TCP/IP proxy recorder. As a prerequisite, you must be able to control the server name and TCP port number the client application connects to. Typically, these settings can be found in the registry or in a configuration file. Then, you can configure Silk Performer and the client application so that traffic is explicitly routed over Silk Performer's recorder.

Assuming that the client connects to MYSERVER on port 5012, you would change these settings to LOCALHOST and port 49152 (or any other unused TCP port) in the registry or appropriate configuration file. In Silk Performer, choose Settings/System/Recorder/Proxies. Add a new proxy, and configure it as

a TCP proxy listening on port 49152, connecting to the remote host MYSERVER on port 5012:



In this way, the following scenario is configured:



The client application doesn't notice a difference in performance when the recorder is running - this is in contrast to the scenario in which the client is connected directly to the server.

Choosing Appropriate Recorder Settings

Socksifying such an application is the most commonly accepted approach. There are however some scenarios in which using a TCP proxy recorder is the better solution:

- Using the TCP proxy mechanism, Silk Performer doesn't have to run on the same machine as the client application. This works if for example both client and server are Unix applications. Just configure them in such a way that they connect to each other via the Silk Performer proxy residing on a Windows box.
- Sometimes, there are numerous client application processes all connected to the same server. In such instances it may be quicker to set up a TCP proxy than to it is to search NT's task manager for all processes that are generating traffic.
- Configuration is easy if you only need to record on one or a small number of TCP ports.

Summary

You should now be prepared for your own load test projects involving legacy or custom TCP based protocols.

This chapter has explored all the major scripting challenges - dynamic server responses, different code tables, string manipulation of binary data, etc. - and has taken a look at two important protocols (Telnet and TN3270e) from a load-testing perspective.

Taken together, the TCP/IP TrueLog Explorer, the extended set of TCP/IP functions, and rule-based recording represent a strong toolset for efficient development of load test scripts for both legacy and custom TCP/IP based protocols.

9

Load Testing Microsoft Based Distributed Applications

What you will learn

This chapter contains the following sections:

Overview

This chapter explains how Silk Performer handles various Microsoft technologies that are frequently used to build e-business applications. Microsoft's sample e-business application, Duwamish Online, is used for all demonstrations. This application allows Silk Performer's HTTP (S), COM and ODBC capabilities to be demonstrated in a comprehensive, real-world load test scenario. In addition to the better-known front-end Web tests, this chapter explains how you can use Silk Performer to examine middle-tier and back-end applications to identify performance problems.

Introduction

In an ongoing effort to meet customers' needs, today's e-business applications have become increasingly complex and involve numerous technologies across multiple tiers. Microsoft technologies, such as Internet Information Server (IIS), Active Server Pages (ASP) and COM, are often applied to enhance the use of Internet portals. Complex application structures result - making it necessary in many cases to monitor and load test middle- and back-end tiers in addition to presentation tiers (front end).

Duwamish Online ("Duwamish" for short) is a virtual "real world" application that's useful for demonstrating the capabilities of Silk Performer in load testing fundamental MS technologies - especially Internet Information Server, COM+ Application Server and MSQL Server. Duwamish Online is a sample online store that sells books, apparel, and related items. It serves as a tutorial application for MSDN Online to demonstrate the use of MS techniques in building complete e-business applications. The version of Duwamish referenced in this chapter simulates an early stage of development in which Duwamish's middle-tier lacks performance optimization, and therefore reveals a middle-tier bottleneck during load testing.

The Sample Application

Logical Architecture of Duwamish Online

Figure 11 shows the basic architecture of Duwamish Online. On the front-end (the presentation layer), Duwamish offers its services by means of an Active Server Pages (ASP) application. The middle-tier implements the bulk of the customer services. It consists of two layers: the workflow layer (WFL), which is used directly (and partially implemented) by the ASP application, and the

business logic layer (BLL). All data are exchanged between the ASP application and the middle-tier in XML format as parameters of COM functions. Products for sale on Duwamish are stored in a database on the back-end that's managed by Microsoft SQL Server 2000. The connection to the database is made in the data access layer (DAL) using Microsoft ActiveX Data Objects (ADO). ADO, in turn, connects the middle-tier and the back-end via an ODBC interface.

Beyond the synchronous portion of the middle-tier - which enables catalog browsing, account management, and purchase transactions - Duwamish includes two asynchronous layers: the QWFL (queued workflow layer) and the FWFL (fulfillment/logistic workflow layer). The QWFL performs credit card authentication and payment processing. The FWFL handles order fulfillment through an external logistic service provider. Communication between the synchronous and asynchronous layers is handled by Microsoft Message Queue Server (MSMQ). The Windows 2000 Task Scheduler regularly starts asynchronous tasks to process received requests.

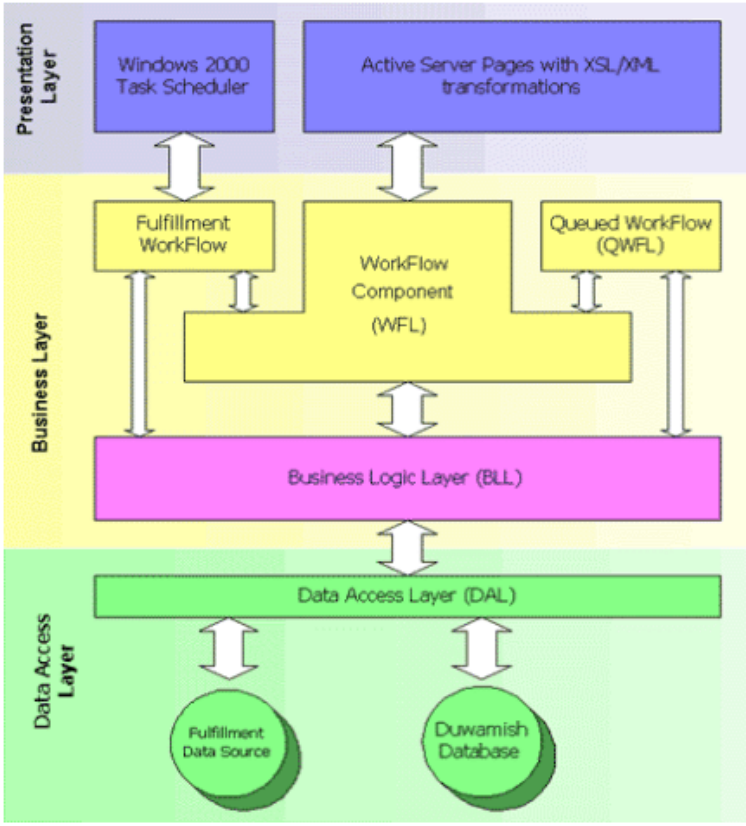


Figure 11 - Duwamish logical architecture

This figure shows an overview of the logical parts of the Duwamish application.

Physical Architecture of Duwamish Online

Figure 11 offers as overview of the logical architecture of Duwamish. Unfortunately, the logical architecture does not provide enough information regarding how Duwamish is deployed.

Figure 12 depicts a more detailed view of Duwamish when it is deployed. The Web server (the front-end) has been installed with the synchronous middle-tier components on one machine (Test Machine 1 in Figure 12); the asynchronous middle-tier components and the database (the back-end) run on a different machine (Test Machine 2 in Figure 12).

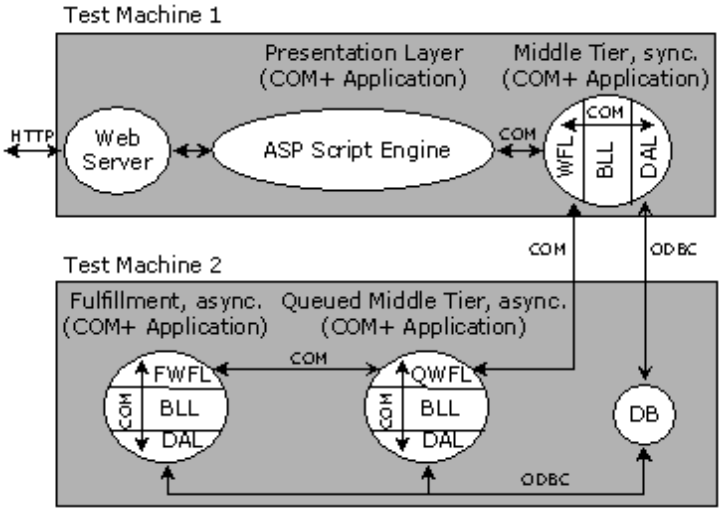


Figure 12 - Duwamish deployment

Circles represent processes; rectangles represent machines. The component-DLLs include WFL (Workflow Layer), BLL (Business Logic Layer), DAL (Data Access Layer), FWFL (Fulfillment Workflow Layer), and QWFL (Queued Workflow Layer). Silk Performer enables load tests of servers that support HTTP, and ODBC. For more information regarding deployment options for Duwamish, see Duwamish Online: Web Farm Installation.

As you can see from Figure 12, the Duwamish installation deploys four instances of the COM+ application server (dllhost.exe):

- Business presentation (ASP Script Engine on Test Machine 1): The ASP (Active Server Pages) application runs in a process (in an instance of the COM+ application server) separate from the Web server. This configuration protects the Web server from risks associated with the ASP application crashing.
- Customer services (Middle-Tier on Test Machine 1): This application provides customer services such as product search and account setup.
- Order processing (Queued Middle-Tier on Test Machine 2): This application implements all the services that are required for processing orders - including authorizing and billing credit cards for purchases after product delivery. The application performs these tasks asynchronously because they consume a considerable amount of time and don't require immediate feedback.
- Product delivery (Fulfillment on Test Machine 2): This application handles communication with a third-party fulfillment company. Such a company maintains product inventory in their warehouse and ships products to customers. All communication tasks, such as the uploading of order forms and the reporting of order status, are performed asynchronously (no immediate feedback is required).

Test Environment

In addition to the two machines that host Duwamish, four other machines are required for the installation of the load test environment (see Figure 13). On one machine the entire load test process is controlled and test results are evaluated; this machine runs both Silk Performer's Controller and the Performance Explorer for server monitoring and results evaluation. Three Silk Performer agent machines have also been set up for the Web load tests and the load tests of the middle-tier and back-end (DB server). For more information on various setup options for the Duwamish sample application, please see Duwamish Online: Web Farm Installation.

Test Goals and Procedures

These initial load tests evaluate a version of Duwamish that has intentionally been designed with performance problems - to allow Silk Performer to detect a bottleneck in the application's middle-tier. Subsequent scalability testing of an improved version of Duwamish reveals that the program performs without error.

During the load test attention is concentrated on the synchronous part of the middle-tier (Customer Services, see above and Figure 14) as it must

immediately react to user inquiries, many of which don't result in product purchases. The asynchronous part (Order Processing and Product Delivery, see above) is responsible for managing and monitoring product orders. Due to the relatively long period of time required to ship products, these tasks don't require immediate feedback.

To identify performance bottlenecks in the middle-tier, the entire application must be put under real load. Only after typical usage scenarios have been specified can accurate load tests identify inferior performance of specific components.

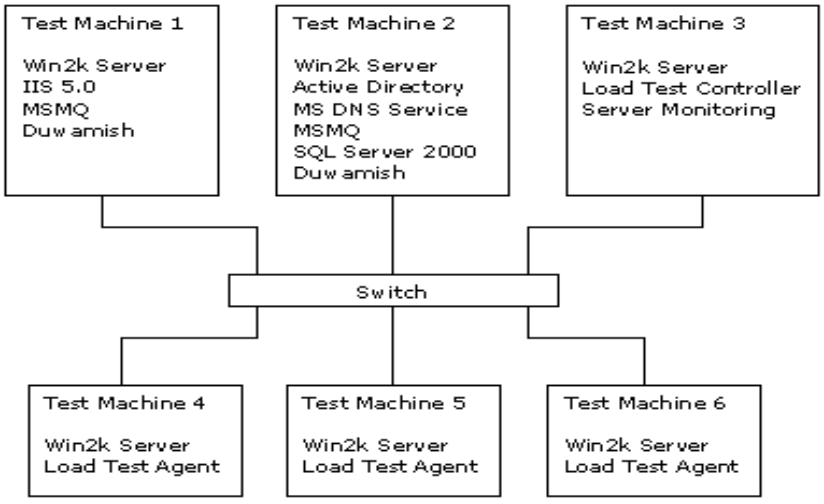


Figure 13 - Load Test Environment

All test machines are equipped with a single 800MHz processor, 512Mbyte RAM, and 100MBit Ethernet. The machines are connected to a Compaq 5411 1GBit Ethernet switch.

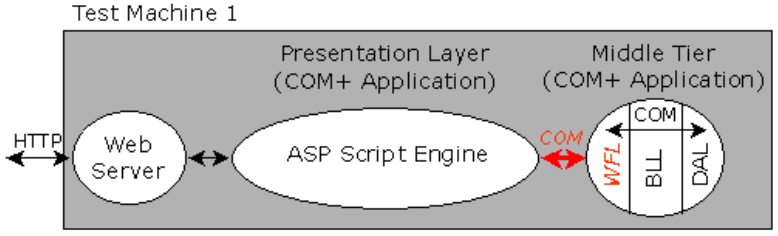


Figure 14 - Test the bottleneck

The synchronous part of Duwamish's middle tier (represented by the COM component WFL) contains the bottleneck.

Specification of Test Scenarios

Finding an accurate test scenario (usage scenario) for a system that isn't yet online can be challenging. Specifications for usage scenarios are comprised of two parts:

1. Use case distribution:
   Market research institutes can be helpful in providing statistics regarding typical usage scenarios for comparable online applications. For the sake of this demonstration, consider the following usage distribution (which is based on past experience and comparable application statistics):
   - 62% Category searchSearch for an Item by Category
   - 21% Keyword searchSearch for an Item by Keyword
   - 10% Home PageOpen only the Home Page
   - 5% Create an accountCreate a User Account
   - 2% Order an itemBuy a product
2. Use case actions:
   The sequence of actions that make up each use case (for example "Category search" and "Create an account") must be specified. In a test script such actions are represented by, for example, calls to Silk Performer's Web Page API. The actions that a use case involves depend on the purpose of each use case and the design of the test object (Duwamish). Action sequences must be carefully defined because they affect the value of load tests.

For more information about usage scenarios, see Duwamish Online: Capacity Planning.

First Load Test

The first load test evaluates the entire application from the Web front-end (see section "Front-End Test"). The use of critical system resources is monitored - for example CPU utilization - on all machines on which Duwamish is deployed. Any part of the system that reaches the CPU usage threshold contains a bottleneck. Closer examination of such parts reveals exactly where the bottlenecks occur. In this example, a bottleneck is located in the middle-tier (see section "Structure of an ASP Application").

Front-End Test

To execute accurate load tests that reflect real world Web server usage, it's important to simulate different kinds of users (customers) with different usage profiles. To do this, test scripts that describe varied and randomized user behavior are required. For a realistic usage scenario, users who access Duwamish via different network devices (e.g., T1 and different modems) are simulated using Silk Performer.

Creating a Web Load Test Scenario

The first step in running a Web load test with Silk Performer involves creating a new project of type **Web business transaction (HTML/HTTP)**. Neither the standard settings or the application profile need to be changed because the standard Web browser (Internet Explorer) is used to browse Duwamish's online store.

The **Model Script** button is selected in Silk Performer's workflow bar to begin recording a script that describes the behavior of a typical Duwamish customer. Later, other scripts that model the behavior of other types of customers will be recorded. In the **Workflow - Model Script** dialog, **Internet Explorer** is indicated as the application to be recorded, and the URL that Internet Explorer (IE) is to contact upon start up is entered. After clicking **Start recording**, Internet Explorer and Silk Performer's recorder open.

The next step is to record the usage scenarios, as specified in section "Specification of Test Scenarios". A separate script is created for each usage scenario; this way the browser cache can't delete generated test scripts. The following steps are performed for each usage scenario:

1. Start the browser.
2. Clear the browser's cache.
3. Clear Silk Performer's recorder log tab and script tab.
4. Load the Duwamish home page in the browser and click through the site's pages as required for a particular usage scenario.
5. Stop recording and save the generated script.

After recording all usage scenarios in separate test scripts, scripts are merged into a single script that's used to drive the load test. This way each recorded usage scenario becomes a separate Silk Performer transaction.

This process results in a relatively unrealistic load test that carries out user actions in exactly the sequence in which they are recorded. What's required is a script that simulates a group of users with similar behavior, not identical behavior. To achieve this, the generated script must be customized in three ways:

1. Randomizing user input:
   In the real world, different users buy different products. Therefore, user input (e.g., the products a user searches for or the products a user buys) must be customized. Virtual users select products randomly from sets of available products. For additional information regarding customizing Silk Performer scripts, see the "Load Testing a Web Application" tutorial.
2. Establishing user groups:
   In addition to randomizing user input, the test script must be customized in such a way that it reproduces the different usage scenarios specified in section "Specification of Test Scenarios". Five different user groups are established, each performing different transactions based on what such user types would likely do in the real world.

   Note The number of transactions that each user performs during a simulation must also be specified, either in absolute numbers (as with load tests applying the queuing workload model) or in relation to the transactions other users perform (as with load tests applying the steady workload model). However, as only the increasing workload model for front-end tests is used in this example, there's no need to bother with the number of transactions that each user performs.
3. Customizing the connection speed:
   As mentioned in section "Front-End Test", it's unrealistic to assume that all virtual users will connect to the Duwamish Web server with the same connection speed. So, during each user's initial transaction a random bandwidth is selected using the WebSetSpeed () BDL function. A randomly selected bandwidth is then passed to the function as a parameter. As a result, the network speed simulation for each user is initialized with a different value. For a better understanding of this customization process, please examine the attached sample files *<cdrom_drive>:\Extras\WhitePaper_Microsoft Technologies_ ProjectFiles.zip*.

Now that the test script has been customized, a model that determines the way that Silk Performer should distribute workload during test runs must be selected. As we have no idea how many concurrent users the installation of Duwamish can handle, Silk Performer must use the increasing workload model. Increasing workloads help explore the limits of server capacity. Here an increasing workload model that begins with 3 users and increases according to the specified user distribution to a total of 100 users is used (see Figure 15). Although one agent machine would be satisfactory, the virtual users are distributed over three agent machines, as it's expected that Duwamish will be required to handle more users in future test runs.

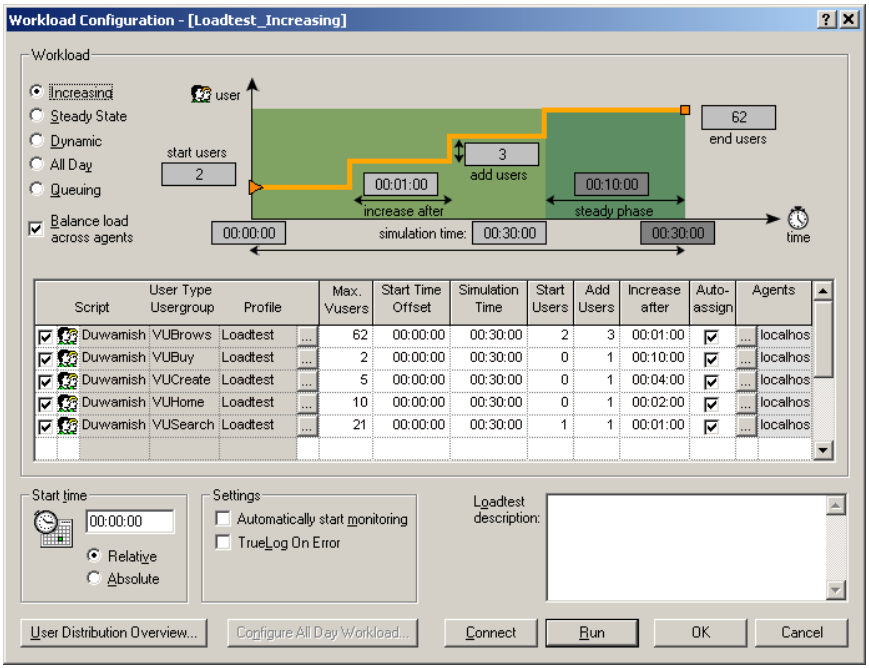


Figure 15 - Setup of an increasing workload

Escalating from 3 to 100 Users (62 VUBrowse + 2 VUBuy + 5 VUCreate + 10 VUHome + 21 VUSearch). The total number of users is distributed among the 5 user groups following the user distribution, which is specified in section "Specification of Test Scenarios" (note the "Max. Vusers" column). The total duration of the increasing phase is 20 minutes (1200 sec), whereas the entire test lasts 30 minutes (1800 sec). Silk Performer uses the default profile for all user groups (note the "Profile" column). Therefore, Silk Performer will consider think times during test runs.

Collecting System Data

As mentioned in the above scenario, there is a bottleneck in the Duwamish application that must be located. Increasing workloads are well suited to such a task. Performance data regarding the increasing number of transactions must be cross-referenced with performance data regarding the usage of system resources on the machines that host Duwamish. Relevant transaction performance data include actual throughput, response times, transaction errors, and similar data. Silk Performer collects such data automatically. However, Silk Performer's Performance Explorer must be configured to collect data regarding the usage of system resources. For the sake of this discussion, the most relevant processes are those of the Web server, the ASP script engine and the COM+ application server

that implements the customer service portion of the middle-tier (compare with section "Physical Architecture of Duwamish Online"). It's not expected that the database will cause problems, therefore there's no need to advise Performance Explorer to monitor the database server process.

Open Performance Explorer and create a new Monitor Report. The system data sources that should be captured during the load test (compare with Figure 16) are:

- Total CPU usage of the Web server/middle-tier machine
- CPU usage of the Web server's main process (inetinfo.exe)
- CPU usage of the Web server's ASP engine (dllhost.exe)
- CPU usage of the middle-tier process (dllhost.exe)

These data sources should be added to the open Monitor Report and **Write Monitor Data** should be enabled. Performance Explorer will now generate a TSD file with the specified counters. When these preparations are complete, the test can be run.

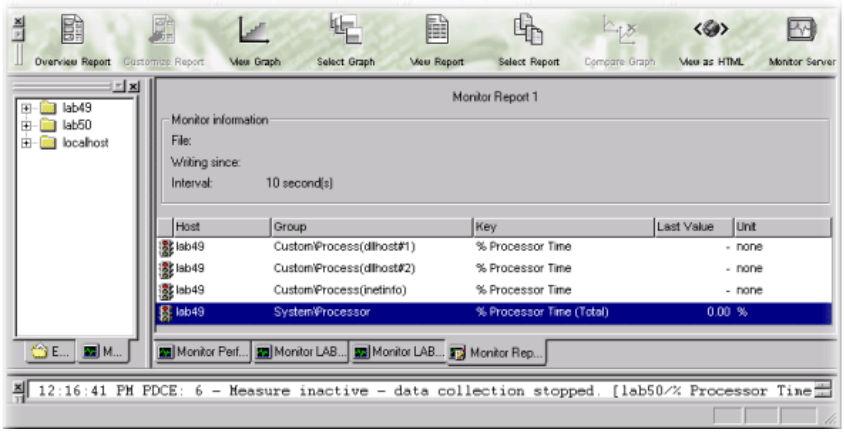


Figure 16 - Monitor the server processes CPU utilization

Inetinfo - IIS main process; dllhost#1 - ASP Script Engine; dllhost#2 - Duwamish Middle-Tier (synchronous part of Machine 1 in Figure 12 and Figure 14)

Analyzing the Results

After the load test, all generated time series data are placed in a new Performance Explorer workspace and the data series that represent the CPU usage of the different server processes are compared in a graph (see Figure Figure 17). Analyzing this graph, one can see that the machine hosting the Web server and the middle-tier reaches its peak capacity (CPU usage 100%) at about 60 concurrent users. One can also identify the middle-tier as the bottleneck in

the system. The process that hosts the middle-tier objects uses more than 50% of the servers CPU and blocks everything else, including the throughput, once a certain level is reached.

Now the middle-tier has been identified as containing a bottleneck. However, the middle-tier performs many tasks. One needs to determine which task in the middle-tier is responsible for the high resource usage. To pinpoint the problem, we need more performance data about each COM function (task) in the middle-tier. By isolating the problematic task we can expect to gain insight into which optimization will provide the most significant performance gain. To obtain performance data about each middle-tier task, a test that focuses on only the middle-tier needs to be set up.

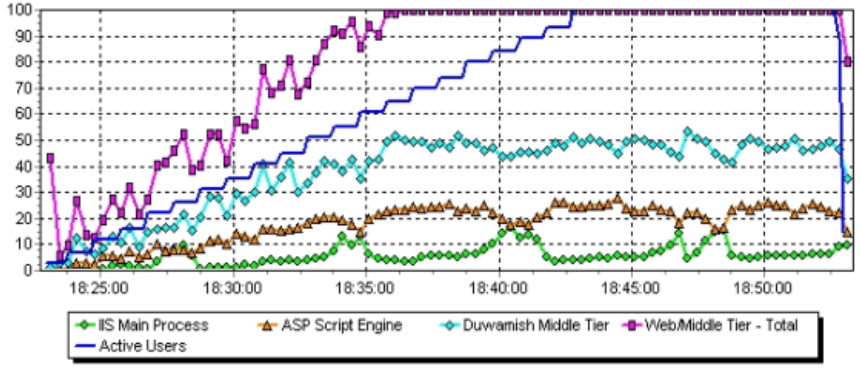


Figure 17 - CPU Usage of different server processes under increasing load

The middle-tier process (the COM+ application server dllhost.exe) utilizes more than 50% of the server's CPU capacity.

Structure of an ASP Application

In a standard IIS 5.0 configuration, an ASP application is configured to run in a process that's separate from the IIS main process. This process is actually the COM+ application server (dllhost.exe), which hosts the ASP script engine, which in turn interprets ASP applications. This configuration ensures that the IIS main process will keep running and be able to reinitialize the ASP application if the application server crashes.

When using default parameters, IIS 5.0 spawns up to 25 independent worker threads. So, as many as 25 threads may be responsible for handling incoming custom requests. In addition, IIS 5.0 offers an intelligent mechanism that limits the number of concurrent worker threads to a value that utilizes the CPU as efficiently as possible. When the queue of open requests reaches a specified limit, no further worker threads are created because they would lead to higher CPU load, which in turn would negatively impact the responsiveness of the Web

server. The application server that hosts the ASP application deploys the same number of threads as the IIS.

For a realistic load test, it's essential to simulate the COM+ application server that hosts the ASP application as accurately as possible. The first step toward this goal is the creation of a COM load test script that simulates a single ASP worker thread. Following that, Silk Performer's replay engine needs to be set up to run as many threads as the application server hosting the ASP script engine runs.

Preparing the Recording Session

To create the COM load test script, Silk Performer's COM recorder captures the traffic that's produced by the ASP script engine under the load generated by a single user. As mentioned above, the ASP script engine is loaded in an instance of the COM+ application server dllhost.exe. Because several instances of the application server can run on one machine, each possibly hosting an ASP script engine, the recorder doesn't know which instance of the application server to monitor. Therefore, the Duwamish ASP application must be reconfigured in such a way that it's loaded into its own distinguished instance of the COM+ application server. That will be the one COM+ application server that the recorder can uniquely hook into.

The Duwamish ASP application needs to be separated so that it runs in its dedicated COM+ application server. As this is not a common procedure, the details are outlined below:

1 Open **Internet Information Services** (Start/Administrative Tools) and, in the **Action** menu, open **Properties** for the Duwamish virtual directory (see Figure 18).

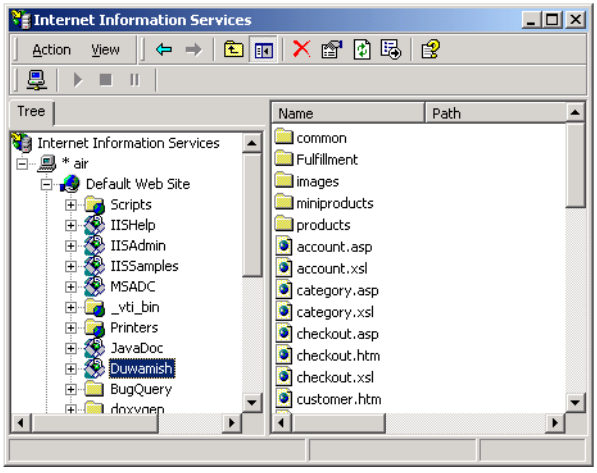


Figure 18 - Configuring the Duwamish ASP application

Each virtual directory installed in the Internet Information Server can be configured independently.

2 On the **Virtual Directory** tab, change the "Application Protection" property to "High (Isolated)," indicating that the ASP script engine for this application will run in its own instance of the COM+ application server (see Figure 19).

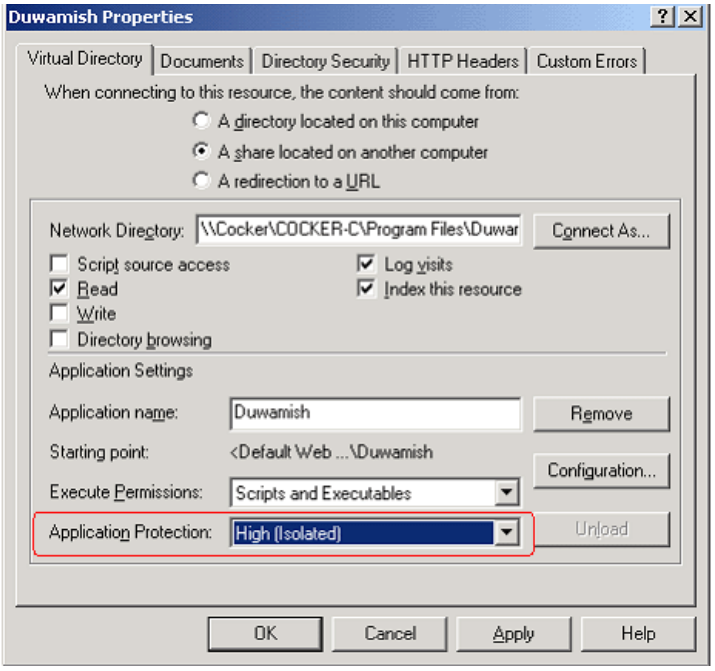


Figure 19 - Duwamish ASP properties

By changing the isolation level to "High (Isolated)", the ASP application no longer shares a COM+ application server with other ASP applications on the system - it uses its own instance.

Once the new properties have been saved, Duwamish appears in the **Component Services** manager as a new COM+ application. Open the Component Services manager (Start/Administrative Tools) to configure the newly created application as required. In the Component Services manager,

search for the entry "IIS-{Default Web Site//Root/Duwamish}", which is located in the "COM+ Applications" folder (see Figure 20).

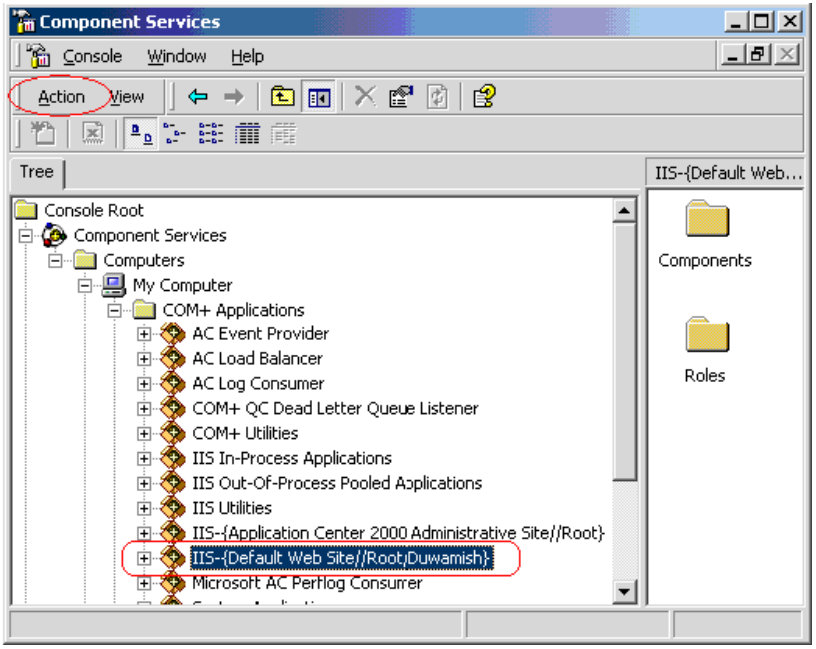


Figure 20 - Configuring a COM+ application

"IIS-{Default Web Site// Root/Duwamish}" is a COM+ application. Each COM+ application installed on the system can be configured using the system's Component Services manager.

In the menu bar of the Component Services Control application (see Figure 20), select Action and open Properties. Under the Advanced tab, select Enable 3G support (see Figure 21). This setting is commonly used to give applications additional memory, but the benefit here is that the application will no longer run in the dllhost.exe process, but rather in dllhst3g.exe. This enables the recorder to hook the Duwamish application process in dllhst3g.exe without any interference

from other applications, because dllhost3g.exe isn't used as a COM+ application server on typical W2k machines.

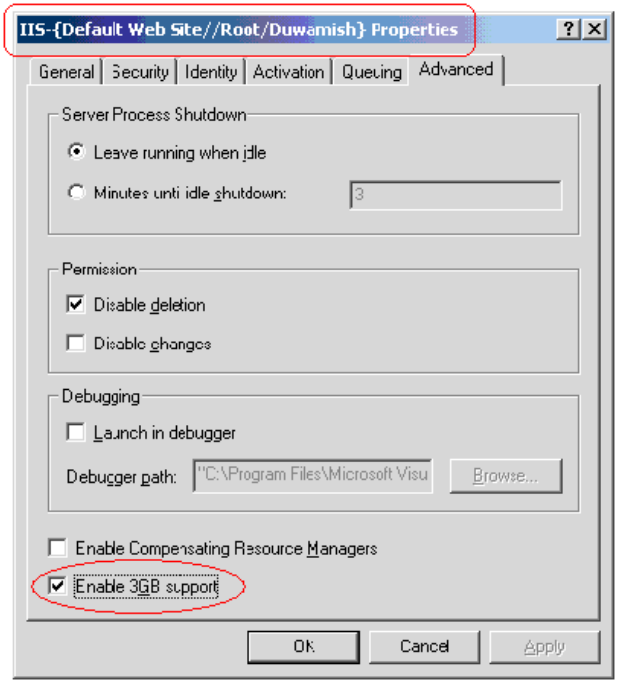


Figure 21 - COM+ properties for Duwamish ASP application

By enabling 3GB support, the ASP application of Duwamish "IIS-{Default Web Site//Root/Duwamish}" runs in a unique application server dllhst3g.exe.

Once the changes are applied, return to Silk Performer and create a new application profile to record the ASP application's calls to Duwamish's middle-tier. Give the application profile a name (e.g., "Duwamish ASP application") and specify the unique COM+ application server dllhst3g.exe as the client application (located in C:\WINNT\System32).

Since the middle-tier provides its services as functions of COM interfaces, change the application type to **Custom Application** and select **COM** as API (see Figure 22). This instructs Silk Performer's recorder to monitor each COM

call issued by Duwamish's ASP application. However, in this configuration, the recorder still probably can't generate test scripts.

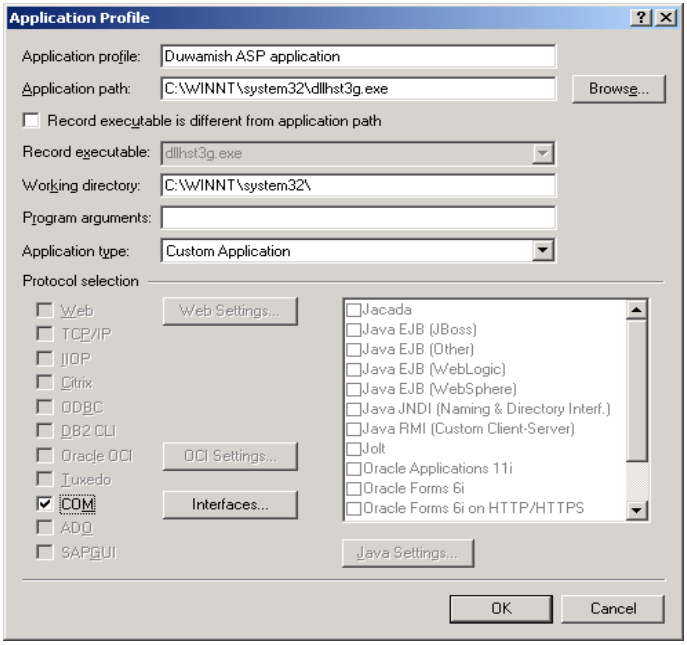


Figure 22 - Creating a profile for a COM application

Silk Performer's recorder needs to capture all of the Duwamish application's COM traffic.

To generate test scripts, the recorder needs descriptions of the COM interfaces (types) in the form of COM type libraries. Specifically the recorder needs to be provided with the type library that describes the relevant COM interfaces. For the requirements of this load test, the interfaces of the middle-tier's outermost COM component WFL (recall Figure 14) are most relevant. The corresponding type library is contained as a resource in the component container D5WFL.DLL. Click the **Interfaces** button to open the Server Interfaces dialog and load all interface descriptions from D5WFL.DLL (see Figure 23).

Note "D5" stands for Duwamish Phase 5. Duwamish online is the 5th most powerful version in a series of Duwamish implementations. So for example, WFL in Figure 12 actually means D5WFL. Likewise, the names of all the other components of Duwamish

online (BLL, DAL, QWFL, FWFL) begin with D5. In the case of Duwamish, all component containers (DLLs) include COM-interface descriptions (type libraries) as resources.

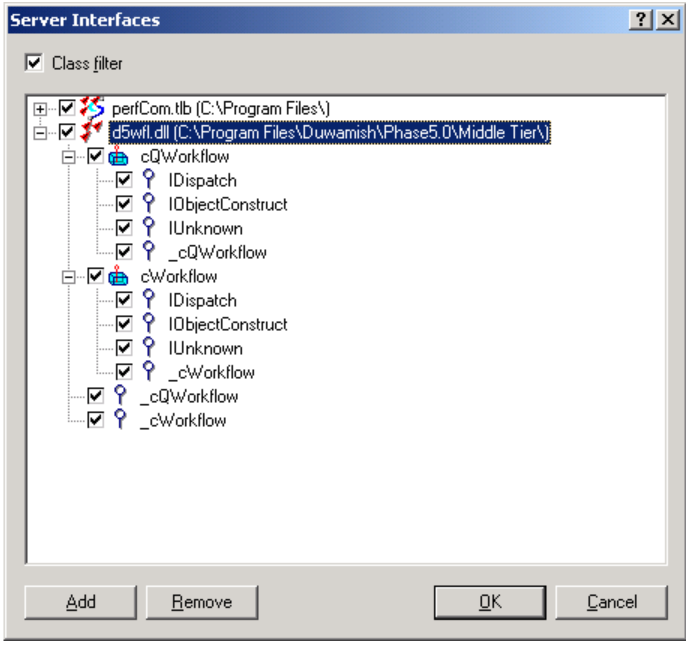


Figure 23 - Add the Duwamish workflow type library

Silk Performer's COM recorder needs the type library to correctly interpret the COM traffic. Silk Performer's replay engine also needs the type library, but only for type checking.

Recording the ASP Application

Now the COM traffic between the ASP application at the front- end and the workflow components in the middle-tier can be recorded. The recorder only needs to hook the right COM+ application server (dllhost3g.exe). If the application server at the front-end is still running, it must be stopped. Open the IIS service manager again, open the **Properties** for the Duwamish application, and click the **Unload** button in the bottom right of the **Properties** dialog. This shuts down the application server containing the ASP script engine. Following that, start up the COM recorder, open the Web browser and click on the Duwamish Web page. This launches the application server, but this time it's dllhost3g.exe rather than dllhost.exe. Because of the earlier-defined settings, the recorder automatically gets a hold of the application server and begins to generate a test script.

Now the usage scenarios are recorded as specified in section "Specification of Test Scenarios". Again, a separate script is created for each usage scenario. This ensures that virtually every call that relates to a certain transaction will be captured. Then follow the steps of a usage scenario (as specified in section "Specification of Test Scenarios"), generate and save the resulting script, shutdown the Duwamish ASP application, clear the recorder tabs and start up the ASP application again. With this procedure you get a separate script for each (specified) Duwamish use case. Finally, the different test scripts are merged into a single script with one transaction for each use case. As with the previous Web load test, the new load test script must be customized.
This time Customizing user data and Customizing user behavior are customized.

1. Customizing user data:
   When running HTTP load tests with Silk Performer, the customization process is relatively simple. HTTP is a simple protocol that allows data transfer by a fixed set of standardized functions. That contrasts with COM, which is a generic protocol that allows data transfer with a set of application specific functions. Session identifiers, for instance, can easily be detected and customized in HTTP test scripts, but not in COM test scripts, as they are modeled as part of application specific interfaces (as in Duwamish). In Duwamish, the middle-tier assigns a unique ID to each user during initial login. This ID also appears in recorded COM scripts. Therefore the script needs to be modified so that each virtual user uses the ID that was assigned by the middle-tier at runtime. Without this, the script won't replay because the hard coded ID will be invalid and the server will reject the requests. Additionally, modifications similar to those that were applied in the HTTP load test script need to be applied and the products that the virtual user buys must be randomized.

2. Customizing user behavior:
   As with the HTTP load test, the script needs to be modified to simulate the behavior of a real client. In this case, the client isn't a real user sitting in front of a Web browser, but rather an ASP script engine in a COM+ application server. That complicates things because the test scripts won't reflect the intuitive behavior of Web front-end users, but rather the artificial behavior of an ASP script engine. With hundreds of users connected to a server on which just a few concurrent ASP engines are running, each ASP engine handles several user requests in parallel. This means that an ASP engine must first handle a part of user request A, then the same engine must handle a part of user request B. Consequently, each ASP script engine handles all request types. Therefore, different Silk Performer user groups don't need to be created to simulate the ASP script engines. In this case it's sufficient to establish one user group in which all

the different transactions are performed. For each transaction type, simply specify the different execution counters according to the distribution specified in section "Specification of Test Scenarios".

Running the COM Test

Despite having recorded and customized the test script, preparation for the load test is not yet complete. The number of IIS worker threads that the COM replay engine should simulate needs to be defined because these worker threads can be seen as the true clients of the middle-tier. However the exact number of IIS worker threads isn't known. Therefore, a test with an increasing workload must be run to determine the number of threads in the COM replay engine that, during a test run, would produce the specified load in the same manner that IIS 5.0 would during normal operation of Duwamish.

The increasing load test begins with one virtual user (a single thread in IIS) and steadily increases the number of virtual users (the number of worker threads in IIS) to a maximum of 25. Following this, Performance Explorer is set up to monitor the CPU usage of the middle-tier server so that the CPU usage data can be correlated with data regarding transaction performance. Figure 24 shows that the CPU of the middle-tier is loaded to about 70% with 5 virtual users. This indicates that 5 users will create an adequate amount of load for our second test run.

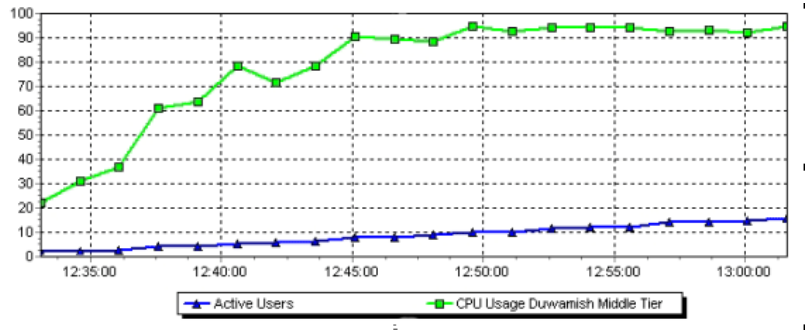


Figure 24 - CPU usage of the middle-tier under increasing load.

For the next test, the workload model is changed from increasing to queuing, the maximum amount of concurrent users is set to 5, a total simulation time of 3600sec is specified, and the transactions are run according to the specified distribution plan. With this model, the test run provides an almost realistic workload for the middle-tier during an hour of operation. This test helps determine if the middle-tier can handle the required number of transactions and - more significant to this test - reveals which middle-tier task(s) use the CPU the most.

Note To run all virtual users in a single replay engine (like the real client, the presentation layer of Duwamish, where all worker threads run in IIS 5.0 or the application server hosting the ASP script engine, see Figure 14), the Silk Performer system setting "Virtual users per process" needs to be set to "5."

Analyzing the Results

Again, the goal here is to pinpoint the most time-consuming middle-tier tasks (COM functions) during a typical usage scenario. These tasks will be candidates for optimization in the next development cycle of Duwamish. So, the most important thing is to compare the response times of the different COM function calls, the number of total calls for the individual functions, and the total call time for all functions of each specific type. The Performance Explorer provides the necessary data (see Table 6 - 1).

COM Function Name	Number of Calls	Response Time in sec			
		Sum	Min	Avg	Max
GetCategories (Late Binding)	9.600	4493,535	0,020	0,0468	17,716
GetDetails (Late Binding)	13.050	2851,269	0,040	0,218	19,748
ProcessOrder (Late Binding)	300	2735,608	0,090	9,119	21,010
UpdateSession (Late Binding)	4.648	595,433	0,020	0,128	3,024
GetSession (Late Binding)	4.650	514,837	0,020	0,111	17,595
GetSearchResults (Late Binding)	3.150	176,041	0,030	0,056	0,491
CreateNewAccount (Late Binding)	750	154,296	0,040	0,206	1,633
InsertSession (Late Binding)	1.050	141,513	0,020	0,135	1,082
LogOnAccount (Late Binding)	300	38,612	0,020	0,129	0,551
GetShippingAddresses (Late Binding)	300	32,293	0,020	0,108	0,511
GetCreditCards (Late Binding)	300	31,584	0,020	0,105	1,012

Table 6 - 1: COM Function Response Times

When running a COM load test, Silk Performer automatically gathers response times of COM function calls. These response times can be examined using Performance Explorer.

The "*GetCategories*" COM function is in first place in terms of total call time during the one hour load test. In second place, with a huge amount of total call time, is the "*GetDetails*" function. Both of these functions belong to the use case (task) "*Category search*." Table 6 - 1 reveals that there are COM function calls that, on average, take more time to be executed. The reason that "*GetCategories*" and "*GetDetails*" take the most overall time however is that they are executed much more frequently than are the others.

Applying Optimization

These test results point out that catalog browsing and retrieval operations create the bottleneck in this system. Each of these operations performs well - they slow down system performance simply because they are called often. So, one can reasonably conclude that an optimization of the browsing and retrieval operations would deliver a huge performance benefit to the system. This is why the developers of Duwamish inserted a cache in their application. The purpose of the cache is to store the XML strings that are returned by the catalog browsing actions. Future requests for the same category don't go to the middle-tier (see Figure 25) if the cache already contains the required data.

In the Duwamish example, as in most every other Web store, data in the catalog doesn't change that often - possibly once a week or less. Therefore, it's useful to cache even dynamically created data and only flush the cache when the product set changes. Figure 26 shows the physical architecture of Duwamish with the newly introduced cache component.

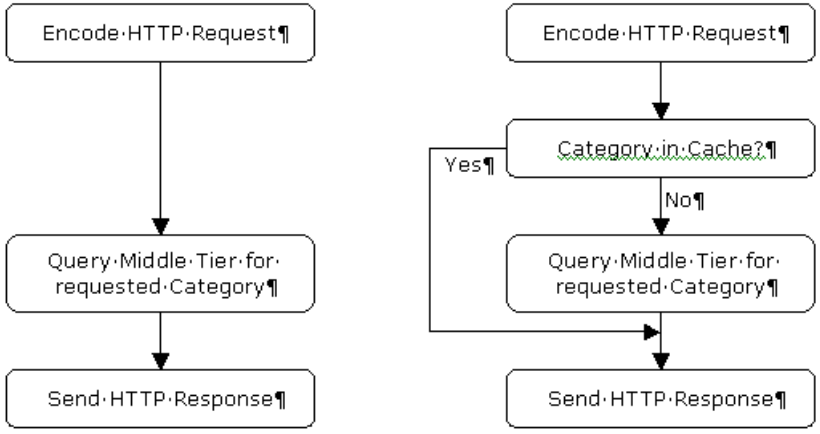


Figure 25 - Category search with/without cache

With the cache in place, the ASP application queries the middle-tier for a specific category or product only once after the product set is updated. The

presentation layer forgoes a roundtrip to the middle-tier when the cache contains the category or product.

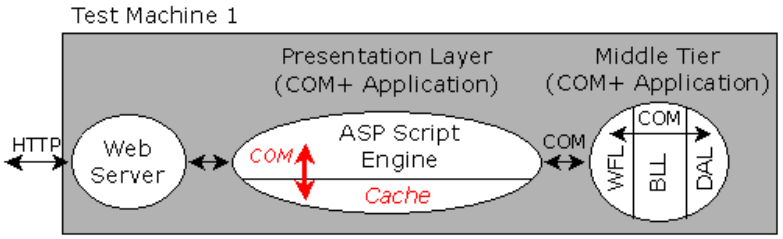


Figure 26 - Physical architecture of the ASP application with cache

The cache component is directly loaded into the process (instance of COM+ application server) of the ASP script engine. This configuration avoids out-of-process round trips.

Second Load Test

With the cache in place, a second system load test is run from Duwamish's front-end (at the HTTP level) because the impact of the cache on overall system performance needs to be checked. Then the efficiency of the optimized version of Duwamish can be compared with the original version. The same Silk Performer setup that was used for the first Web load test can be used for the remainder of this load test.

Performance Comparison

To detect and interpret deviation, test results should be compared side by side. After the test run, a new Performance Explorer workspace is created and the two TSD files of interest (HTTP load test with and without cache) are added. When creating a new chart, enable the **Overlay series** checkbox in the charts property dialog (right click in the chart area and, under the **Chart** tab, select **Properties**). This way the results of different test runs can be dragged and dropped into a single chart. Performance Explorer assumes that all graphs begin at the same time. As a result, the following chart, which relates transactions per second in both configurations with the number of virtual users, is presented (see Figure 27).

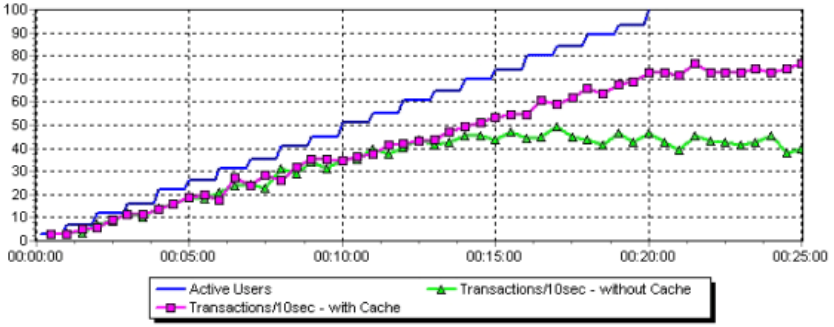


Figure 27 - Throughput comparison of the Duwamish application with and without cache

The amount of concurrent users increases by 5 each minute.

One can readily see that the system with the enabled cache component scales to a wider range than the system without the cache. In the chart without the cache, the maximum throughput is reached when about 50 to 60 concurrent users connect to Duwamish, whereas with the enabled cache, the throughput rises along with the number of users.

One more load test is run to determine the maximum number of users that Duwamish can handle with the cache in place. Again, a test with increasing load is set up, however this time with a higher number of virtual users in the last stage. Figure 28 shows the result of this load test. The maximum number of transactions per second is reached at about 150 concurrent users.

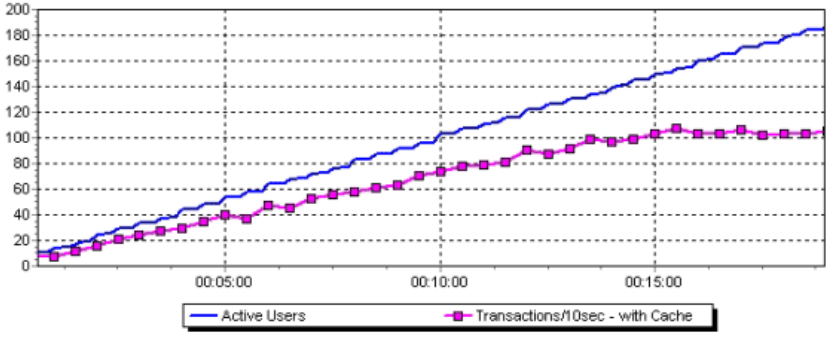


Figure 28 - Throughput with cache

Maximum throughput with the cache enabled is reached at about 150 concurrent users.

It can be concluded that the cache component considerably improves the system's performance. As soon as the different catalog queries are cached, the ASP engine no longer needs to go to the middle-tier. This puts less strain on the

middle-tier components and results in a considerable performance gain for all the middle-tier tasks.

Back-End Test

With SilkPerformer the ODBC interface is used to run load tests against the DBS. To do this the ODBC traffic between the DAL component and the SQL server must be recorded. This can be achieved with SilkPerformer's ODBC support by monitoring the ODBC API calls that the client (in this case the Duwamish DAL) issues. With the resulting script, a subsequent load test can be run against the DBS to identify ODBC statements that don't perform well with this particular test data and usage scenario.

1. Load test at the DAL interface:
   In this example the performance of the DBS is tested via the COM component DAL, DAL.DLL represents the server and BLL.DLL represents the client. BLL and DAL are both COM components running in the same process: in an instance of the COM+ application server. From the client's (BLL's) point of view, DAL is a typical COM in-process server. For a load test of the DBS at the data access layer (DAL), a COM load test must be set up with an instance of the COM+ application server as the client process (or dllhost3g.exe to uniquely identify the process the recorder should hook, see section "Preparing the Recording Session" for details). And Silk Performer must be parameterized with the server's type library (which is contained in DAL.DLL). In addition to this filter criteria, Silk Performer's recorder may be forced to focus only on calls of in-process servers - to keep the generated BDL script as small as possible (disable Recorder Settings/ COM/Filter/In-process server). With such a generated script, the load test can proceed as usual.
2. Load test at the ODBC interface:
   The DB server could just as easily be tested at the ODBC level. To do that the ODBC traffic between the DAL component and the SQL server would be recorded. This could be achieved with Silk Performer's ODBC support by monitoring the ODBC API calls that the client (in this case the Duwamish DAL) issues. With the resulting script, a subsequent load test could be run against the DBS to identify ODBC statements that don't perform well with this particular test data and usage scenario.

Testing Duwamish Online - A Case Study

This section recounts some of the authors' experiences in implementing the procedures outlined in this chapter:

"When we ran the first Web front-end test and increased the number of virtual users, we were confronted with a significant performance problem. What we experienced was a rapid increase of transaction response times, while at the same time the CPU usage on the server decreased considerably (see Figure 29). We were unsure of the cause of this problem: Had we set up the load test correctly? Did we have all the necessary information regarding Duwamish's configuration? Was there a bug in Duwamish?

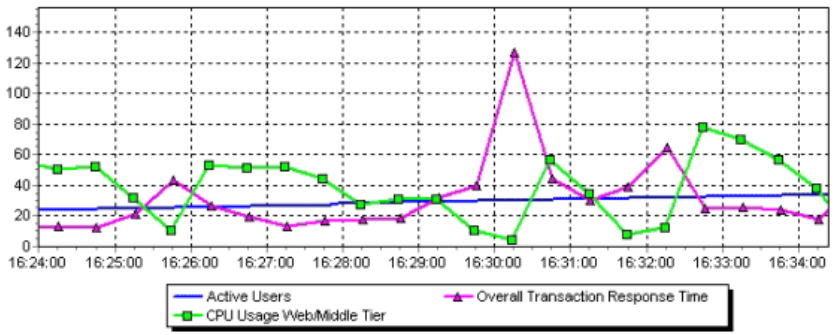


Figure 29 - Results of the first increasing load test

We experienced simultaneous increased response time and decreased CPU usage. Was there a locking problem in the system?

These are challenging times for testers, especially when they can't contact developers directly. We thoroughly checked our Duwamish configuration and found no obvious errors in the setup. We concluded that the problem must be in the Duwamish system or the Web server itself. As such odd behavior often indicates a locking problem on a server, we wondered if perhaps all worker threads were waiting for the same resources. Indeed, after researching the problem further, we found an article about Contention and Scalability in Duwamish that discusses the same problem we'd encountered. The authors invested a huge amount of time in debugging the Duwamish middle-tier and discovered that the ASP engine in IIS 5.0 does in fact have a locking problem. The ASP function Request.QueryString() (among others) waits for resources it doesn't need and causes performance bottlenecks. This bug has since been reported to the MS development team but, to our knowledge, no fix is yet available.

What could we do to bring the test to a positive conclusion? With regard to our testing goal, a positive conclusion would be one that clearly shows the

advantages of the optimized version of Duwamish over the original version. However because of the locking problem, we couldn't test Duwamish at all!

Intensive testing showed that the problem only presents itself when more than a specific number (about 5 to 7) of virtual users performs the order transaction simultaneously. The problem is that the URL query string becomes quite long when it proceeds through the order transaction. To avoid the locking problem in which a number of virtual users perform identical order transactions simultaneously, we introduced 5 different user groups in which each group performs only a single specific transaction. In this way, we attained a distribution of different transactions that avoids the risk of having many users perform the same order transaction simultaneously - only 2 out of 100 virtual users order items (perform the shopping transaction). Thus we were able to perform the load tests as described.

Though with this slight modification of the user distribution we weren't able to perform an accurate load test. In real-world production conditions, it's possible that the Duwamish application could experience exactly the type of load we've avoided with such an unrealistic workload model. Therefore, one must keep in mind that these test results are not real world test results. Therefore, the correct conclusion of our tests is that the Duwamish system, deployed as described, cannot handle real world traffic because of a bug in the ASP script engine - an important error in the implementation of Duwamish. So, the bottom line is that our tests were successful in every respect, because the end goal of any test should be to find errors!"

Glossary

This section contains a short description of some of the abbreviations used in this document.

ASP	Active Server Pages
BLL	Business Logic Layer
DAL	Data Access Layer
FWFL	Fulfillment Workflow Layer
IIS	Internet Information Server
MSMQ	Microsoft Message Queue
QWFL	Queued Workflow Layer
WFL	Workflow Layer
ODBC	Open Database Connectivity

13

Load Testing Siebel 6 CRM Applications

Introduction

A Guide to Customizing Silk Performer BDL Scripts for Siebel 6 (Siebel 2000) Applications.

What you will learn

This chapter contains the following sections:

Overview

This chapter explains how to use Silk Performer for load testing Siebel 6 (also known as Siebel 2000, version 6) CRM applications. It covers the testing of Siebel 6 Dedicated Client installations for Oracle and DB2, using Silk Performer's Oracle OCI and IBM DB2/CLI support. It briefly explains the Siebel architecture and shows how to configure Silk Performer and Siebel 6 Dedicated Clients. This chapter also provides guidelines for customizing Siebel scripts to run successfully in multi-user load tests using Silk Performer's Siebel functions, and steps through the customization of a typical Siebel transaction. Ensuring database integrity with Silk Performer's Siebel functions is also

covered. Finally, this chapter provides best practices for customizing Siebel database scripts.

This chapter does not explain how to install/configure a Siebel CRM application, use a Siebel CRM application, or optimize the performance of a Siebel CRM application.

This chapter is intended for experienced Silk Performer users with knowledge of databases, SQL and Silk Performer's database support for Oracle, ODBC, and DB2/CLI. Knowledge of a Siebel CRM application is also preferable.

Architecture of Siebel 6 CRM Applications

This chapter contains an overview of the Siebel client and how it works. Figure 1 illustrates the architecture of a typical Siebel deployment. Following the figure is a summary of the components that make up the Siebel client and descriptions of each type of client.

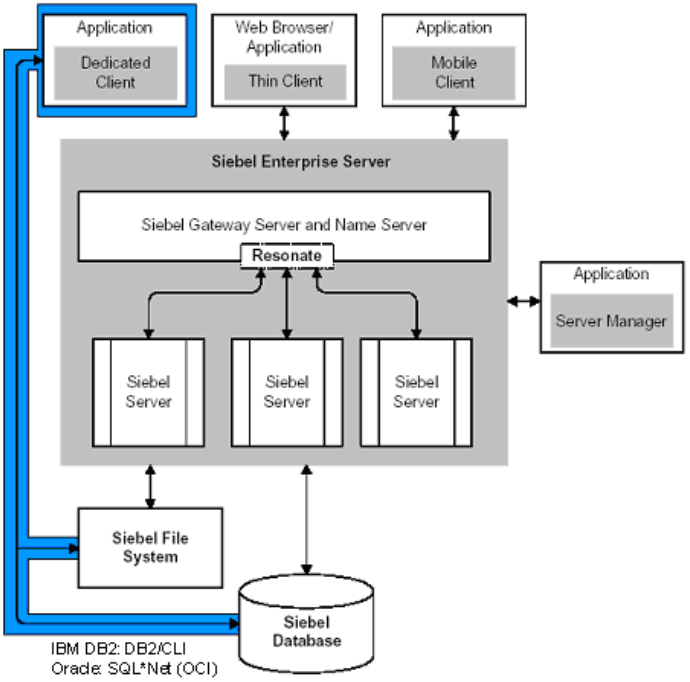


Figure 30 - Architecture of a typical Siebel 6 CRM application

Siebel Client Types

Siebel eBusiness Applications support three types of clients:

Siebel Dedicated Clients

The most common deployment of a Siebel 6 client is a Siebel Dedicated Client.

In Siebel Dedicated Clients, all layers of the Siebel eBusiness Application architecture, except for the database and the file system, reside on the user's personal computer. Figure 1 shows the architecture of a Siebel deployment and demonstrates at a high level how a dedicated client interacts with the Siebel database (area denoted in blue).

Siebel Dedicated Clients connect to the Siebel database using the protocol of the database server. For IBM DB2 database servers, DB2/CLI (DB2 Call Level Interface) is used. For Oracle database servers, SQL*Net/OCI (Oracle Call Interface) is used. For SQLServer database servers, ODBC (Open Database Connectivity) is used.

Silk Performer supports Siebel Dedicated Clients for Oracle, DB2, and SQLServer. Though this chapter concentrates on Siebel installations using DB2 and Oracle databases, it is relevant for Siebel installations using SQLServer databases.

Siebel Mobile Clients

In Siebel Mobile Clients, all layers of the Siebel eBusiness Application architecture reside on the user's computer, and a local database is stored on each mobile machine. Siebel Mobile Clients only connect to the Siebel Enterprise Server to synchronize the mobile databases with the central Siebel database. They use a proprietary TCP/IP based protocol to communicate with the Siebel Enterprise Server. The testing of Siebel Mobile Clients is not covered in this chapter.

Siebel Thin Clients

The Siebel thin client model differs from the default Siebel Dedicated Client model in several ways. Thin clients do not store data locally. Thin clients connect only to the Gateway Server and not to other servers. The Siebel Server executes all business logic for thin clients. There are two basic types of thin clients:

- Siebel Thin Client (Java and Windows ActiveX): Siebel Thin clients communicate with the Siebel Enterprise Server using a proprietary TCP/IP based protocol. Silk Performer's TCP/IP support can be used to load test Siebel Thin clients, however that process is not covered in this chapter.

- Siebel HTML Thin Client: Siebel HTML Thin Client is a technology used for deploying Siebel .COM Applications where Siebel HTML Thin Clients communicate with the Siebel Web Server using HTTP/HTML. The testing of Siebel HTML Thin Clients is supported by Silk Performer's Web support. For further information, refer to Silk Performer's Web support documentation.

Installation and Requirements

Siebel 6 Dedicated Client software must be installed to record and replay Siebel applications with Silk Performer. When using multiple Silk Performer agents, Siebel Dedicated Client software must be installed and running on all Silk Performer agent machines.

Configuring Siebel 6 Client Software / DB2

All Siebel client installations must use identical data sources for each agent. Confirm this using the ODBC Data Source Administrator (accessible from the Administrator Tools menu):

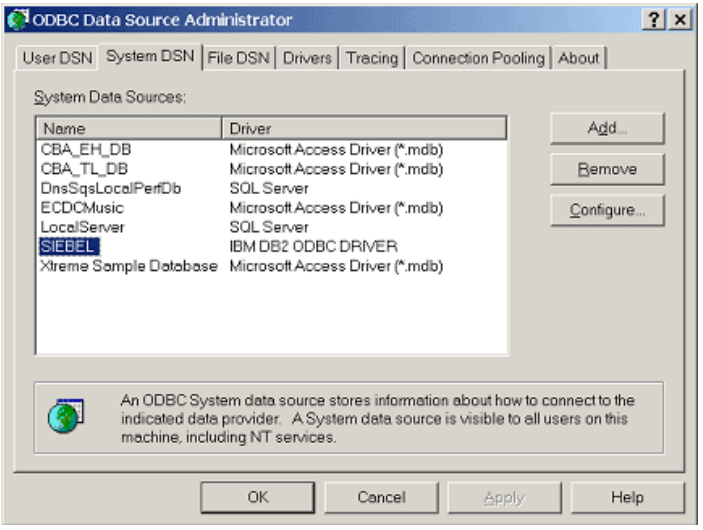


Figure 31 - Siebel Data Source Configuration for DB2

Configuring Siebel 6/DB2 for Recording

With the default installation of Siebel 6 (through version 6.3) Silk Performer is not able to record the Siebel client application (Siebel.exe). To enable recording,

the default database library used by the client must be changed from sscdd50.dll to sscddcli.dll:

- Ensure that the library sscddcli.dll is located in the Siebel installation path (e.g., C:\sea\bin\).
- Determine which configuration file (*.cfg file) is used by the Siebel client application that is to be recorded. To do this, examine the properties of the Start menu shortcut that launches the Siebel application. The entry will look similar to the following:

  Target: C:\sea\BIN\siebel.exe /c C:\sea\bin**scomm.cfg**
- Open the configuration file (in this case: C:\sea\bin\scomm.cfg) with a text editor. Go to the [Server] section and change the DLL entry from sscdd50.dll to sscddcli.dll. This replaces the old DB2 communication library, which uses embedded SQL, with the new DB2 communication library, which uses DB2/CLI. This change has no impact on the Siebel application's performance or functionality. The old library is used for backward compatibility with DB2 version 5.

Sample configuration file:

```
...
[Server]
Docked                    =TRUE
ConnectString             =SIEBEL
TableOwner                =SIEBEL
;;DLL                     =sscdd50.dll
DLL                       =sscddcli.dll
SqlStyle                  =DB2
...
```

Configuring Siebel 6 Client Software / Oracle

All Siebel client installations must use identical database aliases for each agent. Confirm this using the Net8 Easy Config tool, (accessible from the Oracle client software):

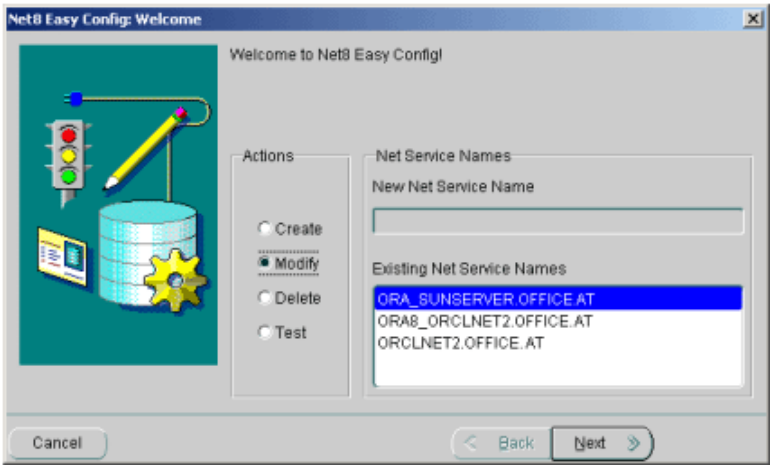


Figure 32 - Siebel Database Alias Configuration for Oracle

The installation of Siebel 6 (through version 6.3) requests the Database Alias and Table Owner in the Database Identification dialog. For that reason, Oracle client software should be installed before Siebel 6 and a Net Service Name should be configured to connect to the machine on which the Siebel database is running.

If Oracle client software is not installed before the installation of Siebel 6 Client software, or before an appropriate Database Alias is configured, installation of the Siebel 6 Client may continue, however the client will not be available for use. In such an instance, the Oracle client installation should be done subsequently.

Be sure that the Net Service Name and connection parameters of the Database Alias are identical on all machines used as Silk Performer Agents.

Configuring Siebel 6/Oracle for Recording

Depending on the different Siebel 6 client versions (through version 6.3.0.110), Silk Performer may not be able to record the Siebel client application (siebel.exe) with the default configuration. While recording Siebel 6.3.0.0 works fine with the default configuration, Siebel 6.3.0.110 requires some adjustments. The default database library used by the client must be changed from sscdo80.dll to sscdo73.dll:

- Ensure that the library sscdo73.dll is located in the Siebel installation path (e.g., C:\sea\client\bin\).
- Determine which configuration file (*.cfg file) is used by the Siebel client application that is to be recorded. To do this, examine the properties of the Start menu shortcut that launches the Siebel application. The entry will look similar to the following:

  Target: C:\sea\client\BIN\siebel.exe /c C:\sea\client\bin\scomm.cfg

  If there is no configuration file specified there (e.g. with Siebel client version 6.3.0.110), then find the .cfg file whose ApplicationTitle entry in the [Siebel] section shows the same string (e.g. Siebel eCommunications) as the window title of the Login dialog box to the Siebel application:

```
[Siebel]
RepositoryFile          = siebel.srf
ApplicationName          = Siebel Power Communications
ApplicationTitle      = Siebel eCommunications
ApplicationSplashText   = eCommunications
ComponentName            = Siebel Communications Client
```

Figure 33 - Login dialog box to the Siebel application

- Open the configuration file (in this case: C:\sea\bin\scomm.cfg) with a text editor. Go to the [Server] section and change the DLL entry from sscdo80.dll to sscdo73.dll. This replaces the default Oracle communication library with an updated version.

Sample configuration file:

```
[Server]
Docked                  =TRUE
ConnectString           =ORA_sunserver
TableOwner              =siebel
;;DLL                    =sscdo80.dll
DLL                     =sscdo73.dll
SqlStyle                =Oracle
...
```

Key Management with Siebel Applications

Why Database Keys Are Important for Script Customization

A recorded Silk Performer script from a database application such as Siebel 6, which inserts new data into a database, usually will not replay correctly without some modification. Replay may produce errors and undesirable results because scripts will attempt to insert exactly the same data during replay as during recording. Inserting the same data into a database more than once may cause primary key constraint violations in the database, resulting in replay errors.

Such errors can easily be identified because they cause tests to halt immediately after errors are detected. In such cases Siebel 6/DB2 generates the following type of error message:

> Odbc***(ODBC: 5 - ODBC error, 23505 (-803) : [IBM][CLI Driver][DB2/NT] SQL0803N One or more values in the INSERT statement, UPDATE statement, or foreign key update caused by a DELETE statement are not valid because the primary key, unique constraint…

For this reason, scripts that insert data must be customized before they can be replayed successfully. To customize database scripts, it is essential to understand the technique that is used by the application to ensure primary/foreign key consistency.

Siebel Keys (Rowids)

Siebel CRM applications use a sophisticated technique to ensure consistent, unique primary keys (primary keys are also referred to as "rowids" in Siebel documentation) for inserted data.

Format of Siebel Primary Keys (Rowids):

The format for primary keys in Siebel databases is the same for all primary keys. A primary key consists of a prefix and a suffix separated by a "-" character. The prefix is the same for the entire database; by default, a value of 1 is used for the prefix. The suffix is a string that represents a number with base 36. A base 36 number uses the characters "0" to "9" for base 36 digits from 0 to 9 and "A" to "Z" for base 36 digits from 10 to 35.

A base 36 value of:

A1Z

Correlates to a base 10 value of:

$$10 \times 36^2 + 1 \times 36^1 + 35 \times 36^0 = 12960 + 36 + 35 = \mathbf{13031}$$

Typical Siebel keys include:

- 1-AF0
- 1-19UZ
- 2-ZT5
- 1-10

Characteristics of Siebel Primary Keys:

All primary keys share the same SQL field name "ROW_ID". Siebel applications maintain available primary keys within a single table named "S_SSA_ID". In addition to being unique within each table, primary key values are also unique to each database.

Key Reservation Algorithm:

To create a new primary key value, a Siebel client reads the value of the next available primary key in the table "S_SSA_ID" (column "NEXT_SUFFIX") and immediately increments that value by a chosen value, for example 100- thereby reserving the next 100 values for use by the client.

For these 100 primary key values the Siebel client does not need to contact the database to retrieve valid primary keys, as they have already been reserved. Unused reserved key values are not returned, which creates "holes" in the key value sequence.

Example

- Siebel Client A:
  Login
- Siebel Client A:
  Reserves a range of rowids (selects table S_SSA_ID and updates table S_SSA_ID with the next available rowid): Reserved rowids:
  1-A00 to **1-A2S**, next available rowid: **1-A2T**
- Siebel Client A:
  Inserts row into table using the first reserved rowid value: **1-A00**
- Siebel Client A:
  Inserts second row into table using the second reserved rowid value: **1-A01**
- Siebel Client B:
  Login
- Siebel Client B:
  Reserves a range of rowids: **1-A2T** to **1-A3L**, next available rowid: **1-A3M**

- Siebel Client B:
  Inserts row into table using the first reserved rowid value: 1-A2T
- Siebel Client A:
  Inserts third row into table using the third reserved rowid value: **1-A02**
- Siebel Client B:
  Inserts second row into table using the second reserved rowid value: **1-A2U**
- Siebel Client A:
  Logout
- Siebel Client B:
  Logout

Formatting Scripts for Siebel Applications

Silk Performer Scripts for Siebel 6/DB2

A typical Silk Performer script for Siebel 6/DB2 appears below:

Global Variables

```
benchmark SilkPerformer Recorder

use "Kernel.bdh"
use "Odbc.bdh"

dcluser
  user
    VUser
  transactions
    TInit           : begin;
    TMain           : 1;

var
  c1      : cursor;
  ...
  c27     : cursor;
  c28     : cursor;
  ghDbc1  : number;
  ghEnv1  : number;
```

Siebel Dedicated Clients use a database connection (identified by the database handle ghDbc1 in the above example) and numerous database cursors (identified by c1 … cnn).

Database Login

```
transaction TMain
var
begin
  OdbcAlloc(SQL_HANDLE_ENV, ghEnv1);
  OdbcSetEnvAttr(ghEnv1, SQL_ATTR_ODBC_VERSION,
                 SQL_OV_ODBC3);
  OdbcAlloc(SQL_HANDLE_DBC, ghDbc1, ghEnv1);
  OdbcConnect(ghDbc1, "DSN=SIEBEL;UID=SEA_USER;
              PWD=mypwd;PATCH2=5,15;DBALIAS=SIEBEL;");
  OdbcSetConnectAttr(ghDbc1, SQL_ATTR_TXN_ISOLATION,
                     SQL_TXN_READ_UNCOMMITTED);
  OdbcSetConnectAttr(ghDbc1, SQL_ATTR_AUTOCOMMIT,
                     SQL_AUTOCOMMIT_ON);
  OdbcOpen(c1, ghDbc1);

  /***
  TMain_SQL001:
    SET CURRENT QUERY OPTIMIZATION = 3;
  ***/
  OdbcExecDirect(c1, TMain_SQL001);
  OdbcSetConnectAttr(ghDbc1, SQL_ATTR_TXN_ISOLATION,
                     SQL_TXN_READ_UNCOMMITTED);
```

The first operation between the Siebel Dedicated Client and the Siebel database is the Login (OdbcConnect/OraLogon). The second parameter of OdbcConnect specifies the data source connect string that is used to connect to the Siebel database. Siebel uses different UIDs (database User ID's) to connect different users to a database. UID and PWD attributes in the data source connect string can be changed to customize user logins.

SELECT Statements

```
/***
TMain_SQL002:
  SELECT T2.PR_POSTN_ID,T1.LAST_UPD,T4.NAME,T1.CREATED_BY,
         T1.EMP_ID,T2.LAST_NAME,T1.CONFLICT_ID,…
         T1.ROW_ID,T2.FST_NAME,T4.BU_ID,T3.OU_ID,T3.NAME,
         T4.BASE_CURCY_CD,T1.POSITION_ID,...
         T2.UPG_COMP_ID,T1.LAST_UPD_BY,T2.JOB_TITLE
  FROM SIEBEL.S_EMP_POSTN T1 INNER JOIN SIEBEL.S_EMPLOYEE
         T2 ON T1.EMP_ID = T2.ROW_ID INNER JOIN
         SIEBEL.S_POSTN T3 ON T1.POSITION_ID = T3.
         ROW_ID INNER JOIN SIEBEL.S_ORG_INT T4 ON T3.OU_ID =
         T4.ROW_ID LEFT OUTER JOIN SIEBEL.S_ORG_INT T5 ON T4.
         BU_ID = T5.ROW_ID LEFT OUTER JOIN
         SIEBEL.S_POSTN T6 ON T2.PR_POSTN_ID = T6.ROW_ID
  WHERE (T2.LOGIN = ?) FOR FETCH ONLY;
***/
```

```
    OdbcPrepare(c1, TMain_SQL002);
    OdbcDefine(c1, "1", SQL_C_CHAR, 16);
    ...
    OdbcDefine(c1, "21", SQL_C_CHAR, 76);
    OdbcBind(c1, ":1", SQL_C_CHAR, 51, SQL_CHAR, 50);
    OdbcSetString(c1, ":1", "SEA_USER");
    OdbcExecute(c1);
    OdbcFetch(c1, SQL_FETCH_ALL, 1, SQL_FETCH_NEXT); // 1 rows fetched

//    |1       |2              |3              |4  |5    |6          |7  |8              |9
// ---|--------|---------------|---------------|---|-----|-----------|---|---------------|--------
//   1|0VT-7T6D|2002-04-11 11:3|Siebel Administ|0-1|1-3BT|Siebel User|0  |2002-04-11 11:3|0VT-7T6D

    OdbcClose(c1);
```

SELECT statements are the SQL statements most frequently called by Siebel Dedicated Clients. For example, a Siebel transaction might involve connecting to the database, opening the Customer Accounts screen, and adding a new customer consisting of 29 SELECT statements, 4 INSERT statements, and 2 UPDATE statements.

SQL Result Sets

The Silk Performer recorder saves data that is returned from the execution of SQL queries during recording as BDL comments, in a tabular format within a BDL script. Such tables are useful for identifying correlations between the output data of SQL queries and input data of subsequent SQL commands.

```
OdbcFetch(c6, SQL_FETCH_ALL, 1, SQL_FETCH_NEXT); // 4 rows fetched

//    |1              |2  |3  |4              |5              |6     |7  |8              |9
// ---|---------------|---|---|---------------|---------------|------|---|---------------|---
//   1|1980-01-01 00:0|0-1|0  |Normal         |1980-01-01 00:0|0-4IX5|2  |Normal         |0-1
//   2|1980-01-01 00:0|0-1|0  |High           |1980-01-01 00:0|0-4IX4|3  |High           |0-1
//   3|1980-01-01 00:0|0-1|0  |Urgent         |1980-01-01 00:0|0-4IX3|6  |Urgent         |0-1
//   4|1980-01-01 00:0|0-1|0  |Urgent with Ale|1980-01-01 00:0|0-4O5O|1  |Urgent with Ale|0-1
```

Maximum column width and number of rows displayed in the result set can be adjusted using profile settings (Menu: Settings > Active Profile - Record - Script - Protocols - Database).

Reading Output Parameters

Values from result sets can be read using the *OdbcGet*/*OraGet* functions. The following statement reads the sixth column of the third row of the above result set, fetched by cursor c6 (0-4IX3):

```
sValue := OdbcGetString(c6, "6", 3);
```

Setting Input Parameters

Siebel uses parameter markers (input parameters) to parameterize SQL commands. This makes customization easy, as the data that need to be customized are parameters of the *OdbcSet*/*OraSet* functions scripted by Silk Performer. In the code above, the string "SEA_USER" is bound using the *OdbcSetString* function to the SQL field T2.LOGIN of SQL statement TMain_SQL002.

Tip To easily identify BDL statements that are relevant for customization, color all *OdbcSet*/*OraSet* functions in scripts by adding them to the custom keywords list (Menu: Settings > System… - Workbench - Layout - Custom Keywords).

Length of Siebel Scripts

Siebel BDL scripts can become quite large. For example, a Siebel BDL script that connects to a database, opens a Customer Accounts screen and inserts a new customer may have about 2600 lines of BDL code.

Silk Performer Scripts for Siebel 6/Oracle

Silk Performer scripts for Siebel 6/Oracle are similar to Siebel 6/DB2 scripts. The major difference between them is that instead of BDL ODBC functions, *) BDL ORA functions are used. The functions relevant for customization (*OdbcSet*/*OraSet*, *OdbcGet*/*OraGet*) have similar parameters. Therefore someone familiar with Siebel 6/DB2 can easily handle Siebel 6/Oracle.

*) DB2/CLI and ODBC share the same specification for a callable SQL interface. Therefore Silk Performer uses the same API functions for IBM DB2/CLI access as it does for ODBC access.

Recording and Customizing a Siebel Application

As the application logic used to generate consistent rowids is implemented in the Siebel client application, a recorded Siebel script containing the rowid values at the time of the recording is generated. Replaying such scripts may produce errors or lead to inconsistent data in a Siebel database. Silk Performer provides special Siebel functions that realistically mimic the behavior of a Siebel client application by implementing the Siebel key generation algorithm. Customizing a Siebel script means replacing hard coded rowid values with dynamic, runtime-generated values. The following section covers the process of customizing Siebel scripts.

Step 1: Configure Siebel 6/DB2 for Recording

See "Installation and Requirements" for more details.

Step 2: Create a Silk Performer Project

Depending on the database system used, create a Silk Performer project using the application type "Siebel 6/Oracle" or "Siebel 6/DB2". Choosing a Siebel 6 application type automatically configures the Silk Performer recorder for the Siebel 6 Dedicated Client application (Siebel.exe).

Step 3: Record the Siebel Application

Open the recorder and launch the Siebel application from the Start menu.

Tip Insert "New Timer Session" for each major step to be recorded (for example, choosing a menu, entering a new value, etc).

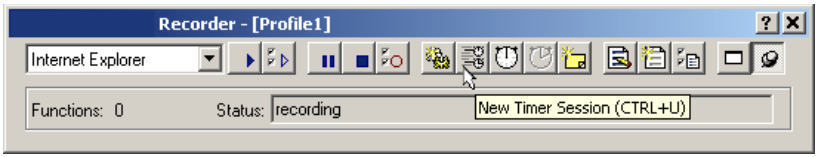


Figure 34 - Silk Performer Recorder

Using Timer sessions to comment on the recording process assists script customization by identifying, which sections of script belong to each recorded action. For each timer session (each *MeasureStart*/*MeasureStop* pair), two entries are made in the Active Script pane of Silk Performer's Workbench- making it easy to navigate to various sections of the script. Commenting recording sessions is more important for recording database applications such as Siebel 6 than it is for recording Web applications. As database scripts often become quite large, it is helpful to have a mechanism for correlating recorded actions with their respective script sections. Additionally, timings are defined for actions that may later be used in load tests.

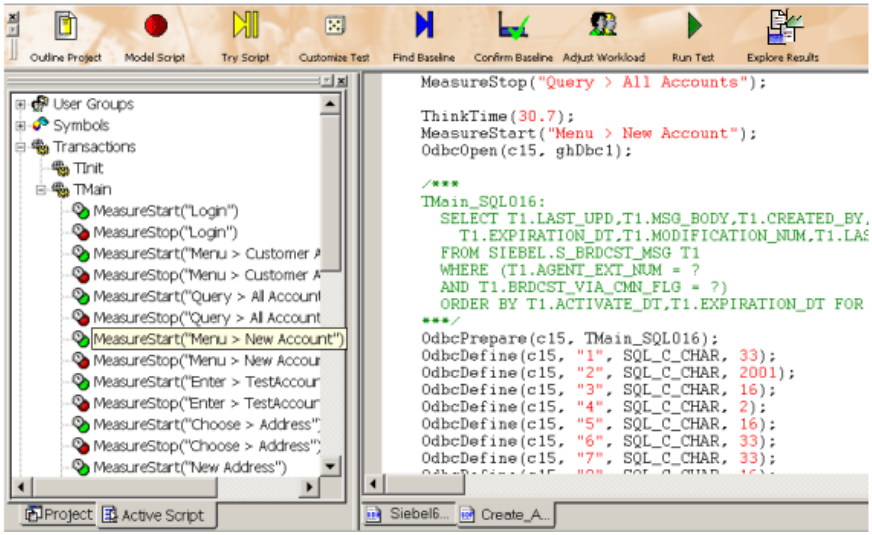


Figure 35 - Active Script pane (double click to navigate to scripts)

Tip Write down all data entered during recording sessions. During script customization this helps identify entered data in recorded scripts.

To ensure consistent scripts, always record full Siebel sessions (from login to logout/exit of the Siebel application).

Step 4: Try the Script

If data was not modified or entered during the recording session, the recorded script will replay without requiring modification. To make it realistic however, the script should be parameterized.

If new data was entered during the recording session, such as creating a new account or adding a new activity to an account, the script will produce errors as it twice attempts to insert the same data. In such an instance customize the key/rowid management of the script.

Step 5: Search for Siebel Rowid Reservations

Siebel rowids are maintained in the table "**S_SSA_ID**" of the Siebel database. Search for occurrences of "S_SSA_ID" to see if Siebel rowids are reserved in a script. Usually a SQL SELECT command is followed by a SQL UPDATE command for table S_SSA_ID within a script. Go to the first occurrence of "S_SSA_ID".

Rowid reservation code in a recorded Siebel 6/DB2 script

```
OdbcOpen(c19, ghDbc1);
MeasureStart("---Rowid reservation---");

/***
TMain_SQL020:
  SELECT T1.LAST_UPD,T1.CREATED_BY,T1.CONFLICT_
ID,T1.CREATED,
    T1.NEXT_SUFFIX,T1.ROW_ID,T1.NEXT_PREFIX,T1.CORPORATE_
PREFIX,
    T1.MODIFICATION_NUM,T1.NEXT_FILE_SUFFIX,T1.LAST_UPD_BY
  FROM SIEBEL.S_SSA_ID T1 FOR FETCH ONLY OPTIMIZE FOR 1 ROW;
***/
OdbcPrepare(c19, TMain_SQL020);
OdbcDefine(c19, "1", SQL_C_CHAR, 33);
OdbcDefine(c19, "2", SQL_C_CHAR, 16);
OdbcDefine(c19, "3", SQL_C_CHAR, 16);
OdbcDefine(c19, "4", SQL_C_CHAR, 33);
OdbcDefine(c19, "5", SQL_C_CHAR, 16);
OdbcDefine(c19, "6", SQL_C_CHAR, 16);
OdbcDefine(c19, "7", SQL_C_CHAR, 16);
OdbcDefine(c19, "8", SQL_C_CHAR, 16);
OdbcDefine(c19, "9", SQL_C_CHAR, 17);
OdbcDefine(c19, "10", SQL_C_CHAR, 16);
OdbcDefine(c19, "11", SQL_C_CHAR, 16);
OdbcExecute(c19);
OdbcFetch(c19, 1, 1, SQL_FETCH_NEXT); // 1 rows fetched
```

```
//   |1            |2  |3 |4            |5  |6   |7 |8 |9  |10|11
// --|-------------|---|--|-------------|---|----|--|--|---|--|-----
//  1|2002-04-12 13:2|0-1|0 |1980-01-01 00:0|4SL|0-11|2 |1 |174|9 |1-3BT

OdbcSetConnectAttr(ghDbc1, SQL_ATTR_AUTOCOMMIT, SQL_
AUTOCOMMIT_OFF);
OdbcOpen(c20, ghDbc1);
OdbcBind(c20, ":1", SQL_C_CHAR, 16, SQL_CHAR, 15);
OdbcBind(c20, ":2", SQL_C_CHAR, 16, SQL_CHAR, 15);
OdbcBind(c20, ":3", SQL_C_CHAR, 17, SQL_NUMERIC, 16, 0);
OdbcBind(c20, ":4", SQL_C_CHAR, 33, SQL_TIMESTAMP, 32, 0);
OdbcBind(c20, ":5", SQL_C_CHAR, 16, SQL_CHAR, 15);
OdbcBind(c20, ":6", SQL_C_CHAR, 17, SQL_NUMERIC, 16, 0);
OdbcSetString(c20, ":1", "1-3BT");
OdbcSetString(c20, ":2", "4VD");
OdbcSetString(c20, ":3", "175");
OdbcSetString(c20, ":4", "2002-04-12 13:32:56");
OdbcSetString(c20, ":5", "0-11");
OdbcSetString(c20, ":6", "174");
/***
TMain_SQL021:
  UPDATE SIEBEL.S_SSA_ID SET LAST_UPD_BY = ?,
    NEXT_SUFFIX = ?,MODIFICATION_NUM = ?,LAST_UPD = ?
  WHERE ROW_ID = ?
  AND MODIFICATION_NUM = ? ;
***/
OdbcExecDirect(c20, TMain_SQL021);
OdbcClose(c20, SQL_DROP);
OdbcCommit(ghDbc1);
OdbcSetConnectAttr(ghDbc1, SQL_ATTR_AUTOCOMMIT, SQL_
AUTOCOMMIT_ON);
MeasureStop("---Rowid reservation---");
```

Using the above code section as an example, embed the section beginning after the ***OdbcOpen*** call, before the SELECT statement through to the ***OdbcSetConnectAttr***(…, SQL_AUTOCOMMIT_ON) call after the UPDATE statement, within a *MeasureStart*/*MeasureStop*("---Rowid Reservation---"). This makes it easier to navigate to the rowid reservation section using the Active Script pane.

Step 6: Identify Rowid Values Used in the Script

The result set table of the "SELECT … FROM S_SSA_ID" statement holds the first rowid value (more precisely, the suffix of the first rowid value) that will be used in the application to insert new data. This value is found in the fifth column of the result set table (column: NEXT_SUFFIX). In this example, the value is 4SL. Now search for the next occurrence of this value in the script. The value will be in a script line such as the following:

```
OdbcSetString(c14, ":6", "1-4SL");
```

In this case the complete rowid (prefix + suffix) that will be used to insert data is **1-4SL**. To find all subsequent rowids that were used for inserting data, search for a phrase such as "**1-4S**" in the script. In this example, the following rowid values were found:

```
OdbcSetString(c21, ":1", "1-4SL");
OdbcSetString(c25, ":7", "1-4SN");
OdbcSetString(c25, ":10", "1-4SL");
OdbcSetString(c26, ":3", "1-4SL");
OdbcSetString(c27, ":4", "1-4SO");
OdbcSetString(c27, ":8", "1-4SL");
OdbcSetString(c27, ":4", "1-4SL");
OdbcSetString(c27, ":6", "1-4SM");
OdbcSetString(c27, ":4", "1-4SL");
OdbcSetString(c27, ":22", "1-4SL");
OdbcSetString(c27, ":37", "1-4SL");
OdbcSetString(c27, ":40", "1-4SL");
OdbcSetString(c27, ":2", "1-4SL");
OdbcSetString(c27, ":2", "1-4SL");
```

The following new rowid values were used in the script:

1-4SL, 1-4SM, 1-4SN, 1-4SO

These are the values that must be customized to produce a consistent script that replays without primary key constraint violation errors.

Step 7: Replace Static Rowid Reservation Code with Silk Performer's Siebel Functions

Silk Performer provides special Siebel functions that realistically mimic the behavior of a Siebel client application by implementing the Siebel key generation algorithm. By replacing the static code for the row reservation with Silk Performer's Siebel functions, dynamic rowid values are generated, replacing the static values found in the script.

Silk Performer's Siebel functions for IBM DB2 are implemented in the include file Siebel6_DB2.bdh. For Oracle, they are implemented in Siebel6_ORA.bdh. Include this file at the beginning of the script:

```
// CUSTOMIZED
use "Siebel6_DB2.bdh"
```

For each rowid that is to be customized, create a BDL string variable such as sRowid_4SL, sRowid_4SM, sRowid_4SN, sRowid_4SO:

```
// CUSTOMIZED
var
  sRowid_4SL : string;
  sRowid_4SM : string;
  sRowid_4SN : string;
```

```
    sRowid_4SO : string;
```

Replace the code that was embedded between MeasureStart("---Rowid Reservation---") and MeasureStop("---Rowid Reservation---") with the following code (in blue below):

```
OdbcOpen(c19, ghDbc1);

// CUSTOMIZED
MeasureStart("---Rowid reservation---");
SiebelAllocateRowId_DB2(ghDbc1);
sRowid_4SL := SiebelNextRowId_DB2();
sRowid_4SM := SiebelNextRowId_DB2();
sRowid_4SN := SiebelNextRowId_DB2();
sRowid_4SO := SiebelNextRowId_DB2();

/*** CUSTOMIZED: comment until OdbcSetConnectAttr(… SQL_
AUTOCOMMIT_ON);
/***
TMain_SQL020:
  SELECT T1.LAST_UPD,T1.CREATED_BY,T1.CONFLICT_
ID,T1.CREATED,
    T1.NEXT_SUFFIX,T1.ROW_ID,T1.NEXT_
PREFIX,T1.CORPORATE_PREFIX,
    T1.MODIFICATION_NUM,T1.NEXT_FILE_SUFFIX,T1.LAST_UPD_
BY
  FROM SIEBEL.S_SSA_ID T1 FOR FETCH ONLY OPTIMIZE FOR 1
ROW;
***/
...
/***
TMain_SQL021:
  UPDATE SIEBEL.S_SSA_ID SET LAST_UPD_BY = ?,
    NEXT_SUFFIX = ?,MODIFICATION_NUM = ?,LAST_UPD = ?
  WHERE ROW_ID = ?
  AND MODIFICATION_NUM = ? ;
***/
OdbcExecDirect(c20, TMain_SQL021);
OdbcClose(c20, SQL_DROP);
OdbcCommit(ghDbc1);
OdbcSetConnectAttr(ghDbc1, SQL_ATTR_AUTOCOMMIT, SQL_
AUTOCOMMIT_ON);

CUSTOMIZED ***/

MeasureStop("---Rowid reservation---");
```

SiebelAllocateRowId_DB2 (use ***SiebelAllocateRowId_ORA*** for Oracle) reserves rowids by selecting and updating the Siebel S_SSA_ID table in the

same way that the Siebel client application does. After calling ***SiebelAllocateRowId_DB2***, get new values for rowids by calling ***SiebelNextRowId_DB2*** (use SiebelNextRowId_ORA for Oracle).

Step 8: Replace Static Rowid Values with Dynamic Rowid Variables

The last step for customizing rowids is to replace all static occurrences of identified rowids with rowid variables.

Using the previous example:

- Search for "1-4SL" and replace all occurrences with "sRowiId_4SL"
- Search for "1-4SM" and replace all occurrences with "sRowiId_4SM"
- Search for "1-4SN" and replace all occurrences with "sRowiId_4SN"
- Search for "1-4SO" and replace all occurrences with "sRowiId_4SO"

New rowid values are usually only found in Bind functions, such as *OdbcSetString* for DB2 and *OraSetString* for Oracle.

The replacement looks like this:

```
OdbcSetString(c21, ":1", sRowid_4SL);
OdbcSetString(c25, ":7", sRowid_4SN);
OdbcSetString(c25, ":10", sRowid_4SL);
OdbcSetString(c26, ":3", sRowid_4SL);
OdbcSetString(c27, ":4", sRowid_4SO);
OdbcSetString(c27, ":8", sRowid_4SL);
OdbcSetString(c27, ":4", sRowid_4SL);
OdbcSetString(c27, ":6", sRowid_4SM);
OdbcSetString(c27, ":4", sRowid_4SL);
OdbcSetString(c27, ":22", sRowid_4SL);
OdbcSetString(c27, ":37", sRowid_4SL);
OdbcSetString(c27, ":40", sRowid_4SL);
OdbcSetString(c27, ":2", sRowid_4SL);
OdbcSetString(c27, ":2", sRowid_4SL);
```

Step 9: Replace Static Input Data with Customized Data

After customizing the rowid handling in the application, customize the user data. This step is rather easy as the data entered during recording can be found as parameters of *OdbcSetString* functions (*OraSetString* for Oracle). In the example, a new account with the name "TestAccount" was created. As a simple customization, replace the account name with a randomized account name using a random string pattern. Searching for "TestAccount" returns the following occurrences:

```
OdbcSetString(c27, ":25", "TestAccount");
OdbcSetString(c27, ":1", "TestAccount");
```

Create a variable to be used in the *OdbcSetString* functions instead of the string constant "TestAccount".

```
var
  sAccountName : string;

  ...
  OdbcSetString(c27, ":25", sAccountName);
  OdbcSetString(c27, ":1", sAccountName);
```

Generate a random variable using the random variable wizard (type RndStr) and assign the random variable rAccountName to the variable sAccountName at the beginning of the transaction.

```
dclrand
  rAccountName : RndStr(10..30);

  transaction InsertAccount
  begin
    sAccountName := rAccountName;
  ...
```

Step 10: Test the Script

Run a try-script test to check if the customized script works. Check if the Siebel functions work correctly by using the Virtual User output window of the controller, or the Output file of the virtual user (.wrt file). If the functions work correctly, messages such as those listed below will appear.

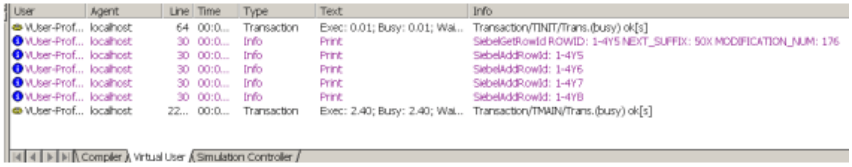


Figure 36 - Virtual User output for Siebel functions

References

SIEB01 Siebel eBusiness Applications, Siebel Client Installation and Administration Guide, Siebel 2000, Version 6.2, September 2000, Siebel Systems, Inc., 2207 Bridgepointe Parkway, San Mateo, CA 94404

TGUIDE01 IBM-DB2 Siebel e-Business Solution, The Technology Guide Series, www.techguide.com

IBM01 Siebel 2000 Database Implementation on OS/390 Using NT Siebel Servers, IBM Redbook, September 2000, IBM Corporation, International Technical

Support Organization, Dept. HYJ Mail Station P099, 2455 South Road
Poughkeepsie, NY 12601-5400

14

Load Testing Siebel 7 CRM Applications

What you will learn

This chapter contains the following sections:

Overview

This chapter explains how to use Silk Performer to load test Siebel 7 (also known as Siebel 2001, Version 7) CRM applications. It covers the testing of Siebel 7 thin client (Web client) installations that utilize front ends comprised of HTML and applets running within a Web browser (e.g., MS Internet Explorer).

This chapter explains how with Silk Performer's Web Recorder:

- There is no need for session info customization.
- There is no need to manually insert parsing functions for database keys.
- Scripts work correctly, even with transactions that insert new records into databases.
- Scripts are prepared for easy randomization.

Also covered are:

- Best practices for properly preparing recording sessions
- Basic architecture of Siebel 7 thin client installations

This chapter is intended for experienced users who are knowledgeable about load testing Web (HTTP/HTML based) applications with Silk Performer.

Siebel 7 CRM Application Architecture

This chapter contains an overview of the Siebel client. Figure 1 illustrates the architecture of a typical Siebel 7 deployment.

Overview

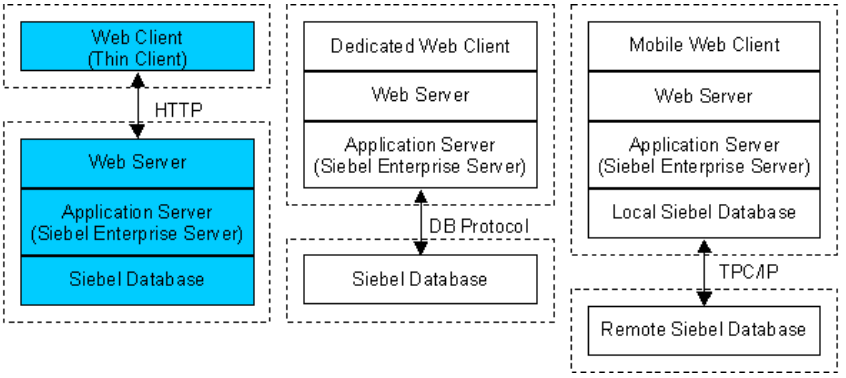


Figure 37 - Siebel 7 CRM Application Architecture

This chapter covers the portion of Figure 37 highlighted in blue - the Web Client (*Thin Client*). This is the default deployment for Siebel 7 installations.

The Web client is a HTTP-, HTML-, Java applet based application that doesn't require a client-side software installation.

The load testing of Siebel 7 thin client installations is achieved by recording and replaying *HTTP* traffic generated by browsers and Java applets.

Dedicated Web clients and mobile Web clients can also be tested using Silk Performer, though those topics aren't covered in this chapter.

HTTP Traffic

To understand recorded scripts, it's helpful to distinguish between two types of HTTP traffic:

- HTTP traffic generated by browsers
- HTTP traffic generated by Java applets

HTTP traffic generated by browsers is similar to HTTP traffic generated by typical Web applications (HTML, pictures, etc).

HTTP traffic generated by Java applets consists primarily of POST requests sent to servers. The bodies of HTTP requests have the MIME type "*application/x-www-form-urlencoded*", which is the same MIME type used for form submissions in HTML based applications. HTTP response bodies however are in a proprietary Siebel format.

The first example below shows the body of a typical Java applet HTTP request. And the other shows the corresponding BDL form definition that is generated by the Web Recorder for this HTTP request.

Example: HTTP Request Body from a Java Applet

```
SWERPC=1&SWECount=5&SWECmd=InvokeMethod&SWEMethod=GetPrefere
nce&SWEInputPropSet=%400%600%603%600%60GetPreference%603%60%
60pCategory%60SWE%3AListAppletLayout%60pPrefName%60%60SWEJSX
Info%60false%60
```

Example: BDL Form Definition for a HTTP Request Body from a Java Applet

```
SALES_START_SWE018 <ENCODE_CUSTOM> :
  "SWERPC"            := "1",
  "SWECount"          := "5",
  "SWECmd"            := "InvokeMethod",
  "SWEMethod"         := "GetPreference",
  "SWEInputPropSet"   := "@0`0`3`0`GetPreference`3``pCategory` SWE:"
                         "ListAppletLayout`pPrefName``SWEJSXInfo`false`";
```

Configuring Silk Performer

Installation

The Siebel 7 Web client doesn't require an explicit software installation on the client computer. The client runs within a Web browser (e.g., MS Internet Explorer).

Browsers may however ask for confirmation when Java applets are downloaded and installed. Such an example is shown in Figure 38.

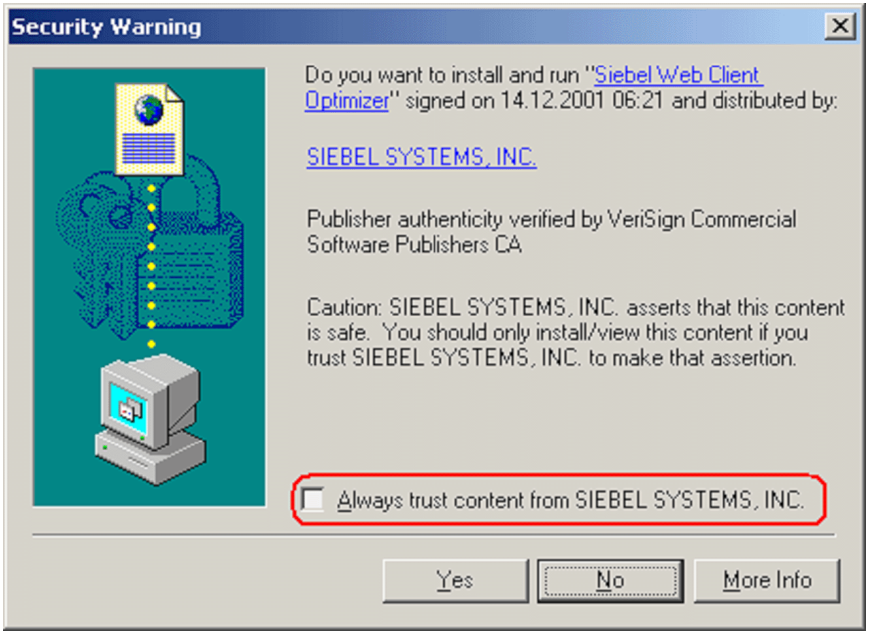


Figure 38 - MS Internet Explorer Security Warning

Check the "Always trust content from SIEBEL SYSTEMS, INC." checkbox and click **Yes**.

To avoid unnecessary security warnings in the future, add the name of the Siebel 7 Web server to the "Trusted Sites" zone of MS Internet Explorer's security settings.

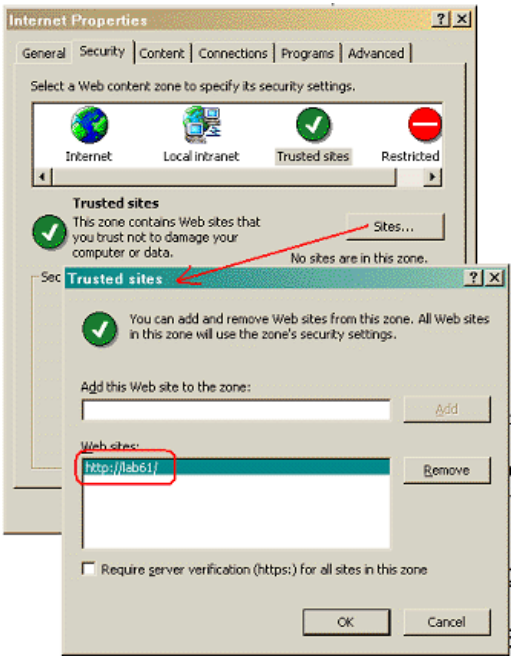


Figure 39 - Adding Siebel 7 Web Server to "Trusted Sites" Zone

Note Enter the complete address of the server, including "http://" or "https://".

Siebel Project Type

Create a Silk Performer project based on the "ERP/CRM, Siebel 7 Web Client" project type, as shown in Figure 40. Otherwise, Silk Performer's Siebel support

won't be activated, which will lead to HTTP/HTML scripts that require heavy customization before they can be executed successfully.

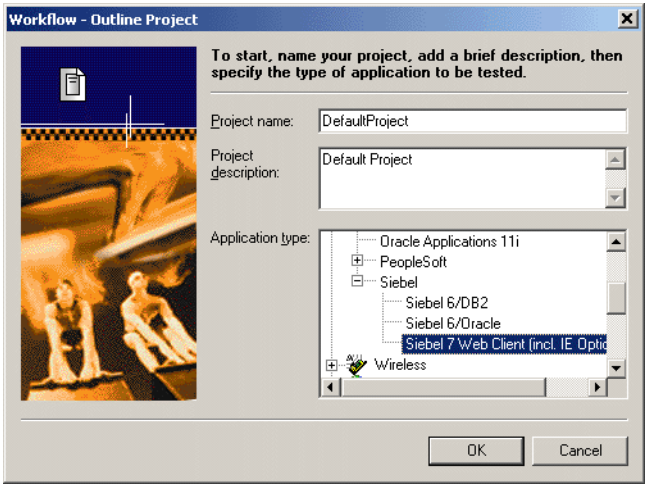


Figure 40 - Specify Project Type - Siebel 7 Web Client

Web Recorder

By creating a Silk Performer project, as illustrated in Figure 40, the Web Recorder is prepared to record a Siebel 7 Web client - no additional settings are required.

However, if the Siebel 7 installation uses load balancing through varying server names, enable the option "Dynamic URL parsing" in the "Advanced Context Management Settings" dialog, as shown in Figure 41.

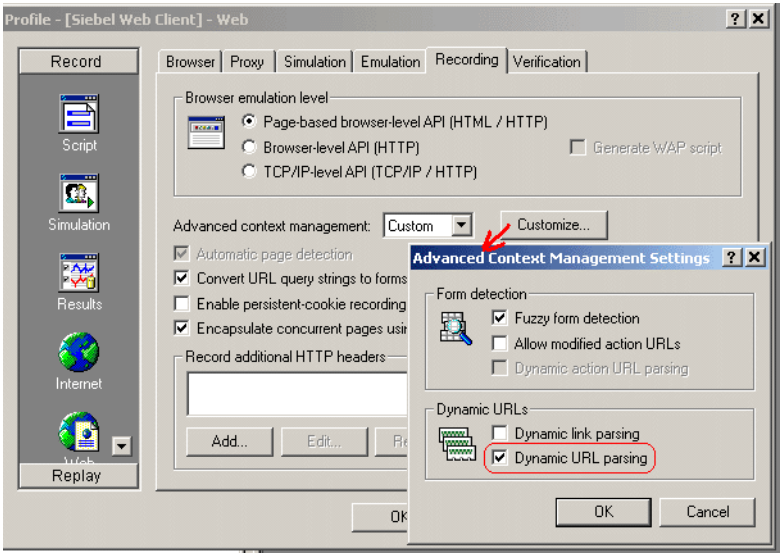


Figure 41 - Profile Settings for Siebel Server with Load Balancing

Scripts that use the host name "standardhost" in place of a real host name can be replayed against different servers by simply changing the *standardhost* profile

setting. The recorder can generate such scripts if the host name is entered in the profile settings dialog before recording takes place, as shown in Figure 42.

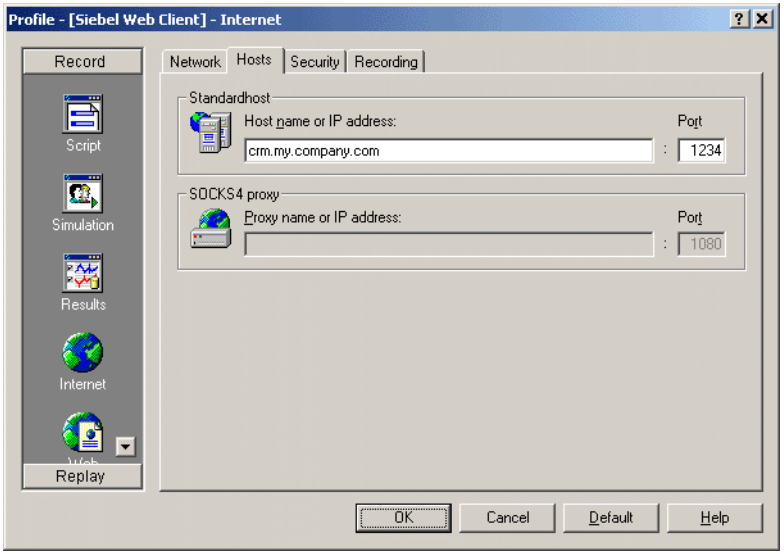


Figure 42 - Enter Standardhost Name and Port Before Recording

TrueLog Explorer

For best results in the "Source Differences" tab of TrueLog Explorer, add the following characters to both the "Compare tags" and "Compare tags HTML" lists within the "Settings / Options" dialog, as shown in Figure 43:

- Ampersand:&
- Asterisk:*
- Reverse single quote:`

- At:@

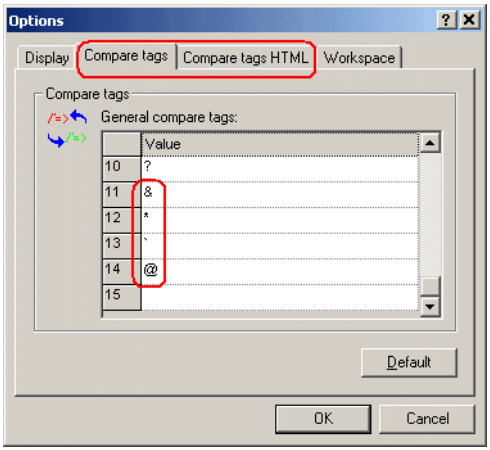


Figure 43 - Options Dialog Settings for TrueLog Explorer

Dynamic Information in Siebel 7 Web Client HTTP Traffic

HTTP traffic generated by Siebel 7 Web clients incorporates dynamic information that must be addressed to ensure successful and accurate replay.

Error Detection

Unfortunately, failure to handle such dynamic data doesn't always generate error messages during script replay. This is because the Siebel 7 Web Server doesn't use HTTP error status codes to indicate errors. Instead, it sends error notifications to the applet, which in turn displays an error message indicating the condition.

This may lead to a false impression of successful replay when in fact load generated on the database tier of the application was different - more than likely less - than it should have been.

The example below shows a HTTP response body sent from a server to a Java applet that describes an error condition. Such a response is returned with the HTTP status code "200 OK" - falsely suggesting that the HTTP request was handled successfully.

Example: Error Condition Response Sent from Server to Applet

```
@0`0`3`2``0`UC`1`SWECount`10`Status`Error`0`1`Errors`0`2`0`L
evel0`0`ErrMsg`An error happened during restoring the
context for requested
location`ErrCode`27631`0`0`Notifications`0`
```

Silk Performer scripts can detect such application level errors. The Web Recorder generates scripts that contain such checks by adding the lines highlighted in the example below:

Example: Application Level Error Detection in Recorded Scripts

```
use "WebAPI.bdh"
use "Siebel7Web.bdh"

// ...

  transaction TInit
  begin
    WebSetBrowser(WEB_BROWSER_MSIE55);
    WebFormDefineEncodeCustom("base=ENCODE_FORM;"
                              " +'@'; -'!()$,'; -''''");
    Siebel7WebInit();
  end TInit;
```

Session ID's

Siebel 7 Web Server uses session ID's to track user sessions.

Servers can be configured to send session ID's either in cookies or in URLs (form fields or query strings; this is the default configuration).

The example below shows a HTTP response header with a "*Set-Cookie*" header for a session ID. Session ID's in cookies are handled automatically by Silk Performer.

Example: Siebel Session ID in a Cookie

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 12 Mar 2002 17:26:01 GMT
Content-Language: en
Cache-Control: no-cache
Content-Type: text/html;charset=windows-1252
Content-Length: 3043
Set-Cookie: _sn=!1.428.556d.3c8e3a29; Version=1; Path=/sales
```

The example below shows a fragment of a HTML frameset with a frame that incorporates a session ID in a URL.

Example: Siebel Session ID in a URL

```
<frameset>
  <frame name="_sweclient">
```

```
  <frame name="_swe" src="http://lab61/sales/start.swe?
                          SWEFrame=top._swe&SWECount=1&
                          _sn=!1.6c0.447b.3ceccd1b
                          &SWECmd=GetCachedFrame">
</frameset>
```

Silk Performer's Web Recorder generates scripts that accurately handle such session ID's.

The next example shows fragments of a recorded script that handles the above session ID using the following techniques:

- A variable **gsSid** is declared to hold the session ID.
- A parsing function **WebParseDataBoundEx** is inserted before the function call **WebPageLink** for parsing of the session ID - which is included in the server response of the **WebPageLink** function (see example above).
- The parsed result can be used wherever it's required. In this example, it's required for a form submission.
- Informational comments enhance readability.

Example: Fragments of a Recorded Script with Session ID's

```
var
  gsSid : string; // !1.6c0.447b.3ceccd1b

// ...

WebParseDataBoundEx(gsSid, sizeof(gsSid), "&_sn=", 1,
                    "&", WEB_FLAG_IGNORE_WHITE_SPACE);
WebPageLink("replace", "Siebel Sales (#2)", 1, "_sweapp");

WebPageForm("http://lab61/sales/start.swe",
            SALES_START_SWE003, "Unnamed page (#2)");

// ...

dclform
  SALES_START_SWE003 <ENCODE_CUSTOM> :
    "SWERPC"     := "1",
    "SWECount"   := "1",
    "_sn"          := gsSid,// value: "!1.6c0.447b.3ceccd1b"
    "SWEJSXInfo" := "false",
    "SWECmd"     := "InvokeMethod",
    "SWEService" := "SWE Command Manager",
    "SWEMethod"  := "PrepareGlobalMenu";
```

Timestamps

HTTP communication between Java applets and the Siebel Web Server includes timestamps, which are strings that tell the number of milliseconds since Jan 1, 1970.

Proper replays must include correct timestamps. Silk Performer's kernel API (file "*kernel.bdh*") provides the function *GetTimeStamp*(), which is used to obtain accurate timestamp strings.

The Web Recorder recognizes timestamps and generates scripts that use them.

The first example below shows the body of a Java applet HTTP request that incorporates a timestamp.

The other example shows the corresponding BDL form generated by the Web Recorder. The value of the timestamp has been replaced by a call to the function GetTimeStamp().

Example: HTTP Request Body with Timestamp

```
SWEUserName=undisclosed&SWEPassword=undisclosed&SWEForm=SWEE
ntryForm&SWENeedContext=false&SWECount=0&SWECmd=ExecuteLogin
&SWETimeStamp=1023377037797
```

Example: Corresponding BDL Form with Timestamp

```
dclform
  SWEENTRYFORM002:
    "SWEUserName"     := "undisclosed", // changed
    "SWEPassword"     := "undisclosed", // changed
    "SWEForm"         := "SWEEntryForm", // added
    "SWENeedContext" := "false", // added
    "SWECount"        := "0", // added
    "SWECmd"          := "ExecuteLogin", // added
    "SWETimeStamp"         := GetTimeStamp(); // added, value: "1023377037797"
```

URL Encoding

According to HTTP specifications, unsafe characters included in transmitted data with the MIME type "*application/x-www-form-urlencoded*" must be encoded (escaped). This is achieved by replacing unsafe characters with hexadecimal equivalents and preceding them with percent (%) symbols.

Silk Performer provides four standard encoding types: *ENCODE_FORM*, *ENCODE_ESCAPE*, *ENCODE_BLANKS* and *ENCODE_NONE*. These encoding types differ in terms of the characters they escape.

Siebel 7 Web client Java applets apply a unique encoding type that differs from Silk Performer's standard encoding types.

For this reason, Silk Performer provides a new encoding type, *ENCODE_CUSTOM*, which can be configured with the function *WebFormDefineEncodeCustom*.

Siebel uses an encoding type that differs from the standard encoding type *ENCODE_FORM* in the following respects:

- Characters not escaped: **!()$,'**
- Characters escaped: **@**

Silk Performer's Web Recorder detects when a HTTP request applies this special encoding type, and then generates a script using the following techniques:

- Function *WebFormDefineEncodeCustom*() is used to define the encoding type *ENCODE_CUSTOM* in the *TInit* transaction.
- The encoding type *ENCODE_CUSTOM* is used in form definitions.

The next example shows fragments of a recorded script that utilizes these techniques.

Example: Fragments of a Recorded Script that utilizes ENCODE_CUSTOM

```
dcltrans
  transaction TInit
  begin
    WebSetBrowser(WEB_BROWSER_MSIE55);
    WebFormDefineEncodeCustom("base=ENCODE_FORM;"
                              " +'@'; -'!()$,'");
  end TInit;

// ...

dclform
  SALES_START_SWE011 <ENCODE_CUSTOM> :
    "SWERPC"                    := "1",
    "SWECount"                  := "4",
    "SWECmd"                    := "InvokeMethod",
    "SWEMethod"                 := "GetPreference",
    "SWEInputPropSet"           :=
              "@0`0`3`0`GetPreference`3``pCategory`"
              "Behavior`pPrefName``SWEJSXInfo`false`";
```

User Input

Generally, it's desirable if user input recorded during recording sessions can be easily identified and changed (e.g., randomized) in recorded scripts.

This is especially true for Siebel 7 Web client scripts, because Java applets often send values for all input fields to the server. If a new database record is created (e.g., for a new customer) and the record is subsequently edited or viewed, input (e.g., customer name) may appear multiple times in recorded scripts.

Silk Performer's Web Recorder:

- Detects user input by assuming that form field values are user input when one of the following is true:
  - Values are enclosed in $ symbols
  - Values are enclosed in underscores (_)
  - Values begin with the character sequence "i.".
- Generates a variable for each user input and uses that variable in a script, in place of the original value.

Example: Fragments of a Recorded Script that Includes User Input

```
var
  gsInputNewName     : string init "$NewName$";
  gsInputNewSite     : string init "_NewSite_";
  gsInputiHttp_x_com : string init "i.http://x.com";

// ...

dclform
  SALES_START_SWE020 <ENCODE_CUSTOM> :
    "SWEMethod"    := "GetQuickPickInfo",
    "SWEViewId"    := "",
    "SWEView"      := "Account List View",
    "SWEApplet"    := "Account List Applet",
    "SWEField"     := "s_1_2_46_0",
    "SWERow"       := "0",
    "SWEReqRowId"  := "1",
    "s_1_2_38_0"   := "N",
    "s_1_2_39_0"    := gsInputNewName, // value: "$NewName$"
    "s_1_2_40_0"    := gsInputNewSite, // value: "_NewSite_"
    "s_1_2_41_0"   := "(999) 999-9123",
    "s_1_2_37_0"   := gsInputiHttp_x_com,
                          // value: "i.http://x.com"
    "s_1_2_49_0"     := "",
    "s_1_2_46_0"     := "",
    "s_1_2_44_0"     := "",
    "SWEBCVals"      :=
         "@0`0`0`1``3``2`0`FieldValues`3``FieldArray"
         "`4*Name8*Location17*Main Phone Number"
         "`ValueArray`14*" + gsInputNewName +
         "14*" + gsInputNewSite + "10*9999999123`";
```

Choose input values during recording based on these descriptions so that the Web Recorder will detect them, as shown in Figure 44.

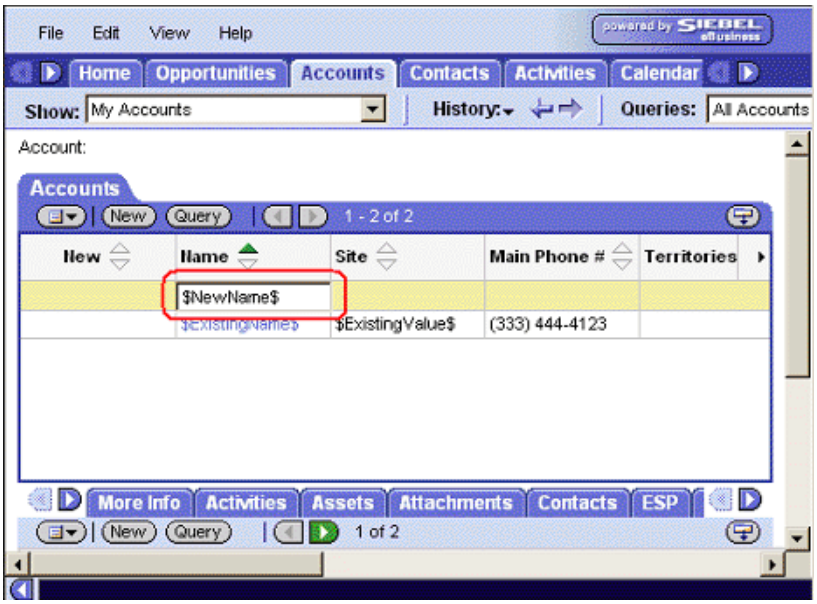


Figure 44 - Recording Siebel 7 - Input Value Enclosed in $ Symbols

Dynamic Data

Siebel HTTP traffic includes a variety of dynamic data that is sent to servers within HTTP requests. Such data can be parsed from previous server responses and substituted into scripts in place of hard coded values.

This includes:

- Existing database record values
- Row ID's

Siebel uses database keys to identify records in databases. Such keys are present in HTTP traffic emanating from both browsers and applets. Database keys are also known as "*Row ID's*".

Accurate handling of row ID's and other dynamic data is essential for successful replay.

Row ID's and other dynamic data can be included in HTML documents (see first example) or in responses to HTTP requests from Java applets (see other example).

Dynamic data are always contained in value arrays (lists of values for single rows to be displayed in the Siebel GUI).

Example: Value Array in JavaScript Code, within a HTML Document

```
<script>
  row = new top._swescript.JSSRecord();
  row.id = "1-9Q1";
  row.valueArray = ["Foo","Bar","1234567890","","Active",
                    "http://www.foo.com/bar","","","USD",
                    "11/26/2002","","USD","11/26/2002",
                    "","","","","N","N","","1-9Q1"];
  S_BC1.Rec(row);
</script>
```

Example: Value Array in an Applet Response

```
@0`0`3`2``0`UC`1`Status`Completed`SWEC`10`0`24`Notifications
`0`2`0``0`OP`bn`bc`S_
BC1`7`0``0`type`SWEIRowSelection`OP`g`br`0`cr`6`bc`S_
BC1`size`7`ArgsArray`20*Account Entry
Applet1*1`7`0``0`type`SWEIRowSelection`OP`g`br`0`cr`6`bc`S_
BC1`size`7`ArgsArray`19*Account List
Applet1*11*01*01*01*01*0`7`0``0`type`SWEIPrivFlds`OP`g`br
`0`cr`6`bc`S_BC1`size`7`ArgsArray`19*Account List
Applet11**BlankLine11*?11**BlankLine21*?9**HTML
URL1*?15**HTML
RecNavNxt1*?`7`0``0`type`SWEICanInvokeMethod`OP`g`br`0`cr`6`
bc`S_BC1`size`7`ArgsArray`19*Account List
Applet1*01*11*11*01*21*11*31*01*41*11*51*11*61*11*71*11*81*1
1*91*12*101*02*111*02*121*12*131*02*141*12*151*12*161*12*171
*12*181*12*191*12*201*12*211*12*221*12*231*12*241*12*251*02*
261*12*271*12*281*02*291*02*301*02*311*02*321*02*331*02*341*
12*351*12*361*1`7`1``0`OP`iw`index`7`br`0`cr`6`bc`S_
BC1`size`7`ar`0`1`0`FieldValues`0`ValueArray`3*Foo3*Bar10*12
345678900*6*Active22*http://www.foo.com/bar0*0*3*USD10*11/
26/20020*3*USD10*11/26/20020*0*0*0*1*N1*N0*5*1-
9N9`8`0``0`OP`dw`index`7`br`0`cr`6`bc`S_
BC1`nr`1`size`7`ar`0`8`0``0`value`0`OP`sc`br`0`cr`6`bc`S_
BC1`size`7`ar`0`state`n`8`0``0`value`1`OP`sc`br`0`cr`6`bc`S_
BC1`size`7`ar`0`state`n`8`0``0`value`7`OP`sc`br`0`cr`6`bc`S_
BC1`size`7`ar`0`state`cr`8`0``0`value`1`OP`sc`br`0`cr`6`bc`S
_
BC1`size`7`ar`0`state`nrk`8`0``0`value`13`OP`sc`br`0`cr`6`bc
`S_BC1`size`7`ar`0`state`nr`6`0``0`OP`nd`br`0`cr`6`bc`S_
BC1`size`7`ar`0`2`0``0`OP`en`bc`S_BC1`0`3`
```

Dynamic information typically appears in the "*dclform*" section of scripts.

When required, the Web Recorder automatically generates a parsing function (*WebParseDataBoundEx*) and substitutes parsed values wherever they appear in scripts. Parsing functions parse for complete value arrays in HTML and applet

responses. The Recorder uses an appropriate tokenizing function **SiebelTokenHtml** or **SiebelTokenApplet** to retrive individual tokens from parsed value arrays. See Silk Performer Online Help for a detailed description of these tokenizing functions.

The next example shows fragments of a recorded script that utilizes the following techniques:

- Variables are declared for the value array (**gsRowValArray** and **gsRowValArray_001**).
- Parsing functions **WebParseDataBoundEx** are inserted to parse the value arrays - which are included in the server response of the subsequent function (see examples above).
- The parsed result can be used wherever it's required. In this example, individual tokens of the parsed values occur in various locations in form definitions.
- Informational comments enhance readability.

Example: Script Fragment Utilizing Parsing Functions

```
var
  gsRowValArray     : string; // = ["Foo","Bar","1234567890","","Active",
                              //    "http://www.foo.com/bar","","","USD",
                              //    "11/26/2002","","USD","11/26/2002",
                              //    "","","","","N","N","","1-9Q1"];
  gsRowValArray_001 : string; // 3*Foo3*Bar10*12345678900*6*Active
                              // 22*http://www.foo.com/bar
                              // 0*0*3*USD10*11/26/20020*3*USD10*11/26/2002
                    // 0*0*0*0*1*N1*N0*5*1-9N9

// ...

  WebParseDataBoundEx(gsRowValArray, sizeof(gsRowValArray),
            "row.valueArray", 2, "S_", WEB_FLAG_IGNORE_WHITE_SPACE, 1);
// function call where parsing function is in effect

  WebParseDataBoundEx(gsRowValArray_001, sizeof(gsRowValArray_001),
                      "ValueArray`", 1, "`", WEB_FLAG_IGNORE_WHITE_SPACE, 1);
// function call where parsing function is in effect

// ...

dclform

// ...

SALES_ENU_START_SWE016 <ENCODE_CUSTOM> :
  "SWEMethod"  := "Drilldown",
  "SWEView"    := "Account List View",
  "SWEApplet"  := "Account List Applet",
  "SWEReqRowId":= "1",
   "s_1_2_40_0"      := SiebelTokenHtml(gsRowValArray, 0), // value: "Foo"
   "s_1_2_41_0"      := SiebelTokenHtml(gsRowValArray, 1), // value: "Bar"
  "s_1_2_42_0" := "(123) 456-7890",
  "s_1_2_51_0" := "",
   "s_1_2_47_0"      := SiebelTokenHtml(gsRowValArray, 4), // value: "Active"
  "s_1_2_45_0" := SiebelTokenHtml(gsRowValArray, 5),
                      //value:"http://www.foo.com/bar"
  "SWECmd"     := "InvokeMethod",
  "SWERowId"   := SiebelTokenHtml(gsRowValArray, 20), // value: "1-9Q1"
  "SWETS"      := GetTimeStamp(); // value: "1038305654969"
```

```
// ...

SALES_ENU_START_SWE022 <ENCODE_CUSTOM> :
  "SWEMethod"  := "Drilldown",
  "SWEView"    := "Account List View",
  "SWEApplet"  := "Account List Applet",
  "SWEReqRowId":= "1",
   "s_1_2_40_0"   := SiebelTokenApplet(gsRowValArray_001, 0), // value: "Foo"
   "s_1_2_41_0"   := SiebelTokenApplet(gsRowValArray_001, 1), // value: "Bar"
  "s_1_2_42_0" := "(123) 456-7890",
  "s_1_2_51_0" := "",
  "s_1_2_47_0" := SiebelTokenApplet(gsRowValArray_001, 4), // value: "Active"
  "s_1_2_45_0" := SiebelTokenApplet(gsRowValArray_001,5),
                                    // value: "http://www.foo.com/bar"
  "SWECmd"     := "InvokeMethod",
  "SWERowId"   := SiebelTokenApplet(gsRowValArray_001, 20), // value: "1-9N9"
  "SWETS"      := GetTimeStamp(); // value: "1038305711331"
```

Reformatting Functions

The Siebel 7 Web client reformats phone numbers and date/time values that are sent by the server or entered by users. Therefore the format of such values in client-sent HTTP request bodies is different from the format that is used in server-sent HTTP response bodies (see first example).

Example: Phone Number in Server-Response Body

```
10*987140255510*78140110000*0*0*22*main.contact@MyCompany.com5*1-FIH
```

Without modification, the format used in request bodies would also be used in recorded scripts (see second example), making it impossible to replace such values with parsed values.

Example: Phone Number in Recorded Script, Reformatted by the Siebel Web Client

```
"s_1_1_25_0" := "(987) 140-2555";
```

To allow such reformatted strings to be replaced with parsed values, Silk Performer's Web recorder records an appropriate reformatting function that mimics the Siebel Web client's reformatting and records actual values in the same format that is used in server responses.

Example: Reformatting Function for Phone Numbers

```
"s_1_1_25_0" := SiebelPhone("9871402555");
```

Unless a value is an input value, values can generally be replaced with parsed values in a second step. The Recorder actually records a parsed value instead of a hard-coded string.

Example: Using a Parsed Value as a Parameter of a Reformatting Function

```
"s_1_1_25_0" := SiebelPhone(SiebelTokenApplet(gsParsed, 5));
```

The following reformatting functions are available and are automatically recorded as required:

- SiebelPhone
- SiebelDate
- SiebelTime
- SiebelDateTime
- SiebelDecodeJsString
- SiebelParam

See *Silk Performer Online Help* for a detailed description of these functions.

Meaningful Timer Names

The Recorder extracts meaningful timer names from form fields. This makes it easier for human readers to interpret script, TrueLog and performance reports.

The next example includes examples of intuitive timer names:

Example: Meaningful Timer Names

```
WebPageForm("http://standardhost/sales_enu/start.swe",
            SALES_ENU_START_SWE022,
            "Account List Applet: InvokeMethod: Drilldown");

// ...

dclform

// ...

SALES_ENU_START_SWE022 <ENCODE_CUSTOM> :
  "SWEMethod"   := "Drilldown",
  "SWEView"     := "Account List View",
  "SWEApplet"   := "Account List Applet",
// more form fields
  "SWECmd"      := "InvokeMethod",
  "SWERowId"     := SiebelTokenApplet(gsRowValArray_001, 20),
  "SWERowIds"   := "",
  "SWEP"        := "",
  "SWETS"        := GetTimeStamp(); // value: "1038305711331"
```

Tips, Hints, & Best Practices

Section "Dynamic Information in Siebel 7 Web Client HTTP Traffic" of this chapter explained techniques employed by Silk Performer's Web Recorder to generate working scripts that don't require customization.

Due to peculiarities of the Siebel 7 Web client however, there are some pitfalls you may encounter.

This chapter explains the common pitfalls, offers hints, and explains best practices for successfully load testing Siebel 7 Web clients.

Read-Only Transactions

A read-only transaction is a transaction in which new records aren't inserted and data isn't altered.

Recorded scripts for read-only transactions run without modification, based on the techniques described in section "Dynamic Information in Siebel 7 Web Client HTTP Traffic".

Such scripts may however contain parsing functions for value arrays. Parsing functions may, for example, parse the value array of the 3rd record in a table. If during replay this table contains fewer than 3 records, no value array can be parsed and the result may be replay errors.

Such situations can be avoided by adjusting the occurrence parameter of parsing functions so that existing value arrays are parsed.

The better solution however would be to ensure that each table in a database contains adequate records during load tests - thereby avoiding the problem altogether.

Read/Write/Update Transactions

A writing transaction is a transaction that inserts new records or changes the values of existing records.

Required Script Customizations

Scripts for writing transactions require only minimal customization to ensure successful replay. This is because value arrays for new records can be parsed from server responses.

When modifications are required for successful replay, it's generally because many tables in Siebel 7 don't allow duplicate records. The circumstances by which Siebel 7 considers records to be duplicates vary between tables. In most

instances, a certain combination of record fields must be different for Siebel 7 to consider a record unique.

For successful replay, these fields must be changed, even with *Try Script* runs. This can be done easily when the Web Recorder creates these values as variables (see section "User Input").

For *Try Script* runs, it's sufficient to manually change these values directly in scripts. For load tests however, data files must be prepared and scripts must be customized in such a way that each virtual user inserts records with unique value combinations.

Beware of Empty Tables

HTTP traffic generated by the insertion of records into empty tables differs significantly from HTTP traffic generated by inserting records into tables that are already populated with one or more records.

For this reason, scripts that record the insertion of records into empty tables can't later be used during replay sessions once tables have been populated - and vice versa.

To avoid such problems, ensure that you don't insert records into empty tables during recording sessions. Also ensure that tables are not emptied before or during load tests.

Note that the following scenario is not *affected* by these constraints:

- Creating new records in tables that include other records (e.g., accounts).
- Creating new records in a table associated with the records just created (e.g., creating new notes for new accounts). While the notes table for newly created accounts is empty, the notes table of newly created accounts during script replay is also empty - so such scripts won't cause problems.

Summary

If you follow the recommendations outlined in this chapter, using Silk Performer to load test Siebel 7 Web client applications will be as easy as load testing standard Web applications.

15

Load Testing PeopleSoft 8

What you will learn

This chapter contains the following sections:

Overview

This chapter explains how to use Silk Performer to load test PeopleSoft 8 applications.

It explains how to use the PeopleSoft 8 SilkEssential package to record and customize PeopleSoft 8 scripts for realistic simulation of virtual users.

Topics include:

- Automatic context management
- Detection and handling of application level errors
- Optional features via manual script enhancement

Configuring Silk Performer

Specifying Project Type

Create a Silk Performer project based on the ERP/CRM/PeopleSoft/PeopleSoft 8 project type, as shown in Figure 45.

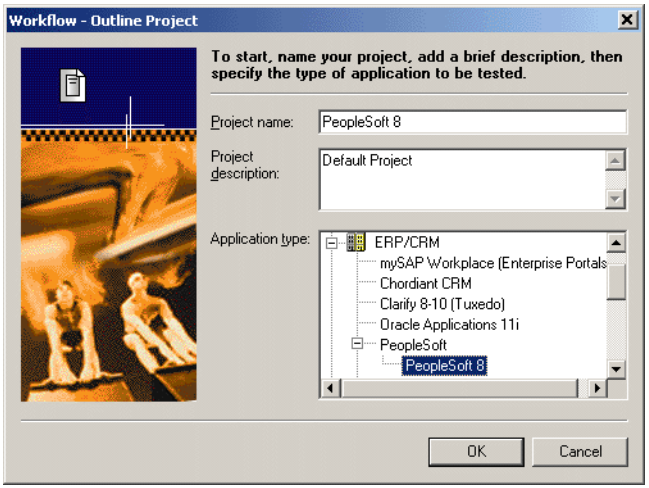


Figure 45 - Specifying project type - PeopleSoft 8

Script Modeling

Script modeling for PeopleSoft 8 transactions is straightforward. Recorded scripts work without any required modification, however by following a few standard customization steps after recording, you can get more out of recorded scripts. Recorded scripts are designed to make the process quick and easy.

Recording

Before recording, make sure that the project has been created based on the PeopleSoft 8 project type, as shown in Figure 45.

Record user interactions in a single BDL transaction. Ensure that the transaction begins with the sign-in process and ends with the sign-out process. Don't use your browser's Back button as this is generally problematic for highly state-dependent Web applications such as PeopleSoft.

Understanding Recorded Scripts

PeopleSoft scripts have a standard structure. Apart from the sign-in and sign-out processes, there are basically two types of user interaction involved:

- Navigation in the menu tree
- Work in the work area

Inclusion of PeopleSoft API Functions

The recorded script contains a use statement for *PeopleSoft8Api.bdh*, a BDH file that contains PeopleSoft specific API functions. It also contains *PeopleSoftInit()*, the initialization function call in the *TInit* transaction (see the example below).

This initializes the PeopleSoft framework contained in several BDH files of the PeopleSoft SilkEssential package.

It also enables global verification rules to catch PeopleSoft specific application-level errors. This is especially useful when used in conjunction with the Truelog On Error option.

In addition, it enables global parsing rules for the parsing of dynamic form names. Parsed form names are available in the global variable *gsFormMain*.

Example: Initializing PeopleSoft API Functions

```
benchmark SilkPerformerRecorder

use "WebAPI.bdh"
use "PeopleSoft8Api.bdh"

dcluser
  user
    VUser
  transactions
    TInit           : begin;
    TMain           : 1;

var
  // ...

dclrand

dcltrans
  transaction TInit
  begin
    WebSetBrowser(WEB_BROWSER_MSIE6);
    WebModifyHttpHeader("Accept-Language", "en-us");
    WebSetStandardHost("crm.ps.my.company.com");
    PeopleSoftInit();
```

```
    //GetLoginInfoPS("LoginPS.csv", gsUserId, gsPassword);
    //WebSetUserBehavior(WEB_USERBEHAVIOR_FIRST_TIME);
    //WebSetDocumentCache(true, WEB_CACHE_CHECK_SESSION);
  end TInit;
```

Wrapper Functions

The BDH files contain wrapper functions for all page-level API functions. These wrapper functions call the original API functions and perform some additional tasks.

The following wrapper functions are defined:

Wrapper function	Wrapped original function
WebPageUrlPS	WebPageUrl
WebPageLinkPS	WebPageLink
WebPageSubmitPS	WebPageSubmit
WebPageSubmitBinPS	WebPageSubmitBin
WebPageFormPS	WebPageForm
WebPageFileUploadPS	WebPageFileUpload

Sign In and Sign Out

Recorded scripts should begin with the sign-in process and end with the sign-out process, as shown in the example below.

The BDH files define, and the recorder records, the functions *HomepagePS*, *SignInPS* and *SignOutPS*..

Example: Recorded sign in and sign out processes

```
var
  gsHomepageUrl : string init "http://standardhost/psp/ps/
?cmd=login";

transaction TMain
begin
  HomepagePS(gsHomepageUrl, "PeopleSoft 8 Sign-in");
  ThinkTime(6.0);
  SignInPS("login", LOGIN002, "EMPLOYEE"); // Form 1

  // more function calls ...
  SignOutPS(gsSignoutUrl, "PeopleSoft 8 Sign-in");
end TMain;
```

These function calls don't require customization. The *SignInPS* function parses the sign-out URL to the *gsSignoutUrl* variable, and the recorded script uses the

variable in the *SignOutPS* function call. The recorder records the URL of the homepage into a variable for easy customization.

Navigation in the Menu Tree

Navigation in the menu tree is recorded by a *WebPageLinkPS* call that uses a custom hyperlink from a *WebPageParseUrl* call of a previous API call.

Example: Recorded navigation in the menu tree

```
WebPageParseUrl("JavaScript Link in page EMPLOYEE",
                  "DEFINITION\",\"", "\"",
                 WEB_FLAG_IGNORE_WHITE_SPACE);
WebPageLinkPS("Home", "EMPLOYEE"); // Link 1

WebPageLinkPS("JavaScript Link in page EMPLOYEE",
                "CR_PRODUCT_DEFINITION", 3);
```

This process works without any required modification.

Interaction in the Work Area

User interaction in the work area is recorded by a *WebPageSubmitPS* call. The form relies on the form field attribute *USE_HTML_VAL*, and thus ensures proper context management without customization.

The form name may change dynamically. While the form name is usually *main* or *win* (depending on the PeopleSoft version), it may, depending on server load, become *main1, main2, win1, win2*, etc.

Because of this, the recorder records a *WebPageSubmitPS(NULL, ...)* function call that references the form by ordinal number rather than form name.

However, the BDH files implement global parsing for the dynamic form name during script execution, so the actual form name of the current page is always available in the global variable *gsFormMain*.

Example: Recorded script fragment - User interaction within the work area

```
WebPageSubmitPS(NULL, MAIN005,
                "Product Definition", 4); // Form 4
```

The above example shows a *WebPageSubmitPS* that references the dynamic form by its ordinal number on the page.

The next example shows a typical submitted form. There is no need for customization of session or state management here because of the *USE_HTML_VAL* attributes.

The only customization that may be required is the randomization of input values.

Example: Submitted form

```
MAIN008:
  "ICType"                := "" <USE_HTML_VAL> ,
                  // hidden, unchanged, value: "Panel"
  "ICElementNum"          := "" <USE_HTML_VAL> ,
```

```
                // hidden, unchanged, value: "0"
"ICStateNum"            := "" <USE_HTML_VAL> ,
                // hidden, unchanged, value: "5"
"ICAction"              := "#ICSearch",
                // hidden, changed(!)
"ICXPos"                := "" <USE_HTML_VAL> ,
                // hidden, unchanged, value: "0"
"ICYPos"                := "" <USE_HTML_VAL> ,
                // hidden, unchanged, value: "0"
"ICFocus"               := "" <USE_HTML_VAL> ,
                // hidden, unchanged, value: ""
"ICChanged"             := "" <USE_HTML_VAL> ,
                // hidden, unchanged, value: "-1"
"RBF_PRD_DF_SRCH_SETID"     := "" <USE_HTML_VAL> ,
                // unchanged, value: ""
"RBF_PRD_DF_SRCH_PRODUCT_ID" := "" <USE_HTML_VAL> ;
                // unchanged, value: "NEXT"
```

Script Customization

The PeopleSoft SilkEssential package contains some utility functions that can be accessed via manual script enhancement to get more from recorded scripts. This chapter describes these optional steps.

Customizing Think Times

Silk Performer's Recorder records think times as they occur during recording. Often however this is not what is required.

The *PeopleSoftApi.bdh* file provides a substitute for the *ThinkTime* API function that's called *ThinkTimePS*.

```
function ThinkTimePS(fTime           : float optional;
                     bForceThinkTime : boolean optional)
```

To customize the script, use the *Replace* dialog (select *Replace* from the *Edit* menu) to replace all ThinkTime function calls with *ThinkTimePS* function calls.

The behaviour of the ThinkTimePS function depends on the value of the bForceThinkTime parameter.

When *bForceThinkTime* is set to *false* (the default), *ThinkTimePS* ignores the value of the *fTime* parameter and calls the *ThinkTime* API function with the value of the *ThinkTimePS* project attribute. The behaviour follows the usual thinktime-related profile settings (e.g., stresstest, random thinktime, exponential/uniform distribution, thinktime limited to x seconds, etc).

When *bForceThinkTime* is set to *true*, *ThinkTimePS* calls the *ThinkTime* API function with the value of the parameter *fTime* and the *OPT_THINKTIME_*

EXACT option. This results in a thinktime that is exactly as specified, regardless of profile settings and project attributes.

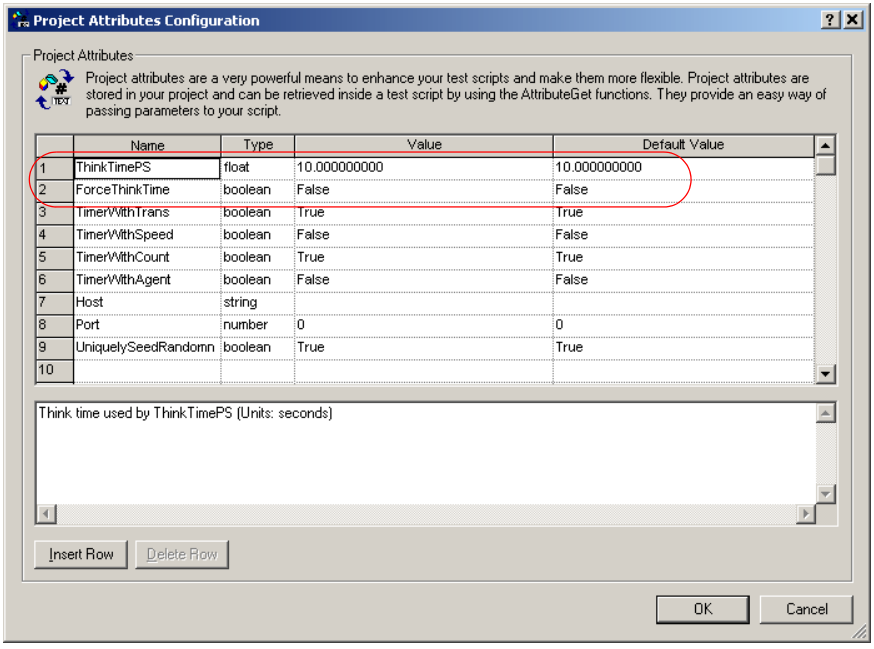


Figure 46 - Think time related project attributes

There are two project attributes available for customizing the behavior of the *ThinkTimePS* function. Select *Project Attributes* from the *Project* menu to access these attributes.

- ThinkTimePS (float):Think time value used by *ThinkTimePS* when *bForceThinkTime* is false. Unit: Seconds;
  Preset: 10.0
- ForceThinkTime (boolean):Allows for global override of the *bForceThinkTime* parameter of the *ThinkTimePS* function.
  Preset: false

Uniquely Seed Randomness

To improve the randomness of script execution, it can be advantageous to uniquely seed the random number generator for each virtual user.

This is done by changing the value of the *UniquelySeedRandomness* project attribute to true.

Customizing Timer Names

For easier results analysis, it's desirable to have timer names that can be sorted based on their occurrence in scripts and amended with additional information. To accomplish this, the wrapper functions amend orginal timer names with additional information.

Project attributes can be used to include any combination of the following items in amended timer names:

- Agent name
- Transaction name
- Modem emulation speed
- Page counter for the current transaction

Figure 47 shows the project attributes that can be used to specify desired items.

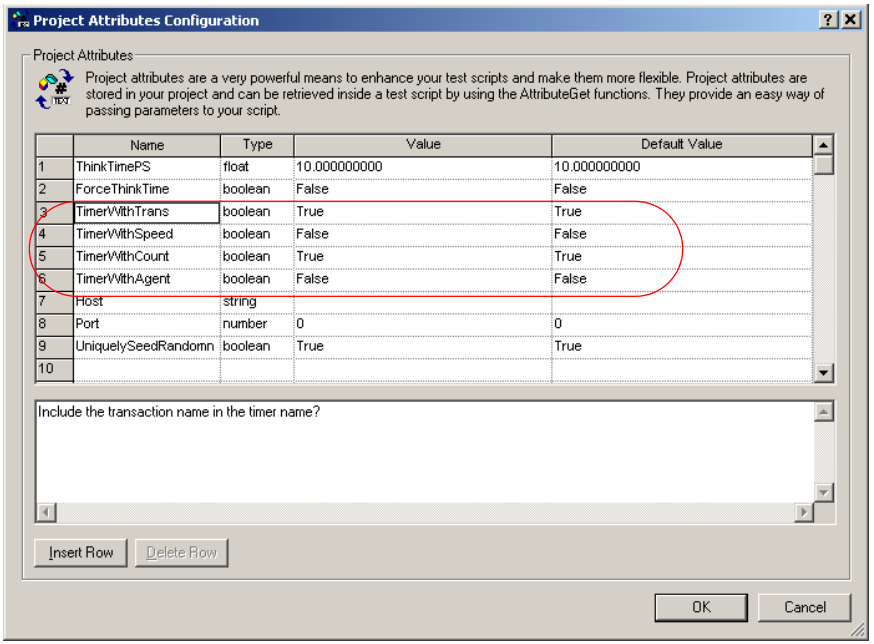


Figure 47 - Timer name related project attributes

The *WebPage*PS* wrapper functions keep track of internal page numbers, which are prepended to timer names when this option is selected in the project attributes.

While page numbers are automatically reset to '0' at the end of each transaction, you can manually reset the page counter at any time by calling the *ResetPageCount* function.

You may also call the *IncPageCount* function to manually increment the internal page counter. This is useful for keeping a consistent page counter after an unbalanced *if* statement.

The *WebPage*PS* wrapper functions also allow you to specify a specific value for the page counter. This value may be passed as the parameter where the frame name is passed in the original wrapped functions (Note: Wrapper functions don't allow you to pass a frame name; this rule is also obeyed by the recorder). You may pass the special value *PAGE_NUMBER_KEEP* to reuse the current page counter, rather than increment it.

Targeting a different server

Recorded scripts contain a call to *WebSetStandardHost* in the *TInit* transaction:

Example: Call to WebSetStandardHost as recorded

```
transaction TInit
begin
  // ...
  WebSetStandardHost("crm.ps.my.company.com");
  // ...
  PeopleSoftInit();
end TInit;
```

There are three options for targeting a different server:

1. Edit the recorded *WebSetStandardHost* function call.
2. Delete the recorded *WebSetStandardHost* function call and specify a standardhost in the profile settings.
3. Specify a different standardhost in project attributes, as shown in Figure 48. These project attributes will be evaluated in the *PeopleSoftInit()* function call and will therefore override the recorded *WebSetStandardHost* function call.

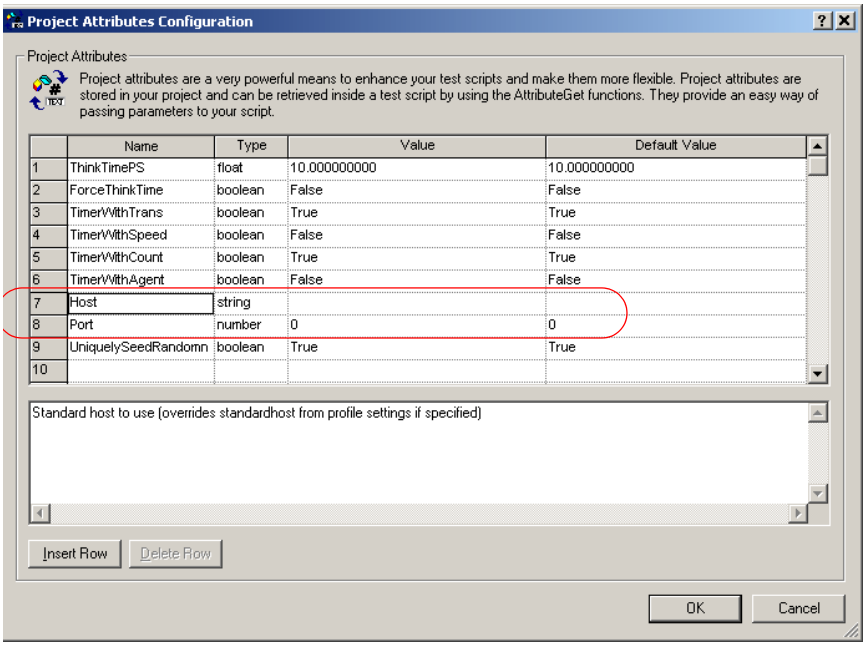


Figure 48 - Standardhost related project attributes

Enable server-side tracing

PeopleSoft 8 offers the option to enable detailed server-side tracing for individual login sessions. Scripts should be recorded without tracing being enabled, because this would enable tracing for all virtual users.

Instead, server-side tracing can be enabled on a virtual user basis by inserting a call to the *EnableTracingPS* function in the *TInit* transaction, as shown in the sample below. The form used for login must be passed to this function. It is important to note that tracing should only be enabled for individual virtual users. Enabling tracing for all virtual users would impose significant overhead on servers and could skew test results.

Enabling tracing for one virtual user

```
  transaction TInit
  begin
// ...
    PeopleSoftInit();
    if GetUserId() = 1 then
      EnableTracingPS(LOGIN001);
    end;
  end TInit;

dclform
  LOGIN001:
```

```
"httpPort"          := "" <USE_HTML_VAL> ,
"timezoneOffset"    := "-60",
"userid"            := gsUserId,
"pwd"               := gsPassword,
"Submit"            := "" <USE_HTML_VAL> ;
```

Tracing options can be modified by editing the *Tracing.csv* file, which is available in the *Data Files* node of the project tree view. See Figure 49.

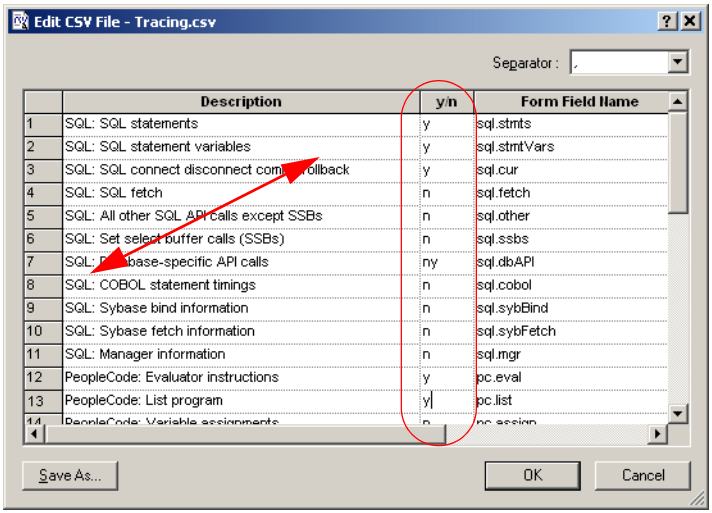


Figure 49 - Editing tracing options in Tracing.csv

Randomizing table row selection

PeopleSoft pages often contain tables (e.g., lists of search results). Clicking an item within such a table returns a form where the *ICAction* field is set to *#ICRowX*, where X denotes the ordinal number of the selected row.

Example: Form submitted while selecting an item in a table

```
dclform
  MAIN003:
    "ICType"        := "" <USE_HTML_VAL> ,
    "ICElementNum"  := "" <USE_HTML_VAL> ,
    "ICStateNum"    := "" <USE_HTML_VAL> ,
    "ICAction"      := "#ICRow9", // hidden, changed(!)
    "ICXPos"        := "" <USE_HTML_VAL> ,
    "ICYPos"        := "" <USE_HTML_VAL> ,
    "ICFocus"       := "" <USE_HTML_VAL> ,
    "ICChanged"     := "" <USE_HTML_VAL> ,
// ...
```

To accurately randomize or customize such forms you must determine the number of table rows.

There are four functions that assist with this:

- *GetMaxRowNr* returns the maximum valid row number (e.g., in a table with 47 rows the maximum valid row number is 46, since row numbers are zero-based). If the current page does not contain a table, it returns -1.
- *GetRowCount* returns the number of rows on the current page. If the current page does not contain a table, it returns 0.
- *GetRndRowStr* returns a valid, random row string.
- *FindICRow*(*sStringToFind* : *string*) : *string* returns the row string of the first row that contains the given text.

The sample below shows an example that uses the *GetRndRowStr* function to randomize the selection of an item in a table:

Example: Randomizing the selection of an item in a table

```
dclform
  MAIN003:
// ..
    "ICStateNum"    := "" <USE_HTML_VAL> ,
    "ICAction"    := GetRndRowStr(),
    "ICXPos"        := "" <USE_HTML_VAL> ,
// ...
```

Application Level Errors

General

PeopleSoft doesn't use HTTP response status codes to indicate application-level errors. Instead, it returns HTML with status code *200 Success*, even when the HTML contains error messages.

There are two error message types:

- Error messages embedded in HTML
- Error messages in the parameters of JavaScript functions (*Alert*) that display dialog boxes.

Additionally, not all messages that are displayed with an *Alert* function are error messages. Some messages are simply informational (e.g., R*ecord has been saved*).

It doesn't make sense to continue transactions after severe errors occur.

Benefits

Using the PeopleSoft 8 SilkEssential package, recorded scripts are automatically able to handle application level errors.

When severe errors occur, errors of *SEVERITY_TRANS_EXIT* severity are raised, and virtual users gracefully terminate their transactions by signing out.

It is strongly recommended that the *Truelog On Error* option be enabled during load tests to fully benefit from this feature.

Customizing Error Messages

Lists of specific error messages are contained in the *AppErrors.csv* and *AlertSeverities.csv* CSV files, which can be edited by double clicking them in the *Data Files* section of the project tree view.

These lists are based on significant consulting experience and meet the needs of most users. If however you feel that certain error messages are missing, or you wish to add additional error messages, you can customize these files by following the steps outlined in the following section.

Customizing Error Messages in HTML

The *AppErrors.csv* file contains error messages that cause errors to be raised when they occur in HTTP responses. Each row defines one error message across three columns:

- *Severity*: Specifies the severity of error that is to be raised. Values in this column must begin with S, I, W, E, or T (signifying the severities *Success*, *Informational*, *Warning*, *Error* or *Transaction Exit*).
- *Data or Html*: Specifies whether the entire HTTP response should be searched for the error message (API function *WebVerifyData*), or whether only the visible HTML content should be searched for the error message (API function *WebVerifyHtml*). Values in this column must begin with either D or H.
- *Error String*: The text of the error message.

  If you wish to catch a set of error messages that have a common substring, it's good practice to enter only one entry where the *Error String* is the common substring.

  Example: If you have e.g. 100 different error messages which all begin with "Microsoft SQL error:", it is sufficient to have one list entry with the *Error String* "Microsoft SQL error:".

The *AppErrors.csv* file can be edited to meet your needs.

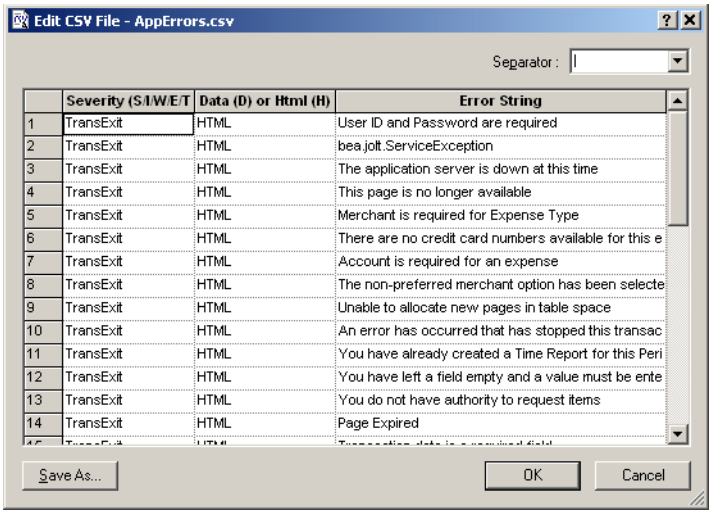


Figure 50 - Customizing application level errors

Customizing Alerts

Alerts are pop-up windows that are implemented by a JavaScript function called *Alert*. This function is called from the onload section of HTML pages.

Example: Alert in a HTML response

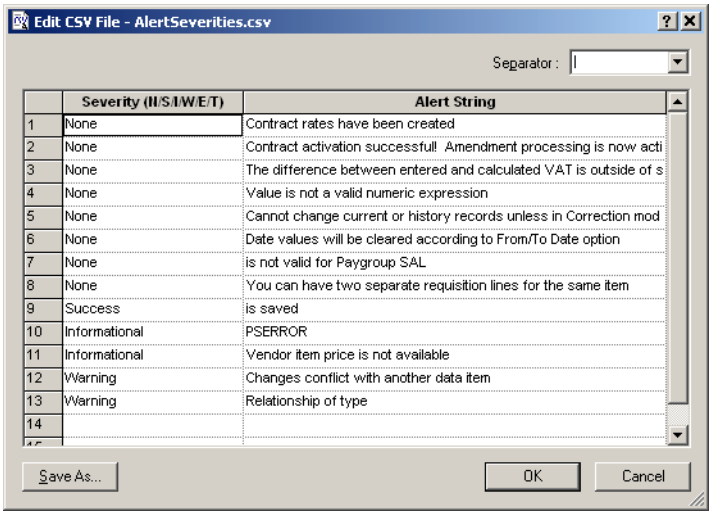
```
<body·class='PSPAGE'··onload="
  processing_main(0,3000);
  setKeyEventHandler_main();
  self.scroll(0,0);
  setEventHandlers_main('ICFirstAnchor_main',
                        ·'ICLastAnchor_main',·false);
  setupTimeout();
alert(&#039;Highlighted·fields·are·required.·(15,30)
      Enter·data·into·the·highlighted·fields.&#039;);"
>
```

The error detection mechanism detects any alert contained in a HTML response and treats it as being an error of *SEVERITY_TRANS_EXIT* severity. Known alerts that are to be ignored or reported with another severity can be specified in the *AlertSeverities.csv* file.

Each known alert message is represented by a single row in the *AlertSeverities.csv* file, across two columns:

- *Severity*: Specifies the severity of error that is to be raised. Values in this column must begin with N, S, I, W, E, or T (signifying the severities *None (=ignore)*, *Success*, *Informational*, *Warning*, *Error* or *Transaction Exit*).

- *Alert String*: The text of the alert string (or fragment thereof).



	Severity (I/S/I/W/E/T)	Alert String
1	None	Contract rates have been created
2	None	Contract activation successful! Amendment processing is now acti
3	None	The difference between entered and calculated VAT is outside of s
4	None	Value is not a valid numeric expression
5	None	Cannot change current or history records unless in Correction mod
6	None	Date values will be cleared according to From/To Date option
7	None	is not valid for Paygroup SAL
8	None	You can have two separate requisition lines for the same item
9	Success	is saved
10	Informational	PSERROR
11	Informational	Vendor item price is not available
12	Warning	Changes conflict with another data item
13	Warning	Relationship of type
14		

Figure 51 - Customizing alert severities

Alert messages are reported with the actual alert text, prefixed with the *gsAlertMsgPrefix* global variable. This variable contains an empty string by default, but can be assigned any value at anytime during script execution to meet your reporting needs.

Parameterization

The recorder can create variables for certain strings that appear in scripts. This makes script customization and randomization easier.

Sign In Data

The user names and passwords that are used for sign-in are created as variables at the tops of scripts. The recorder also records a commented call to the *GetLoginInfoPS* function in the TInit transaction. The *GetLoginInfoPS* function retrieves values for the *gsUserId* and *gsPassword* variables from the *LoginPS.csv* file, which can be edited by double clicking it in the *Data Files* section of the project tree view.

Customization of login data is done by uncommenting the recorded *GetLoginInfoPS* function call and populating the *LoginPS.csv* file with valid user accounts from your PeopleSoft application.

Example: Sign in data created as variables

```
var
  gsUserId    : string init "Admin";
  gsPassword  : string init "Secret";

  // ...
transaction TInit
  begin
    // ...
    PeopleSoftInit();
    //GetLoginInfoPS("LoginPS.csv", gsUserId, gsPassword);
    // ...
  end TInit;

  transaction TMain
  begin

  // ...

    SignInPS("login", LOGIN001, "EMPLOYEE"); // Form 1

  // ...

  end TMain;

dclform

  // ...

  LOGIN001:
    "httpPort"         := "" <USE_HTML_VAL> ,
                       // hidden, unchanged, value: ""
    "timezoneOffset"   := "-120",
                       // hidden, changed(!)
    "userid"           := gsUserId,
                       // changed, value: "Admin"
    "pwd"              := gsPassword,
                       // changed, value: "Secret"
    "Submit"           := "" <USE_HTML_VAL> ;
                       // unchanged, value: "Sign In"
```

Input Values

Form field values that begin and end with underscores, or begin with *inp.*, are treated as input values. The recorder generates variables at the tops of scripts for these values. This makes randomization of input values easier.

Example: Variables created for input values

```
var
  gsInput_NewProduct      : string init "_New Product_";
  gsInput_inp_Customer    : string init "inp.Customer";

  // ...

dclform

  // ...

  MAIN009:
    "ICType"           := "" <USE_HTML_VAL> ,
                      // hidden, unchanged, value: "Panel"
    "ICElementNum"     := "" <USE_HTML_VAL> ,
                       // hidden, unchanged, value: "0"
    "ICStateNum"       := "" <USE_HTML_VAL> ,
                       // hidden, unchanged, value: "6"
    "ICAction"         := "#ICSave",
                       // hidden, changed(!)
    "ICXPos"           := "" <USE_HTML_VAL> ,
                       // hidden, unchanged, value: "0"
    "ICYPos"           := "" <USE_HTML_VAL> ,
                       // hidden, unchanged, value: "0"
    "ICFocus"          := "" <USE_HTML_VAL> ,
                       // hidden, unchanged, value: ""
    "ICChanged"        := "" <USE_HTML_VAL> ,
                       // hidden, unchanged, value: "0"
    "ICFind"           := "" <USE_HTML_VAL> ,
                       // hidden, unchanged, value: ""
    "PROD_ITEM_DESCR"  := gsInput_NewProduct,
                       // changed, value: "_New Product_"
    "PROD_ITEM_EFF_STATUS" := "A",
                       // added
    "RBF_ARRA_WRK_PROD_ATTR_CATEGORY" := "",
                       // added
    "PRD_KIT_ARR_FLG1" := "" <USE_HTML_VAL> ,
                       // unchanged, value: "S"
    "DESCR$0"           := "" <USE_HTML_VAL> ,
                       // unchanged, value: ""
    "RBF_ARRA_PRD_VW_DESCR$0"    := "" <USE_HTML_VAL> ,
                       // unchanged, value: ""
    "RBF_PROD_ARGMT_RAISE_PCT$0" := "" <USE_HTML_VAL> ,
                       // unchanged, value: ""
    "RBF_PROD_ARGMT_REDUCTION_PCT$0" := "" <USE_HTML_VAL> ,
                       // unchanged, value: ""
```