

# SERVICE MANAGEMENT USING THE APPLICATION RESPONSE MEASUREMENT API WITHOUT APPLICATION SOURCE CODE MODIFICATION

Martin Haworth, Resource and Performance Management Solutions  
Network and System Management Division, Hewlett-Packard Company  
June 1997

*A key challenge facing IT Manager's is the measurement and management of end user response times. The ARM API provides a mechanism for addressing this key Service Management issue by the instrumentation of application code with response time markers. The availability of the ARM API has not, however, solved this problem for the many thousands of applications where source code changes are not possible. This presentation describes an alternative mechanism of utilizing the ARM API that provides the ability to acquire end user response times for applications without source code modification.*

## Introduction

A key challenge facing IT Manager's in the Open Systems arena is the measurement and management of end user response times. This challenge exists due to the lack of any mechanism to readily capture application response times with non-intrusive measurement tools. The Application Response Measurement Application Programming Interface (ARM API) provides a mechanism for addressing this key Service Management issue during the development of an application, or in those situations where source code changes can be made to an existing application.

The availability of the ARM API has hitherto not, however, solved the problem for the many thousands of applications that are already deployed and where source code changes are not possible. Examples of such applications include packaged solutions (where the users must wait until the application vendor instruments the application) and applications that are considered functionally stable, with no planned investment in development. This document and the accompanying presentation discuss an alternative mechanism of utilising the ARM API that provides the ability to acquire indicative end user response times for applications where source code modification is not a tenable option.

## The Application Response Measurement API

In June 1996 Hewlett-Packard and Tivoli announced the availability of the Application Response Measurement

Application Programming Interface (ARM API). ARM provides a straightforward method for developers to instrument application source code. The ARM API defines a set of six library procedure calls that programmers can utilise within their source code to initialise the ARM subsystem and define the beginning and end of Business Transactions:

<i>arm_init</i>	Initialise ARM environment for your application
<i>arm_getid</i>	Name each transaction that will be monitored
<i>arm_start</i>	Signify the start of a unique transaction instance
<i>arm_update</i>	Update statistics for a long running transaction (optional)
<i>arm_stop</i>	Signify the end of a unique transaction instance
<i>arm_end</i>	Clean up the ARM environment prior to application shutdown.

For a more detailed description of the calls and their usage, please consult the documentation in the ARM Software Developer's Kit, which is available free of charge from Hewlett-Packard or Tivoli (<http://www.hp.com/go/arm>, <http://www.tivoli.com/ARM>). Special Interest Group's focused on the ARM API are also hosted by the US and UK Computer Measurement Group's.

Once an application has been instrumented using ARM, any of the industry strength Service Management products that comply with the ARM API can then collect and manage transaction end to end response time and volumetric information.

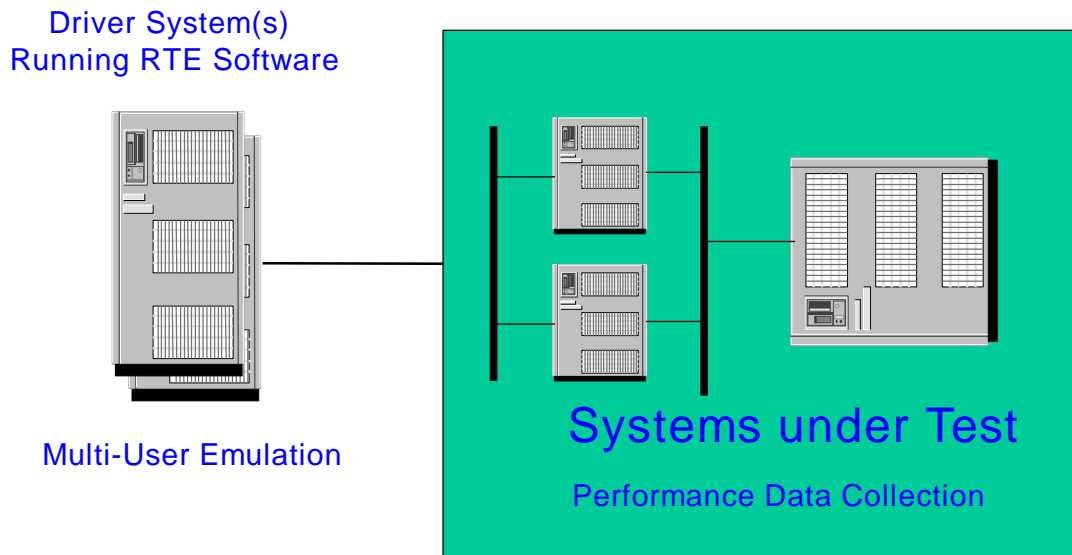
But what about those applications where the source code modifications required to instrument with ARM calls are not practical? As a Service Manager we would still like to be able to measure and manage the response times of these applications...

### Remote Terminal Emulation technology

Remote Terminal Emulation (RTE) is a technique traditionally employed in benchmarking and performance stress testing facilities to simulate large populations of users accessing and interacting with one or many applications. The environment for a simple performance benchmark exercise is illustrated in Figure 1.

scripts are editable, in some cases as C code, and generally it is possible to modify the scripts produced to include calls to external routines, such as the ARM API. In addition to the information required to replay the dialogue between the user and the application, the scripts also contain timing information, which controls the pacing of the business transactions. To replay a script, it is compiled using a special environment provided by the software vendor, which links additional library modules in to the script to facilitate its execution and monitoring during benchmarks where many hundreds, or even thousands, of scripts run concurrently.

Figure 1  
Performance Benchmark Environment



A number of specialised RTE software packages are available which facilitate the generation and management of the emulated user scripts which define each of the simulated online sessions. RTE packages are available to drive applications employing traditional "dumb" character mode terminals, client-server architectures, X-terminal display and Internet Browser interfaces, as well as specialised modules for application such as SAP R/3.

RTE packages provide facilities for capturing a user script by recording the real dialogue between a user terminal or workstation and the application/database servers. The

An RTE script may use 50% (or more) of the resources of a normal, "real" online user and so the hardware resources required to support the RTE scripts for a run of, perhaps, 1000 users can be considerable. During performance testing, a separate RTE driver system is therefore used to ensure that no additional resource requirements are placed upon the system being tested. A key point worth noting is that while it is normal to use a separate system (or set of systems) to run the RTE emulation software, this is not a technical requirement; The RTE scripts can also be run on the System under Test or indeed a system running a live application in a production environment.

## Using RTE Software as a Service Management Monitor

We have already established two key points in the preceding text;

- RTE scripts can run on the same system as the application software.
- Many leading RTE tools produce scripts that are editable and can make calls to external library routines (such as the ARM API).

It is therefore possible to use RTE software as part of a toolkit to build “robots”, which drive a live production application automatically, and measure the service levels being delivered. The use of robots in this context is not, of itself, revolutionary. Indeed this approach has been implemented for a number of years by organisations seeking to gain control of the service levels being delivered in distributed environments. The main problem, however, is that a significant degree of development investment has typically been required to achieve this objective. Where organisations have chosen to implement response time robots it has been necessary for them to write the script capture and replay software (although some organisations already use the RTE toolsets broadly described in this document) and to also provide the measurement infrastructure for capturing, logging and reporting response times. The use of “off the shelf” RTE packages in conjunction with the ARM API and measurement subsystems that support ARM means that the only development investment required by organisations wishing to develop response time robots is in recording and fine tuning scripts that are specific to the applications being monitored.

For an application that is deployed in production and for which we would like to monitor end user response times (without any changes to its source code) the following approach is therefore proposed.

- 1) Record one or more RTE scripts that exercise the key business transactions of the application on the live database. Scripts should be recorded and executed against the live application and databases, as we want to gain as realistic a measure as possible of the service being delivered to the service users. The script sessions can either be kept quite short, and then executed repetitively, or can be quite long (in terms of elapsed time to run) and then be run less frequently. In order to aid re-usability of the scripts, it is most useful if the scripts can undo any changes they make to the database. For example, if a script exercises the transactions “add order”, and “change order”, it is prudent to also include a corresponding “delete

order” transaction, even if this is not of particular interest from a service management perspective.

- 2) Edit the scripts to include calls to the ARM API for each business transaction that is exercised and compile them, linking the ARM library into the script compile process. Most scripts produced by RTE packages already have constructs within the code that aid in the identification of the points at which ARM calls should be inserted. Naturally it would be advantageous if the RTE packages could automatically embed ARM calls within the scripts, however at present this functionality is not available. As is the case with the ARM instrumentation of leading packaged solutions from software vendors, business demands from paying customers could eventually persuade the leading RTE vendors to provide this facility. If the RTE software provided automatic ARM instrumentation the scripts would only require editing to ensure that meaningful transaction naming conventions were being utilised in the ARM calls.
- 3) Schedule the scripts to run at an appropriate interval using your installations chosen scheduling software (or alternatively, and for example, the Un\*x *crontab* or Windows NT *at* command facility).
- 4) Utilise an ARM compliant monitoring or service management package to capture, raise exception alarms, log and report the service levels experienced by the scripts driving the application, with integration into your chosen Enterprise System Management.

This approach provides the mechanism to implement a “Service Management Monitor” for an application that provides a realistic measure of the typical service levels being experienced by its users. It does not, however, provide the level of granularity that can be gained by actual instrumentation of the application, where service level degradations can be detected for individual users or groups of users. It is also limited in that, unlike direct instrumentation with ARM, it cannot provide the overall business transaction throughput rates for the entire application, as the transaction throughput of the Service Management Monitor is dictated by the structure and frequency of execution of the scripts (and is therefore artificial in nature). However, degradation in the transaction throughput of the Service Management Monitor, caused by poor system performance, can be measured and used as an indicative Service Level metric.

These shortcomings can be partially addressed by running multiple scripts, perhaps in different geographic locations for a distributed client-server application where network response time is an important component of response time that will vary for different user populations.

Figure 2 shows an example scenario of an application running in a tiered client-server environment. The

application can be accessed by PC's running the client component of the application, or by "dumb" terminals where the client component of the application runs on the application servers.

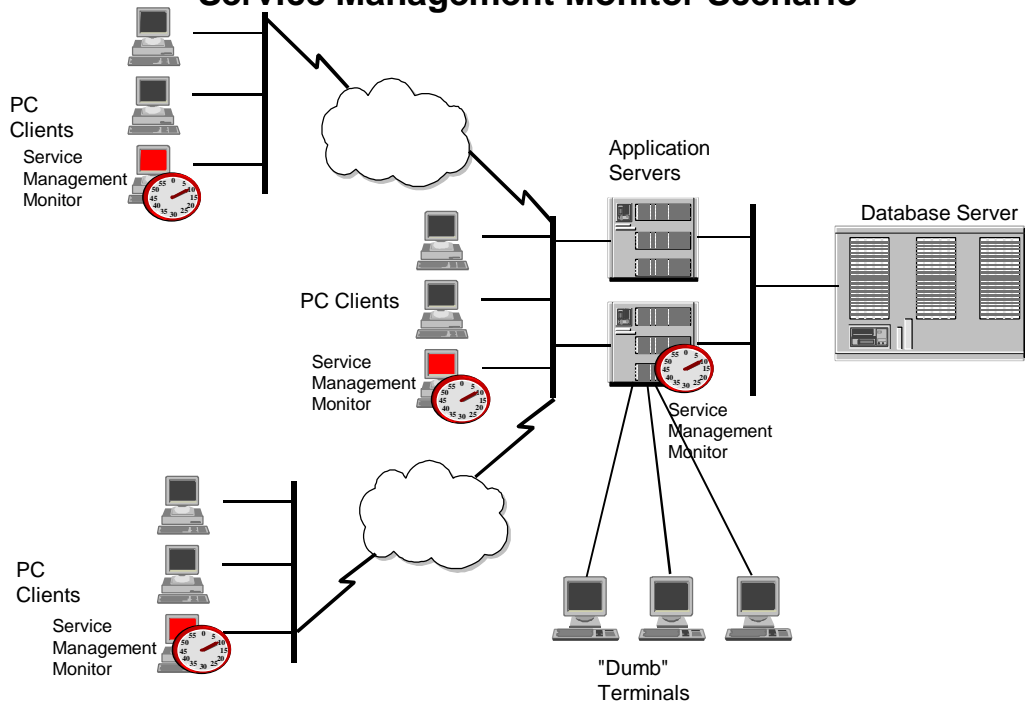
In this environment, it would be advisable to monitor the end user response times at a number of places in the infrastructure in order to be able to identify service degradation resulting from failures in remote LAN's, the LAN/WAN bridges etc. Each of the Service Management Monitors installed in the environment is therefore providing valuable information about the service levels being delivered to specific subsets of the user population.

The number of Service Management Monitors that you choose to deploy within your environment will, of course, be driven by a number of factors including:

- The physical infrastructure.
- The definitions of your application Service Level Objectives.
- The criticality of the Service to specific segments of the user population.

**Figure 2**

**Service Management Monitor Scenario**



**Summary**

The concept of a Service Management Monitor is not revolutionary within the industry, however it has historically required a significant investment of development time to implement the software infrastructure required. In the worst case an organisation would need to develop script capture and replay software in addition to the code to measure, log and report on response times. The investment required has resulted in only a few, typically very large, organisations adopting this

approach. The availability of robust Remote Terminal Emulation software and the ARM API with support from mainstream performance management vendors provides a far more compelling argument for the Service Management Monitor. Organisations wishing to implement Service Management Monitors need now only develop the component of the "response time robot" that is specific to their application and business, allowing them to focus on delivering business solutions rather than becoming a software testing and service management solution development department.

## Appendix A

### Example of an RTE script with embedded ARM API calls

The text below is a sample fragment from an RTE script with ARM instrumentation embedded to time how long it takes to login to a Un\*x system.

```
#include "arm.h"

/* Do various RTE initialisation things */
Set(CDELAY);          /* Put typing delay between characters */
Typerate(50);         /* Typing delay in CPS */
Thinkuniform(0.1,0.5); /* Think delay at every Xmit() */
Seed(getpid());       /* Seed random number generator */
Timeout(10, CONTINUE); /* What to do if Rcv() takes too long */

/* Now lets define our ARM variables and initialise them */
arm_tran_id_t login_tran_id = -1; /* Define a unique identifier for the login transaction */
arm_appl_id_t appl_id; /* Define an identifier for the application id */
arm_start_handle_t tran_handle; /* define a variable for the transaction ID */

void init()
{
    appl_id=arm_init("ARM/RTE sample script", "", 0,0,0);
    login_tran_id = arm_getid(appl_id, "User_Login", "Sample Login Transaction",0,0,0);
    if (login_tran_id <= -1) printf("Failed to register User_Login transaction.\n");
} /* init */

/* OK, all initialisation is done, lets login to the system – send a Carriage return to get a prompt */
Xmit("^M");

/* Now lets start a transaction to time our login */
tran_handle = arm_start(login_tran_id,0,0,0);

/* Wait for the login response from the system : ^M is a carriage return, ^J a linefeed */
Rcv("^M^JHP-UX hpuxtest A.09.04 U 9000/856 (ttyp4)^M^J^M^Jlogin: ");

/* Send our user name followed by a carriage return */
Xmit("armtest^M");

/* ... and wait for a Password prompt */
Rcv("^M^JPassword:");

/* Now send our password */
Xmit("ARMAPI^M");

/* and wait for the Un*x prompt */
Rcv("$ ");

/* We are now logged on, so we can stop the login transaction */
arm_stop(tran_handle, ARM_GOOD, 0,0,0);

/* If this were the end of our script we would now call : arm_end(appl_id,0,0,0); */
```