Abstract

Client/Server (C/S) technology is being taken up at an incredible rate. Almost every development organisation has incorporated C/S as part of their IT strategy. It appears that C/S will be the dominant architecture taking IT into the next millennium. Although C/S technology is gaining acceptance rapidly and development organisations get better at building such systems, performance issues remain as an outstanding risk when a system meets its functional requirements.

This paper sets out the reasons why system performance is a risk to the success of C/S projects. A process has been outlined which the authors have used to plan, prepare and execute automated performance tests. The principles involved in organising a performance test have been set out and an overview of the tools and techniques that can be used for testing two and three-tier C/S systems presented.

In planning, preparing and executing performance tests, there are several aspects of the task which can cause difficulties. The problems that are encountered most often relate to the stability of the software and the test environment. Unfortunately, testers are often required to work with a shared environment with software that is imperfect or unfinished. These issues are discussed and some practical guidelines are proposed.

Prerequisite Key Words: none

Topic Descriptors: Performance Testing, Client/Server.
1. INTRODUCTION

1.1 Client/Server Technology

Client/Server (C/S) technology is being taken up as never before. Almost every development organisation has incorporated C/S as the backbone of their IT strategy. In the US, almost all organisations are retraining developers in the new techniques and implementing C/S projects. Take up in the UK is less rapid, but not far behind. It appears that C/S will be the dominant architecture taking IT into the next millennium.

As the emerging technology of the 90s, C/S has been made out to be the solution to many of the problems IT organisations face. C/S systems were touted as being cheaper to build and run, providing the flexibility businesses required in the 90s, the best route to Open Systems and the easiest way to replace ageing legacy mainframes.

Although most larger organisations have developed C/S systems as pilot projects, in the early 90s the success rate of C/S development projects was very poor. Many projects were abandoned before completion, either because of their complexity or through lack of developer experience. Other projects failed because mainframe development methodologies were used and sometimes because essential disciplines (such as configuration management and software testing) were not used. Expectations of C/S technology are now more realistic as so many early projects proved to be difficult learning experiences.

Nowadays, the success rate has clearly improved. However, a fifth of all C/S systems fail to meet their requirements in terms of number of transactions, concurrent user access and/or message loadings. C/S performance is emerging as one of the major risks to success and is growing in importance.

C/S technology is promoted as a more efficient (and hence economic) means of procuring processing power and placing it where it is needed. In a C/S architecture, processing is distributed between:

- **The client**, which deals with presentation logic (the GUI) and application ‘front end’.
- **Application servers** which process ‘business logic’ (a broad term encompassing data caching, messaging or other processing better suited to a dedicated server).
- **Database server(s)**, which run the DBMS software and are optimised to process SQL requests received from clients or other servers.

Most C/S systems are two-tier architectures and only use clients and database servers.

1.2 Why is Client/Server Performance an Issue?

We can identify four main issues associated with the performance in C/S systems:

- Large volumes of network traffic caused by ‘intelligent clients’.
- Increased number of ‘architectural layers’ of which a system is composed.
- Delays between distributed processes communicating across networks.
- The increased number of suppliers of architectural components who must be dealt with.

These causes are discussed in the rest of this section.
1.2.1 Intelligent Clients

Client workstations usually submit messages across the network for processing by the database server. For single row actions the dialogues between the client and the server are relatively simple. However, there are many circumstances where the simple dialogue becomes much more complicated and drastically increase the level of network traffic.

One example is where a simple query, constructed by the application, allows wild-card searches to be performed. Forms-based applications usually fetch only the data that can be displayed on the screen. However, on a workstation, it is quite common for the application to fetch all the data in the query and store it in arrays for later use. Not only must the database fetch all the table rows, the data must be transferred over the network, and the workstation must allocate large amounts of memory (with consequent paging onto disk) to process the query. Although the user may only ever see the first few rows displayed, very large amounts of network, server and client resources are used and the response time is slow.

It should also be noted that middleware which implements the connectivity between client and database processes running on separate hosts can often impose considerable overheads of its own. For example, some middleware introduces many additional messages beyond those that transfer SQL statements in one direction and data in the other. Some middleware reduces the number of messages by fetching batches of rows together, rather than individually.

1.2.2 Increased Number of Architectural Layers

In a traditional system, one might be able to identify only five to seven distinct layers between the user and the data managed within the database. A typical comparison between a terminal/host based system and two and three tier architectures is presented in Table 1 below.

<table>
<thead>
<tr>
<th>Terminal/Host</th>
<th>Two Tier C/S</th>
<th>Three Tier C/S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal firmware</td>
<td>Display driver</td>
<td>Display driver</td>
</tr>
<tr>
<td></td>
<td>GUI</td>
<td>GUI</td>
</tr>
<tr>
<td></td>
<td>Application code</td>
<td>Application code</td>
</tr>
<tr>
<td></td>
<td>Toolkit Layer</td>
<td>Toolkit Layer</td>
</tr>
<tr>
<td></td>
<td>DB Connectivity</td>
<td>Middleware</td>
</tr>
<tr>
<td></td>
<td>Client O/S</td>
<td>Client O/S</td>
</tr>
<tr>
<td></td>
<td>Network Transport</td>
<td>Network Transport</td>
</tr>
<tr>
<td>Terminal Driver</td>
<td>LAN/WAN</td>
<td>LAN/WAN</td>
</tr>
<tr>
<td>Forms Management s/w</td>
<td>Network Transport</td>
<td>Network Transport</td>
</tr>
<tr>
<td>Application code</td>
<td>SQL Services</td>
<td>DB Connectivity</td>
</tr>
<tr>
<td>SQL Services</td>
<td>DBMS</td>
<td>SQL Services</td>
</tr>
<tr>
<td>DBMS</td>
<td>DB Server O/S</td>
<td>DBMS</td>
</tr>
<tr>
<td>Host O/S</td>
<td></td>
<td>DB Server O/S</td>
</tr>
<tr>
<td>7 Layers</td>
<td>13 Layers</td>
<td>20 Layers</td>
</tr>
</tbody>
</table>

Table 1. Comparison of Architectural Layers.

The significance of the number of layers is that the overall functionality of all the layers has increased, that there are many more interfaces between layers, and that the software in each layer is unlikely to be optimal because of assumptions made by the different developers usually from different suppliers of each of the layers.
1.2.3 Distributed Processes

In traditional systems, user processes running applications communicate directly with the DBMS process using the system bus which would normally operate at extremely high speed.

In a C/S system, there are similar processes (user and DBMS) operating, but on separate processors connected by a network. On a Wide Area Network (WAN), network delays between these two processes are likely to be much greater and might be of the order of 10 or 20 milliseconds. Under normal circumstances, this would not be noticeable to a user. However, where a transaction involves hundreds of messages being exchanged, the overall delay might be measured in seconds, and this delay is much more significant when added to the delays within the servers and clients.

1.2.4 More Suppliers

Because C/S architectures often have more than ten layers, or an ‘open’ C/S architecture is being implemented, it is common for the layered products to be produced by several different suppliers. When a C/S application is found to have performance problems it may not be at all clear which supplier’s product is to blame.

2. CLIENT/SERVER PERFORMANCE TESTING PROCESS

Unlike host-based systems, it is usually not possible to model (and predict) the performance of a C/S system because of its increased complexity. Usually, some simple, informal tests of an untried architecture are performed during system development to give some indication of the actual performance under real loads. Such informal tests may give some confidence, but are unreliable when it comes to predicting response times under production loads.

Performance testing using a simulated load (sized in accordance with the users’ business volume estimates) with response time measurements compared with agreed users requirements is the only practical method of predicting whether a system will perform acceptably.

Although it is possible for performance tests to be conducted with testers executing manual test scripts, this paper is concerned with performance tests which use automated test running tools. Automated test running tools make use of test scripts which define the actions required to simulate a user’s activity on a client application or messages sent by a client across the network to servers. Most proprietary test running tools have their own script language which are, in many ways, like programming languages.

2.1 Performance Testing Objectives

The objectives of a performance test are to demonstrate that the system meets requirements for transaction throughput and response times simultaneously. More formally, we can define the primary objective as:

“To demonstrate that the system functions to specification with acceptable response times while processing the required transaction volumes on a production sized database.”

The main deliverables from such a test, prior to execution, are automated test scripts and an infrastructure to be used to execute automated tests for extended periods. This infrastructure is an asset, and an expensive one too, so it pays to make as much use of this infrastructure as possible.

Fortunately, the test infrastructure is a test bed which can be used for other tests with broader objectives which we can summarise as:
- **Assessing the system’s capacity for growth** - the load and response data gained from the tests can be used to validate the capacity planning model and assist decision making.

- **Identifying weak points in the architecture** - the controlled load can be increased to extreme levels to stress the architecture and break it - bottlenecks and weak components can be fixed or replaced.

- **Detect obscure bugs in software** - tests executed for extended periods can cause failures caused by memory leaks and reveal obscure contention problems or conflicts.

- **Tuning the system** - repeat runs of tests can be performed to verify that tuning activities are having the desired effect - improving performance.

- **Verifying resilience and reliability** - executing tests at production loads for extended periods is the only way to assess the system’s resilience and reliability to ensure required service levels are likely to be met.

The test infrastructure can be used to address all these objectives and other variation on these themes. A comprehensive test strategy would define a test infrastructure to enable all these objectives to be met.

### 2.2 Pre-Requisites for Performance Testing

We can identify five pre-requisites for a performance test. Not all of these need be in place prior to planning or preparing the test (although this might be helpful), but rather, the list below defines what is required before a test can be executed.

#### 2.2.1 Quantitative, Relevant, Measurable, Realistic, Achievable Requirements

As a foundation to all tests, performance objectives, or requirements, should be agreed prior to the test so that a determination of whether the system meets requirements can be made. Requirements for system throughput or response times, in order to be useful as a baseline to compare performance results, should have the following attributes. They must be:

- **Quantitative** - expressed in quantifiable terms such that when response times are measured, a sensible comparison can be made. For example, response time requirements should be expressed as a number of seconds, minutes or hours.

- **Relevant** - a response time must be relevant to a business process. For example, a response time might be defined within the context of a telesales operator capturing customer enquiry details and so should be suitably quick, or a report generated as part of a monthly management reporting process and which might have an acceptable delay of ten minutes.

- **Measurable** - a response time should be defined such that it can be measured using a tool or stopwatch and at reasonable cost. It will not be practical to measure the response time of every transaction in the system in the finest detail.

- **Realistic** - response time requirements should be justifiable when compared with the durations of the activities within the business process the system supports. Clearly, it is not reasonable to demand sub-second response times for every system function, where some functions relate to monthly or occasional business processes which might actually take many minutes or hours to prepare or complete.

- **Achievable** - response times should take some account of the cost of achieving them. There is little point in agreeing to response times which are clearly unachievable for a reasonable cost (i.e. within the budget for the system).
2.2.2 Stable System

A test team attempting to construct a performance test of a system whose software is of poor quality is unlikely to be successful. If the software crashes regularly it will probably not withstand the relatively minor stress of repeated use. Testers will not be able to record scripts in the first instance, or may not be able to execute a test for a reasonable length of time before the software, middleware or operating systems crash.

Performance tests stress all architectural components to some degree, but for performance testing to produce useful results the system infrastructure should be both reliable and resilient.

2.2.3 Realistic Test Environment

The test environment should ideally be the production environment or a close simulation and be dedicated to the performance test team for the duration of the test. Often this is not possible. However, for the results of the test to be useful, the test environment should be comparable to the final production environment. Even with an environment which is somewhat different from the production environment, it should still be possible to interpret the results obtained using a model of the system to predict, with some confidence, the behaviour of the target environment. A test environment which bears no similarity to the final environment may be useful for finding obscure errors in the code, but is, however, useless for a performance test.

A simple example where a compromise might be acceptable would be where only one server is available for testing but where the final architecture will balance the load between two identical servers. Reducing the load imposed to half during the test might provide a good test from the point of view of a server, but might, however, understate the load on the network. In all cases, the compromise environment to be used should be discussed with the technical architect who may be able to provide the required interpretations.

The performance test will be built to provide loads which simulate defined load profiles and can also be adjusted to impose higher loads. If the environment is such that, say, a 20% error in any results obtained from tests are expected, extra confidence may be gained by adjusting the load imposed by 20% (or more) to see if performance is still acceptable. Although not entirely scientific, such tests should increase confidence in the final system as delivered if the tests show performance to be acceptable.

2.2.4 Controlled Test Environment

Performance testers require stability not only in the hardware and software in terms of its reliability and resilience, but also need changes in the environment or software under test to be minimised. Automated scripts are extremely sensitive to changes in the behaviour of the software under test. Test scripts designed to drive client software GUIs are prone to fail immediately, if the interface is changed even slightly. Changes in the operating system environment or database are equally likely to disrupt test preparation as well as execution and should be strictly controlled. The test team should ideally have the ability to refuse and postpone upgrades in any component of the architecture until they are ready to incorporate changes to their tests. Changes intended to improve performance or the reliability of the environment would normally be accepted as they become available.

2.2.5 Performance Testing Toolkit

The execution of a performance test must be, by its nature, completely automated. However, there are requirements for tools throughout the test process. Test tools are considered in more
detail later, but the five main tool requirements for our ‘Performance Testing Toolkit’ are summarised here:

- **Test Database Creation/Maintenance** - to create the large volumes of data on the database which will be required for the test. Usually SQL or ‘Procedural SQL’ database tools.

- **Load generation** - tools can be of two basic types, either a test running tool which drives the client application, or a test driver which simulates clients workstations.

- **Application Running Tool** - test running tool which drives the application under test and records response time measurements. (May be the same tool used for load generation).

- **Resource Monitoring** - utilities which can monitor and log both client and server system resources, network traffic, database activity.

- **Results Analysis and Reporting** - test running and resource monitoring tools can capture large volumes of results data. Although many such tools offer facilities for analysis, it is often useful to be able to combine results from these various sources and produce combined summary test reports. This can usually be achieved using PC spreadsheet, database and word processing tools.

### 2.3 Performance Requirements

Performance requirements normally comprise three components:

- Response time requirements.
- Transaction volumes detailed in ‘Load Profiles’.
- Database volumes.

#### 2.3.1 Response Time Requirements

When asked to specify performance requirements, users normally focus attention on response times, and often wish to define requirements in terms of generic response times. A single response time requirement for all transactions might be simple to define from the users point of view, but is unreasonable. Some functions are critical and require short response times, but others are less critical and response time requirements can be less stringent.

Some guidelines for defining response time requirements are presented here:

- **For an accurate representation of the performance experienced by a live user, response times should be defined as the period between a user requesting the system to do something (e.g. clicking on a button) to the system returning control to the user.**

- **Requirements can often vary in criticality according to the different business scenarios envisaged. As a consequence, quick responses are not always required. Business scenarios are often matched with load profiles (see section 2.3.2).**

- **Generic requirements** are described as ‘catch all’ thresholds. Examples of generic requirements are times to ‘perform a screen update’, ‘scroll through a page of data’, ‘navigate between screens’

- **Specific requirements** define the requirements for identified system transactions. Examples would be the time ‘to register a new purchase order in screen A0101’

- **Response times for specific system functions should be considered in the context of the business process the system supports. As a rule of thumb, if a business process is of short**
duration, e.g. logging a customer call, response times should be suitably brief. If a business process is of longer duration, e.g. preparing a monthly report, longer delays ought to be acceptable.

- Requirements are usually specified in terms of acceptable maximum, average or 95 percentile times.

Response times should be broken down into types: generic and specific, where appropriate. Generic response times can be defined for system updates, queries or reports and are often qualified by complexity. Response time requirements for specific system functions should be stated separately.

The test team should set out to measure response times for all specific requirements and a selection of transactions which provide two or three examples of generic requirements.

### 2.3.2 Load Profiles

The second component of performance requirements is a schedule of load profiles. A load profile is a definition of the level of system loading expected to occur during a specific business scenario. Business scenarios might cover different situations when the users’ organisation has different levels of activity or involve a varying mix of activities which must be supported by the system.

Examples of business scenarios might be:

- Average load, busy hour, busy 5 minutes - useful where the mix of activities is relatively constant, but the volume of tasks undertaken varies.
- Normal, end of month, end of year - where an organisation’s activities change over time with peaks occurring at specific periods.
- Quiescent, local fault, widespread emergency - where a support organisation might have quiet periods interspersed with occasional peaks and must cater for 1 in 200 year disasters.

A comprehensive load profile specification will identify the following for each business scenario:

- User types or roles.
- Identification of all locations.
- Distribution (numbers) of users of each type at each location.
- Business processes (or system transactions) performed by each user type at each location and the estimated transaction rate.

Table 2 below is an extract from a typical load profile specification.
## Scenario: Major Fault

<table>
<thead>
<tr>
<th>ID</th>
<th>Transaction</th>
<th>User Type</th>
<th>No. Users</th>
<th>Location</th>
<th>TXN rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>Log Customer Fault</td>
<td>Telesales</td>
<td>100</td>
<td>BHM</td>
<td>20/hr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Telesales</td>
<td>80</td>
<td>BTL</td>
<td>15/hr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Telesales</td>
<td>140</td>
<td>WEM</td>
<td>25/hr</td>
</tr>
<tr>
<td>24</td>
<td>Allocate Fault</td>
<td>Fault Controller</td>
<td>5</td>
<td>BHM</td>
<td>10/hr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fault Controller</td>
<td>7</td>
<td>WEM</td>
<td>14/hr</td>
</tr>
<tr>
<td>25</td>
<td>Escalate Fault</td>
<td>Section leader</td>
<td>10</td>
<td>BHM</td>
<td>5/hr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section leader</td>
<td>10</td>
<td>WEM</td>
<td>10/hr</td>
</tr>
<tr>
<td>26</td>
<td>Clear Fault</td>
<td>Fault Controller</td>
<td>5</td>
<td>BHM</td>
<td>10/hr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fault Controller</td>
<td>7</td>
<td>WEM</td>
<td>14/hr</td>
</tr>
</tbody>
</table>

Table 2. Example Load Profile.

### 2.3.3 Database Volumes

Data volumes, defining the numbers of table rows which should be present in the database after a specified period of live running complete the load profile. Typically, data volumes estimated to exist after one year’s use of the system are used, but two year volumes or greater might be used in some circumstances, depending on the business application.

### 2.4 Process

We can identify a four stage test process. An additional stage, tuning, can be identified. Tuning can be compared with the bug fixing activity that usually accompanies functional test activities. Tuning may involve changes to the architectural infrastructure and often does not affect the functionality of the system under test. A schematic of the test process is presented in Figure 1 below. The five stages in the process are described in outline in Figure 2.

![Figure 1. High Level Performance Test Process.](image)
Specification
- Documentation of performance requirements including
  - database volumes
  - load profiles having relevance to the business
  - response time requirements.
- Preparation of a schedule of load profile tests to be performed (e.g. normal, busy hour, busy 5 minutes or some other scheme).
- Inventory of system transactions comprising the loads to be tested.
- Inventory of system transactions to be executed and response times measured.
- Description of analyses and reports to be produced.

Preparation
- Preparation of a test database with appropriate data volumes.
- Scripting of system transactions to comprise the load.
- Scripting of system transactions whose response is to be measured (possibly the same as the load transactions).
- Development of Workload Definitions (i.e. the implementations of Load Profiles).
- Preparation of test data to parameterise automated scripts.

Execution
- Execution of interim tests.
- Execution of performance tests.
- Repeat test runs, as required.

Analysis
- Collection and archiving of test results.
- Preparation of tabular and graphical analyses.
- Preparation of reports including interpretation and recommendations.

Tuning
- Sundry changes to application software, middleware, database organisation.
- Changes to server system parameters.
- Upgrades to client or server hardware, network capacity or routing.

Figure 2. Performance Test Process Outline.

2.5 Incremental Test Development

Test development is usually performed incrementally and follows a RAD-like process. The process has four stages:

- Each test script is prepared and tested in isolation to debug it.
- Scripts are integrated into the development version of the workload and the workload is executed to test that the new script is compatible.
- As the workload grows, the developing test framework is continually refined, debugged and made more reliable. Experience and familiarity with the tools also grows, and the process used is fine-tuned.
- When the last script is integrated into the workload, the test is executed as a ‘dry run’ to ensure it is completely repeatable and reliable, and ready for the formal tests.

Interim tests can provide useful results:

- Runs of the partial workload and test transactions may expose performance problems. These can be reported and acted upon within the development groups or by network, system or database administrators.
- Tests of low volume loads can also provide an early indication of network traffic and potential bottlenecks when the test is scaled up.
Poor response times can be caused by poor application design and can be investigated and cleared up by the developers earlier. Inefficient SQL can also be identified and optimised.

Repeatable test scripts can be run for extended periods as soak tests. Such tests can reveal errors, such as memory leaks, which would not normally be found during functional tests.

2.6 Test Execution

The execution of formal performance tests requires some stage management or co-ordination. As the time approaches to execute the test, team members who will execute the test as well as those who will monitor the test must be warned, well in advance. The ‘test monitoring’ team members are often working in dispersed locations and need to be kept very well informed if the test is to run smoothly and all results are to be captured correctly. The test monitoring team members need to be aware of the time window in which the test will be run and when they should start and stop their monitoring tools. They also need to be aware of how much time they have to archive their data, pre-process it and make it available to the person who will analyse the data fully and produce the required reports.

Beyond the co-ordination of the various team members, performance tests tend to follow a standard routine shown in Figure 3 below.

1. Preparation of database (restore from tape, if required).
2. Prepare test environment as required and verify its state.
4. Start the load simulation and observe system monitor(s).
5. When the load is stable, start the application test running tool and response time measurement.
6. Monitor the test closely for the duration of the test.
7. If the test running tools do not stop automatically, terminate the test when the test period ends.
8. Stop monitoring tools and save results.
9. Archive all captured results, and ensure all results data is backed up securely.
10. Produce interim reports, confer with other team members concerning any anomalies.
11. Prepare analyses and reports.

Figure 3. Performance Test Execution Procedure.

When a test run is complete, it is common for some tuning activity to be performed. If a test is a repeat test, it is essential that any changes in environment are recorded, so that any differences in system behaviour, and hence performance results can be matched with the changes in configuration. As a rule, it is wise to change only one thing at a time so that when differences in behaviour are detected, they can be traced back to the changes made.

2.7 Results Analysis and Reporting

The application test running tool will capture a series of response times for each transaction executed. The most typical report for a test run will summarise these measurements and for each measurement taken the following will be reported:

- The count of measurements.
- Minimum response time.
- Maximum response time.
- Mean response time.
- 95th percentile response time.
The 95th percentile, it should be noted, is the time within which 95 percent of the measurements occur. Other percentiles are sometimes used, but this depends on the format of the response time requirements. The required response times are usually presented on the same report for comparison.

The other main requirement that must be verified by the test is system throughput. The load generation tool should record the count of each transaction type for the period of the test. Dividing these counts by the duration of the test gives the transaction rate or throughput actually achieved. These rates should match the load profile simulated - but might not if the system responds slowly. If the transaction load rate depends on delays between transactions, a slow response will increase the delay between transactions and slow the rate. The throughput will also be less than intended if the system simply cannot support the load applied.

It is common to execute a series of test runs at varying load. Using the results of a series of tests, a graph of response time for a transaction plotted against the load applied can be prepared. Such graphs provide an indication of the rate of degradation in performance as load is increased, and the maximum throughput that can be achieved, while providing acceptable response times.

Where a test driver is used to submit SQL statements to the database server across the network, the response times of each individual SQL statement can be recorded. A report of SQL statements in descending order of response time is a very good indicator of those SQL statements which would benefit from some optimisation and database tables or views which may not have been correctly defined (e.g. indices not set up).

Resource monitoring tools usually have statistical or graphical reporting facilities which plot resource usage over time. Enhanced reports of resource usage versus load applied are very useful, and can assist identification of bottlenecks in a system architecture.

3. TOOLS AND TECHNIQUES FOR EXECUTING PERFORMANCE TESTS

The Performance Testing Toolkit mentioned earlier, identified the range of features required of the tools used in conducting performance tests. In this section we concentrate on test execution and describe in more detail the options open to the performance testing team and the trade-offs which are inevitable.

3.1 Test Architecture

Figure 4 presents a simple schematic of the architecture of a performance test.

We can identify three major components of functionality required to execute a test:

- Client application running.
- Load generation.
- Resource monitoring.
3.2 Client Application Running

The first component required is a tool to execute selected system transactions and measure the response times, as they would be experienced by the user. There are many tools on the market nowadays, often referred to as Capture-Replay or Test Running tools which have powerful test script recording and replay features. They usually have an associated test script language which can be used to customise the simple test scripts. Typically, scripts are amended to make them ‘data driven’. This ensures they are repeatable and also that they input a variety of data during a test run.

Client Application Running tools can co-reside on the client hardware or require their own separate host. The two configurations are presented in Figure 5. A comparison of the two types is made in Table 3.
### Table 3. Comparison of Client-Resident and Separate Tool Host Client Application Running Tools.

<table>
<thead>
<tr>
<th>Client Resident Tool</th>
<th>Separate Host Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Most common on PCs.</td>
<td>• Available for both PCs and UNIX (X workstations).</td>
</tr>
<tr>
<td>• Intrusive on client, may absorb considerable amount of client resources.</td>
<td>• Not intrusive on client (although some tools require ‘agents’ to be present on the SUT).</td>
</tr>
<tr>
<td>• Not intrusive on network.</td>
<td>• Intrusive on network as all messages pass between tool host and SUT.</td>
</tr>
<tr>
<td>• Easy to use. Only one screen, keyboard, mouse required.</td>
<td>• More complicated to use as two machines are required.</td>
</tr>
<tr>
<td>• Usually geared to single station functional tests.</td>
<td>• Usually geared to performance tests.</td>
</tr>
<tr>
<td>• Usually lack facilities to regulate load and measure response times.</td>
<td>• Usually include facilities for load regulation and response time measurement.</td>
</tr>
</tbody>
</table>

### 3.3 Load Generation

Figure 6 below presents a schematic representation of a C/S architecture with an indication of where load generation tools might insert transactions to simulate a load. All load generators operate at a certain layer within the architecture.

![Figure 6. Load Generation and C/S Architecture.](image)

We can identify three categories of load generation tools based on the level at which they subject the architecture to a load:

- **User Emulation** - These tools simulate users and operate at the highest GUI layer. They exercise the complete architecture, but test scripting can be complicated. Proprietary test running tools operate at this level.

- **Test Drivers** - These tools simulate clients by submitting client messages across the network to the database server and operate at an intermediate layer. Not all architectural layers are exercised, but test scripting is often simpler. This option obviates the need for
many client workstations but requires its own host (usually a server). This configuration stresses the network, but may not provide a realistically distributed load.

- **Server Based Processes** - which subject the database on the server to prepared SQL bypassing most of the architecture, including the network. Although simple and inexpensive, such tests simulate the load on the server only and are useful for database tuning purposes. This type of load generation is suitable for host based systems but is losing popularity as a C/S test technique. It will not be discussed further.

Table 4 presents an outline of the issues to consider when selecting a Load Generation solution.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Consideration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Costs/availability</td>
<td>- Are tools for your preferred hardware/operating system platform available?</td>
</tr>
<tr>
<td></td>
<td>- Is tool functionality adequate?</td>
</tr>
<tr>
<td></td>
<td>- Cost of tool acceptable?</td>
</tr>
<tr>
<td></td>
<td>- Cost of test tool hosts (if required) acceptable?</td>
</tr>
<tr>
<td>Complexity and skills</td>
<td>- Are the tools difficult to learn, configure, use?</td>
</tr>
<tr>
<td></td>
<td>- Are skills available internally or in the market at all?</td>
</tr>
<tr>
<td></td>
<td>- Is specialist knowledge of proprietary middleware or other messaging</td>
</tr>
<tr>
<td></td>
<td>technologies required?</td>
</tr>
<tr>
<td>Simulation realism</td>
<td>- Required architectural components exercised?</td>
</tr>
<tr>
<td></td>
<td>- Required distribution of load possible?</td>
</tr>
<tr>
<td></td>
<td>- Required load volumes possible?</td>
</tr>
<tr>
<td>Tool intrusiveness</td>
<td>- Intrusive on client?</td>
</tr>
<tr>
<td></td>
<td>- Intrusive on servers?</td>
</tr>
<tr>
<td></td>
<td>- Intrusive on network?</td>
</tr>
<tr>
<td></td>
<td>- Is it possible to measure intrusiveness and compensate?</td>
</tr>
</tbody>
</table>

**Table 4. Load Generation Considerations.**

### 3.3.1 Load Generation via User Emulation

Figure 7 presents a schematic of load generation performed using a tool which emulates user activity. Usually, the tool host communicates with a process running on the workstation and indirectly with the client user interface to execute the application under test and prepared system transactions. The tool host waits for the responses as seen by the user and (usually) captures response times.

In an X Windows environment, commercial tools can also simulate X servers. The X server is really a process which runs on the workstation or a server, if X terminals are used. If the tool can simulate the X server, it can also simulate ‘pseudo’ X terminals, so the number of terminals actually driven in the test need no longer be limited to the number of physical workstations available. Certainly, for simple X terminal this is a very effective solution.

Where client software resides on an X workstation, however, a host for the client application process(es) must be found. For a simple application, it might be possible to establish more than one instance of the application process on a single physical workstation, but although this may enable the architecture to be fully exercised, response times for the multiple applications running on the overloaded workstation will no longer be realistic.
All results are captured on the tool host and most tools provide facilities for analysis and reporting of results.

3.3.2 Load Generation Using Test Drivers

Test drivers are used when it is impractical to build a test environment with sufficient numbers of client workstations or PCs, and where commercial tools cannot provide ‘pseudo’ terminal facilities. This situation is most common in PC environments. Tools are only just emerging which (it is claimed) can simulate a user’s interaction with a PC user interface. Figure 8 presents a schematic of load generation using a test driver based on a separate host. In essence, a test driver simulates one or more clients by issuing prepared messages across the network to application or database servers.

The test driver requires its own host to operate and needs to be networked to simulate the required client activity. For small scale tests a PC and remote system management software might be appropriate. For larger tests where many clients must be simulated, a UNIX server would be required. In a large project with many servers, it might be possible to borrow a spare server for the duration of the test.

A simple test driver program, capable of submitting SQL or remote procedure calls usually has limited functionality, but the most essential requirements are to:

- Establish a network connection.
- Read a ‘standard input’ or a prepared data file for commands or the messages to be transmitted.
- Acknowledge responses from the remote server.
- Be configurable to send messages at pre-defined rates.
Other requirements, where a more sophisticated driver is required would be to:

- Record response times for the transactions submitted to the server.
- Clone itself and synchronise between different instances, where the need is to establish hundreds of dummy clients.

If a simple harness is built, it can be complemented by an automated test running tool. The test tool can be used to drive the required number of test harness processes and control the rate at which transactions are submitted. An additional server may be required to host the test tool if it does not reside on the same machine as the test driver software.

![Diagram showing load generation using test drivers.](Figure 8. Load Generation Using Test Drivers.)

### 3.4 System Resource Monitoring

During the tests, every resource monitoring tool at your disposal should be used. The reason for this is simple. The only time that system monitoring can be usefully performed, prior to release into production, is during the performance tests and performance tests are expensive. As a rule, it is better to capture statistics and discard them if not required, than not to have them at all. A full complement of tools and the resources they monitor would include:

- **PC client monitor** - Windows and DOS memory resources, disk space, loaded DLLs, TSRs.
- **UNIX clients and servers monitors** - CPU usage and i/o by process, free memory, page swap rates.
- **Network monitor** - at a high level network usage v bandwidth, at a low level individual data packets might be traced for a detailed analysis.
- **Database monitor** - processes accessing the database, locks requested and granted, deadlocks, logical and physical i/o.

One other aspect, worth mentioning at this point, is that of instrumentation. Application code and middleware can be instrumented to capture information. Typical data which can be usefully logged are:

- SQL or RPC messages sent to servers or processed by middleware and
- Response times or time-stamps for specified activities.

Instrumentation logs can be extremely useful in capturing the messages being sent across the network (for later use with test drivers) or tracking down delays in specific architectural layers.
4. PERFORMANCE TESTING IN PRACTICE

In this section we describe, using two test cases, some of the practicalities of C/S performance testing. The system used for the first test case is based on PCs running MS Windows. The other system runs in a DEC UNIX and Motif environment. These two systems formed part of a very large C/S development involving five integrated applications which the authors were called upon to performance test.

Prior to our involvement, the project had already adopted SQA Robot to test PC applications and the UNIX based PreVue-X performance testing tool from Performance Awareness Corporation used for X-Windows based applications. Although both of these tools are sophisticated with excellent facilities, some innovation and lateral thinking is always required to get a comprehensive test off the ground.

4.1 Performance Testing a Two-Tier Architecture

Most C/S systems to date have adopted two-tier architectures. The tiers refer to the simplest C/S split which usually places the user interface and application code on the client and the database (with sometimes some business logic) on the server. A generic two-tier architecture useful for our purposes is presented in Figure 11. The figure also shows an example architecture.

![Figure 9. 2 Tier Architecture](image)

As can be seen in the example, the client PCs ran MS DOS and Windows and a Powerbuilder application using an ODBC driver to access to the database. The networking was all TCP/IP based. The database servers were DEC Alphas running OpenVMS and Oracle RDB.

The example architecture is a real system designed to support 100 concurrent users which was performance tested by the authors. The application provides database maintenance, ad-hoc query and reporting facilities for a complex database (>300 tables). In this case, the client workstations were to be distributed to 24 sites and the brief was to performance test the architecture before the system was rolled out. A test environment was set up with five PCs.
configured to run the system to be used for both functional and performance testing and a
dedicated database with a large volume of data on a production server.

The main problem to be overcome here was that only a handful of PCs were available for
testing so a ‘User Emulation’ approach to load generation was not practical. The approach
adopted can be summarised as follows:

- SQA Robot was used to record automated scripts to drive the application on individual
  PCs.
- A test driver program, running on a UNIX server, would be used to simulate the other
  clients.
- The UNIX PreVue testing tool would be used to execute the required number of test
  driver instances to generate the required load.

A comprehensive set of load profile and response time requirements already existed and was
used to specify the load to be simulated and the test transactions to be executed and
measured. A large volume of test data was already available as there was a continuing project
to capture and convert data to be loaded in the database, so a production-sized database was
readily available.

The test was, in simple terms, developed in seven stages:

1. SQA Robot has no facilities to take performance measurements, so some custom routines
   were developed in Visual Basic to capture response times and log them in a defined
   format. A general-purpose test harness was also developed allowing the configuration of
tests to be controlled using standard MS Windows initialisation files.

2. An MS Access database was developed to hold the test results and produce reports. It was
   anticipated that a large number of tests would be performed over time, so we decided to
   make report-generation as automated as possible.

3. SQA Robot test scripts were recorded which covered all the transactions which would
   ultimately comprise the required load profiles. The test scripts were customised to make
   them ‘data-driven’ and the response time measurement routines called at the required
   points in the script.

4. The SQA Robot scripts were executed with ODBC logging turned on. ODBC logs
   contain all the ODBC commands being processed by the driver and can be used to
capture the SQL statements being generated by the application. A fragment of an ODBC
   log is shown in Figure 10. Using the DOS FIND command and some MS Word macros,
it was possible to extract the SQL statements for each test script in a few minutes.

5. Sample code supplied with the ORACLE RDB SQL Server product was used to create a
   simple test driver program to run under UNIX. The program had a command line
   interface which could submit SQL statements across the network using identical message
   formats as the PC clients.

6. PreVue scripts were used to build a suite of tasks, each being driven by a different
datafile containing the SQL statements to be executed. The PreVue scripts were enhanced
to provide a simple parameterising facility to make them data driven. This was essential if
the scripts were to be repeatable and not to conflict with each other.

7. Finally, the tests were conducted using PreVue to generate the background load, and
   SQA Robot running test transactions and taking response time measurements.
Figure 10. Extract of an ODBC log file.

All the results (both PreVue statistics and SQA Robot measurements) were loaded into the Access database and reports produced. The SQL used to generate the background load was also loaded into the Access database and each SQL statement (over 700) could be matched with the response times as seen by PreVue when the statement was executed automatically. In this way, the worst performing SQL could be identified, optimised, and re-coded in the application.

Tests revealed many instances of badly designed SQL which was subsequently optimised. Several statements taking minutes were improved to be sub-second. Several incorrect views were identified (which also did not impose optimising techniques). Some table indexes which had not been set up were also identified and put right.

4.2 Performance Testing a Three-Tier Architecture

Recently, the three-tier architecture has begun to emerge as the architecture of choice for large scale C/S systems. In this case, the client-tier is where the user interface and application code resides; the middle tier executes business logic, caches commonly used data and optimises calls to the database server; the third tier is the database server which of course hosts the database.

A generic three-tier architecture is presented in Figure 11. The figure also shows an example architecture.

As can be seen in the example, the client workstations ran UNIX and the Motif GUI and an application written in C++. The application sends Remote Procedure Calls to the Application Server, which processes some calls locally, but in many cases, makes calls to the database server using Dynamic SQL calls to the database server. This system is integrated with the 2-tier system described above and shares the same database servers.

This architecture is a system designed to support 70 concurrent users. The application provides a graphical display of a large regional utility network and allows real-time access to a large plant records database combined with digital drawings produced by a GIS system (not considered here). In this case also, the client workstations were to be distributed to 24 sites and the brief was to performance test the architecture before the system was rolled out. A test environment was set up with two workstations configured to run the system to be used for both functional and performance testing and the same database as before.
Fortunately, in the same building as the test centre, a training room with additional workstations was available for testing. The nature of the application is that users make very deliberate use of the system and do not generate a large volume of transactions. It was considered a reasonable compromise to simulate 70 active users on a smaller number of workstations operating more quickly. The approach adopted was to use the PreVue-X UNIX performance testing tool to drive both physical and some pseudo workstations to simulate the load profiles required. PreVue-X also has comprehensive response time measurement and results analysis utilities and these were also used for reporting.

As before, a comprehensive set of load profile and response time requirements already existed and were used to specify the load to be simulated.

Although the three-tier architecture under test is more complicated, the test process was actually simpler than that used for the 2-tier test. The test followed 3 stages:

1. PreVue-X test scripts were recorded which covered all the transactions which would make up the required load. The load comprised a small number of high-use transactions.

2. The PreVue-X workload definition was developed in stages and eventually to generate the complete load profile required.

3. Finally, the tests were executed using PreVue tool and utilities.
The MS Access database developed to hold the 2-tier system test results was re-used. All the results were loaded into the Access database as before and reports produced. As part of this exercise, significant use was made of middleware instrumentation to assist diagnoses of poor response times.

5. PERFORMANCE TESTING PITFALLS AND GUIDELINES

In planning, preparing and executing performance tests, there are several aspects of the task which can cause difficulties. The problems that are encountered most often relate to the software and environment. The predominant issue that concerns the performance tester is stability. Unfortunately, performance testers are often required to work with software that is imperfect or unfinished. These issues are discussed and some practical guidelines are proposed.

5.1 Software Quality

In many projects, the time allowed for functional and non-functional testing (including performance testing) is ‘squeezed’. Too little time is allocated overall, and developers often regard the system test period as ‘contingency’. Under any circumstance, the time allowed for testing is reduced, and the quality of the software is poorer than required.

When the test team receive the software to test, and attempt to record test scripts, the scripts themselves will probably not stretch the application in terms of its functionality. The paths taken through the application will be designed to execute specific transactions successfully. As a test script is recorded, made repeatable and then run repeatedly, bugs which were not caught during functional testing may begin to emerge.

One typical problem found during this period is that repeated runs of specific scripts may gradually absorb more and more resources on the client, leading to a failure, when a resource, usually memory, runs out. Program crashes often occur when repeated use of specific features within the application causes counters or internal array bounds to be exceeded. Sometimes these problems can be bypassed by using different paths through the software, but more often, these scripts have to be postponed until the software errors can be fixed.

5.2 Software Maturity

C/S technology is often used in conjunction with a Rapid Application Development (RAD) method. RAD methodologies are gaining in popularity with development organisations. This may be good news for the developers who are released from much of the formality of the established structured methodologies, but is often bad news for testers. Some RAD projects encourage the ‘code a little, test a little, release a little’ approach. Sometimes, the software is never actually ‘finished’ and may undergo last minute changes well into the System Test period.

For testers attempting to record automated test scripts, changing software is their worst enemy. A software release may fix errors and make it possible to record a new scripts, but it may, by introducing changes in functionality or the user interface, make existing scripts useless. Functional testers may be able to accommodate daily or weekly releases of code which fix problems. The testers creating automated scripts probably won’t.

Fortunately, it is sometimes possible for the performance testers not to have to keep test scripts up to date with every release of the application software. If the changes being made and released fix functional errors, the performance of the software to be tested is probably not affected so it may be acceptable not to accept that version of software and wait for a later
release. Of course, if a release fixes a problem which will now allows a test script to be created, or the release fixes a performance problem, new versions should be accepted.

5.3 Configuration Management

In a traditional development project, it is usually only the application software which is released into the test environment. Changes are usually made to fix specific problems and come from a single source. The Configuration Manager in most cases need only copy files to specified locations on the test machine and the release is done.

In a C/S environment, the job of the Configuration Manager is more complicated. Software configuration changes span database schema changes, server middleware, client middleware as well as client application software. Further, many client applications, particularly on PCs, require local initialisation or configuration files to be installed. Altogether, C/S installations have more software components, from more suppliers to be installed in more, possibly remote, locations.

Without very good installation records and version control, assisted by system management tools, configuration management issues can be a major headache. From the point of view of the tester, (and in our experience) it is worthwhile keeping one’s own records of the version numbers of every component in the test environment.

5.4 Dedicated Environment

During test preparation, testers will be recording, editing and replaying automated test scripts. These activities should not disturb or be disturbed by the activities of other users on a shared system. However, when a single test script is integrated into the complete workload and the full load simulation run, other users of the system will probably be very badly affected by the sudden application of a such large load on the system.

If at all possible, the test team should have access to a dedicated environment for test development. It need hardly be stated, that when the actual tests are run, there should be no other activity on the test environment.

5.5 Other Potential Problems

Underestimation of the effort required to prepare and conduct a performance can lead to problems. Performance testing a C/S system is a complex activity, which usually has to be completed in a very limited timescale. Few project managers have direct experience of the tasks involved in preparing and executing such tests. As a result, they usually underestimate the length of time it takes to build the infrastructure required to conduct the test. If this is the case, tests are unlikely to be ready to execute in the time available.

Over ambition, at least early in the project, is common. Project managers often assume that databases have to be populated with valid data, that every transaction must be incorporated into the load and every response time measured. As usual, the 80/20 rule applies: 80% of the database volume will be taken up by 20% of the system tables. 80% of the system load will be generated by 20% of the system transactions. Only 20% of system transactions need be measured. Experienced testers would probably assume a 90/10 rule. Inexperienced managers seem to mix up the 90 and the 10.

Using tools to execute automated tests does not require highly specialised skills, but as with most software development and testing activities there are principles which, if adhered to, should allow reasonably competent testers to build a performance test. It is common for
managers or testers with no test automation experience, to assume that the test process consists of two stages: test scripting and test running. As should be clear to the reader the process is more complicated and actually is more akin to a small software development project in its own right. On top of this, the testers may have to build or customise the tools they use.

When software developers who have designed, coded and functionally tested an application are asked to build an automated test suite for a performance test, their main difficulty is their lack of testing experience. Experienced testers who have no experience of the SUT however, usually need a period to familiarise themselves with the system to be tested. Allowance for this should be made as in the early stages of test development, testers will have to grapple with the vagaries of the SUT before they can start to record scripts.

Building a performance test database involves generating thousands or millions of database rows in selected tables. There are two risks involved in this activity. The first is that in creating the invented data in the database tables, the referential integrity of the database is not maintained. The second risk is that business rules, for example, reconciliation of financial fields in different tables are not adhered to.

In both cases, the load simulation may not be compromised, but the application may not be able to handle such inconsistencies and fail. In these circumstances, test scripts developed on a small coherent database will no longer work on a prepared production size database. Clearly, it is very helpful for the person preparing the test database to understand the database design and the operation of the application.

This problem can of course be helped if the database itself has the referential constraints implemented and will reject invalid data (often, these facilities are not used because they impose a significant performance overhead). When using procedural SQL to create database rows, the usual technique is to replicate existing database rows with a new unique primary key. In most cases this method will work satisfactorily, but is not guaranteed in all situations.
6. CONCLUSION

This paper has set out the reasons why system performance is a risk to the success of C/S application development projects. A process has been outlined which the authors have used to plan, prepare and execute automated performance tests. The principles involved in organising a performance test have been set out and an overview of the tools and techniques that can be used for testing two and three-tier C/S system presented.

Performance testing C/S systems is a complex activity and there are many practical problems (many of which will be familiar to functional testers). People with performance testing skills are not yet plentiful, although a small number of consultancies and the tool vendors, of course, can provide help.

In summary we can make the following conclusions:

- Automated performance tests are expensive to build, but are an extremely valuable asset.
- Database, system and network administrators cannot create their own tests, so should be intimately involved in the staging of all tests to maximise the value of the testing.
- There are logistical, organisational and technical problems with performance testing - many issues can be avoided if the principles outlined here are recognised and followed.
- The approach to testing two and three-tier systems is similar, although the architectures differ in their complexity.
- Proprietary test tools help, but improvisation and innovation is often required to make a test ‘happen’.
- Tools which can eliminate the need for custom-written test drivers are beginning to emerge.