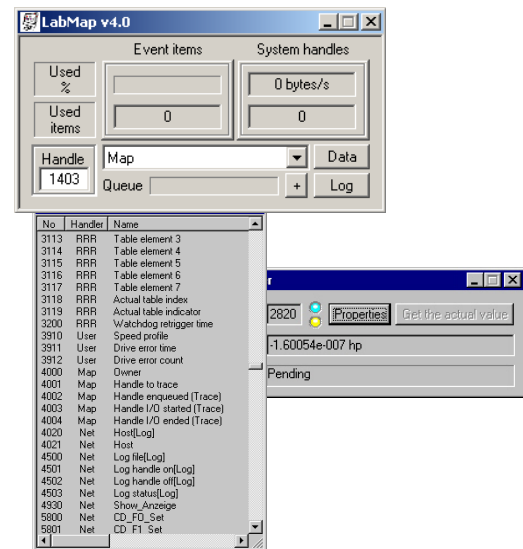
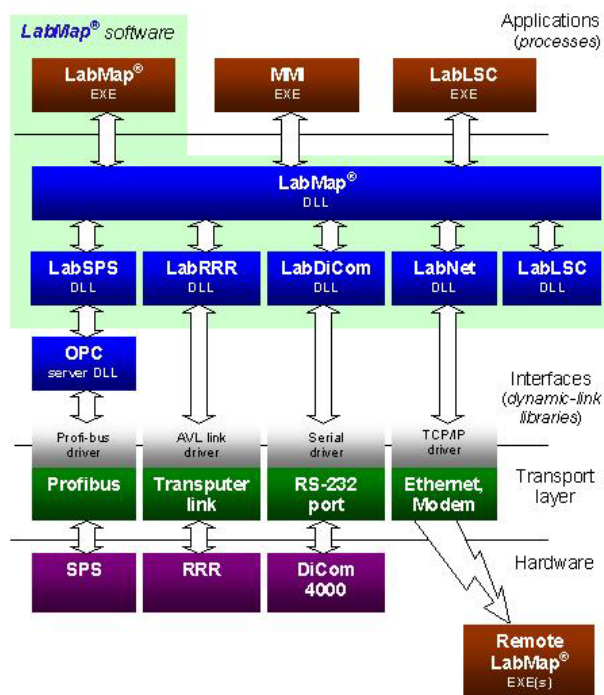


Benchmarking of LabMap[®]



Authors:

Dr. D. Kazakov,
cbb software GmbH, Lübeck, Germany

Mr. Osei-Agyemang Jr.,
Kwame Nkrumah University of Science and Technology, Kumasi, Ghana

April 2004

Table of Contents

Table of Contents	1
List of Tables, Figures and Code Listings.....	3
Chapter 1 Introduction	4
1.1 Overview of a typical control system.....	4
1.2 Overview of LabMap [2].....	4
1.3 Distributed Control systems with LabMap’s network interface	6
1.4 Exploring LabMap Alternatives.....	6
1.4.1 Common Object Request Broker Architecture CORBA.....	6
1.4.2 OLE for Process Control (OPC).....	7
1.4.3 RTPS Middleware (NDDS) promoted by IDA	7
1.5 Advantages of LabMap	7
Chapter 2 Analysis and Design.....	9
2.1 Technical Overview of LabMap	9
2.1.1 Data types supported by LabMap.....	9
2.1.2 Communicating with LabMap [2].....	9
2.1.3 LabMap and its network interface, LabNet.....	10
2.2 Factors that affect the performance of LabMap in a distributed control system.....	10
2.3 Determining Round Trip Times	11
2.3.1 Timing the Round Trip times	11
2.3.2 Determining the time taken to execute QueryPerformanceCounter	12
2.4 Data sending.....	13
2.4.1 Case I: Client sending data to server.....	13
2.4.2 Case II: Data sent from Server to Client	15

Chapter 3	Procedure.....	16
3.1	Experimental Set up	16
3.2	Registers for the tests	16
3.2.1	User defined Registers [1].....	16
3.2.2	Net Registers	16
3.2.3	Creating the Registers	17
3.2.4	Other registry settings [1].....	20
3.2.4.1	Server configuration.....	20
3.2.4.2	Client configuration.....	20
3.3	The process procedure.....	21
3.4	Measuring Overwrites	29
Chapter 4	Results and Discussions	30
4.1	Effect of number of handles on performance.....	30
4.2	Effect of timer interval on performance.....	32
4.3	Effect of data type on performance	32
Appendix A	Results of experiment.....	33
A.1	Results for integers.....	33
A.2	Results for real numbers.....	34
A.3	Results for Strings	35
Appendix B	Sample Charts of data obtained.....	36
B.1	3-D surface for the Local Round Trip Time for the integer data type	36
B.2	3-D surface for the Mean Remote Round Trip Time for the integer data type.....	37
Appendix C	References	38

List of Tables, Figures and Code Listings

Figure 1.1	LabMap middleware decoupling the hardware layer and control applications .	5
Figure 2.1	Data sent to server and notification received from client.....	11
Figure 2.2.	Queuing process in client	13
Figure 2.3	Client sending data from queue to server.....	14
Figure 3.1	Data sending and callback notification procedure	23
Figure 3.2	Array to hold times.....	25
Figure 3.3	Timer and Callback procedure	28
Figure 4.1	Chart of Mean RTT in ms for integer data type at timer interval of 10 ms	30
Figure 4.2	Chart of Mean RTT in ms for integer data type at timer interval of 50ms	31
Figure 4.3	Chart of Mean RTT in ms for integer data type at timer interval of 100ms	31
Table 2.1	Results obtained from tests to determine time taken to execute the QueryPerformanceCounter	12
Listing 3.1	Code to delete existing handles.....	17
Listing 3.2	Code to create handles.....	19
Listing 3.3	Code to allocate memory of dynamic arrays.....	25
Listing 3.4	Timer procedure	26
Listing 3.5	Callback procedure.....	26

Chapter 1 Introduction

This paper outlines a study performed to measure and evaluate the performance in terms of speed of LabMap, as a middleware for control processes. This is to give system implementers and developers an idea of how a long it takes for a response to a change in a system variable to be received. The study is mainly based on the performance of LabMap middleware in a typical distributed system.

1.1 Overview of a typical control system

Most modern control processes are such that all Input/Output operations are not centralised at one particular location. The Input/Output operations are said to be distributed. Contrary to the Input/Output operations, the control of the system is usually centralised or may be implemented with a cluster of centralised control points. The Input/Output operations are therefore connected to the centralised control system via a communication network. Most delays in the whole system can be attributed to communication delays. The whole system including the communication network, communication with the hardware and the communication protocol is usually referred to as the Software Bus. There are many implementations of software buses that are in use in typical control processes. Examples include

- NICAN from National Instruments
- ModBus
- DiCOMM
- Profibus

1.2 Overview of LabMap [2]

LabMap is a middleware that abstracts the hardware layer from the control application. There are many hardware devices and software buses on the market presently, each with its own proprietary communication protocol. Since there is no standard interface that allows easy interoperability among them, application developers are saddled with the problem of developing an interface or drivers for each hardware device or software bus.

What LabMap does is to provide an interface such that the control application development is decoupled from the hardware layer. The application communicates with the hardware through registers and LabMap carries out the actual sending of the data through the appropriate protocol and interfaces for the given hardware.

Though the purpose of LabMap is mainly to hide the hardware from the control application, it goes further to abstract the location of the hardware involved, with its network module, LabNet, such that the control application communicates with remote hardware as if it communicates with a local hardware. LabMap therefore makes the implementation of control systems easier by separating the control application development from the hardware assembly. This is illustrated in Figure 1.1.

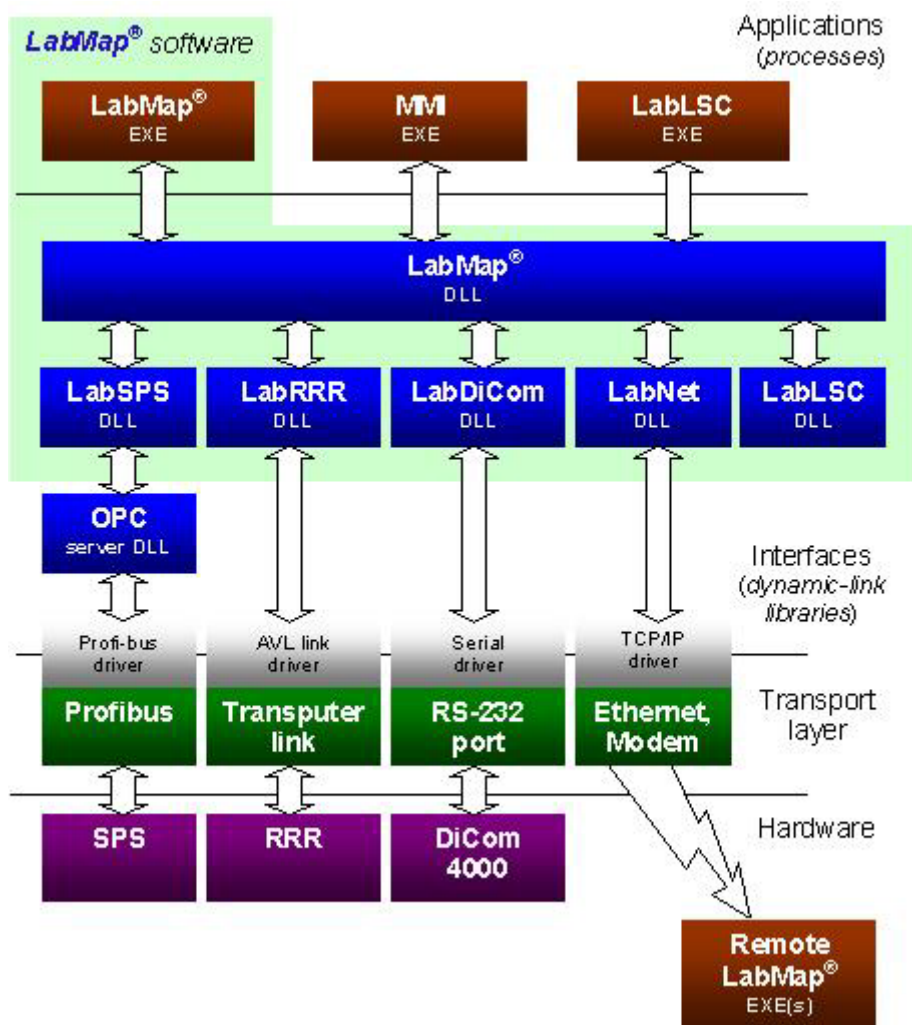


Figure 1.1 LabMap middleware decoupling the hardware layer and control applications

A change in the hardware or its location does not mean a change in the control application but just a reconfiguration in LabMap

1.3 Distributed Control systems with LabMap's network interface

LabMap's network interface is implemented via Windows Sockets communicating with Transmission Control Protocol/Internet Protocol (TCP/IP). Direct socket programming has been known to result in very efficient applications. Although socket programming has been known to be very low-level and quite a daunting task for complex applications, LabMap does a good job of hiding all the complexities involved. LabMap goes further to encapsulate all the low-level communication process such that communication with a remote hardware is like communicating with a local hardware.

1.4 Exploring LabMap Alternatives

When designing and implementing distributed control systems, LabMap certainly isn't the only choice. Other alternatives exist by which such systems can be built. Depending on the nature of the system - ranging from its complexity to the platform(s) it runs on -there are a number of alternatives to consider. The following describe some of the known alternatives.

1.4.1 Common Object Request Broker Architecture CORBA

CORBA is an object-oriented architecture promoted by the Object Management Group (OMG). CORBA is a standard used mainly for implementing distributed applications and systems [6]. Some research has gone into implementing CORBA for distributed control systems. Further research has gone into implementing a real-time version of CORBA. CORBA is cross-platform and can be implemented in any of the many programming languages that support CORBA. It's cross-platform nature makes it very viable when considering systems that will run across Unix/Linux, Sun Solaris and Microsoft Windows platforms. This is one of the main strengths of CORBA.

1.4.2 OLE for Process Control (OPC)

OPC is based on Microsoft's (Distributed) Component Object Model (COM/DCOM) architecture. OPC consists of a standard set of interfaces, properties and methods for process control and provides a common interface for communicating with diverse process-control devices, regardless of controlling software or devices in the process. DCOM, the underlying technology for OPC, can be said to be a highly-optimised protocol that extends COM to networks thus making it possible for remote objects to be accessed as if they were local [4]. Compared to CORBA, its limitation will be in the fact that DCOM is a Microsoft Windows technology and OPC applications are somehow limited to the Microsoft Windows platforms. Work is currently going on involving some software concerns as Software AG (Darmstadt, Germany) trying to extend DCOM to non-Microsoft platforms.

1.4.3 RTPS Middleware (NDDS) promoted by IDA

Network Data Delivery Service (NDDS) is network middleware for distributed real-time applications [5]. It is backed by Interface for Distributed Automation (IDA). The NDDS architecture is based on the publish-subscribe model for data distribution. Unlike LabMap, OPC and CORBA, it uses TCP/UDP protocol. The network traffic is hence greatly reduced and efficiency is increased.

1.5 Advantages of LabMap

Some of the advantages of LabMap are as follows;

i. Ease of custom application development

Client application (example SCADA, HMI) developers have full access to system data through LabMap and do not need to have full knowledge of the industrial network, the kind of hardware used, where which hardware will be placed and possible changes in the system. They can therefore focus on application functionality instead of device connectivity.

ii. Maintenance of hardware

The maintenance of hardware is made easier. Defective hardware can easily be replaced with an upgraded hardware or hardware from a different vendor without a

change in the client applications but just a reconfiguration of the LabMap middleware.

iii. Expandability and flexibility of systems

Systems developed with LabMap are very scalable and flexible. New hardware can easily be added by describing the new hardware in terms of system variables.

iv. Vendor lock in is eliminated

Different hardware from different vendors can all be integrated in a single control system (A very important feature if one wants to avoid vendor lock-in). The system implementer is therefore at will to choose any kind of hardware that he/she deems fit.

v. High Efficiency of LabMap

LabMap implements a very thin protocol on top of TCP/IP. This coupled with the fact that LabMap is multithreaded makes it highly efficient.

vi. Simple configuration of LabMap

Configuration of LabMap is very simple and therefore reduces troubleshooting and maintenance times.

vii. Ease of application development

Applications based on LabMap are easy to develop. They are also easy to debug since LabMap represents data as system variables. LabMap client applications can be developed with any programming tool that can use Windows Dynamic Link Libraries like C/C++, Delphi and Visual Basic. Through Java's native methods, client applications can also be developed for Windows OS.

viii. Future Enhancements to LabMap

cbb Software GmbH intends to extend LabMap to many other application areas. Work is currently going on in developing Embedded LabMap (LabMap for embedded systems).

Chapter 2 Analysis and Design

2.1 Technical Overview of LabMap

LabMap is a middleware that is used to transmit data. The data transmitted represents the state of variables in the system. LabMap uses what are called handles or registers to represent these state variables. Control applications can therefore be said to communicate with LabMap through a set of variables (registers) called handles. Applications developed on LabMap respond to changes in these state variables and react accordingly. It can therefore be said that data is transmitted through these registers. The implementation of these registers has been encapsulated such that a register may represent the state variables that are available on a remote computer. Data transmission to and from a remote register is the same as it is for a local register. Each register has a type, value, timestamp and Input/Output direction.

2.1.1 Data types supported by LabMap

There are three main data types used to send data through LabMap and these are:

- Integers (4-byte integer)
- Real Numbers (corresponding to 4-bytes floating point numbers)
- String (usually sent as a pointer to a null-terminated array of characters)

2.1.2 Communicating with LabMap [2]

There are four basic I/O requests supported for each register:

- **Get.** The value of a register can be read in a safe way that warrants consistency of the read value bits and the timestamp. The application is relieved from the burden of locking the value in presence of other tasks accessing it concurrently.
- **Set.** The value of a register can be set in a safe way. No I/O initiated by this request.
- **Request** (for output registers only). A new value of a register can be requested from the underlying hardware. The application does not know which actions are necessary to request the new value. It is the responsibility of the driver. The request is asynchronous and the application is not blocked until I/O completion. However it may enter a non-busy waiting for the I/O completion.

- **Send** (for input registers only). The actual value of a register can be sent to the underlying hardware. Like in the case of **request** the application is unaware of the actions the hardware undertakes upon **send**. The application may enter a non-busy waiting for the I/O completion.

LabMap exposes an Application Programmer Interface (API) for communicating with it. The three main LabMap functions used for data transfer operations between a client application and a remote server application in the tests performed were:

- LabMapSendInt
- LabMapSendReal
- LabMapSendString

2.1.3 LabMap and its network interface, LabNet

The remote access interface is provided by the LabNet.dll. LabMap uses a client/server architecture for communication. The client requests data from the server and the server provides (serves) the client with the data.

2.2 Factors that affect the performance of LabMap in a distributed control system

There are many variables that come into play when performance of LabMap and other network applications in general are concerned. The main factors to be considered are the effect of the following on overall performance of LabMap

1. Number of handles to be updated
2. Number of client applications
3. Network/ Bandwidth factor
4. Operating system dependencies
5. Data type being transferred
6. The rate at which data in the registers needs to be updated

2.3 Determining Round Trip Times

The Round Trip Time can be said to be the time it takes for LabMap to send data to a remote server and back. Data was sent in the form of a new value to a remote register through LabMap and the time taken for a callback notification to be received is computed as the Round Trip Time. This is as shown in Figure 2.1.

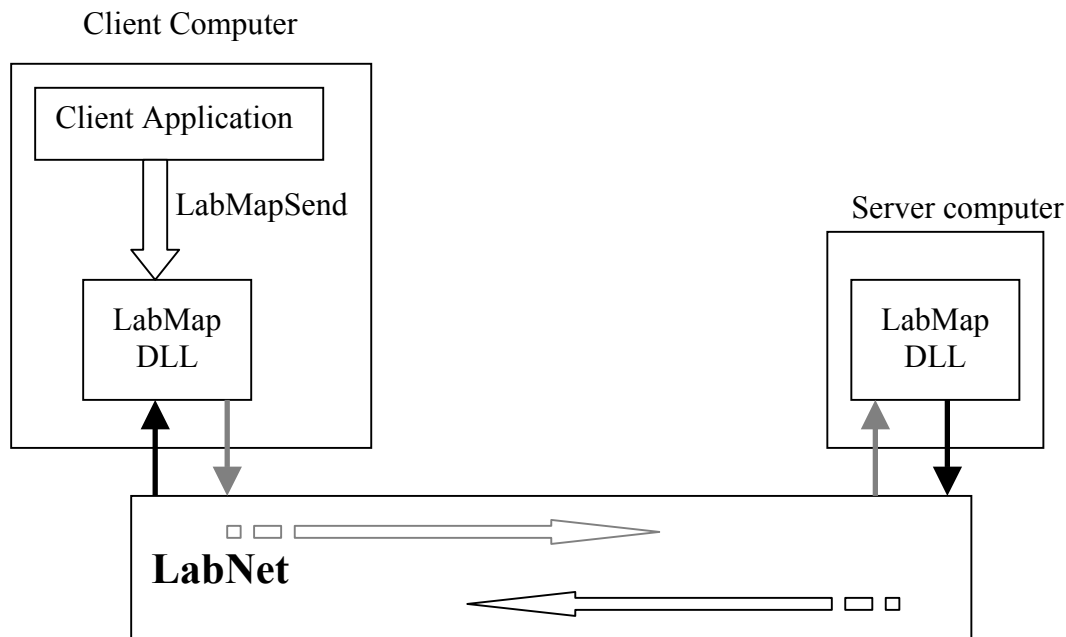


Figure 2.1 Data sent to server and notification received from client

2.3.1 Timing the Round Trip times

Timing of the LabMap Round Trip Times had to be carried out using the most accurate means possible since typical time measurements were expected to be in the milliseconds range. The hardware of the systems used supported the high-resolution performance counter that provides high-resolution elapsed times. The QueryPerformanceFrequency function was used in determining the frequency of this counter and was found out to be 1193182 Hz (counts per second). The QueryPerformanceCounter function retrieves the current value of the high-resolution performance counter [3]. This function is called immediately before and after the execution of the process that is to be timed. The number of high-resolution counts can be determined by taking the difference between these two values. The time taken in seconds is then calculated by taking the product of the number of high-resolution counts and the high-resolution frequency.

$$\text{Time in ms} = \frac{(\text{Time after} - \text{Time before}) * 1000}{\text{High resolution frequency}}$$

2.3.2 Determining the time taken to execute QueryPerformanceCounter

The execution of the QueryPerformanceCounter function also takes some time. It was therefore important to find out how much time is taken by that function and whether it will have a significant effect on the measured results. Previous experiments carried out in [1] state the time to execute the QueryPerformanceCounter function as 6 microseconds.

In order to determine the time taken to execute one QueryPerformanceCounter function, the function was called a hundred times consecutively and the total time taken for those calls to be executed was calculated. The time for a single call can thus be determined by dividing that time by the number of calls. The experiment was repeated for

Number of calls, N = 1000, 10000 and 100000

The results are as tabulated in Table 2.1

	Number of function calls, N			
	100	1000	10000	100000
Total time in ms	0.49444	4.9749	49.767	509.67
Average time in ms	0.00494	0.00497	0.00498	0.00510

Table 2.1 Results obtained from tests to determine time taken to execute the QueryPerformanceCounter

The average value of 5 microseconds acquired is quite agreeable considering the fact that higher speed computers were used in this experiment. The time compared to the typical times expected from calls to LabMap is very small and it is safe enough to entirely neglect it.

2.4 Data sending

2.4.1 Case I: Client sending data to server

In a particular control system, it is desired that data from the client be sent to the server and in the order in which the data was generated. This will allow the server to act on the data the way it was meant to be. This means that the server knows of the various states which occurred within a given time period.

LabMap has a data structure that references all the registers and the data they contain. When data is sent to LabMap through a LabMapSend function for instance, LabMap first calls the corresponding LabMapSet function that sets the register to the new value. LabMap then puts the register number in a (First-In-First-Out) queue where it waits for its turn to be sent to the server. This is as illustrated using an arbitrary integer handle with number 45 in Figure 2.2.

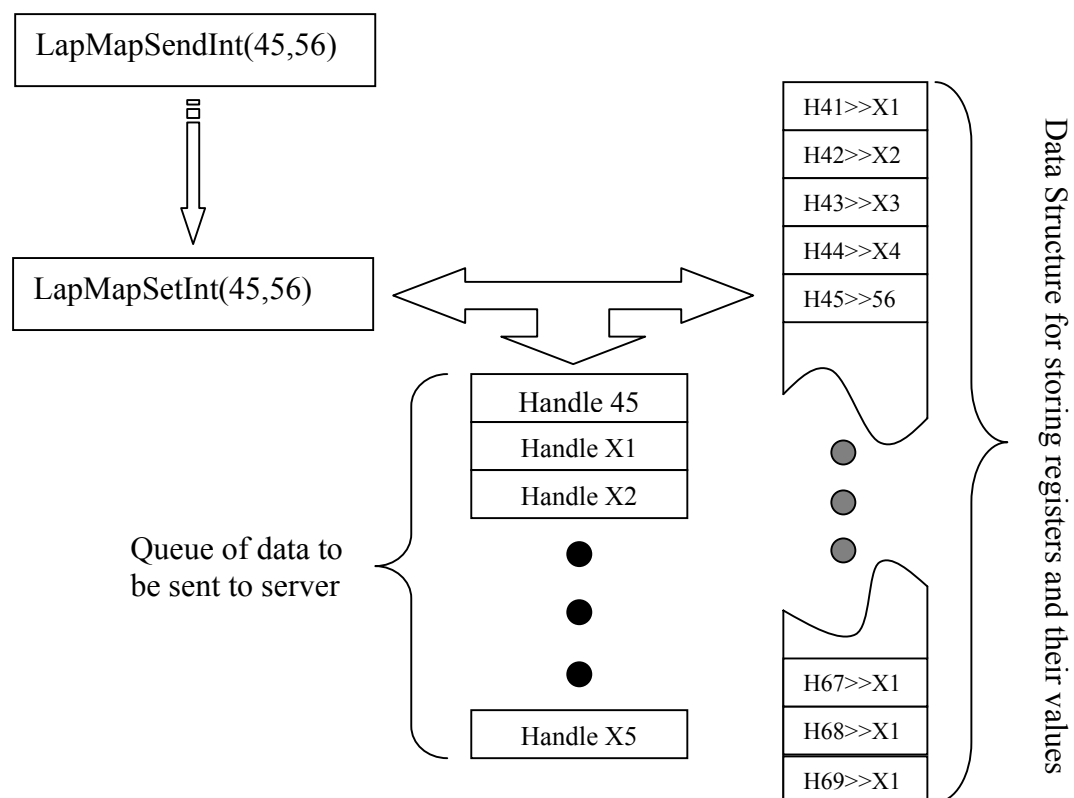


Figure 2.2. *Queuing process in client*

When network and system conditions allow, LabMap creates a data structure by adding the handle number of the register and the data it contains and sends it over the network via a Windows socket connection to the server. If for instance, within the time interval that the

register number is put in the queue and the time the data is sent, the register value is set to a new value, say 57 in this case, then the register value 57 will rather be sent. In such case, it is said that an overwrite has occurred.

This process is illustrated as in Figure 2.3.

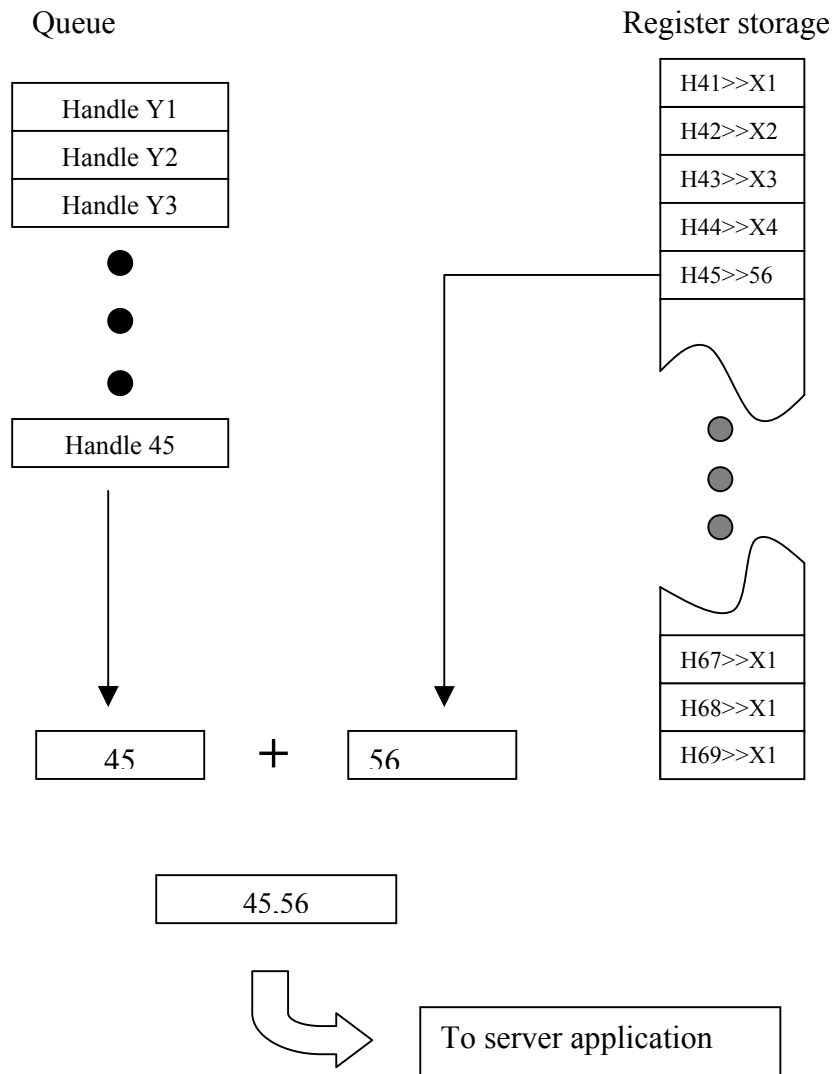


Figure 2.3 Client sending data from queue to server

2.4.2 Case II: Data sent from Server to Client

LabMap implements a slightly different approach when it comes to communication from the server to the client. In a particular control system, client applications such as HMI and SCADA applications will always want to know the most current state of the system. LabMap always sends to the clients the most current values.

On the server side, LabMap maintains a data structure for the storage of registers and their values as in the case described before in Section 2.4.1. If a register value is changed by LabMapSet for instance, LabMap sets the register value to this new value and moves the data structure representing the register into the Send Queue data structure. The main difference between the first case (data from client to server) and the second case (data from server to client) is that a handle can be represented more than twice in the send queue in the first case but only once in the latter case. This is to ensure that the server application is not overburdened with many updates. The send queue is limited in size to the number of handles in the system.

Chapter 3 Procedure

3.1 Experimental Set up

Three computers were used in the tests, all with the following specifications

- Intel Pentium III Processor of speed 1000MHz
- 256 MB Random Access Memory
- Windows NT 4 Operating System
- All connected via a 10Mb/s Local Area Network

3.2 Registers for the tests

LabMap has several kinds of registers. For the experiment, LabUser and LabNet handles were used in this experiment. LabUser handles were used on the server and LabNet handles were used on the client machine mapped to the LabUser handles on the server.

3.2.1 User defined Registers [1]

The LabUser.dll becomes the handler when User is specified as the handler for the register. There is no reaction time required for this type of registers and send and request operations are immediately satisfied. These handles were used on the server machine because the send/request queries are immediately handled and no time is spent processing the data in these handles. Thus the time for a round trip is not affected by any processing on the server.

There are no parameters involved in the creation of User handles.

3.2.2 Net Registers

On the client computers, LabNet handles were created. Net handles are sometimes referred to as remote handles because their data source is remote, the handle to which it is mapped on to. A LabMapSend request on the client machine is therefore relayed to the remote machine to update that particular remote register. LabNet handles may need additional parameters that define the particular remote handle it is mapped onto and the network name of the computer on which the remote handle is defined. An example is

[20.Log]

where 20 is the handle number of the remote register the net handle maps onto and Log is the name of the server computer where the remote register is.

In the case where a net handle maps unto a remote handle with the same handle number, the handle number of the remote register can be ignored. The parameter specified therefore is the network name of the remote computer on which the handle is defined. The example above therefore becomes

[Log]

where Log is as above and the handle number of the remote register is implicitly defined in the handle number of the handle.

In the experiment, the net handles were mapped onto handles with the same number so the latter approach was used in creating the handles.

3.2.3 Creating the Registers

In order to create the many handles used in the experiment, there had to be a small application that could prepare all the handles in question automatically. For all experiments, the lower 1000 handles were used, that is registers with Handle numbers 1 to 1000. The application first deleted all the registers within the range 1 to 1000 and then recreated them. The LabMap function used for the deletion was LabMapDelete whose only parameter is the handle number of the register to be deleted. The whole procedure is as shown in Listing 3.1.

```

var
  i, HandleStatus:Integer;
begin
  for i:=1 to 1000 do
    begin
      HandleStatus := LabMapGetStatus(i,0); //Get status of handle
      if (HandleInUse = (HandleStatus and HandleInUse)) then //If handle exists
        LabMapDelete(i); //Delete handle
      end;
      ShowMessage('Done');
    end;
end;

```

Listing 3.1 Code to delete existing handles

The function iterates from 1 to 1000 and checks if a handle exists for each iteration. The LabMapGetStatus function returns the status of a register. The value returned by the LabMapGetStatus is masked with the HandleInUse mask. Each defined handle has this bit set.

Therefore, masking the returned value with the HandleInUse will return true if the register has been defined. If the register is defined, then it is deleted.

After the register numbers have been deallocated, the application creates the handles 1 to 1000 for use in the experiment. LabMapCreate function was used to create the registers. The definition for LabMapCreate is as follows

```
function LabMapCreate
(
  HandleNo      : PLongWord,
  const Name    : PChar,
  const Server  : PChar,
  Status        : Cardinal,
  TimeOut       : Cardinal,
  Each          : Cardinal,
  const InputUnit : PChar,
  const OutputUnit : PChar
):Integer;
```

This function creates a new register whose handle number is returned through the HandleNo parameter. The Name specifies the name of the handle. Any additional parameters for the particular handle are concatenated to the name to be given to the handle in square brackets. This applies to the Net handles that need at least the remote computer name specified in its parameters. The server specifies the name of the interface that serves the register. This is “User” in the case of the LabUser handles created on the remote computer and “Net” on the client computer. The Status parameter specifies among other things the data type of the register to be created, whether it is an Input or Output register and whether a Timeout is specified for the register or not.

The code for automatically creating the about 1000 handles is as shown in Listing 3.2.

```

var
  i:Integer;
  PHandleNo:PLongWord;
begin
  GetMem(PHandleNo,1024);           //Allocate Memory for pointer to HandleNo
  try

    for i:=1 to 1000 do
      if(Sender = btnUserCreate)then //true if User Handle is to be created
        LabMapCreate(PHandleNo,
          PChar('Handle-'+IntToStr(i)), //no parameter specified
          'User',
          HandleInput or HandleInteger or HandleNoTimeout,
          0,
          0,
          "
          "
          );           //create User handle
      else
        LabMapCreate(PHandleNo,
          PChar('Handle-'+IntToStr(i) + '[Log]'), //Log specified as parameter
          'Net',
          HandleInput or HandleInteger or HandleNoTimeout,
          0,
          0,
          "
          "
          );           //create Net handle

    finally
      FreeMem(PHandleNo); //Deallocate memory for pointer toHandleNo
    end;
end;

```

Listing 3.2 Code to create handles

An example illustrating the name for a register with number 567 that was created is as follows

User	Net
567	567[Log]

where Log is the name of the remote computer used in the experiment.

3.2.4 Other registry settings [1]

For LabMap to be able to transmit data between the client and the server, there has to be other configurations done in the registry of both the client and the server.

3.2.4.1 Server configuration

The server is that which accepts connections from clients. The MaxConnections property specifies the maximum number of clients that can be connected simultaneously to the server while the Port property specifies the TCP/IP port used to accept connections from the clients.

3.2.4.2 Client configuration

A client must have a connection to the server before communication can take place between the two. A client may have more than one connection to a particular server and may also have different connections to different servers. Each connection is denoted by an alias that is a case-sensitive text string indicating the remote sever. For each alias, the following registry keys are defined.

Registry Entry	Example	Purpose
Server. <Host>	Server. <Log>	Specifies the TCP/IP address of the remote server
Port. <Host>	Port. <Log>	Specifies the TCP/IP communication port used for the connection

In addition, for each such alias, a virtual input string register named 'Host [[<Host>](#)]' must be declared. In our example, it would have the name **Host [[Log](#)]**. This register is initialized with the contents of the registry key Server.<Log>.

3.3 The process procedure

For a particular number of handles, N and a timer interval of t ms, the following scenario is what takes place. The client application first sets the LabMapCallback function that will be receiving notifications of updates made to handles. The main idea is to send new values to the registers and wait to receive notifications from LabMap that the new value has been applied to the register. To do this, the application sets a LabMapCallback function that is notified of any changes to any register value. This is done with the LabMapSetCallback function for the particular type of data type involved. This function takes as parameter, the memory address of the call back function. The function is defined for the integer datatype as follows:

```
function LabMapSetIntCallback : Integer;  
(  
    LabMapIntCallBack : Pointer  
);
```

This function sets the callback routine to process changes in integer register values. The call back procedure is called each time the value of an integer register is changed. The call back function must have the stdcall calling convention and is defined as follows;

```
procedure LabMapIntCallBack  
(  
    HandleNo    : Cardinal;  
    Value       : Integer;  
    Const Stamp : TimeStampArray;  
    Overrun     : Integer  
);stdcall;
```

The HandleNo is the handle number of the register whose value changed. The parameter Value is the new value of the register. Corresponding SetCallback functions and callback procedures exist for Real and String register types.

In the experiment, LabMapSend is called with a new value on the client. LabMap sends the first notification through the call back function in the client application when it sets the value of the register to a new value. LabMap then relays this new value to the remote register. LabMap on the server sets the remote register to the new value. LabMap then sends messages to all clients that are mapped to that particular register informing them of a value

change. When LabMap on the local machine receives this information, it tries to set the value of the register. But since the second update value is the same as the already existing value, it does not update the register again. Since the Round Trip Time for LabMap to send data to the server and back will be determined by LabMap notifying the client application on receipt of the message from the remote server, it is imperative that the second notification is sent even if there is no change in value. To force LabMap to apply updates even if there is no change, the LabMapCatchAll function must be called for that particular register with a value other than 0 for the *CatchAllFlag* parameter. This causes a second notification to be sent from LabMap to the client application. The definition of LabMapCatchAll is as follows:

```
LabMapCatchAll (HandleNo : Cardinal ; CatchAllFlag : Integer);
```

The whole scenario is as shown in the Figure 3.1

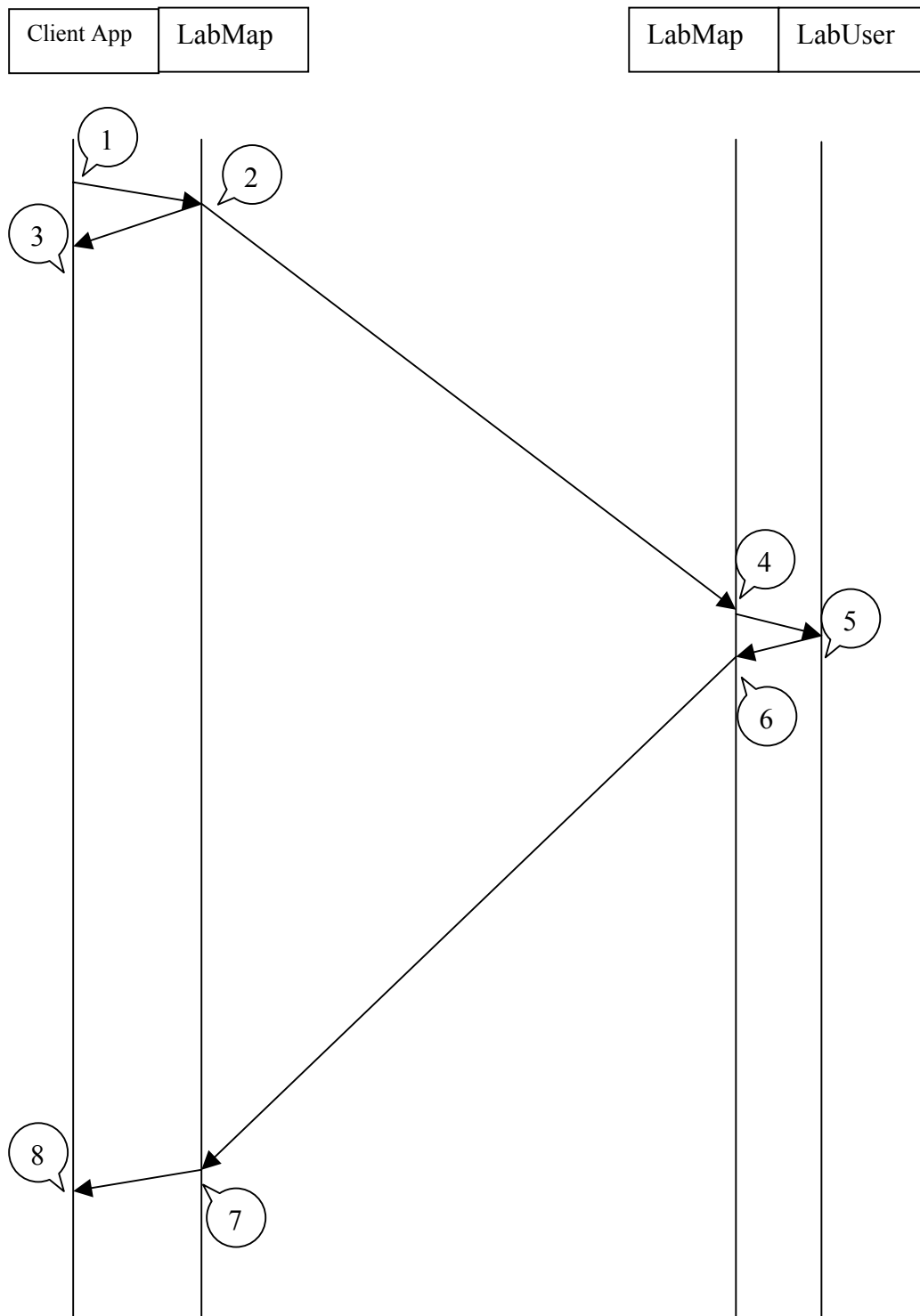


Figure 3.1 Data sending and callback notification procedure

1. Client Application takes the performance count of the high-resolution timer, records it and sends a new value to a register in LabMap.

2. LabMap sets the value of the register to the new value and sends notification to client application through the callback function. LabMap also relays the value change over the network to the remote computer.
3. Client application receives callback from LabMap with the Handle Number of the updated register and the new value. The performance count of the high-resolution counter is then taken upon receipt and stored. Time between 1 and 3 is the Local Round Trip Time.
4. LabMap on the remote machine receives the notification from the client.
5. LabMap sets the value of the register to the new value.
6. LabMap broadcasts the change in value of the handle.
7. LabMap on the client machine receives the broadcast from the remote machine and sends the second callback to the client application
8. The client application receives the callback from LabMap as in step 3 and records the performance count of the high-resolution counter. Time between 1 and 8 is the Remote Round Trip Time.

The client application sends a thousand different values to LabMap and calculates the average value and the variance of all the values recorded. As much as possible, the client application is not supposed to perform much computations during the timing process but just to store the performance count of the high resolution counter. All the performance counts before the value is sent to LabMap and when the first and second callback notifications are received from LabMap have to be stored for each value sent and for each of the handles. The queryperformancecounter function usually takes a large integer (a 64-bit integer) data type. This corresponds to the Int64 data type in Pascal. It is known that each handle has 1000 different values sent to it but the number of handles is specified by the user just before the test is conducted. A 2-dimensional array was used to store the performance counts. For each handle, an array of dimension, 1000 was defined to hold the performance counts at each new value sent. Since the number of handles is not known and can be varied before each test, a dynamic array of arrays for each handle was created and the memory for the dynamic array is allocated before the start of the test for each handle. Another such boolean array is created to tell LabMap whether a particular notification received corresponds to the first or second callback. Figure 3.2 illustrates such an array.

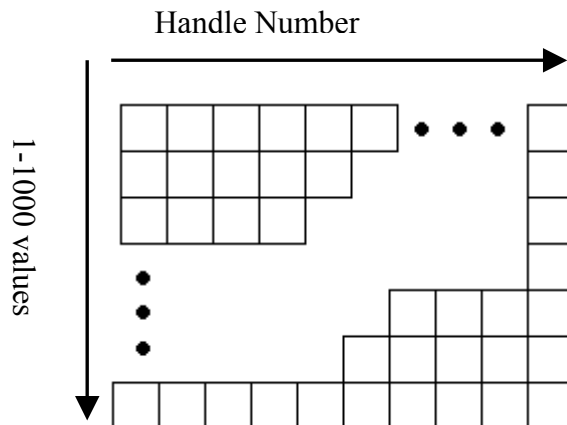


Figure 3.2 Array to hold times

Listing 3.3 is an extract of the array allocation procedure that is executed just before the test is carried out.

```
var
  StartTimes,lcEndtimes,rmEndTimes:array of array[1..1000] of Int64;
//Definition of dynamic arrays for start, local and remote callbacks times
  Remote: array of array [1..1000] of Boolean;
//Definition of dynamic array for flags set when first callback is received

//Allocation of memory for NumHandles registers
  SetLength (rmEndTimes,NumHandles);
  SetLength (lcEndTimes,NumHandles);
  SetLength (Remote,NumHandles);
  SetLength (StartTimes,NumHandles);
```

Listing 3.3 Code to allocate memory of dynamic arrays

After all the necessary preparation including the memory allocation has been done, a timer procedure is started. Whenever this timer times out, it sends a new value to all the handles that are undergoing the test. For each of the NumHandles registers, the function records the performance count of the high-resolution counter and then sends a new value. After that, the function adds one to the value that is sent out. This process is repeated until values 1 to 1000 has been sent. The timer procedure is as shown in Listing 3.4.

```

var
  i:Integer;
begin
  for i:=0 to NumHandles-1 do
    begin
      QueryPerformanceCounter(StartTimes[i,setValue]);
      LabMapSendInt(i+1,setValue);
    end;
    Inc(setValue);
    Timer1.Enabled:=setValue<=1000;
  end;

```

Listing 3.4 Timer procedure

The callback waits for notifications from LabMap. Each callback contains information such as the Handle Number and new value of the register. If the register is one of the registers in the test, the performance count of the high-resolution count is taken. If the first callback has been received for that particular handle number and value, the performance count is stored in the remote endtime array. It is stored in the local endtime array if otherwise and the corresponding flag is set. Listing 3.5 shows the callback procedure.

```

var
  mEnd:Int64;
begin
  if HandleNo <= NumHandles then
    begin
      QueryPerformanceCounter(mEnd);
      if Remote[HandleNo-1,Value] then
        rmEndTimes[HandleNo-1,Value]:=mEnd
      else
        begin
          Remote[HandleNo-1,Value]:=True;
          lcEndtimes[HandleNo-1,Value]:=mEnd;
        end;
    end;
  end;
end;

```

Listing 3.5 Callback procedure

An example for the whole process is shown in the diagram for a register with handle number 20 and value 345 sent.

1. Take QueryPerformance Count just and put in StartTimes dynamic array.
2. Send value 345 to handle number 345
3. Callback is received, take QueryPerformance Count
4. Check if flag is set.
5. If not set, then it is a first callback, put value in first callback array
6. Set flag to show that first callback has been received
7. Callback is received, take QueryPerformance Count
8. Check if flag is set
9. If flag set, then it is a second callback, put value in the second callback array

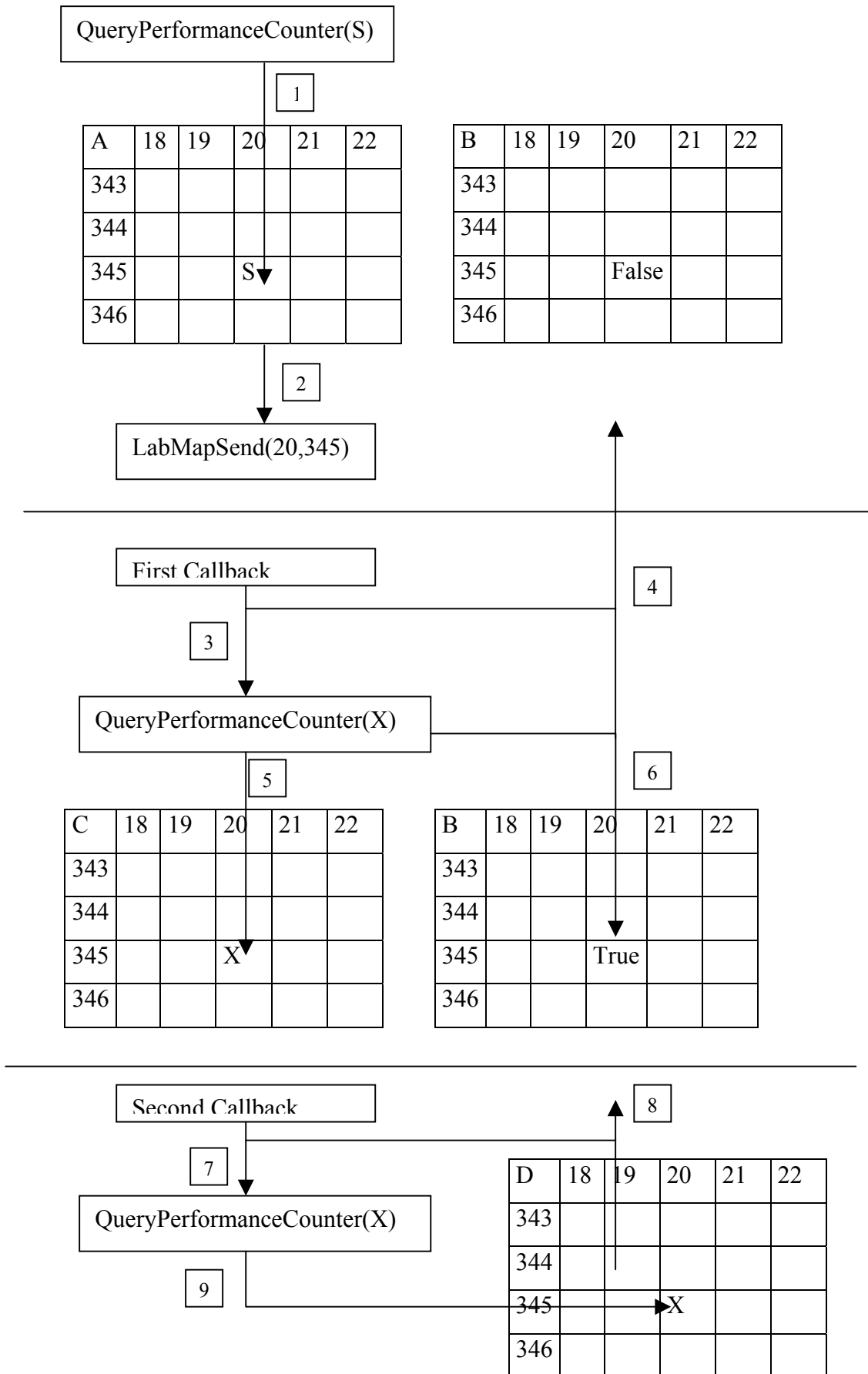


Figure 3.3 Timer and Callback procedure

Legend

- Array A: Start Times array
- Array B: Boolean Flag Array
- Array C: Local End Times Array
- Array D: Remote End Times Array

After all the values have been sent to all handles in question, the round trip return times are computed for each value sent for each handle. This is computed as follows

$$RTT = \frac{EndTime - StartTime}{1193.182} ms$$

The mean and variance for all these values are computed then computed and tabulated.

3.4 Measuring Overwrites

The number of overwrites that occurred in the server application is also computed as a percentage and computed. The basic assumption is that all first callbacks are notifications received from the local copy of LabMap. If a second callback for a particular handle number and value is not received, then it can be concluded with the afore-mentioned assumption that an overwrite occurred. Since all values in the second callback dynamic array are initialised to zero before the test, any cell in that matrix with a value of zero implies an overwrite.

Chapter 4 Results and Discussions

All the results are tabulated in Appendix A. It should be noted that while some factors had a clear impact on the performance of the system, other factors, which are outside the scope of LabMap, also have a major role to play in the performance.

The effect of the various factors on the performance are as discussed below.

4.1 Effect of number of handles on performance

It was noted that in general, the performance degraded with increasing number of handles. This is the normal expectation because the more the number of handles, the more resources in terms of network bandwidth and processor clock cycles needed by the Windows Operating System to satisfy all data communication operations. Chart 4.1 shows the Mean Round Trip Time for the integer data type at intervals of 10, 50 and 100 ms.. It can be seen that the curve approaches a linear curve as the timer interval increases.

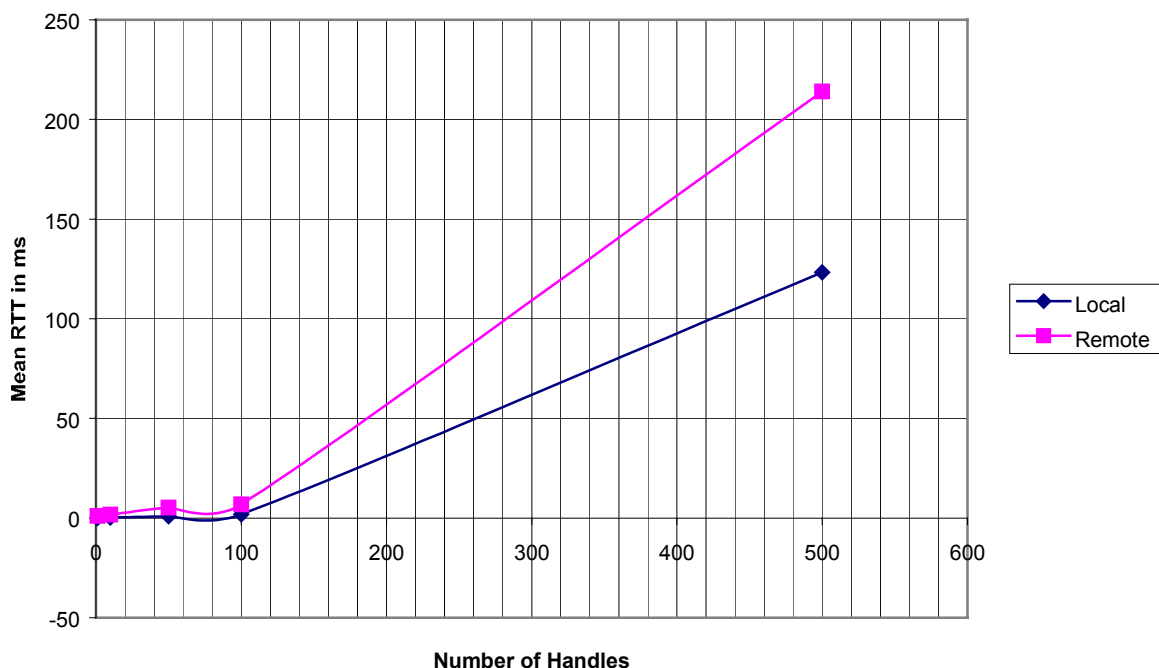


Figure 4.1 Chart of Mean RTT in ms for integer data type at timer interval of 10 ms

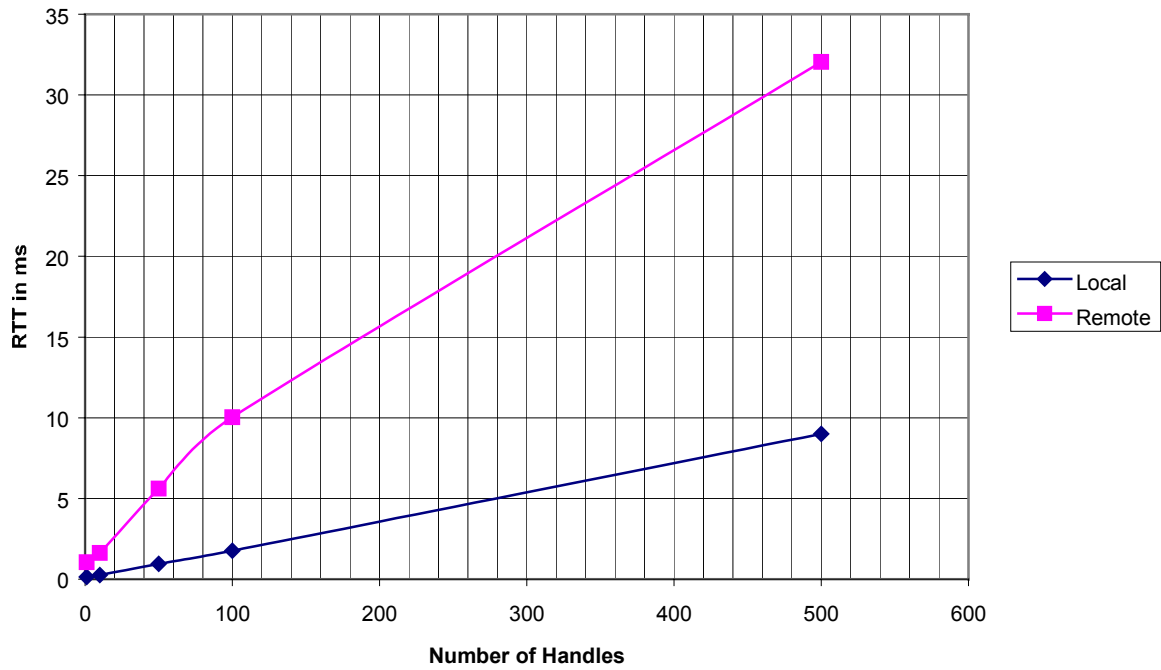


Figure 4.2 Chart of Mean RTT in ms for integer data type at timer interval of 50ms

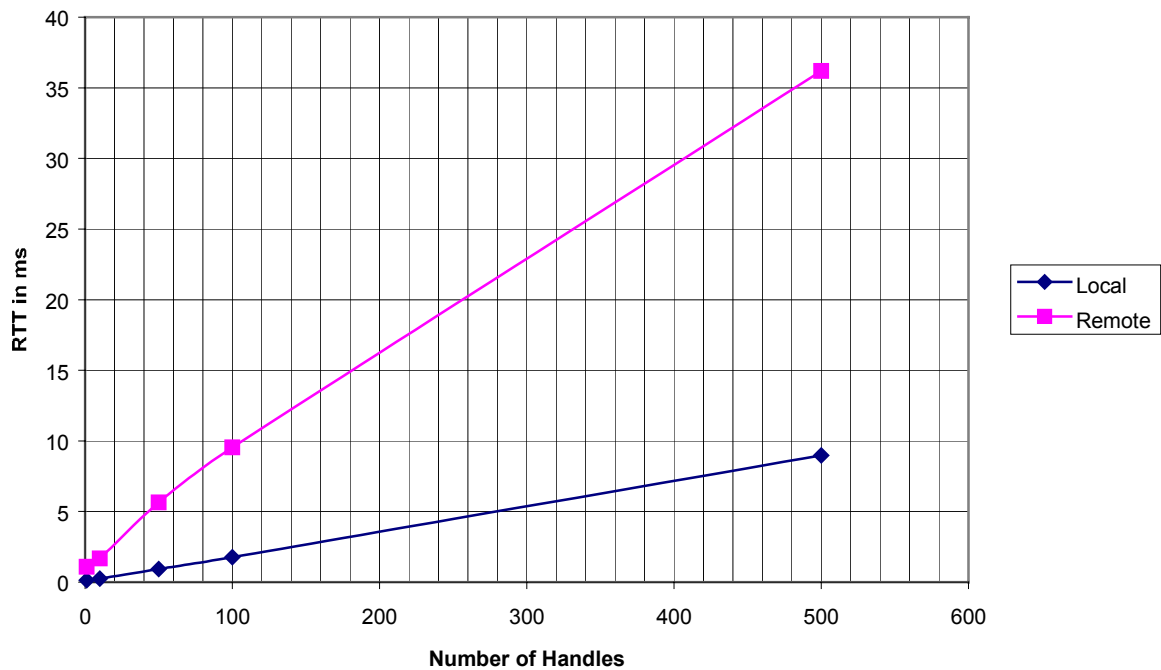


Figure 4.3 Chart of Mean RTT in ms for integer data type at timer interval of 100ms

4.2 Effect of timer interval on performance

It was noted that there was no particular correlation between timer intervals and the general performance. It was noted that the performance depended on both the timer interval and the number of handles. These two factors can be said to be the deciding factor as to the time intervals between which LabMap sends data to the Windows socket buffer.

If this time interval is long enough, then the TCP/IP sockets have to be reinitialised anytime data is to be sent. Another possible reason is the socket buffer. The Windows socket buffer is such that it waits for enough data before actually sending the data. Some data therefore might have to wait for an additional amount of time in the buffer before it is actually sent. This phenomenon can be noticed at timer intervals of 200 ms, where the performance was not so predictable.

4.3 Effect of data type on performance

It was noted that the data type involved did not have so much effect on the average round trip time. Possible reasons include the fact that the data structures that contain the data have other data that makes the effect of the size of the actual data to be sent on system performance insignificant. Another possible reason is the effect of the time interval between two data-send sessions on Windows sockets and the socket buffers as explained in 4.2, effect of timer interval on the performance. On the whole, it can be said that the data type involved did not make much difference.

Appendix A Results of experiment

A.1 Results for integers

T = 10 ms

	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.125	0.257	0.924	1.944	123.23	1.128	1.698	5.244	6.789	213.97
Overwrite	0	0	0	0	0	0.3	0.65	10.286	45.665	97.506
Var	0.028	0.068	0.107	0.701	2667.3	0.349	10.929	7.247	53.289	55611

T = 50 ms

	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.134	0.259	0.947	1.764	8.998	1.059	1.632	5.618	10.031	32.052
Overwrite	0	0	0	0	0	0	0.23	11.664	11.149	52.274
Var	0.004	0.019	0.099	0.356	12.281	0.086	20.159	19.123	36.429	216.13

T = 100 ms

	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.139	0.26	0.933	1.775	8.964	1.086	1.681	5.636	9.551	36.188
Overwrite	0	0	0	0	0	0	0.33	11.894	9.653	8.251
Var	0.002	0.013	0.098	0.366	10.999	0.375	22.994	27.677	23.007	566.07

T = 200 ms

	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.158	0.262	0.926	1.755	8.662	1.114	113.79	9.75	8.876	91.962
Overwrite	0	0	0	0	0	0	0	0	9.431	5.178
Var	0.003	0.016	0.1	0.463	8.907	0.076	3077.6	5521.6	97.671	694.44

A.2 Results for real numbers

T = 10 ms

	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.135	0.276	1.04	2.106	135.64	1.139	1.863	5.616	7.575	182.5
Overwrite	0	0	0	0.001	0	0.2	4.03	12.678	63.374	96.766
Var	0.088	0.097	0.258	0.677	2592.3	0.27	16.474	41.877	114.58	28992

T = 50 ms

	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.124	0.269	0.947	1.933	9.812	1.041	1.643	5.618	10.514	31.484
Overwrite	0	0	0	0	0	0	0.27	11.664	10.836	52.196
Var	0.001	0.007	0.099	0.324	11.997	0.19	15.129	19.123	38.485	201.65

T = 100 ms

	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.132	0.308	1.001	1.92	9.728	1.05	1.786	6.04	10.815	40.626
Overwrite	0	0	0	0	0	0	0.19	12.97	12.388	8.753
Var	0.001	0.02	0.085	0.322	9.741	0.115	17.763	38.791	42.557	513.14

T = 200 ms

	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.136	0.277	1.004	2.115	9.329	1.065	104.9	108.58	132.55	94.712
Overwrite	0	0	0	0	0	0	0	0	0	6.234
Var	0	0.0057	0.083	2.925	7.147	0.081	2620.6	4313.9	3483.8	644.96

A.3 Results for Strings

T = 10 ms

	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.18	0.325	1.261	2.712	162.56	1.212	1.979	5.238	7.581	184.23
Overwrite	0	0	0	0.002	0	0.4	1.56	18.342	59.074	96.665
Var	0.335	0.083	0.417	2.514	5571.7	0.122	29.813	27.562	41.867	39108

T = 50 ms

	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.156	0.316	1.21	2.282	12.169	1.104	1.805	6.761	11	32.839
Overwrite	0	0	0	0	0.0004	0	1.53	10.324	11.598	58.584
Var	0.049	0.111	0.304	0.724	29.304	0.528	12.303	20.28	34.338	255.42

T = 100 ms

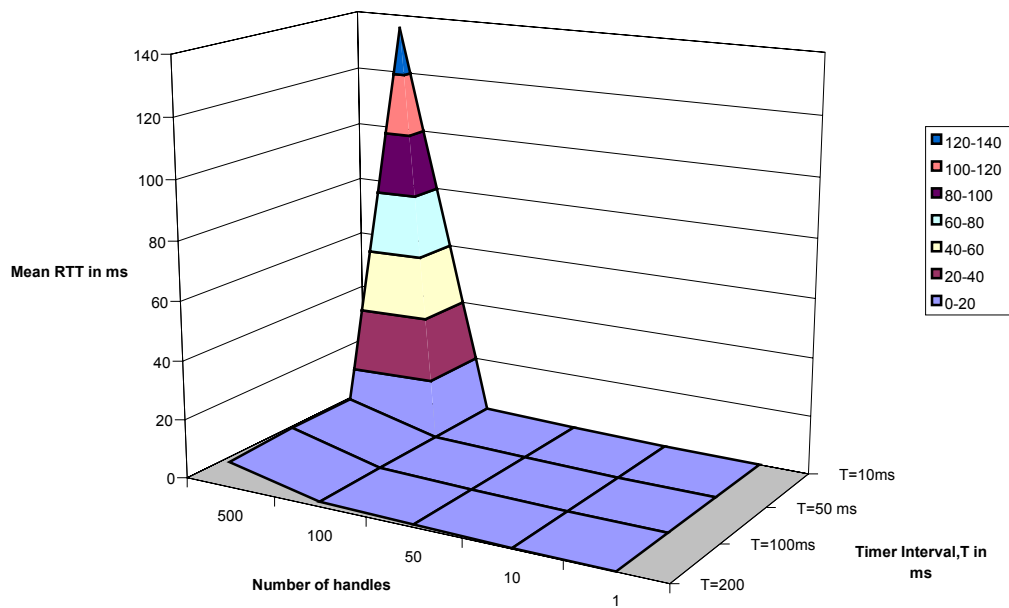
	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.821	1.24	1.196	2.291	11.588	2.194	2.609	6.436	10.87	41.921
Overwrite	0	0	0	0	0	0	0.11	14.72	10.861	6.735
Var	0.228	0.39	0.189	0.725	20.37	0.677	33.888	30.17	43.935	255.16

T = 200 ms

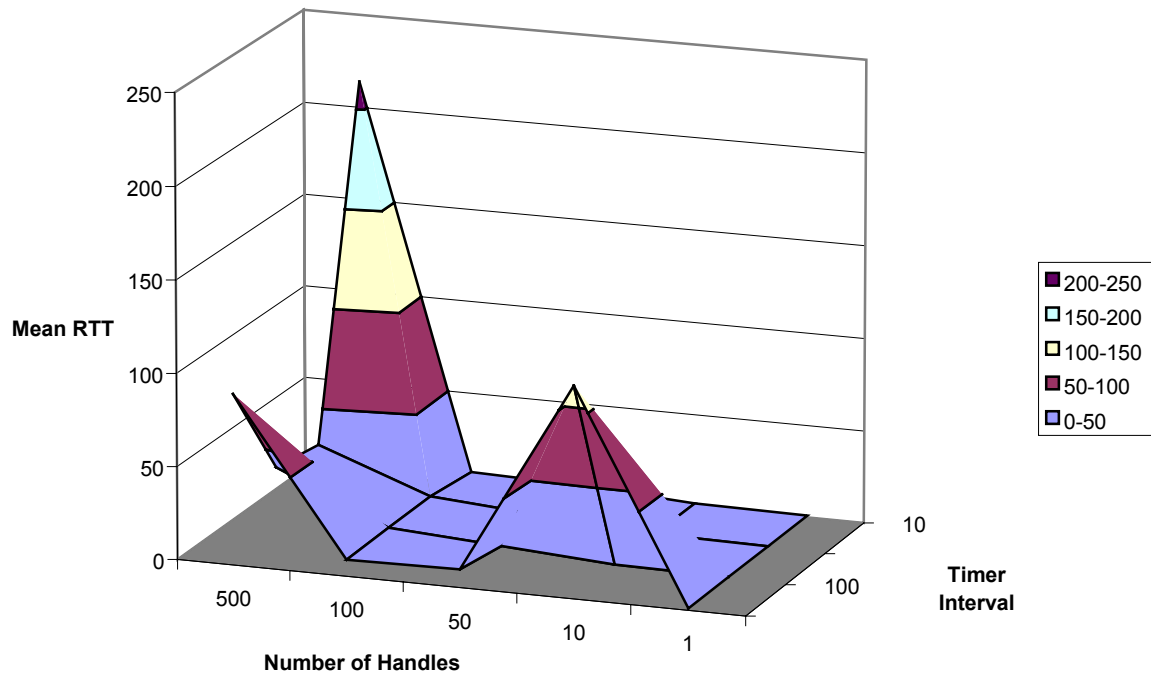
	Local					Remote				
	1	10	50	100	500	1	10	50	100	500
Mean	0.195	0.315	1.206	2.297	11.663	1.752	88.953	33.229	75.893	105.91
Overwrite	0	0	0	0	0	0	0	21.954	21.854	7.039
Var	0.002	0.012	0.191	0.727	18.546	0.154	1849.5	2083.8	2774.5	598.71

Appendix B Sample Charts of data obtained

B.1 3-D surface for the Local Round Trip Time for the integer data type



B.2 3-D surface for the Mean Remote Round Trip Time for the integer data type



Appendix C References

1. Gonzalez O., Ramamritham K., Sen S., Shen C. and Shirgurkar S.
Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations, 1998
2. Kazakov, D.A.
LabRRR Handbook Eleventh Edition – cbb Software GmbH
3. Microsoft Corporation
Microsoft Developer Network Library October 2001 Edition
4. OPC Foundation
OPC Technical Overview, 1998
5. Real-Time Innovations
Network Data Delivery Services Tutorial, July 2000.
6. Rosenberger, J.
Teach Yourself CORBA In 14 Days SAMS Publishing, September 1998