Level: Intermediate
Works with: Lotus Workplace Messaging, Rational Suite TestStudio
Updated: 06-Oct-2003

by
Charles
Raymond
Smith

IBM Lotus Workplace Messaging (LWM) 1.0 provides low-cost messaging that extends your existing enterprise email to users traditionally without access to a dedicated workspace or desktop. As part of the product development process, we performed extensive testing on LWM 1.0. This included load and scalability analysis using computer load generation, performance measurement, and system testing. Previously, this testing would have been done using a combination of home-grown and third-party tools, which were often complex, expensive, and incompatible with externally obtained products. However, IBM's recent acquisition of Rational gives us the opportunity to apply a highly evolved integrated tool set that can be applied to a wide range of product load and scalability goals. See the technical article, "Using Rational Suite TestStudio to analyze a Domino application."

Rational Suite TestStudio provides multiple ways to create a workload for LWM 1.0. One way involves using TestStudio's record function to record the activities of a single LWM user and then to generalize this record to simulate a large group. The Performance Perspectives column, "Creating a Lotus Workplace Messaging test scenario with Rational Suite TestStudio" explains how to do this. This strategy works well when using HTTP to communicate because TestStudio supports this protocol through its record and playback features.

You can also build a workload by creating scripts with TestStudio's Virtual User (VU) programming language as explained in this article. In the example we describe, our workload communicates with LWM through SMTP, which TestStudio does not support natively through its record feature. However, VU provides a programmatic interface that allows us to support SMTP communications. In this article, we discuss how we created this workload and how it can be used to perform load and capacity testing in other environments. Equally important, we include workload development techniques that can help you create your own workloads.

We assume that you're familiar with LWM and have worked with system analysis tools, such as TestStudio or Server.Load. Some programming experience (especially with the C language) will also help you understand the sample scripts included in this article. To see a demonstration of LWM, visit our live demo on Lotus Labs. For more information on LWM testing, see the Performance Perspectives column, "Optimizing Lotus Workplace Messaging: Early experience." Also, the Rational Developer Domain offers articles and related information about TestStudio and all other Rational products.

LWM 1.0 workload environment

TestStudio, Rational's recording and script generating engine, supports a number of well-known protocols, such as HTTP, ODBC, and IIOP. This means that data transmissions using these protocols can be directly recorded, represented in the Virtual User (VU) language, and played back. This direct support enables

TestStudio to record samples of applications and to make simple modifications to the automatically generated VU scripts, quickly producing a usable workload. (See "Creating a Lotus Workplace Messaging test scenario with Rational Suite TestStudio.")

TestStudio's record feature can be used with a great number of protocols. However, a few that are of interest to LWM 1.0 cannot be recorded and parsed automatically. One of these is Simple Mail Transfer Protocol (SMTP). Although SMTP activity can be recorded, the VU scripts generated are not in the “language” of the protocol. Data that is not recognized as part of a protocol is represented in the VU language as basic socket-level Emulation Commands. Therefore, we did not use the record function as our primary method when building the SMTP workloads described in this article.

Hardware configuration

The sample scripts described in this article were developed and tested against an iSeries server running LWM 1.0 software. The primary development and test environment included:

- iSeries Model 730 running OS/400 Version 5 Release 2
- 8 GB of main memory
- 8-way processor

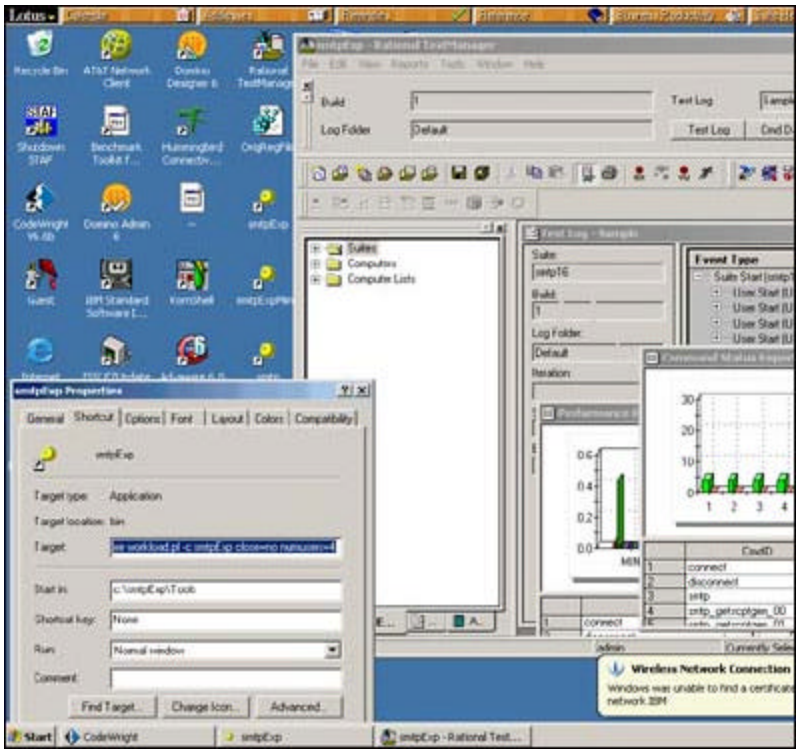
Note that LWM does not require the preceding configuration to run on a server. Also, LWM 1.0 can run on the Windows 2000 and AIX platforms.

Our workload

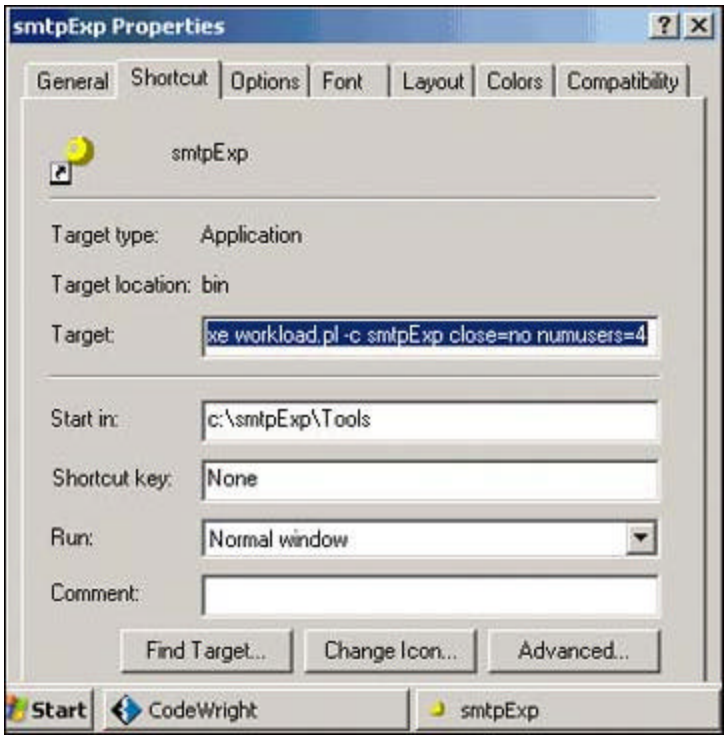
Our workload enhances the standard Rational Suite TestStudio interface in the following ways:

- The workload can be started and run to completion with little or no Rational tool or workload knowledge by the operator.
- Many of the workload operating parameters, such as server name, can be changed (via command line in the desktop shortcut) with no changes to the test script.
- Operating parameters can be stored in a configuration file separate from the test script.

The following illustration shows our workload in action:



The lower left corner of the workload screen includes a properties box for the shortcut smtpExp:



This properties box displays the command line interface used for this workload. To the right of this properties box is the Rational TestManager display. This display remains on the screen due to the close=no command line setting. If you set close=yes in the command line (via the shortcut), the TestManager automatically closes at the end of the run.

The following sections describe an example of how we developed a workload and scalability test for SMTP. We have provided a set of useful techniques and strategies that facilitate similar work for other developers who want to develop load scripts to use with their own products. Our goal is to share our testing experiences with you—while at the same time, assuring you that we at Lotus/IBM subject all of our software products to extensive and rigorous testing before release, using state-of-the-art tools!

Creating the SMTP workload

Our workload generated a stream of emails that follow the SMTP protocol. These emails were directed to a user-supplied SMTP server. (The user's server must have an SMTP processing application running during this process.) The workload performed the following actions:

- Initialization phase:
  - Setup server name
  - Setup user name(s)
  - Connect to server
  - Fail if connection fails

- Activity phase
  - Generate FROM, RCPT, DATA, … data calls
  - Accept expected status, Fail if error
  - Close connection

Note that the exact formats for the SMTP component commands, such as RCPT, DATA, and so on, are explained in the Internet Request For Comment (RFC) referenced in the following section. To represent actual user activities, we included the following randomizations in the email messages we generated:

www.lotus.com/ldd/today.nsf

- Random number of RCPTs (each RCPT command adds an additional mail destination)
- Random user names (RCPT, From) to ensure a more even, real-life distribution
- Random length of mail messages to model actual mail activity

Workload programming tips

Using TestStudio's VU programming language and the Internet Request for Comments (RFC) standards for SMTP and MIME, we developed test scripts for the LWM 1.0 workload (with minimal use of TestStudio's record feature). The RFCs we referenced are described in these specifications for Simple Mail Transfer Protocol (SMTP) and Multipurpose Internet Mail Extensions (MIME).

VU and C programming

When programming our scripts, we discovered VU is very much like the C language. The structure is almost identical, using braces (), {}, []; operators +, -, /, =, &, "*operator* ="; and most other syntax found in traditional C language. VU supports the commenting convention of /* through */ plus the newer // through to the end of the line. The header file inclusion mechanisms of #include <> and #include "" are both supported. Most basic C library functions, such as printf, are available to the VU programmer.

VU offers some improvements to the C language. For example, a string type has been added, replacing char and char []. Operations such as + (concatenate) and == equal test are supported on the string data type. This is especially useful with the very long strings encountered in developing communication strings. It is also useful appending strings to other strings using the += operator. For instance:

```
string file_path = "";
string base = "c:";
string file_sep = "/";
string name = "autoexec.bat";

file_path = base;
file_path += file_sep;
file_path += name;
```

VU programming tips

Based on our experience, we offer the following tips to programmers new to VU:

- To ensure no unwanted collisions between variables of like usage in separate routines, prefix what would be the local variable name with the routine's name. For example:

```
func foo(){

    int foo_a;
    string foo_b;
    }
```

We recommend this because there is no scoping of variable names except for function parameters. If not properly addressed, this can become a major problem as the size of the program increases. Also, with no functional prototypes, all function definitions must appear before the first use of the function. This becomes somewhat cumbersome as the size and complexity of the program increases. Finally, there is no include path specification.

- Place all header files in the script directory.
- Place generic work functions in separate include header files. We did this because there are no separately compiled functions (not counting DLLs).
- Implement your code in small parts and recompile and run them often. VU compilation error diagnostics are often of the form parse error with limited information on the location of the actual mistake. Be advised that the location of the error diagnostic is not always an absolute indication of the actual location of the problem (a common issue with other compilers because it often is essentially impossible to determine accurately where the actual user error may be). Errors such as unterminated quotations ("), undefined variables, and undefined functions give similar parse error diagnostics.
- When modifying code in a header file, make a simple change (for example, add then delete a space

character in the including script file) then recompile. Include or header (.S or .SBH) files are not automatically rescanned when the including file is recompiled. This means that if you make a change in the header file and then recompile the including VU script (.S) file, the change in the header file will not necessarily be seen. Rescanning the header files appears to take place if the including script is changed. (Note that we used .SBH extensions because the Rational Suite TestStudio editor defaults to this extension for project header files. However, many Rational developers also use the .S extension for VU scripts and use .SBH for GUI scripts.)

Allowing for variations

You should plan for the variability in the responses with programmed workloads. Try to avoid fixed length responses often generated by TestStudio's record feature, such as:

```
http_nrecv ["Login1.028"]  6533;       /* 6533/16557 bytes */
```

Instead, use a construct that reads to a pattern, such as end of line:

```
sock_recv [label] "\n";
```

or reads all available input:

```
http_nrecv ["Login1.039"]  100 %% ;    /* 2375 bytes */
```

External program control

A prime goal of our workload development effort was to incorporate flexibility into workload scripts so that they can support various conditions (such as different server names and user names) with a minimal amount of user or programmer intervention. There are a number of mechanisms available for this. We'll briefly discuss two: datapools and header files.

Datapools

TestStudio datapools are a very powerful facility to provide variable information. Datapools let you place information in a separate location (usually a file) whose contents can be changed without changing or recompiling the test script.

To allow for future modularity, we placed each datapool in its own script (.S) file. This lets us use these datapools individually for different test scripts. To implement multiple datapools, we compiled each datapool script file we created by using the datapool setup process (accessible through the TestStudio Edit - Datapool Information menu). However, it is possible to place multiple datapools in a single script file, an option we plan to explore further.

To share definitions, add this file using the include statement:

```
// smtp_message.s      19-May-2003  crs,
// Message info
DATAPOOL_CONFIG "smtp_message"  DP_RANDOM DP_PRIVATE DP_WRAP
{
INCLUDE, "messagelen", "string","100";
}
```

Although the preceding datapool script file is usable, we recommend more information hiding. Place definitions of access functions (functions that give access to the data but hide the particulars) in the datapool script files. This makes the test script more readable and usually makes data source modifications localized to the datapool file alone. Note that with the following example, the only reference to the data in the actual test script file is through generic access routines: smtp_message_open, smtp_message_close, and smtp_message_next. All the datapool specifics are hidden, even the fact that the data access is via a datapool versus some other access technique is localized to the datapool file:

```
// smtp_message.s      19-May-2003  crs,
```



```
// Message info
int DP_MESSAGE;                    // message length
string dp_message = "smtp_message";
DATAPOOL_CONFIG "smtp_message"  DP_RANDOM DP_PRIVATE DP_WRAP
{
   INCLUDE, "messagelen", "string","100";
}
proc smtp_message_open()
   {
   DP_MESSAGE = datapool_open(dp_message);
   }

proc smtp_message_close()
   {
   datapool_close(DP_MESSAGE);
   }

string func smtp_message_next()
   {
   datapool_fetch(DP_MESSAGE);
   return datapool_value(DP_MESSAGE, "messagelen");
   }
```

Note that the method described in this section in some respects reflects a beginner's approach to using Rational tools. However, those of you with more Rational experience can create dynamic datapools by using named files and providing command line options to allow the files' default names to be overridden with user entries in the configuration file and command line. We plan to explore this option further when developing future workloads.

Header files

Header files (usually denoted with the file extension .SBH) are text source files that are logically inserted into the script file, using the following syntax:

include "*name of header file* "

These header files are located in the same directory as the including script file. Note that there is an alternate form (include <file-name>), usually restricted to special files (such as VU.h), that looks in a special place. Header files are often useful to contain commonly used function definitions or variable definitions. However, it can be useful to use the include mechanism to place values on the command line into the VU script. The process goes as follows:

1. A command line script (for instance, a Perl script) decodes command line specifications.
2. The script creates a suitable header file with a fixed name (for example, _command.sbh) containing VU code with assignment statements of the form *variable_name=value.*
3. The primary VU script file contains the following include statement:

   include "_command.sbh";

   This include statement logically places the assignment statements where they will be executed, setting the variables to the appropriate values.
4. The command line script executes the Rational TestManager that, in turn, executes the test script.

The following is an example of a header file:

Command line

```
Perl.exe workload.pl command:cmd_servername=server.ibm.com
```

_command.sbh file – Automatically generated by workload.pl


```
//c:\smtp\TestDatastore\DefaultTestScriptDatastore\TMS_Scripts\_command.sbh
// - Automatically generated - Do not edit by hand
cmd_servername = "server.ibm.com";
```

Rational TestManager Command (Abbreviated) – invoked by workload.pl

```
rtmanager smtp16 /user admin /runsuite /project c:\smtp\smtp.rsp
```

Test Script (Abbreviated)

```
//   smtp16.s    17-Jun-2003  crs, include .sbh files
#include <VU.h>
...
// Default server name
string cmd_servername = "default.ibm.com";
...
{
string servername;          // Local copy
...
#include "_command.sbh"       // Bring in settings
servername = cmd_servername;
...
}
```

Protocol level routines

It is often useful to develop routines that encompass the protocol's structure rather than strictly rely on VU's low-level routines for input and output. One example of this is our routine smtp_get. The procedure smtp_get repeatedly retrieves lines of data until it encounters a line starting with d\d\d <sp> (where d stands for digit, and <sp> is the space character) matching the SMTP defined end of response. The extensive use of such functions, following generally accepted programming practices, improves the readability of the script and greatly reduces the probability of development programming errors. Note that smtp_get calls a protocol level routine socket_get that gets a line of data:

```
// Get smtp response
// till \d\d\d <sp>
// Placing cumulative response in smtp_get_str
proc smtp_get(label)
   string label;
   {
   string smtp_get_str = "";
   string smtp_get_reply_code;
   string smtp_get_tail;
   string smtp_get_desc;
   if (socket_rtns_display_in)
      printf("smtp_get(%s)\n", label);
   while (1)
      {
      socket_get("smtp_get:" + label);    // Get next line - till "\n"
      smtp_get_str += _response;
      if (!match("^([0-9]+)$0([- ])$1(.*)$2$", _response,
         &smtp_get_reply_code, &smtp_get_tail,
         &smtp_get_desc))
         {
         user_exit(-1, "Unexpected smtp response " + _response);
         }
      if (smtp_get_tail == " ")
         {
         return;
         }
```

```
        }
    }
```

Tracing socket level I/O

When developing a new protocol-based workload, it is very easy to make mistakes. These can include unwanted characters in the output or overlooking unexpected characters in the input. It can be very helpful to see what actually is being sent and received. While TestStudio produces log output and there are sniffers and logic analyzers available to display socket-level data transfer, these tools are sometimes difficult to use or do not show activity in a manner closely connected to your script.

We developed a set of simple wrapper routines which, in addition to sending and receiving the appropriate data, optionally display the data in a readable format. The trace routines pretty print socket input and output to virtual users' Virtual Tester Output files. The trace routines are patterned after the VU routines and are given similar names. Some of the trace output comes from higher-level routines (such as smtp_get) that have been instrumented with tracing code similar to that of the low level routines. The general procedure followed in tracing is as follows (indentation indicates conditional execution):

```
If tracing is enabled
Display routine name and user supplied label, if any
If input
Do actual input
If tracing is enabled
   Convert data into readable form
   Print out converted data
If output
   Do actual output
```

For example:

```
…
smtp_get(smtp_connect)
socket_get(smtp_get:smtp_connect); - 55 bytes
220 rchasmil.rchland.ibm.com Nagano SMTP Server Ready\r\n

socket_send();
EHLO joedoe\r\n

smtp_get(smtp_ehlo)
socket_get(smtp_get:smtp_ehlo); - 43 bytes
250-rchasmil.rchland.ibm.com Hello joedoe\r\n

socket_get(smtp_get:smtp_ehlo); - 9 bytes
250-DSN\r\n

socket_get(smtp_get:smtp_ehlo); - 18 bytes
250 SIZE 2000000\r\n
…
```

Conclusion

Rational Suite TestStudio provides a substantial development environment for creating protocol-based performance tests. Our experience has demonstrated that the following set of techniques can be very helpful in the development of workloads outside of the currently supported protocols:

- Learn the capabilities of the VU (Virtual User) language and how they can support your requirements, while paying attention to the differences between VU and its “first cousin” C.
- Judicious design and implementation of protocol-based routines is key in combating the inherent complexity involved with the protocol. Also consider the complexity of the protocol with which you are working.

www.lotus.com/ldd/today.nsf

- Dynamic tracing display of the network traffic is critical for the type of workload described in this article.

We hope you found this article a useful guide to help you develop your own workloads with TestStudio.

ABOUT THE AUTHOR

Charles Raymond (Ray) Smith has worked as a software engineer since the late 1970's, most recently with IBM in Westford, Massachusetts in the Lotus Engineering Test group. When not deep in other tasks, Ray is a local champion for increasing the more effective and extensive use of the Perl scripting language for problem solving. Ray has a wife and three children, the youngest of which is currently attending William and Mary College.

