Component-based Development on the Windows DNA Platform

Post-graduate Thesis
in Information and Communication
Technology

by
LARS BARSTAD
FRODE K. KRISTENSEN

Grimstad, June 2000

Abstract

This report deals with key areas of software development in a distributed environment. Distributed computing systems are becoming increasingly more popular, and the demands for functionality are growing accordingly. With distributed systems being used for so many different purposes, and with so many people depending on it to do their jobs, these systems must be stable and robust to continuously serve the users. If the system fails, for example caused by network problems or by application crash, it can in the worst case paralyze the organization.

With this as a starting point, we have been looking at technologies for developing distributed systems based on the Windows DNA platform. One issue that we have been working on in particular is mechanisms that provide dynamically recovery from failure.

We have learned that by using the right technologies developers can save themselves from a lot of work. After working with Windows DNA during the project period, we feel that it offers a good and solid platform with technologies for developing robust enterprise systems. Our experience is that these technologies are fairly easy to implement, as long as you plan your work well and use a suitable development language.

Preface

This project is a part of the master degree study (sivilingeniør) in information and communication technology at Agder College (Høgskolen i Agder). The thesis was scheduled for five months and counts as ten credits (vekttall).

At this point we would like to thank our technical advisor, John S. Rasmussen, for excellent guidance and helping us stay focused on the goal of our study.

Grimstad, June 2000

Lars Barstad

Frode K. Kristensen

1 Introduction

1.1 Research Motivation

Today, organizations are working towards using computer systems more effectively. The trend shows an extensive use of *distributed computer systems*. More and more businesses run their systems over the network, with all clients accessing the same resources. A distributed system, or enterprise system, combine large numbers of independently executing programs into a seamless whole, and provide services critical to the day-to-day operation of the organization.

Developing these kinds of distributed systems requires a lot more effort and planning than traditional applications running on one single computer. The system needs to handle multiple users accessing the same resources without letting them interfere with each other. Since a distributed system runs over a network, special care must be taken to respond dynamically to failures and recoveries. Other issues like security also require extra attention when messages are sent across a potentially unsafe network.

As we can see, software development for distributed systems presents a number of different challenges.

1.2 Problem Approach

In this report, we will take a closer look at what the *Windows DNA* (Distributed interNet Applications Architecture) platform can provide when it comes to enterprise development. We will focus on common problems like message delivery and transactions over a network. In Windows DNA there are two technologies that deal explicitly with these problems: *Microsoft Message Queue* and *Microsoft Transaction Server*.

Since our work consisted of both research and implementation, we decided to use the following approach for our work:

- Research theory on the subject (i.e. transactions in general)
- Research theory on the technology in question (i.e. Microsoft Transaction Server)
- Design the implementation using industry-standard UML diagrams
- Implement the technology in the selected languages (Visual C++, Visual J++ and Visual Basic)
- Test and evaluate performance on the different implementations

This approach worked well, and helped us stay focused throughout the project.

We have included the original description of our project at the end of this report. Together with our technical advisor we decided to omit our look at SOAP due to time constraints. No information on SOAP is to be found in this report.

This report as well as the source code for our components can be found at:
http://www.reptile.no/diplom/

2 Components and Windows DNA

2.1 What is Windows DNA?

Windows DNA is a collection of software-packages and technologies that together provide a platform for building and deploying distributed systems, both traditional applications and modern web-based systems like *e-commerce* and *intranets*. Microsoft has developed all parts of Windows DNA.

Windows DNA consists of the following applications/technologies:

- Windows NT / Windows 2000
  - Internet Information Services
  - COM/Microsoft Transaction Server (COM + for Windows 2000)
  - Message queuing
  - Indexing services
  - Security Services
  - Network load balancing
  - Universal Data Access
  - XML support
- SQL Server 7.0
- SNA Server 4.0
- Site Server 3.0 Commerce Edition
- Visual Studio 6.0
- BizTalk Server
- Exchange Server 5.5

[source: http://www.microsoft.com/dna]

In this report, we will look at the “heart” of the Windows DNA platform, namely the *Component Object Model* (COM), *Microsoft Transaction Server* (MTS), and *Microsoft Message Queue* (MSMQ). In addition we will look at *Visual Studio 6.0*, the premier tool for creating Windows DNA applications. We will also look at some of the enterprise tools bundled with Visual Studio. These tools are introduced in the next part of this chapter, since they in part form the basis of the following chapters on MTS and MSMQ.

COM is a component technology that allows different components to communicate. This is done with the help of an interface. An interface describes which functions, methods and properties are available in a component. For a given component, the interface will always stay the same from version to version. If new functionality is introduced, a new interface is added to the component while the old interface is retained. This way existing applications will not be broken if a newer version of a component is installed on a computer.

To make it possible for an application on computer A to use a COM component that resides on computer B, *Distributed COM* (DCOM) was introduced. *DCOM* is simply an extension that transparently takes care of creating and using a component on another computer. The developer does not need to take any action to switch from a local component to a remote component. Everything is handled by DCOM.

MTS and MSMQ were introduced as supporting services that solve common challenges within a distributed system. MTS handles transactions, while MSMQ has mechanisms for guaranteed asynchronous message delivery.

With the introduction of Windows 2000, Microsoft also introduced COM+. *COM+* is an extension of COM, which among other things has built-in transaction support. This means that MTS does not exist as a separate component in Windows 2000. The functionality however remains the same. For more detailed information about COM and COM+, visit http://www.microsoft.com/com/.

From a developer’s point of view, components are the smallest building blocks of the Windows DNA platform. All the services you interact with in a Windows DNA solution are component based. For example, you can create an instance of the ADODB component to communicate with a database, or create an instance of ADSI to communicate with Microsoft Exchange Server and so on. By using this technology the developer can leverage all the work that has been put into developing Windows itself, other Microsoft applications and services, as well as a number of third-party tools.

2.2 Visual Modeler

Visual Modeler is a tool used to visually design components and the interaction between components in a typical three-tier solution. As a subset of Rational Rose, Visual Modeler supports UML class diagrams and logic views. In addition it is possible to generate code from a class-diagram, as well as reverse-engineer VB projects.

Our use of Visual Modeler is limited to designing our classes in an environment that does not tie our design to any specific language. We have also tried out the code-generation feature for VB, which works fairly well. The greatest advantage of using tools such as Visual Modeler is that it allows you to focus on the design of your component rather than the implementation. With the code-generation feature you also save some time writing tedious function declaration. The use of modeling tools seems to us like a win-win situation. You spend some time modeling the component, but save roughly the same amount of time when the software generates the function declarations for your code. Result: You get the model for free. More likely than not, it will also help you design a better component.

Even though Visual Modeler is categorized as an Enterprise tool within Visual Studio, Visual Modeler alone is not a powerful enough solution to design a large-scale distributed system. Among the things missing in Visual Modeler is UML Use Case diagrams and activity diagrams.

2.3 Visual Studio Analyzer

When developing distributed systems performance is a key concern. It only takes one poorly programmed component to slow down an entire system. When a distributed system is experiencing a decrease in performance, it is often difficult to locate the bottleneck that is causing the slowdown.

Visual Studio Analyzer is designed to let developers analyze performance in a distributed system, by showing how all components of a distributed, multi-tier application interact. This is done by capturing an event-log of the communication that occurs between components when the application is run. Once the event-log has been created, VSA can be used to display time-diagrams of the interaction, graphically display how the components interact, and apply different filters to see only the interaction of a specific category. A typical category can be for example all database-calls.

VSA also has a feature that allows Performance Monitor counters to be captured from a remote machine. These counters can be displayed together with a graphical representation of the event log, making it possible to detect bottlenecks. Once a bottleneck has been found, other means must be used to find the exact cause of it. If it is a component that is causing the bottleneck, traditional profiler tools can be used. If the bottleneck is caused by hardware (or lack of it) we can use Performance Monitor to detect which resources are being depleted.

3 Microsoft Transaction Server

This chapter is divided into four parts:

- Transaction Technology
- Our Transaction Component
- Implementation
- Discussion

3.1 Transaction Technology

3.1.1 Transaction Theory

In a distributed system it is common that several components are involved in performing a single task. Most of the time when performing such a task, components depend on the successful execution of other components in order to succeed. This is what we call a *transaction*: Either all components succeed in their task, or none succeed. Even though this sounds simple, it would present quite a challenge to the programmers to implement this with traditional software development methods. Each component would need to know about all other components, and for each task it performs check the other components to see if they have succeeded or not. This adds a complexity factor of n*n to the error handling. And on top of that, one key element of distributed systems is the encapsulation and reusability of each component, meaning that a component should not need to know about the other components that are part of the transaction.

Luckily, a more effective approach has been adopted for modern distributed systems – the transaction manager. A transaction manager acts like a supervisor during a transaction. It keeps track of all participating components, collect their status (*success* or *failure*), and either executes or aborts the entire task. Transactions managers on different operating systems differ in how they execute or abort the transaction, so we won't go into more details on that. In our discussion of *Microsoft Transaction Server* (part 3.1.2) however, we will explain the details of that particular implementation.

To further illustrate the concept of transactions, we will look at the classic example - the banking application. A banking application typically transfers money from account A to account B.

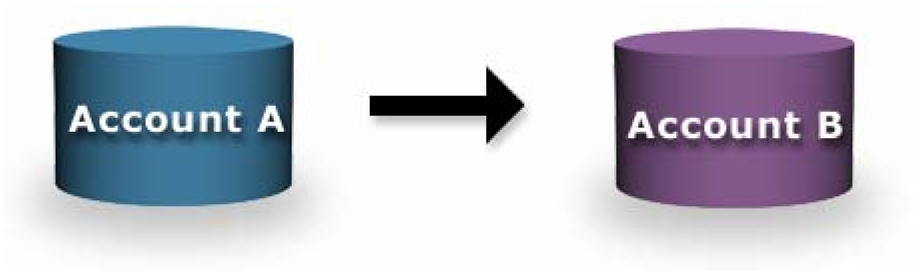


Illustration 3.1 Money Transfer

This illustration shows a transfer from account A to account B.

There are two tasks that need to be done in order to complete the transfer:

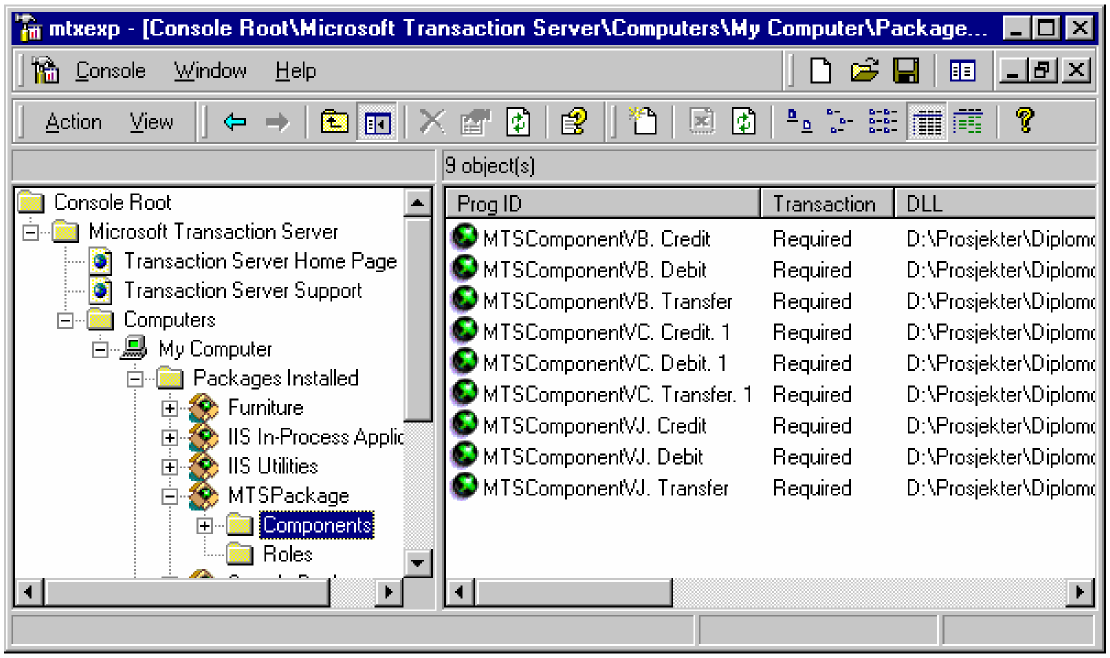
- Withdraw the money from account A.
- Credit the money to account B.

This should be a very simple task, but a closer look reveals that there are several things that can go wrong. If money is withdrawn from account A and the system fails before completing the transfer, the money is gone from account A even if account B has not received them. And if the system credits account B with the transfer amount and then fails, money will not be withdrawn from account A. As we can see, both these cases leave the system in an inconsistent state, which means that money has either been “created” or “disappeared” in the system. The total amount of money in the system is not the same after the failed transaction as it was before. In essence, this is the problem that transaction technology seeks to solve.

3.1.2 Microsoft Transaction Server Overview

Microsoft Transaction Server (MTS) provides a mechanism to handle transactions and ensure data consistency. It makes sure that a transaction is only completed as one whole unit. If it fails during execution, MTS will undo the transaction and leave everything in a consistent state – the state it was in before the transaction started.

Since MTS is built for a distributed environment, it assumes a component-based approach to application development. MTS has its own *run-time environment*, where all components that take advantage of Transaction Server runs. Components that run in this environment are *COM components* stored as *dynamic-link libraries* (DLL). Before a component can participate in a transaction, it needs to be configured as part of a *Component package* within MTS. This is done with an administrative tool called Microsoft Transaction Server Explorer.


mtxexp - [Console Root\Microsoft Transaction Server\Computers\My Computer\Package...
Console
Window
Help
Action
View
9 object(s)

- Console Root
  - Microsoft Transaction Server
    - Transaction Server Home Page
    - Transaction Server Support
    - Computers
      - My Computer
        - Packages Installed
          - Furniture
          - IIS In-Process Applic
          - IIS Utilities
          - MTSPackage
            - Components
            - Roles

Prog ID	Transaction	DLL
MTSComponentVB. Credit	Required	D:\Prosjekter\Diplom
MTSComponentVB. Debit	Required	D:\Prosjekter\Diplom
MTSComponentVB. Transfer	Required	D:\Prosjekter\Diplom
MTSComponentVC. Credit. 1	Required	D:\Prosjekter\Diplom
MTSComponentVC. Debit. 1	Required	D:\Prosjekter\Diplom
MTSComponentVC. Transfer. 1	Required	D:\Prosjekter\Diplom
MTSComponentVJ. Credit	Required	D:\Prosjekter\Diplom
MTSComponentVJ. Debit	Required	D:\Prosjekter\Diplom
MTSComponentVJ. Transfer	Required	D:\Prosjekter\Diplom

Illustration 3.2 Microsoft Transaction Server Explorer

The MTS Explorer is a graphical user interface used to configure and manage MTS components within a distributed computer network. This screenshot shows the components contained in the package called MTSPackage.

MTS supports all COM components, which can be created and implemented with Visual Basic, Visual C++, Visual J++, or any other ActiveX/COM-compatible development tool.

Because MTS components can take advantage of transactions, developers do not need to worry about whether the other components in the transaction are running successfully. All they need to do is to make sure that their component performs it part, and that it indicates failure to MTS if the operation is unsuccessful. The MTS transaction system, working in cooperation with database servers and other types of resource managers, ensures that transactions are *atomic*, *consistent* and have proper *isolation*.

The way this is implemented in MTS is that each component has its own *context* object that is implicitly associated with a given Microsoft Transaction Server object. Context objects contains information about the component’s state in the execution environment, such as transaction, activity and security properties. These objects also simplify the development of components, because they let each component independently acquire its own resources, perform its work, and indicate its own internal state. The state of a component is indicated by calling the MTS-method *SetComplete* or *SetAbort*. The component calls SetComplete when the whole transaction is completed successfully, and SetAbort if an error has occurred and the transaction needs to be rolled back.

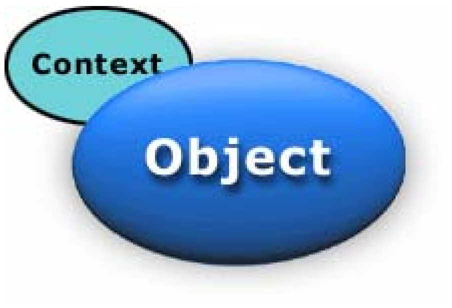


Illustration 3.3 An MTS object and its associated context object

The illustration shows how MTS keeps track of each component by assigning a context object to each.

The development effort to implement transactions for COM components is minimal. Developers do not have to specify *Begin transaction*, *End transaction*, or *Abort transaction* statements in their code. Rather, transactions are implemented by setting a property of an application component. If the component is marked *transactional*, then Transaction Server will build a transaction around its own processing. Any other components referenced by the transactional component will automatically participate in the transaction as well.

Microsoft Transaction Server also serves as an infrastructure product that can deal with many of the requirements in developing and deploying multi-tier applications. An MTS package can be created on one machine, saved, and imported on another machine, thus removing the need for complex component installation software. This allows them to focus on implementing business functions instead.

Using MTS makes it easier to build distributed applications by providing *location transparency*. MTS automatically loads the component into a process environment. A MTS component can be loaded into a client application process (*in-process component*), or into a separate surrogate server process environment, either on the client's computer (*local component*) or on another computer (*remote component*).

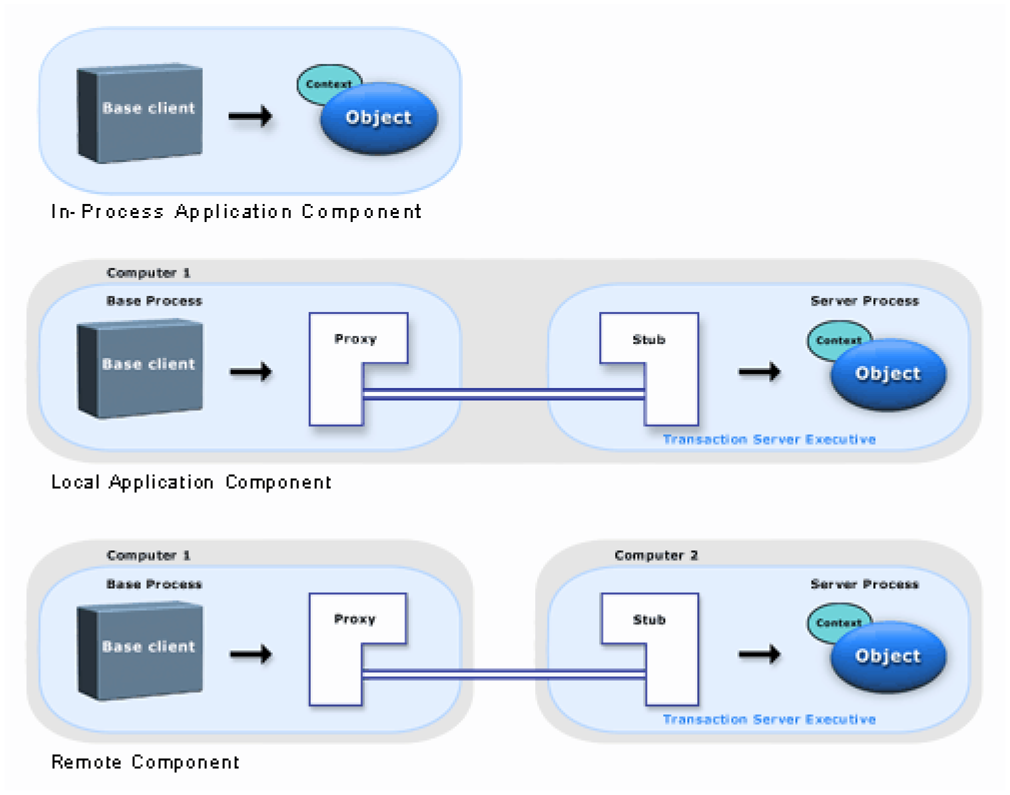


Illustration 3.4 In-process, local, and remote components

MTS supports *component pooling*, which means that instead of creating and destroying a new instance of a component each time it is requested, a collection of objects is recycled. This increases performance by avoiding freeing and allocating memory, and is very beneficial in a large-scale distributed system where there is a lot of component reuse.

Integration with the *Microsoft Distributed Transaction Coordinator* (DTC) provides a robust transaction management infrastructure. Transaction Server provides a transaction monitor that controls transactional access to *resource managers*, such as Microsoft SQL Server, the latest versions of Oracle and databases that support the ODBC interface. Transactions may access a single resource manager, or, through support of DTC protocol, transactions may coordinate and synchronize access to multiple resource managers.

With the DTC, work can be committed as an atomic transaction even if it spans multiple resource managers on separate computers. This transaction support is transparent to the programmer. DTC implements a *two-phase commit protocol* to ensure that the transaction outcome (either commit or abort) is consistent across all resource managers involved in a transaction.

3.2 Our Transaction Component

3.2.1 Challenge

The component we have created solves the classic banking case where we want to transfer money between two accounts. When money is transferred between the accounts, there are a few things that require special attention. We have to make sure that the same amount that is subtracted from the sender is credited the receiver. If anything goes wrong during the transfer, we need to roll back the entire transaction.

3.2.2 Component Design

We designed of three different components: Transfer, Credit and Debit.

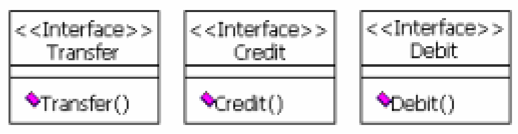


Illustration 3.5 UML Class Diagram

The three components have interfaces called Transfer, Credit and Debit. They only have one member function each, with the same name as the respective interface.

The *Transfer* component handles money transfers from one account to another. To do this, it relies on the two other components: *Credit* and *Debit*. By splitting the task into three components, we emphasize the way each of the components doesn't know (or care) about the status of the other components. Transaction Server handles that.

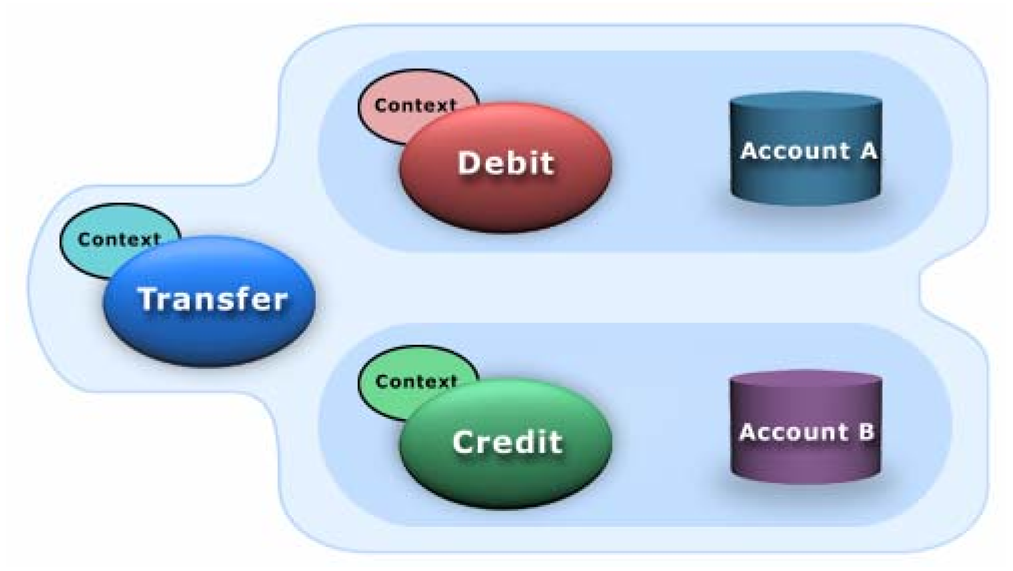


Illustration 3.6 The complete Transfer transactions

This illustration shows three transactions: Transfer, Credit and Debit. The Credit and Debit transactions are both part of the Transfer transaction, in addition to run as individual transactions as well.

3.3 Implementation

3.3.1 Implementation vs. Class Diagram

Since Visual Basic, Visual C++ and Visual J++ have different methods of implementing features such as callbacks and events, we anticipated slight changes between the methods of the same class in the different languages. In the case of the banking application however, no callbacks or events are used, so the implementation (with regards to methods and interfaces) is the same in all languages.

3.3.2 Implementation Issues

3.3.2.1 Visual Basic

Visual Basic is definitely the easiest of the three languages when it comes to component development. Creating a COM component is as easy as running a wizard, selecting ActiveX DLL, and adding the methods of your interface as public functions or subroutines. When creating an MTS component you also need to include a reference to *Microsoft Transaction Server Type Library*, and set the *MTSTransactionMode property* to indicate in which way the component participates in a transaction. It is also advisable to set the project property *Version compatibility* to *Binary Compatibility* so the component isn’t given a new GUID each time it is compiled.

The only thing the programmers has to worry about is remembering that all components that are participating in a transaction must be created with *CreateInstance* instead of *CreateObject*, and that each component must call either *SetComplete* or *SetAbort* to indicate its state. The fact that Visual Basic manages to hide almost all the details of COM, IDL and interfaces is impressing.

3.3.2.2 Visual C++

C/C++ in general is regarded as one of the most robust, high-performance languages available on any computer, and is the industry standard for commercial applications. Visual C++ is the most widely used tool when it comes to building commercial Windows applications.

The high performance of Visual C++ does come at a cost however. To build COM components with Visual C++, you need to know the basic internal workings of COM, as well as have good knowledge of C++. Similar to Visual Basic, Visual C++ has a wizard that helps you get started on your component (ATL COM AppWizard). Unlike VB, VC requires you to start with creating an interface rather than directly editing code.

Note: Starting by describing its interface is a more politically "correct" approach to component-development.

Calling the MTS *CreateInstance* and *SetComplete*/*SetAbort* is actually easier than VB, since VC automatically adds a member variable named *m_spObjectContext* and initializes it with a pointer to the transaction context.

3.3.2.3 Visual J++

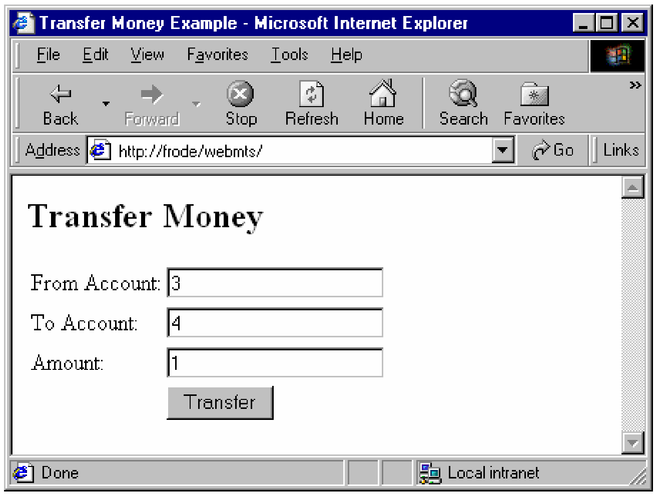
Visual J++ is the newest product in Visual Studio. Using it to create basic COM components are easy, all you need to do is select a checkbox in the option dialog. If you want to create more advanced components though, things get a little more complicated. The first thing we noticed when starting doing more advanced COM programming in Java was that there didn't seem to be a policy on whether or not you should use IDL files. Some examples mentioned the use of IDL files without explaining why or when you need to use them, while others talked about the "ActiveX Wizard for Java" which we to this date have not seen anywhere in the application.

When creating an MTS component we quickly discovered that we needed an interface ID in order to create components using *CreateInstance* (which is necessary for them to be part of the transaction). The documentation said nothing about how the interface ID was assigned or how we could define it ourselves.

Our general impression was that COM in VJ++ was harder than it should be. The documentation was lacking, and there seemed to be a lack of a general strategy for how to create COM components with the tool.

3.3.3 Performance Testing

In order to test the performance for each component we had developed, we used *Microsoft Web Application Stress Tool*. For the testing we created a web page which used our MTS components to transfer money between two accounts.



Transfer Money

From Account: 3

To Account: 4

Amount: 1

Illustration 3.7 Web-based client for the MTS components

This example screenshot shows the parameters for transferring 1 unit of money from account 3 to account 4.

We recorded a script that transferred money from one account to another, and let the stress testing tool run it over and over again for 20 minutes. Six threads were running continuously, which was the optimal stressing limit before the web server got overloaded and performance dropped dramatically. The test computer was not a robust server, but a Pentium 200 workstation running Personal Web Server.

Table 3.1 Stress testing results for the MTS component

The table below shows the results after stress testing three components developed in VC, VJ and VB. Each component was stressed with the exact same job, which was to transfer money from one account to another using MTS and a database connection.

	Requests/Sec	**Requests Total**	**Current Connections**
VB	24.38451219	29987	5.46341463
VC	23.06543209	28031	5.67901234
VJ	22.88352535	27978	5.58821129

The chart below presents the Requests/Sec column from the table above. It may seem somewhat strange that VB yielded best performance. Our VSA analysis indicates that VB takes better advantage of component-pooling than the other languages. As a result of this it is slightly faster.

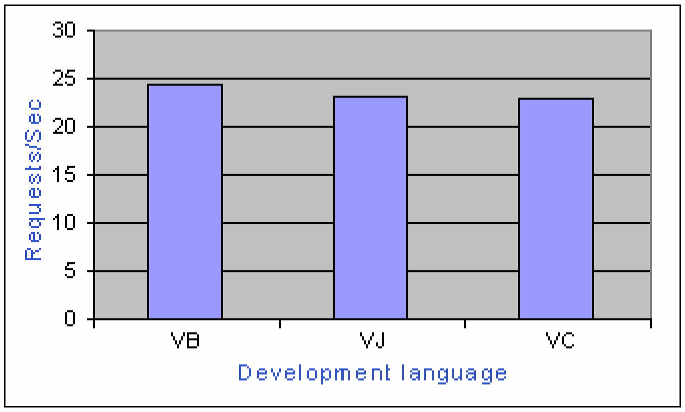


Illustration 3.8 Chart with the stress testing results

Stress testing results showing performance presented as Requests/Sec for the components developed in VC (Visual C++), VJ (Visual J++) and VB (Visual Basic).

3.4 Discussion

Of the three tools we used to create Microsoft Transaction Server components, only one (Visual J++) presented us with implementation problems. That was due to awkward support for COM/MTS in Visual J++ rather than a problem with COM/MTS itself.

We think the three development tools we selected (Visual Basic, Visual C++ and Visual J++) is a fair representation of development tools in general, so MTS should be no harder to use with development tools from other vendors than Microsoft.

During our testing MTS performed well as a transaction manager, and the ease of use makes it easy for programmers to write components that support transactions. Even when there is no need for a transaction, the *object-pooling* capability of MTS makes it worthwhile for components that are frequently used.

The ability to configure MTS components with a dedicated tool makes MTS attractive in an enterprise distributed system, because it allows a network administrator to control and administer the system down to component level, rather than leaving the details to the programmer.

Writing components that take advantage of Microsoft Transaction Server is easy, understanding distributed systems and transaction theory well enough to design effective systems is a bit more demanding. Our conclusion is that MTS is a powerful technology well worth spending some time learning.

4 Microsoft Message Queue

This chapter is divided into four parts:

- Message Queuing Technology
- Our Message Queue Component
- Implementation
- Discussion

4.1 Message Queuing Technology

4.1.1 Message Queuing Theory

With the trend moving toward distributed computing in enterprise environments, it is important to have flexible and reliable communication among applications. Businesses often require independent applications running on different systems to communicate with each other and to exchange messages even though the applications may not be running at the same time.

There are two types of message communication; synchronous and asynchronous. *Synchronous*, or message passing communication, requires that both communicating applications are up and running because data is being passed directly between them. Telephone conversations implement synchronous communication. *Asynchronous*, or message queuing communication, does not have that requirement. It allows applications to communicate indirectly through a message queue. Voice mail, where the message is queued and later retrieved, is an example of asynchronous communication.

Many things can prevent messages from being delivered over a network. Network connections can be broken, the receiver application may have crashed, servers are down or the network can be overloaded. This is very serious, since one of the most critical factors of a distributed system is its uptime, defined as the percentage of time the system is available to its users. If messages cannot be passed over the network, the system is not available.

The consequence of this is that the users will be unable to use the system until all necessary components are operational. For example, consider a data-collection process where information is continuously being entered on a client computer, sent across the network and stored on a server. Just a single network failure or server crash can cause the process to stop because data can no longer be registered. An alternative way of registering data is, of course, to use pen and paper and type in the data later. This is usually not an acceptable option.

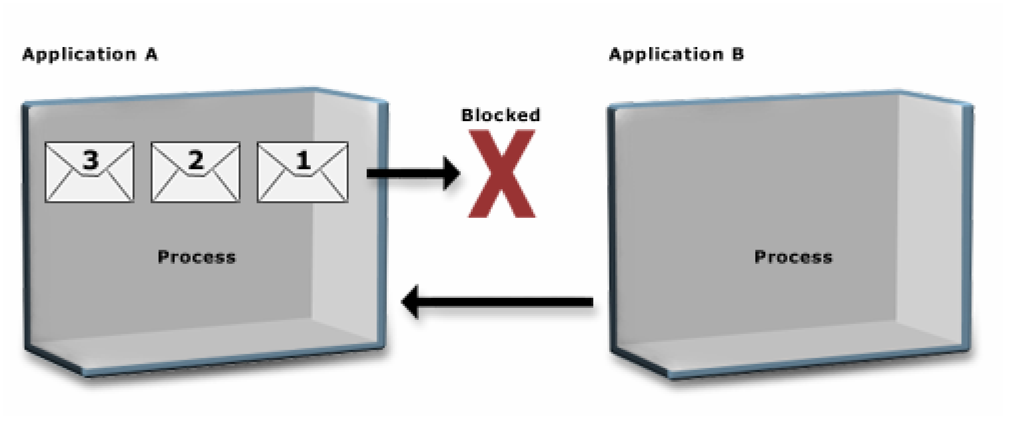


Illustration 4.1 Blocked connection

Blocked connection between two processes preventing them from exchanging messages.

To enable applications to use asynchronous messages, message queue systems have been developed. A *message queue* allows an application to send messages without worrying about whether the receiver is operational or not. Maintaining a separate queue on the sender and the receiver makes this possible. A message will not be removed from the local (sender) queue until the receiver has given an acknowledgement on the reception of the message. This way, even though the network is down, messages will be delivered as soon as the connection between the sender and the receiver is operational.

4.1.2 Microsoft Message Queue Overview

Microsoft Message Queue is a technology that guarantees error-free message delivery between applications in a network, even if you're not able to communicate with the system for a period of time. MSMQ is a standard part of Windows NT Server (introduced with NT Option Pack), like Microsoft Transaction Server (MTS) and Internet Information Server (IIS).

Message queuing assumes asynchronous communication. The client application sends its messages via an interface to a queue, which works as a buffer. The server will invoke the same queue, and collect all the messages when it is ready to process them. In this way, the client application has no direct contact with the server. It only depends on being able to deliver the messages to the queue. Communicating this way makes it a lot easier for the server to process the messages as well, since it only has one client to respond to – the queue.

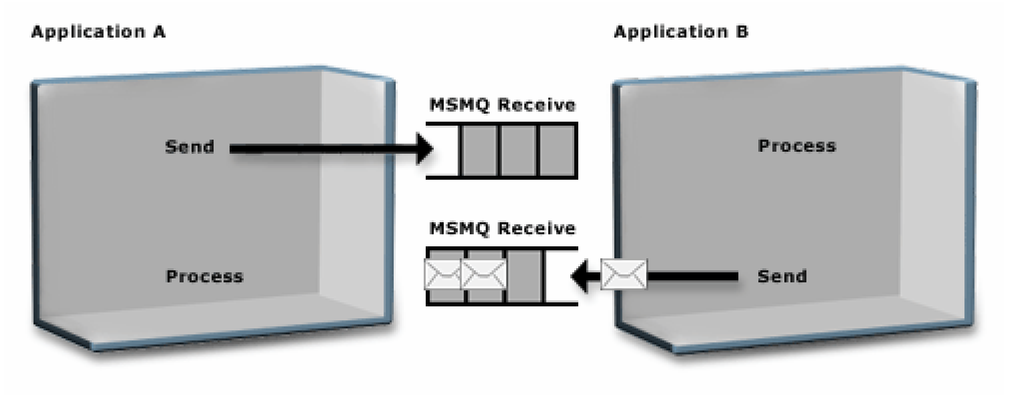
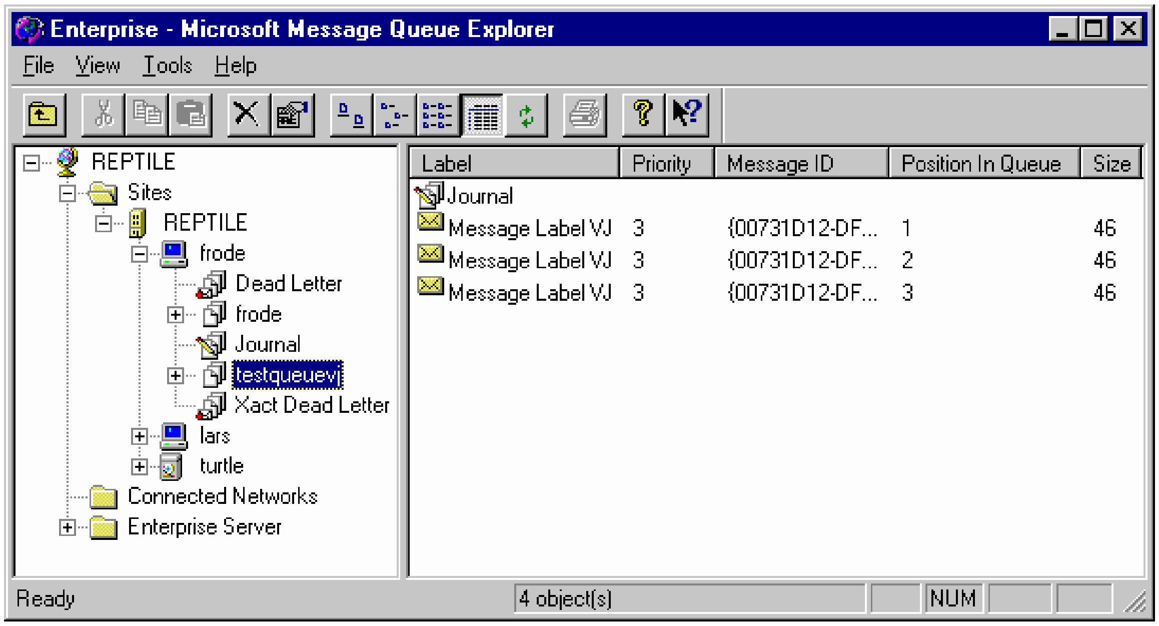


Illustration 4.2 Message delivery via Microsoft Message Queue

The illustration shows two processes exchanging messages using MSMQ.

The administration tool is called MSMQ Explorer. It is used to create, administer and view sites, connected networks, computers and queues on your MSMQ server.



Label	Priority	Message ID	Position In Queue	Size
Journal				
Message Label VJ	3	{00731D12-DF...	1	46
Message Label VJ	3	{00731D12-DF...	2	46
Message Label VJ	3	{00731D12-DF...	3	46

Illustration 4.3 Microsoft Message Queue Explorer

MSMQ consists of three basic components (Illustration 4.4):

- Interfaces
- Messages
- Queues

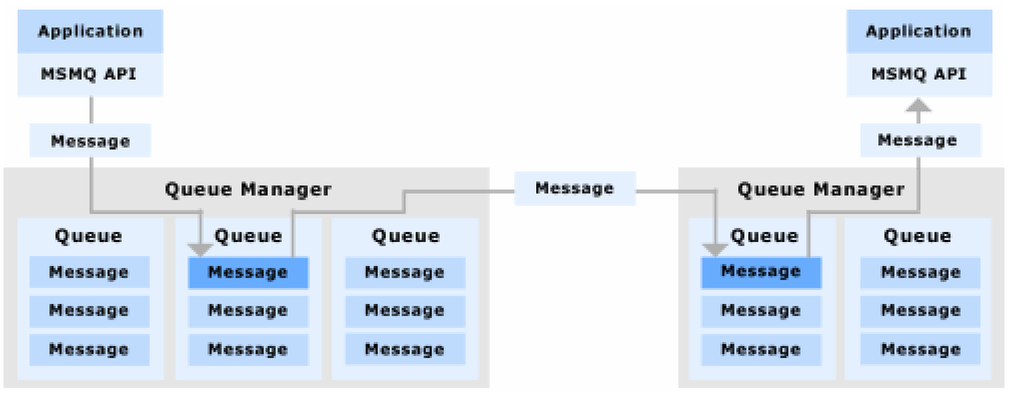


Illustration 4.4 The three basic components in MSMQ

The illustration shows how two applications communicate via MSMQ, and how MSMQ is made of interfaces, messages and queues.

Interfaces are provided so that applications can communicate with MSMQ, and be able to send and receive messages. MSMQ actually provides two APIs. One API is defined as a set of C function calls, and the other API is defined as a set of COM objects. The COM API includes more services, and can be used by all programming languages that support COM.

Messages are created, sent and received by applications. They can either be stored in memory or on disk, where memory storage is faster and disk storage provides more reliability. Each message can be up to 4 MB in size. Some messages are time critical, so if they have not been received by a certain amount of time, it may be appropriate to throw them away. MSMQ takes care of this, and discards the messages that have timed out. Messages can time out either for not being received or not reaching the queue within its time limits. Also, messages can be given a priority between 0 and 7, allowing time critical messages to be processed more quickly. If desired, MSMQ can create a log with all messages processed by a queue. Other great features include encrypting, digitally signatures and more.

Queues are managed by a queue manager, where messages are sent into and received. The queue managers can communicate with each other to send messages from one queue to another. Applications and people with the right permissions can set properties for queues, however, the only property required is its pathname. The pathname is just a character string (such as “machineX/myqueue”) identifying the machine the queue is on and the name of the queue. Other properties define things like type of queue, maximum size in bytes that a queue can hold, routing between queues and more. Queues can also be configured to participate in transactions using Microsoft Transaction Server.

Microsoft Message Queue defines three types of applications (Illustration 4.5):

- MSMQ server
- Independent client
- Dependent client

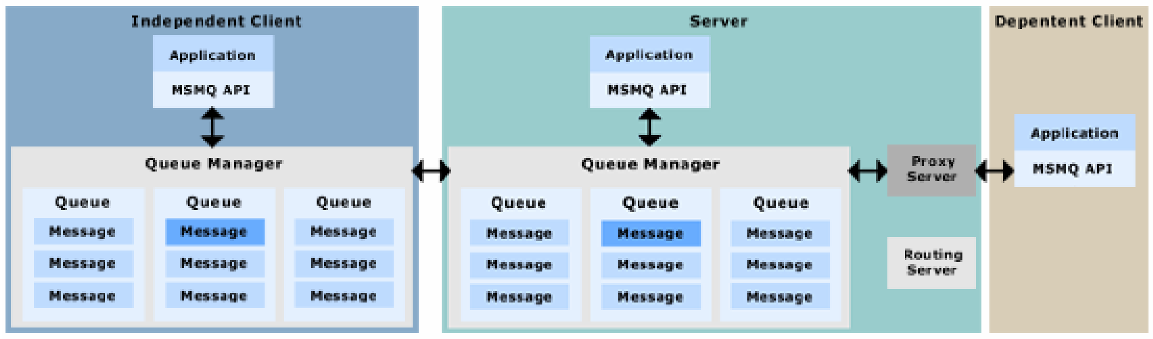


Illustration 4.5 Three types of applications defined by MSMQ

At least one *MSMQ server* must be available in every MSMQ installation. The most important things it contains are queues, a queue manager, support for the MSMQ API and software to route messages between queues.

Independent clients support the MSMQ application programming interface (API), and they also have a queue manager with their own queues. Applications running on an independent client are able to send messages even if they are not online with the MSMQ server, because message sent by the application are stored in the client's queue. This is a great advantage if the network goes down, or if the client is running on a laptop and is not always wired to the network. When an independent client reconnects to the network, its queue manager automatically detects this and forwards all messages to an MSMQ server.

Dependent clients are more limited than independent clients. They provide all MSMQ APIs, but require an available network connection to an MSMQ server in order to operate. For this purpose, the MSMQ servers contain a *proxy function* to support these systems. Dependent clients are intended for systems with permanent connection to a network, such as a desktop machine connected to a LAN. However, there is one advantage with dependent clients. Since dependent clients have no queues on their own, an MSMQ environment with dependent clients is easier to manage than if independent clients were used.

While the basic idea of message queuing is simple, the kinds of things applications want to do with messages are often not so simple. Accordingly, MSMQ provides a powerful set of services for those applications that need it.

4.2 Our Message Queue Component

4.2.1 Challenge

The system we have been looking at is a data collection process, where data is being entered on a client and stored on a server somewhere else in the network. We have used a melting process at a steel mill as our fictive case. During this process, data is being registered by the people supervising the process. This data is very important for further analysis in order to improve the melting process and accurately calculate costs associated with the particular batch being produced. If the network is unavailable, the workers are unable to register the data, and must write down the information by other means (i.e. pen and paper). In a real-time production environment where the focus is on using the most cost-effective production method, unexpected delays can be both costly and create frustration among the workers.

To solve this problem, we need to design a component that let the user register data on the client, regardless of any possible problems with the network connection. Our component must be able to receive data from the client, even if the network fails, and pass it on to the server. The data storage on the server should be transparent to the user.

4.2.2 Component Design

We have designed two components, one for the sending application and one for the receiving application.

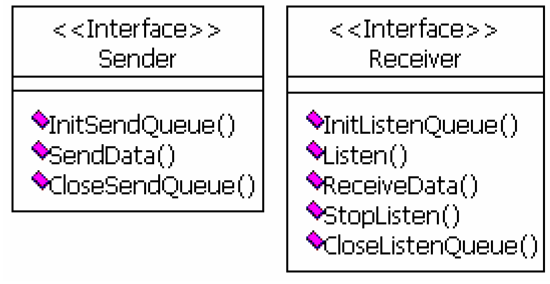


Illustration 4.6 UML Class Diagram

The sender class is the smaller of the two classes, containing only three functions. *InitSendQueue* initializes the queue, *SendData* sends a message to the queue, and *CloseSendQueue* closes the queue when the application exits.

The receiver class is a little more complex, but is still quite simple. We have *InitListenQueue* and *CloseListenQueue* which create and close the queue respectively. Since the receiver is the passive part of the message process, it needs to check the queue regularly to see if any messages has arrived. This is done with the *Listen* function. When a message arrives, the Listen function calls *ReceiveData* to retrieve the message. Listen is called once per message, so after ReceiveData has processed a message, it calls Listen again to wait for the next message. When the application is shut down, *StopListen* is called to abort the Listen routine.

4.3 Implementation

4.3.1 Implementation vs. Class Diagram

The only difference in implementation between the languages was the naming of the callback/event procedure that was called/fired whenever the receiver needed to be notified of a new message. Even though this is more a matter of syntax than a design change, we mention it here to avoid possible confusion if someone choose to read the source code.

4.3.2 Implementation Issues

4.3.2.1 Visual Basic

The Visual Basic component was created as an *ActiveX DLL*. VB hides the details of COM in a wonderful way. We just needed to implement MSMQ , which we did by adding the *Microsoft Message Queue Object Library* to our project. This gave us access to all the objects in the MSMQ API. There were no specific problems in VB.

4.3.2.2 Visual C++

When you know how to create COM components with Visual C++, the threshold for using MSMQ is not high. We created our project using the *ATL COM AppWizard*, which gave us the fundament for our component. Furthermore, in order to get access to the MSMQ API, we included the dynamic link library *mqoa.dll*.

From here, we had no trouble using the MSMQ functionality in our component. It’s the same as implementing MTS; the technology itself is not hard to implement, however, handling COM is a bit trickier in VC than VB.

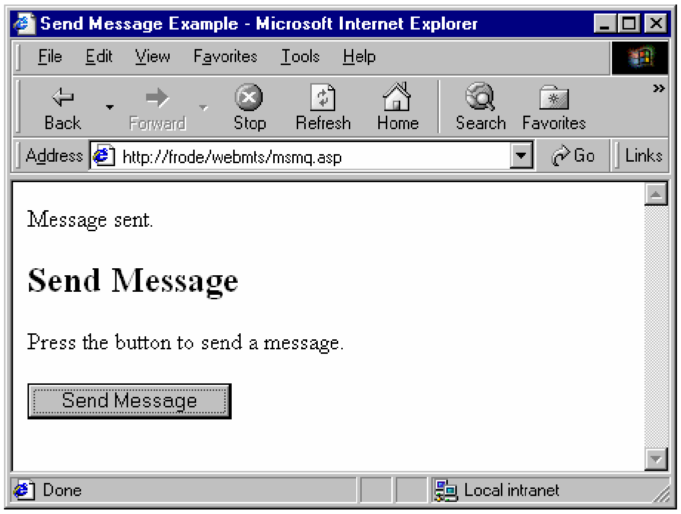
4.3.2.3 Visual J++

The implementation in Visual J++ gave us a bit more trouble than VB and VC. Like in VC, we first created a new project using the ATL COM AppWizard. Then to gain access to MSMQ, we had to add a *COM wrapper* to the project called *Microsoft Message Queue Object Library*.

When it came to calling the MSMQ functions, we learned that the syntax in VJ was a lot "uglier" than in VB and VC. For example, when calling a function called *LookupQueue*, we had to pass nine parameters in VJ while one was sufficient in VB and three in VC. In addition, it was not possible to look up a queue using the queue name in VJ, so we had to look it up using the queue's Type ID. These implementation differences gave us some trouble, since the documentation for VJ was poor compared to the other languages.

4.3.3 Performance Testing

Like we did on the MTS components, we used *Microsoft Web Application Stress Tool* to test the performance for our implementations of the MSMQ component. For that use, we created a simple web-based client.


Send Message Example - Microsoft Internet Explorer

http://frode/webmts/msmq.asp

Message sent.

Send Message

Press the button to send a message.

Illustration 4.7 Web-based client for the MSMQ components

This example screenshot shows client interface that sends a message using the MSMQ component.

The script that we recorded with the stress testing tool simply sent messages to an MSMQ queue using this web client. The script was running for 20 minutes, with nine threads stressing the application continuously (nine threads was the limit before the web server became overloaded, and performance decreased for all components). The test computer was a Pentium 200 workstation running Personal Web Server.

Table 4.1 Stress testing results for the MSMQ component

The table shows the results after stress testing three components developed in VC, VJ and VB. Each component did the exact same job, which was to send a message to a queue in MSMQ.

	Requests/Sec	Requests Total	Current Connections
VC	24.69250914	29994	8.32098765
VJ	18.90695122	23215	8.02439024
VB	16.99414634	20890	8.31707317

The chart below presents the Requests/Sec column from the table above, where we can see that Visual C++ gave best performance in the test.

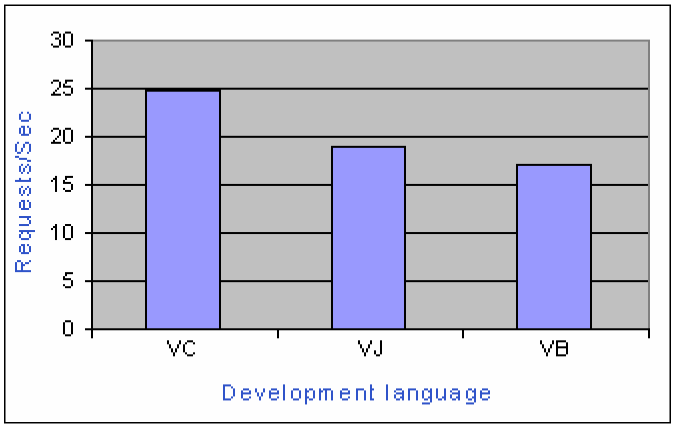


Illustration 4.8 Chart with the stress testing results

Stress testing results showing performance presented as Requests/Sec for the components developed in VC (Visual C++), VJ (Visual J++) and VB (Visual Basic).

4.4 Discussion

Our experiences with implementing the Microsoft Message Queue (MSMQ) in the three languages we selected (Visual Basic, Visual C++, Visual J++) closely match the experiences we had implementing the Microsoft Transaction Server Component.

Of the three languages, Visual Basic was the easiest to work with, hiding all complexity in a well thought out manner. Visual C++ has a steeper learning curve, and you need a more detailed understanding of how COM operates. But in the whole VC presented us with few difficulties. Again Visual J++ turned out to be the rotten apple among the three, with more complex function calls, poor documentation, and what seemed to be less functionality than the other languages. It is possible that there are better ways of using COM and MSMQ from VJ, but with the documentation currently available it’s hard to say.

In general, we found MSMQ easy to work with. The concept of message-queing is straightforward, so that should not be an obstacle for programmers wanting to take advantage of MSMQ. Like MTS, Microsoft Message Queue has much to offer to those developing distributed systems.

5 Conclusion

Our goal has been to look at the performance of components written in a selection of programming languages. We wanted to find out if any languages were more suited to building components than others with regards to performance and complexity. Furthermore, we wanted to evaluate *Microsoft Transaction Server* and *Microsoft Message Queue* to see if they represent good solutions when it comes to *transaction services* and *message queuing*.

While working with distributed systems through this study and through our work experience, we feel that we have an idea of what the status of this area is today and where it is headed. We will share these ideas at the end of this chapter.

When developing the components, Visual Basic was our starting point. VB is widely known for its ease of use and its powerful features, although some developers using more advanced languages often describe it as a toy rather than a real programming language. The reason for starting out with VB was simple – VB hides nearly all the complexity of *COM*, making it the perfect tool for prototyping COM components. When it comes to development time, neither Visual C++ nor Visual J++ stand a change against VB.

Note: Our previous programming experience is about 50% VC, 30% VB and 20% VJ. In our opinion this does not significantly affect our evaluation of the three languages, but ultimately we leave that decision to the reader.

Visual C++ requires the programmer to know the basics of how COM works in order to be an efficient tool. Even though VC includes a wizard that goes a long way in creating the component for you, it presents you with questions on threading model, interface and aggregation, as well as reference explicit COM interfaces. Without the proper knowledge, it is easy to make mistakes that could drastically affect the functionality and performance of your component. Still, it is hard to put down VC because of these things, because it is in the nature of VC to give the programmer total control rather than sacrificing control or performance to make things easier. As the only one of the three languages, VC requires you to start out by declaring the interface of your component through an *IDL file*. All in all, we think Visual C++ is an efficient tool for creating high-performance components, given that the programmer has the necessary skill.

Visual J++ should in theory be the most suitable language for developing components. As a strong object-oriented, modern language with true multi-platform compatibilities, Java has certainly gained its share of attention. Strangely enough, we found VJ to be the tool where we ran into most trouble when implementing the components. The COM support in VJ is more confusing than in the other languages. For example, there does not seem to be a clear strategy on whether or not you should use IDL files when programming in Java. Turning a java class into a component is as easy as selecting a checkbox. The documentation we found on VJ and COM was only dealing with the most basic issues. Documentation about components participating in transactions was lacking.

Overall, we found both Visual Basic and Visual C++ to be good choices for developing COM components. Working with Java and COM was very confusing, and better documentation would be most welcome. In addition, using Java with COM ties your solution to the Windows platform. In the context of this report this is not a problem since we are looking at *Windows DNA*. But when looking at the big picture, tying Java to Windows removes one of the best features of Java, which is true multi-platform support. We recommend that anyone who has chosen Java as their main language for distributed applications at least evaluate other technologies which retain Java’s multi-platform support before going with COM. One such technology is *Enterprise JavaBeans*.

Microsoft Transaction Server is one of the main technologies we have been looking at. After implementing a simple MTS component in three different languages, we feel confident that we have a solid background for evaluating it in terms of ease of use and functionality. For the average programmer the technical side of MTS programming is very easy to learn. To properly understand transaction theory and know when to use it is likely to be more of a challenge. As for the quality of the transaction service, we found it very powerful and robust. The component-pooling capabilities are a further incentive to consider using MTS, even when a transaction is not required.

Likewise, Microsoft Message Queue is easy to use, and message queuing should be a fairly easy concept to understand. Learning to implement the MSMQ technology is a bit more work than learning MTS, mainly because more objects and methods are involved. However, no part of it is too complex for an intermediate programmer. For successful use of MSMQ, it is important to understand when asynchronous communication is important. Using MSMQ in a setting where the component waits for an answer after sending a message must be avoided.

To summarize MTS and MSMQ, we found that both technologies are easy to use, and they include powerful features that make it well worth spending some time learning. MTS and MSMQ may be all it takes to turn an unusable distributed system into a good solution. As the trends move towards more use of distributed systems, and with Windows-based systems in almost every company, MTS and MSMQ will certainly be part of many future solutions.

When developing distributed systems, performance is a key concern. It only takes one poorly developed component to slow down an entire system. When a distributed system is experiencing a decrease in performance, it is often difficult to locate the bottleneck that is causing the slowdown.

To help developers analyze performance in a distributed system, tools like Visual Studio Analyzer have been created. We expected *Visual Studio Analyzer* to give us more performance information that it actually did, so it turned out to play a smaller part in our performance analysis than we initially thought. There are several reasons for this. The most important being that our components spends most of their time using other components like MTS and MSMQ, so the differences between the languages became marginal.

Once we started stress testing our components using Microsoft’s Web Stress Application (nicknamed Homer), this became even clearer. There were a few surprises though – VB actually outperformed VC in the MTS test. We found the reason for this with VSA. VB was able to take better advantage of component-pooling than VC.

Note: Our limited stress testing was performed in a real-world scenario on one of our test-servers, and as such do not meet the standards required for commercial lab-testing.

Distributed systems are the latest trend within the software industry. Even though much has been said and written about distributed systems, our impression is that many spend to little time analyzing and building the right architecture for their solutions. The consequence of this is that many “distributed-systems” are closer to client-server applications than true multi-tier distributed systems.

Without proper planning components tend to remain as part of the system they were built for, and never realize their full potential as reusable components. The main reason for this is that developers use their experiences from building client/server applications, and merely package their classes within components. The same applies to how the development tools are selected. Programmers stick with the tools they are familiar with, rather than evaluate which tools are the best for the job at hand.

From a practical point of view however, it is hard to criticize developers for using the tools they know. Many software projects have stranded because developers overrate their skills at adopting new tools. In other cases it can be hard to justify the expenses involved with learning and buying new tools. Making the correct choices between introducing new technology or leveraging existing knowledge is a critical factor when it comes to building distributed systems.

In our opinion, the essence of a distributed system is proper analysis, design and architecture. It is important to make the right choices when it comes to tools and technologies, and re-use existing components whenever you can. That is what distributed systems are about. With tools like Visual Basic and Visual C++, and technologies like COM, MTS and MSMQ, developers are much more likely to successfully make the transition from client/server development to the new world of distributed systems.

6 References

Windows DNA: http://www.windows.com/dna/

Component Object Model: http://www.windows.com/com/

Visual Studio: http://msdn.microsoft.com/vstudio/

Visual Java: http://msdn.microsoft.com/visualj/

Visual Basic: http://msdn.microsoft.com/vbasic/

Visual C++: http://msdn.microsoft.com/visualc/

NT Option Pack (MSMQ & MTS):
http://www.microsoft.com/NTServer/web/exec/overview/option_pack4.asp

The Evolution of Windows DNA:
http://www.microsoft.com/dna/discover/evolution.asp

MSDN: What Is Visual Studio Analyzer?

MSDN: Visual Studio Analyzer Reference

MSDN: Visual Modeler Reference

MSDN: Developing MTS Components with Java

MSDN: About MSMQ

MSDN: Changes to MSMQ COM Components in MSMQ 2.0

MSDN: Using MSMQ ActiveX Components from Java

CD:
Mastering Distributed Application Design and Development Using Microsoft Visual Studio 6.0
http://msdn.microsoft.com/training/courseware/1298_sp.asp